FEASIBILITY STUDY:

REFERENCE ARCHITECTURE FOR

MACHINE CONTROL SYSTEMS INTEGRATION

Thomas R. Kramer

M. K. Senehi

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

Acknowledgements

Partial funding for the work described in this paper was provided to Catholic University by the National Institute of Standards and Technology under cooperative agreement Number 70NANB2H1213.

In collecting papers to read for the study described in this report, the authors were helped by already having a large collection assembled for the Manufacturing Systems Integration project by Moonkee (Daniel) Min. Several additional papers were provided by Richard Quintero.

Executive Summary

1 Introduction

As industrial equipment becomes more sophisticated, computers and communications more powerful, and robots more capable, the need for a method of unifying diverse machines into coherent systems becomes increasingly urgent. The unification of diverse systems can be accomplished using a machine control system architecture. Without the consistent approach provided by an architecture, integrating variegated equipment into a system that does useful work is a tedious, labor-intensive, error-prone undertaking. Despite the wide acknowledgment of the benefits of a widely applicable machine control architecture and the development of many specific machine control architectures, no broadly applicable architecture has gained widespread acceptance.

To address the need for a widely applicable and broadly accepted machine control architecture, the Robot Systems Division (RSD) and the Factory Automation Systems Division (FASD), branches of the Manufacturing Engineering Laboratory at the National Institute of Standards and Technology (NIST) have been developing and experimenting with architectures for more than sixteen years. This work indicates that there are aspects of control which are common to all control systems in a broad range of applications. These aspects have been captured in a number of control system reference architectures, most particularly the Real-Time Control System (RCS) architecture and the Manufacturing Systems Integration (MSI) architecture. These architectures share many common elements, but there are also some differences.

RSD and FASD are engaged in a joint project to assess the feasibility of formulating a single reference architecture and to outline the architecture. This report is written primarily for the team of researchers charged with developing the joint architecture. The report strives to provide team members with an understanding of the basis on which RCS and MSI were compared, a list of technical issues, a framework for developing the joint architecture, and references to existing work on architectures. The report also provides a preliminary sketch of a joint architecture useful for the applications considered in RSD and FASD. It is expected that such an architecture will be applicable in a wider range of applications besides those considered by RSD and FASD, as well.

Sections 1 and 2 of this report provide an Introduction and Preliminary Definitions. In section 3 we discuss the Definition of an Architecture, giving five elements required for the specification of an architecture. Using these elements, sections 4 and 5 discuss General Architecture issues and Control Architecture Issues, respectively. Section 6 characterizes types of architectures and describes several architectures other than RCS and MSI, to illustrate each type. Section 7 describes the RCS and MSI architectures and assesses their compatibility. A more detailed comparison of the two architectures is given in appendix C. Section 8 outlines the proposed single reference architecture. Section 9 gives conclusions regarding the comparison of architectures, and the formulation of reference architectures. Additional appendices provide a glossary of terms, an annotated bibliography, a list of general architecture issues and control architecture issues, and more.

2 Definition of an Architecture

The definition of an architecture involves tiers of architectural definition and elements of architectural definition.

2.1 Tiers of Architectural Definition

An architecture consists of elements which are more or less concrete. A grouping in which all elements have similar concreteness is called a tier of architectural definition. If we were talking about the architecture of houses, such concepts as surfaces to walk on and load bearing systems might appear at a high tier of architectural definition, while particularizations of these concepts, such as wooden floors to walk on and post-and-beam construction for bearing loads would appear at a lower tier.

In general, the elements defined at a less concrete tier of architectural definition will be more generally applicable than those which are more concrete. Specifying an architecture using several tiers enables the architecture developers to indicate which parts of the specification are intended to be broadly applicable and which are not.

2.2 Elements of an Architecture

At each tier of architectural definition, the definition of an architecture consists of specifying five elements of architectural definition. These are:

- statement of scope and purpose,
- domain analyses,
- architectural specification,
- methodology for architectural development, and
- conformance criteria.

The statement of the scope and purpose of an architecture describes the range of application areas to which the architecture is intended to apply and the general objectives of having an architecture for those areas.

The area of potential application for an architecture is termed its domain. A domain analysis is a systematic examination of the target domain to reveal its essential elements. Commonly used forms of domain analysis are functional analysis, information analysis, and dynamic analysis.

An architectural specification is a prescription of what the pieces (software, languages, execution models, controller models, communications models, computer hardware, machinery, etc.) of an architecture are, how they are connected (logically and physically), and how they interact. The architectural specification is the heart of a machine control system architecture.

A methodology for architectural development is a set of procedures for applying an architecture to an application domain. The architectural specification describes what you are trying to build, and the methodology tells how you build it.

Conformance criteria specify how an element of an architecture is judged to conform to the architecture. Conformance criteria may apply to elements of architectural specification or to methodologies for architectural development.

3 Issues

Over 30 pages of the report are devoted to the initial presentation of dozens of issues. There is no reason to single out a few and no brief way to state them all, so this executive summary simply categorizes, summarizes, or names them. Appendix B is a listing of the issues without explanation. Architecture issues are divided into two sets: general architecture issues and control architecture issues.

3.1 General Architecture issues

The general architecture issues are organized around the elements of architectural definition listed earlier.

The scope of an architecture may be characterized by its extent in several different dimensions: application domain, life cycle, organizational extent, and tiers of architectural definition.

Domain analyses for an architecture should cover functional, information, and dynamic aspects. Appropriate methodologies for conducting the analyses must be used.

Architectural specification issues include: component balance, granularity, architecture definition languages, and a set of miscellany at the lowest tier of architectural definition (hardware, operating system, and processes).

Methodology issues include the use of cyclic development, the use of CASE tools, and the question of how to map architectural components onto software and hardware components at the lowest tier of architectural definition.

Conformance issues include: conformance testing methods, the usefulness of conformance testing, whether conformance to methodologies for architectural development should be required, how conformance classes might be used, and providing for non-conformance.

3.2 Control Architecture Issues

One large set of issues concerns the nature of individual controllers: their functionality, operational states, operational modes, internal workings, interactions with humans, and so forth.

Other sets of issues concern:

- how collections of controllers interact,
- how to specify, generate, and execute tasks,
- what data is required for control and what data handling architecture is suitable for dealing with the data,
- how to provide for process planning, scheduling, and resource allocation,

- how to provide for communications among the components of a control system,
- how to incorporate checks and safeguards in control systems and how to provide for recovering from errors.

Sixteen desirable characteristics of a control architecture have been identified. There is general agreement about what is desirable, although not how to achieve it. Achieving one desirable characteristic may help or hinder in achieving another.

4 Types of Architectures

There is general agreement that three aspects of control system architectures are important: control, communications, and information. These aspects are largely independent but must be integrated for a control system to be effective. Most architectures reviewed for this report focus on the control aspect, but several emphasize information.

Four commonly discussed types of control architecture are: centralized, hierarchical, modified hierarchical, and heterarchical. RCS, MSI, and the proposed joint architecture are all hierarchical. The hallmark of a hierarchical control architecture, of course, is that controllers are arranged in a hierarchy. In the MSI and RCS architectures, controllers are arranged in a special type of hierarchy in which each controller has one superior and zero to many subordinates. Controllers interact through a command-and-status protocol.

5 RCS and MSI

To prepare for formulating a joint architecture, RCS and MSI were studied, and their compatibility was assessed.

5.1 RCS

RCS is an architecture for complex, integrated machine control systems which work in a changing world and keep pace with changes in real time. RCS is intended for applications as diverse as space robotics and discrete parts manufacturing. The elements of an RCS system that do information processing are: sensory processing, world modeling, value judgment, and behavior generation.

The sensory processing function of an RCS system takes sensory data at the lowest hierarchical level, interprets the data, and passes the interpreted data to world modeling.

The world modeling function serves to keep a description of the state of the world. It receives information from sensory processing for updating the world model. It also predicts events and sensory data and answers questions about the world model.

The behavior generation system in RCS is strictly hierarchical. That is, each controller responsible for behavior generation has one superior and zero to many subordinates, for the purposes of performing actions. Superiors interact with subordinates by sending commands to them and receiving status messages from them. Each controller has a number of tasks that it can carry out, and these tasks are understood by the superior of the controller.

The RCS architecture decomposes system activities into hierarchical levels. The levels are characterized by the relative amount of time taken to perform activities and by the relative spatial extent of the activities. Roughly an order of magnitude change in temporal and spatial extent is expected between any two adjacent levels, with activities getting faster and more localized at the lower levels.

5.2 MSI

The goal of the MSI architecture is to integrate the operation of a shop which manufactures discrete metal parts. Particular emphasis is placed by the architecture on the integration of shop planning, scheduling, and control functions in both nominal and error situations. The architecture approaches integration by identifying the systems in the shop which need to be integrated, examining the interactions among the systems, and proposing mechanisms to ensure that these systems function in a cohesive manner.

For systems which interact directly, the MSI architecture defines an architectural unit called a control entity, which consists of a planner and its associated controller. The planner in the control entity is required to support scheduling of plans and may support process planning and batching. The controller in the control entity must support task execution. The controllers interact through hierarchical control. A mechanism for external intelligent intervention, called a guardian, is included in the MSI architecture. A control entity may have as many as five types of interfaces: a planning interface, a controller interface, a guardian to planning interface, a guardian to controller interface, and a planner to controller interface. The MSI architecture can be used with a centralized or a distributed planner, and other combinations of control and planning systems.

For systems which interact indirectly, the MSI architecture specifies that it is sufficient to describe the shared information at a conceptual level and provide guidelines for the access of the information. The description of the shared information is given through a number of information models. The information models and the guidelines for information access form the information architecture of MSI. Three of the most important information models in MSI are the Integrated Production Planning Information Model, which describes the manufacturing environment at a high level of abstraction, the Process Plan Model, and the Production Plan Model. Process and production plans are key vehicles by which information is shared between planning and control systems in the MSI architecture. MSI defines six other information models, as well. Data access guidelines include, for example, the requirement that information which must be shared among systems be placed in a data repository where it is possible for all systems which need this information to access it.

5.3 Compatibility Assessment

RCS and MSI are different enough that a system built with the RCS architecture cannot be expected to be interoperable with one built with the MSI architecture. Many of the differences, however, are the result of differing application requirements. MSI and RCS strengths and weaknesses complement each other. RCS provides for real-time control and operation in a range of environments, from the highly structured to the highly unstructured, which may be highly variable. MSI provides for a high degree of integration of planning, scheduling and resource allocation. MSI specifically addresses error-recovery for resource problems, scheduling difficulties and task failure. It appears to be feasible to define an architecture combining the strengths of RCS and MSI.

6 Proposed Architecture

An outline of a specific proposed architecture is given in this report. An effort involving a larger group of people from RSD and FASD has begun to define a joint architecture fully. That group is not bound by the outline given here, which is given as a starting point. This executive summary summarizes the outline.

The proposed architecture has four tiers of architectural definition:

a domain-independent, application-independent tier (tier one),
a domain-specific, application-independent tier (tier two),
a domain-specific, application-specific tier (tier three),
an implementation tier (tier four).

At each tier of architectural definition, all five elements of an architecture are considered. The lowest tier is not discussed further, as it consists of implementations of the architecture to be built in the future.

The first tier gives many of the guidelines necessary to construct a control system. It is assumed the system being controlled must interact with its environment and react to unpredicted changes in the environment. At this tier, the architecture is intended to be applicable to (at least) factories, robots, autonomous vehicles, construction machines, and mining machines.

The second tier of architectural definition is recommended to be focused on discrete parts manufacturing. This is an important, broad domain requiring the features of existing RSD and FASD architectures, particularly real-time control and integration of control with planning, scheduling, and other required functions.

Tier three, which is domain-specific and application-specific, but is not an implementation, is not delineated further in this report.

7 Conclusion

This report provides a clear definition of an architecture, delineates important issues concerning architectures for machine control systems, presents a sampling of existing architectures, compares the MSI and RCS architectures, concludes that an architecture combining the strengths of MSI and RCS is feasible, and outlines a proposed joint architecture for RSD and FASD. Completing the proposed architecture will require a great deal of work, but the end result will be an architecture which fills the needs for real-time control and information integration.

CONTENTS

FIGURES

TABLES

1 Introduction

As automation of manufacturing systems becomes commonplace, the design, construction, and use of computerized control systems has become an increasingly vital problem. Computerized control systems are typically large, complex systems that are composed of components of lesser complexity which are developed and validated separately.

In a manufacturing environment, components are diverse systems such as production machinery, communications software and hardware, computer hardware, databases, file systems, and production management software.

The purpose of a control architecture is to enable these components to work together in an integrated way to give satisfactory product quality at a reasonable price. At present, for each distinct set of components, a systems integrator defines a unique control architecture. This approach to developing control architectures is expensive, time-consuming and makes diagnosis of system problems difficult. The creation of a control architecture which can be applied to classes of machine control systems—a reference architecture—is therefore highly desirable.

1.1 Benefits of Reference Architectures

As United States manufacturing continues to lose market share in the global market, it is apparent that the manufacturing industry must reduce its costs to be competitive. Consequently, significant effort is being devoted to making manufacturing in the United States more cost-effective.

The Manufacturing Systems Committee of the Department of Defense Manufacturing Technology (DoD ManTech) Advisory Group has recently released a report [Plonsky1] which analyses the distribution of the costs in the manufacturing of defense materials and proposes a plan for focusing ManTech-funded research on specific technical areas to reduce the cost of manufacturing to DoD. While the report focuses upon defense production, the findings of this report regarding costs in the manufacturing environment and technical strategies for reducing costs are relevant to all consumers of manufactured goods.

The report states that, in order to reduce overall manufacturing costs, technology must be developed and deployed that:

(1) produces manufacturing systems that can efficiently make a wide variety of products which are produced in small numbers,

(2) permits the rapid realization of new products.

Since costs involved in the planning, scheduling, and control of factory and supplier processes and operations, the manufacturing support costs, comprise approximately 37% of a manufacturing company’s cost, technologies which reduce these present a substantial opportunity for cost savings.

The study advocates a number of technical strategies to achieve the desired cost-savings. Among these are:

(1) the integration of multi-vendor information systems for customers, vendors and suppliers (enterprise integration),

(2) the development of systems which streamline, automate and integrate manufacturing management (command, control, and communications—C^3),

(3) the development of a comprehensive system for creating quality products (quality management).

The technical barriers for the implementation of these strategies include:

(1) the lack of an information architecture (enterprise integration),

(2) the lack of integration methodologies (C^3),

(3) the lack of manufacturing and industrial engineering support tools (C^3, quality management).

A reference architecture, which provides the framework for components of a complex control system to work together as a whole rather than as a disjoint set, encompasses both information architecture and integration methodologies. Furthermore, a reference architecture can promote the development of interoperable support tools.

In the manufacturing environment, different components of the control system are made by different vendors and are designed to work with humans, but not with other automated systems. Typically, a reference architecture specifies integration rules and standard interfaces among components. By adhering to the standard interfaces and integration rules required by the architecture, different vendors can construct components which are interoperable. Using the interoperable components and system integration rules and methods, components may be integrated to build a machine, groups of machines and people can be integrated to form a workstation, workstations may be integrated to form cells and so on, to any degree of complexity desired. The availability of a reference architecture which defines interoperable components can improve the flexibility, timeliness, reliability, safety and extensibility of control systems.

Once a reference architecture is available which can serve as a standard, tools for building control systems can be constructed and applied, and a body of knowledge about how to apply the architecture can be built. Public availability of the architecture, tools and the knowledge of how to apply them to real-world control problems will greatly reduce the time and cost required for building military and commercial control systems.

1.2 Reference Architectures at NIST

The Manufacturing Engineering Laboratory (MEL) at the National Institute of Standards and Technology (NIST) is conducting research on control of mechanical systems for use in such diverse fields as discrete parts manufacturing, coal mining, under-ice submarining, and space exploration.

As a result of differing requirements in each domain, the characteristics of control systems vary greatly. Nevertheless, more than sixteen years of experience within the Robot Systems Division (RSD) and the Factory Automation Systems Division (FASD) of MEL indicate that there are aspects of control which are common to all control systems in a broad range of domains. These aspects have been captured in a number of control system reference architectures that provide both specifications for the parts of the architecture and their behaviors and methodologies for constructing control systems according to the prescribed specifications. One class of reference architecture (developed by RSD) is the Real-Time Control System (RCS) architecture [Albus1], [Albus2], [Albus3], [Albus5], [Barbera1], [Herman1], [Quintero3] and specializations of it, such as the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Albus4]. FASD reference architectures include the Automated Manufacturing Research Facility (AMRF) control architecture [Jones2], [Jones5], [McLean1], [Simpson1] and the Manufacturing Systems Integration (MSI) architecture [Senehi2], [Senehi3], [Wallace1].

The architectures under active investigation in RSD and FASD share many common features. For example, all consist of a set of controllers arranged in a command hierarchy. In all the architectures each type of controller has its own specialized set of commands it can carry out. All the architectures implement command execution by message passing between controllers, and so forth. But there are also some differences. Timing issues and sensory processing receive more attention in RSD architectures, information integration, scheduling and resource definition issues more in FASD architectures.

An assessment of the feasibility of formulating a single reference architecture using the RCS and MSI architectures has been performed, and it has been determined that such an architecture is possible. RSD and FASD are engaged in a joint project to outline an architecture which includes features of both RCS and MSI. It is expected that such an architecture will be applicable in a wide domain beyond that of the two NIST divisions.

1.3 About This Report

This report documents the work performed in assessing the feasibility of combining the RCS and MSI architectures into a single reference architecture. The report is written primarily for the team of researchers charged with developing the joint architecture. The report strives to provide team members with:

(1) an understanding of the basis on which the two architectures were compared and found to be compatible,

(2) a preliminary list of the technical issues which need to be resolved in formulating a joint architecture,

(3) a framework for developing the joint architecture,

(4) a preliminary sketch of the joint architecture, and

(5) references to existing work on architectures.

The report is not intended as a tutorial and assumes considerable familiarity with control architectures and manufacturing.

1.3.1 Feasibility Assessment Process

In order to assess the feasibility of combining RSD and FASD architectures, the authors first examined approximately one hundred papers about the RSD and FASD architectures, externally developed architectures, and frameworks for architectures. The primary goals of this literature search were to assist the authors in developing a framework for comparing the two architectures and to understand how the NIST architectures relate to other architectures. The literature search was not intended to be comprehensive, but an attempt was made to obtain a wide cross-section of papers on architectures.

The review of the control architecture literature revealed that architectures tend to vary widely in content and emphasis. The authors did not find standard terminology for discussing control architectures, or a standard framework for comparing architectures which was mature enough for immediate use.

To remedy this situation, the authors developed a terminology with which to discuss both architectures and defined a framework for discussing and comparing the two architectures. Using this terminology and framework, the authors developed a set of issues which must be addressed when constructing the joint architecture.

The authors then performed a comparison of RCS and MSI on every issue and combined these results to formulate the conclusion that a joint reference architecture is indeed feasible. Finally, the authors generated an initial formulation of the contents of the joint architecture.

1.3.2 Feasibility Report

The order of the feasibility report differs somewhat from the order in which the work itself was performed. The report presents a more unified view of the conclusions and results of the work than would otherwise have been possible. Sections of the report summarize several key aspects of the work, particularly the issue by issue comparison of RCS and MSI and the results of the literature survey.

Section 2 presents terminology used in the remainder of the report for discussing control architectures. In Section 3, we describe our framework for architectures. Using this framework, we then discuss (Section 4) general architecture issues relevant to any architecture, not just to architectures for control systems. In Section 5, we narrow our focus to issues pertaining to control architectures. There is some overlap between the issues in Section 4 and those in Section 5. Many issues are clearly in one section or the other, but a few could have been placed in either section. Section 6 discusses classifications of architectures and describes several architectures other than RCS and MSI to illustrate each type. Section 7 describes the RCS architecture from RSD, the MSI architecture from FASD, and assesses the compatibility of RCS and MSI. A more detailed comparison of the two architectures is given in Appendix C, using the issues identified in Section 4 and Section 5. Appendix B presents a list of all the issues without discussion, for easy reference. Appendix F presents issues which apply primarily to the RSD architectures.

Section 8 outlines a proposed single reference architecture.

Section 9 gives conclusions regarding the comparison of architectures, and the formulation of reference architectures.

A list of all papers cited in this report is given in the list of references following Section 9. A review of the literature, listing the papers read and the authors' summaries and comments on them is in Appendix D.

Appendix B gives a glossary of terms applicable within this report. Although generated as a result of the literature search, in many cases there is no generally accepted definition for commonly used terms. For these terms, the authors have selected one or more commonly used definitions. No attempt has been made to define every use of each term.

2 Preliminary Definitions

In this report, we will use a number of terms which have a variety of meanings in the manufacturing setting. Although an international standards effort is underway to produce a standard terminology, results of this effort are not yet mature enough for use [ISO4]. To clarify this situation, we give definitions for terms as they are used in this report. Throughout the report, terms which are assigned a specialized meaning are printed in italics the first time that they occur. In some cases, the definition of a term may be further refined later in the report.

An *architecture* gives the design and structure of a system. The class of situations in which an architecture is intended to be used is termed its *domain*. For example, an architecture might apply to the manufacture of discrete parts. An *application* is a subset of one or more situations in the domain of an architecture having similar characteristics. A particular shop, with a specific set of equipment and configuration is an example of an application consisting of a single situation. The class of 3-axis milling machines is an example of an application encompassing several situations. The realization of an architecture in hardware and software for an application will be called an *implementation* of the architecture.

In this report, a *reference architecture* is defined to be a generic architecture for a domain which is broader than a single situation. Henceforth in this report, we shall use architecture and reference architecture interchangeably.

A complete definition of an architecture requires a number of *elements of architectural definition*. Elements of architectural definition are conceptual entities, which may or may not have any physical realization. The elements of architectural definition are discussed in detail in Section 3.

One of the main elements of architectural definition is the *architectural specification*, which describes the architecture. An architectural specification is a prescription of what the pieces (software, languages, execution models, controller models, communications models, computer hardware, machinery, etc.) of an architecture are, how they are connected (logically and physically), and how they interact. The pieces of an architecture described above have specific meaning to the architecture and will be referred to as *architectural units*. Architectural units are frequently defined by giving each one distinct functional characteristics, although this is not the only mode of definition. We shall refer to the realization of an architectural unit in an implementation as a *component* of the implementation.

3 Definition of an Architecture

An architecture consists of architectural units each of which is more or less concrete in nature. Architectural units of similar concreteness can be grouped together to form a cross section of the architecture where all units are at the same level of concreteness (or abstraction). We shall refer to such a grouping as a *tier of architectural definition*, or simply *tier*[1]. The concept of tier of architectural definition appears under different names in [Biemans1, section 2.4], [Bohms1, throughout], [Dornier2, section 3], and [Michaloski1, page 2-2]. A complete specification for an architecture includes an architectural specification for each tier.

At each tier, the definition of an architecture consists of specifying a number of elements of architectural definition. These are:

(1) statement of scope and purpose

(2) domain analyses

(3) architectural specification

(4) methodology for architectural development

(5) conformance criteria

These elements of architectural definition vary in indispensability. For example, an architecture must have an architectural specification, but it is possible to use an architecture which omits conformance criteria. Definitions of existing reference architectures include different subsets of these elements of architectural definition and place emphasis on them in varying degrees. However, an architecture which is completely defined addresses all elements of architectural definition in a balanced fashion.

The remainder of this section expands on the notions of tiers and elements of architectural definition.

3.1 Tiers of Architectural Definition

Tiers of architectural definition, as defined above, are distinguished by their degree of abstraction, with lower tiers being more concrete than upper ones. It is useful to define tiers by identifying some specific aspect (or a set of aspects) that is broad in a higher tier but becomes narrower in the next tier down.

To illustrate the idea of tiers, we consider a hypothetical architecture with three tiers. The highest (most abstract) tier is characterized by being applicable to a broadly-defined domain (discrete parts, perhaps).Within this broadly-defined domain, there are any number of applications which are deemed by the developers of the architecture to

1. An alternative, somewhat more intuitive term for this concept would be *level of abstraction.* The term *tier of architectural definition* was chosen in order to avoid confusion with other uses of the term *level* (notably hierarchical level of control), both in this report and in prevailing control architecture literature. The notion of hierarchical levels of control is used throughout this report and is the focus of the first section of Appendix E.

differ significantly from each other. For each application specified, the middle tier of the architecture specifies the application-dependent features of the architecture and has architectural units which generically identify functions, information and dynamics. For each application, a number of implementation-specific architectures may be described. This lowest tier of architectural definition is implementation-specific and describes the mapping of architectural units onto software and hardware components.

At each less abstract tier of architectural definition, each architectural unit included in the architecture will be described in greater detail. For example, at the highest tier of architectural definition, methods for generating information models might be given, at a less abstract tier of architectural definition, specific information models might be given, and at the least abstract tier, the information models might be implemented as locations in memory or as a database schema. The language used to express the specification may well be different at each tier of the architecture.

Table 1 shows sample sets of architectural units which might be defined at each tier of architectural definition of an architecture which has three tiers. These architectural units are grouped into sets corresponding to the five elements of architectural definition. The table is meant only to give examples; it is not intended to outline any existing or proposed architecture.

Table 1: Sample Architectural Units for a Three-Tiered Architecture

Element of Architectural Definition	Tier of Architectural Definition		
	Top Tier	Middle Tier	Bottom Tier
statement of scope and purpose	domain-dependent, application-independent, statement of scope and purpose	domain-dependent application-dependent, implementation-independent statement of scope and purpose	domain-, application-, and implementation-dependent statement of scope and purpose
domain analyses	activity analysis	domain-specific information analysis	hardware-specific dynamic analysis
architectural specification	the general functionality of system components	logical actuator definitions	actuator hardware specifications
		logical sensor definitions	sensor hardware specifications
	methods for communicating among components	logical connection diagrams	wiring maps
		communications execution model	communications hardware specifications
	templates for information models	information models	database system schemas
	template for task definitions	task definitions	source code for driving actuator
	the type and content of logical control interfaces among controllers	controller hierarchy diagrams	source code for generating commands to subordinates
methodology for architectural development	CASE tool used to define tasks	automatic generator used to write C source code for task	
conformance criteria	correct information modeling language must be used	information models must pass through EXPRESS parser	data files must pass through data file reader

Three tiers are used in Table 1 only because several convenient examples fit into three tiers. The top and middle tiers of Table 1 may be collapsed into one to make a two-tier model.

Different architectures may have different numbers of tiers of architectural definition. For example, the architecture proposed by Dornier described in Appendix E has four explicit tiers of architectural definition. The RCS architecture may reasonably be divided into three tiers and the MSI architecture into two tiers, although neither of the two has explicit tiers in existing descriptions. If all architectures had the same number of tiers of architectural definition, it would be feasible to give a unique name to the architectural specification at each tier (the most abstract tier of architectural specification might be called a canonical form, for example). Since this is not the case, we are using the same term for each tier.

3.2 Statement of Scope and Purpose

The *statement of scope* of an architecture describes the range of areas to which the architecture is intended to be applied. It is useful to identify explicitly items which are out of scope, and to identify general characteristics of the domain which may extend or limit its applicability to other domains. As Biemans and Vissers observe [Biemans1, p. 390], this statement is absent in the majority of proposals for (CIM) architectures.

A *statement of purpose* identifies what the objectives of an architecture are within the given scope. The statement of purpose of an architecture should be a major determinant of the contents of the architecture. If the objective is to achieve interoperability between components of an implementation, it would be expected that definitions of shared information and interfaces between components would be stressed. If the objective is to guarantee real-time performance of the resulting control system, execution models may be stressed.

3.3 Domain Analyses

A critical step which must take place before an architecture can be formulated is to perform analyses of the target domain which reveal its essential characteristics. These analyses are *domain analyses*. The type of analyses done, the order in which the analyses are performed and the language in which the results are expressed are part of the methodology for domain analysis. The results of the domain analyses may be very much different depending on the types of analysis performed and the analysis methodologies used. Architectures often have biases consistent with the view(s) of the domain which the domain analyses examined. It can be difficult to compare architectures which were generated using different domain analysis methods.

Many methods for domain analysis exist. It is beyond the scope of this document to discuss all, or even a broad set of them. In this report, for the purpose of comparing architectures, the authors will adopt a specific set of types of analyses which are widely accepted, described immediately below.

3.3.1 Types of Analysis

An *analysis*, in general, is an examination of the components of some complex and how they relate to one another. The specific types of analysis discussed here conform to that definition.

Commonly used forms of domain analysis are functional analysis, information analysis, and dynamic analysis [Jayaraman1]. Functional and information analysis are particularly well entrenched and have been used in structured programming for many years.

A *functional analysis* of a domain is an analysis of all the activities within the scope of the architecture which a conforming control system is supposed to be able to perform.

An *information analysis* of a domain is an analysis of all the information within the scope of the architecture needed for a conforming control system to function properly.

A *dynamic analysis* of a domain is an analysis of the characteristics of the functions and information in the domain which vary over time during control system operation. It provides qualitative and quantitative information about the sequence, duration and frequency of change in the functions and information of the domain [Jayaraman1, p. 250]. Real-time requirements would be explored in this phase of domain analysis.

3.3.2 Domain Analysis Methodologies

The triple of functional, information, and dynamic analyses is supported, for example, by languages for expressing analysis results developed under the auspices of the United States Air Force’s Integrated Computer-Aided Manufacturing (USAF ICAM) program.[2] The associated methodology specifies that functional analysis is performed first, followed by information analysis and finally, dynamic analysis. Many alternatives are available. A currently popular alternative is object-oriented analysis [Dewhurst1, Chapter 6]. These techniques mandate a cyclical development cycle with function and information decomposition taking place simultaneously and producing “objects” which have both information and functional content. Overall analysis of the dynamics of the system of objects created is not explicit in this methodology.

3.4 Methodology for Architectural Development

It is important for an architecture to have a set of procedures for refining and implementing the architecture. This set of procedures is called the *methodology for architectural development* for the architecture (which we will shorten to *methodology* when the meaning is clear). The architectural specification at each tier of architectural definition is related to, and used in, generation of an architectural specification for the

2. The USAF ICAM effort developed three modeling methodologies: IDEF0 for functional analysis (function modeling)[Mayer1], IDEF1 for information analysis (information modeling) [Mayer2], and IDEF2 for dynamic analysis (simulation modeling) [Mayer3]. The ICAM effort also developed IDEF1X [ICAM1] for data modeling, to support database design.

other related tiers as specified in the methodology for architectural development. If an architecture has more than two tiers of architectural definition, a methodology will be needed to link each two adjacent tiers.

A methodology may specify top-down decomposition, bottom-up composition or some combination of both in constructing the complete architecture. For example, if the code or specifications for the lowest tier is available, as is often the case when dealing with vendor-supplied equipment, an implementation-independent template for the code may be developed. In this case, the methodology would describe how to use the template.

A methodology for producing an architectural specification at a middle tier of architectural definition from a specification at a high tier of architectural definition might include:

(1) performing an activity analysis

(2) using a CASE tool embodying the high-tier specification to define application-specific tasks, sensors, actuators, etc.

A methodology for producing an architectural specification at a low tier of architectural definition from a specification at a middle tier of architectural definition might include:

(1) rules for assignment of software modules to computing hardware

(2) rules for using software templates

(3) timing analysis

(4) performance measurement capabilities

(5) debug mechanisms

If an architecture lacks a methodology for getting between any two tiers of architectural definition, control systems developers must devise their own methods for making the transition.

3.5 Conformance Criteria

Conformance criteria are criteria which specify how an architectural unit at one tier of an architecture conforms to the architectural specifications of a higher tier, or how a process for building part of an architecture conforms to the development methodology given by the architecture for building that part.

Methods for determining conformance of a component of an architecture might include:

(1) reading source code

(2) checking that documents which are supposed to be in computer-processable format are in fact computer-processable

(3) observing an implementation in action

(4) devising test cases and using them to test control systems

(5) examining documentation of development activities

4 General Architecture issues

The issues listed in this section pertain to architectures in general, not just to control architectures. The purpose of discussing these issues is to develop a framework for discussing and comparing reference architectures for control systems. Throughout this section, we will assume that the definition of the architecture has the five elements of architectural definition which were presented in Section 3.

4.1 Balance Among Elements of Architectural Definition

In the literature, it is common to read about architectures which do not have an explicit scope or purpose, or which omit conformance criteria. Some architectures pay great attention to defining the way in which the architecture should be applied to a real world problem, others do not discuss this. What should be the balance of emphasis in the architecture's treatment of each of the five elements of architectural definition?

4.2 Scope Issues

There seem to be at least three dimensions of architecture scope: domain, life cycle, and organizational extent. These terms are described below.

Natural language seems to be most suitable for the statement of scope and purpose. However, it may be helpful, in addition, to use an N-dimensional space spanning some large range and to identify a portion of the space as being within the scope of the architecture. The selection of axes for this N-dimensional space for the classification of architectural efforts has been one focus for both the work of the CIM-OSA project [Jorysz1] and the work of ISO 184 SC5 WG1. [ISO1]

4.2.1 Domain

For deciding whether a situation to which the architecture might be applied falls in the domain of an architecture, the primary subject matter of the situation is not usually a critical factor. Rather, secondary characteristics of the situation (such as real-time performance requirements, importance of safety, and need for resource sharing, among many others) are likely to be the determining characteristics. The classification of such requirements would be a challenge and to date the authors have not seen such a classification. A large issue is how context-free can an architecture, or part thereof be made.

4.2.2 Life Cycle

The *life cycle* of a control system is the stages in the life of the system. One breakdown of life cycle (with subdivisions) includes: design (conceptual design, engineering design), manufacturing (manufacturing engineering, scheduling, production, inspection), use (testing, operation, maintenance, logistics support), decommissioning, and disposal. How much of the life-cycle of a control system should be covered by an architecture?

4.2.3 Organizational Extent

The *organizational extent* of an architecture is the set of related activities of an organization covered by the architecture. MSI, for example, covers planning, scheduling, and production. [Ting1] mentions factory planning, purchasing and distribution, and personnel management in addition to these.

Even if an activity is not covered in detail by an architecture, it will be useful to provide hooks from the covered activities to the uncovered ones known to be related. Having a conceptual model which gives the interrelationships of all organizational data can form the basis for such hooks.

4.2.4 Tiers of Architectural Definition Issues

As previously discussed, an architecture has one or more tiers of architectural definition. A number of issues with respect to formulating these tiers must be raised. How far from theory to implementation should a reference architecture go? How should that continuum be divided?

Developers using architectures without clearly defined tiers of architectural definition are likely to experience difficulties in determining implementation details and deciding conformance. Fiala [Fiala1] made the following comments about RCS, for example.

> *Using the decomposition-around-equipment rule, a different architecture is obtained for a different set of equipment. … Thus, it would be impossible to define "standard modules" and the corresponding interfaces for the general architecture. … it may be possible to have a base document that describes the general principles, and on top of this define applications, which are specific architectures for specific problems.*

The difficulties Fiala cites are solved by having at least two tiers of architectural definition. The upper tier includes an architectural specification (the "base document" Fiala mentions) and a methodology for architectural development for producing things in the lower tier. The lower tier may initially contain nothing, or it may contain a template for an architectural specification and/or a template for a methodology. The upper tier methodology is applied in conjunction with the upper tier specification to create the lower tier specification (either from scratch or by filling in the template, if there is one) and to create the lower tier methodology if one is required (also either from scratch or by filling in a template).

4.3 Domain Analysis Issues

An architecture reflects the domain analyses upon which it was based. There are many choices to be made in choosing both the type of analyses performed and the methodology of the analyses.

4.3.1 Aspects Covered

As suggested in [Bohms1], one dimension along which an architecture can be analyzed is by identifying the aspects covered. This is distinct from scope. An *aspect* is a cross-cutting view of an architecture from some specialized viewpoint, such as information, communications, or control flow. The CAM-I CIM architecture is cited in [Bohms1, page 120] as identifying five aspects (management structure, information structure, function/activity structure, computer systems structure, and physical structure), while the CIM-OSA architecture is cited as selecting four aspects (function, information, resource, and organization) - as shown on the “stepwise generation” axis in Figure 11 on page 223 of this report. Specifying a set of aspects from which to view the problem domain is essential in formulating an architecture, but often they must be inferred from the architectural specification, since they are not explicitly stated.

Existing architectures place varying amounts of emphasis on different aspects. As previously mentioned, an architecture tends to reflect the domain analysis aspects used. The two most widely accepted aspects are functional aspects and information aspects. We have chosen to discuss functional, information and dynamic aspects of architectures. As already noted, more aspects can be identified.

4.3.1.1 Functional Aspects

The *functional aspects* of an architecture describe what a control system conforming to the architecture does. A functional specification would describe what roles components could fill in the architecture and what functions each of these roles would encompass. For example, both MSI and RCS specify controller hierarchies, in which one controller has the role of superior and many controllers have the role of subordinate (to the previously defined superior). Both architectures then define functions that controllers having these roles should perform. For example, superiors must generate commands for subordinate controllers and subordinate controllers must generate status information.

The choice of language for expressing the results of functional analysis is an issue. It may be stated in natural language or in a formal language. Examples of formal languages used for this purpose are Activity Scripting Language (AcSL) [Dornier2], Structured Analysis and Design Technique (SADT)[Ross1], and IDEF0 [Mayer1].

4.3.1.2 Information Aspects

The *information aspects* of an architecture describe the information required for the operation of an implementation of an architecture. Often, required information is expressed only in data structures of the computer-executable languages (C, C++, Ada, etc.) of the implementation. A different approach is to develop conceptual models of the required information. A *conceptual data model* of a set of information is a description of the information, always giving relationships among the members of the set, usually including the data type of the members of the set, and often giving some of the semantic content of the information. Conceptual models are expressed in formal languages designed for this purpose, such as EXPRESS [Spiby1], NIAM (Nijssen Information Analysis Methodology)[Verheijen1] and IDEF1 [Mayer2]. Some compilers exist which can translate a conceptual model into a specific computer language or a database schema. For example, MSI has several EXPRESS models for factory information which MSI requires an implementation to understand [Senehi2], [Ray1], [Catron1], [Barkmeyer2]. These MSI models may be translated automatically by software tools into database schemas for a specific object-oriented database system.

The existence of “domain-independent” information models for architectures is currently a topic for debate. Some efforts (such as STEP and CIM-OSA) have attempted to construct such models, while others confine themselves to the construction of models for more explicitly limited domains [Barkmeyer2], [Fiala4], [Wavering1].

Another open debate is the relationship between functional and information analysis. Some methodologies insist that these are inextricably intertwined, whereas others view the two as separate stages of analysis.

4.3.1.3 Dynamic Aspects.

The ***dynamic aspects*** of a control system describe how the information and function vary over time. Examples of formal languages used for this purpose are IDEF2 [Mayer3] and IDEF3 [Menzel1]. IDEF2 produces a dynamic model appropriate for constructing simulations; IDEF3 produces a dynamic model which captures the behavioral aspects of the system. Execution models for real-time access and update of information are dynamic aspects of the RCS architecture. The sequence of expected messages defined in the MSI’s control entity interface (CEI) specification is a dynamic aspect of the MSI architecture [Wallace1].

4.3.2 Analysis Methodology

There are many different ideas about how domain analysis should be performed. Some methods recommend that information analysis be done first, followed by functional and then dynamic analysis. Other methodologies start with functional analysis and then do information analysis. Still others insist that both be done simultaneously. It is unclear what the best methodology is.

4.4 Architectural Specification Issues

One indispensable element of architectural definition is an architectural specification. Within the architectural specification at each tier, there are many choices to be made.

4.4.1 Granularity

An ***atomic unit*** of an architecture is an architectural unit which the architecture does not break down further into simpler architectural units. The ***granularity*** at a tier of architectural definition is the size of the atomic units which the architectural specification at that tier addresses. Granularity is a characteristic of a tier of architectural definition, not of an entire architecture. An architecture may have different granularity at different tiers.

What degree of granularity is best at each tier of architectural definition? In an architecture with several tiers of architectural definition, is it reasonable to have granularity become finer at lower tiers of architectural definition?

In an architectural specification, the atomic units generally have to interact to do the work of the control system, and a formal specification of how they interact is required. If the size of an atomic unit is small, the number of types of interaction becomes large and defining them all and understanding their dynamics becomes difficult. If the size of an atomic unit is large, the domain of the architecture is likely to be small, and the overall architecture does not add much information above that specified by the capability of the atomic units.

An atomic unit may either have internal subunits or may be a *black box*. When an atomic unit has internal units, we will refer to these units as *submodules*. Submodules are not architectural units, since the submodule has no meaning outside of the context of the architectural unit. When an atomic unit is a black box, the architecture specifies the functions and interfaces of the atomic unit, but does not place requirements on the internals of the atomic unit. Implementations of the architecture are still free to decompose atomic units further by creating submodules and specifying interfaces between the submodules within a single atomic unit. Any submodules so defined, however, may not interface directly with external atomic units or with submodules of external atomic units. They must go through the interface of the atomic unit of which they are a part. Otherwise, they are violating the architecture and will make the atomic unit of which they are a part non-interoperable.

Atomic units may combine to form *molecular units*, whose interactions must also be specified. And, possibly, molecular units may combine to form larger molecular units. Note that any atomic unit or molecular unit that is recognized by an architecture is, by definition, an architectural unit.

The "Processes" issue discussed in Section 4.4.3.3 becomes important when the granularity of an architectural specification reaches the degree of fineness at which processes are defined. It should be noted that, depending how "process" is defined, a single process may contain multiple atomic units, a single process may correspond to one atomic unit, or a single atomic unit may be composed of many processes.

4.4.2 Architecture Definition Languages

What language or languages are suitable for defining architectures?

Most architectures are defined in natural language (all the referenced papers are available in English), but this is often vague. A degree of vagueness is appropriate at a high tier of architectural definition. In fact, several authors explicitly endorse vagueness. Unfortunately, it is often not clear what is vague by intent and what is vague inadvertently. The areas of intentional vagueness should be clearly defined. This is possible in formal modeling languages, but just is not done in natural language.

The elements of architectural definition are very different and therefore it is appropriate to use different formal languages. An architectural specification (what it is) is quite different in nature from a methodology for architectural development (how to build it), for example, so different languages are likely to be used.

A formal modeling language intended to be used for modeling information is called an *information modeling language*. Examples are EXPRESS, NIAM, and IDEF1X. These languages are suitable for defining items of information in an architecture such as messages, catalogs of resources, or process plans. They can be used for modeling other parts of an architecture but were not built for that purpose. They do not make it easy to state in an architectural specification at a high tier of architectural definition what is expected when a lower-tier architectural specification is generated from it. This represents an opportunity for enhancement of these languages.

When an architecture includes several tiers of architectural definition, it will be appropriate to use different languages for the same element of architectural definition at different tiers. For example, at the lowest tier, the architectural specification should be given in a standard computer language (and even that might be split into source code and object code), while at the highest tier a formal modeling language may be suitable. The MSI project (see Section 7.2) has used the EXPRESS information modeling language for defining the ALPS process planning language [Catron1] and has built tools for automatically generating a schema for a commercial database system from an EXPRESS model. One of the authors has written an EXPRESS model for the NASREM architecture at a high tier of architectural definition [Kramer1]. The Dornier architecture (see Section E.2) uses the IDEF0 language for activity analysis and other formal languages for other purposes.

Formal languages for expressing methodologies for architectural development (which are action oriented) are less well developed than those for expressing architectural specifications (which are object oriented).

Formal languages have several advantages over natural languages:

(1) formal languages are much clearer and less ambiguous;

(2) formal languages provide formal methods of extending abstract models into restrictions of the original domain (subtyping, for example);

(3) models constructed in formal languages may be checked algorithmically for logical completeness and syntactic correctness - for some languages, compilers exist which will do these jobs automatically;

(4) with formal languages, compilers may be written which will produce executable computer code or database schemas automatically from statements in the language - many such compilers already exist.

Because of these advantages, there is a strong case that architectures should be stated in formal languages to the extent possible, and stated in natural language to the extent that formal languages are unable to carry the required information. It is desirable that natural language descriptions equivalent to formal language descriptions be written to accompany statements in formal language, so people unfamiliar with the formal language can understand the intent of the formal model.

4.4.3 Bottom Tier of Architectural Definition

At the lowest tier of architectural definition, the architectural specification describes the physical hardware, software etc. The issues at this tier are many.

4.4.3.1 Hardware

What assumptions about the computer and communications hardware on which a control system runs is it reasonable to make in an architecture? How can assumptions about hardware be minimized?

It is possible (easy, in fact) to design an architecture without regard to physical implementation. The hardware available for implementing a control system, however, has a profound effect on whether and how the architecture can be implemented. Typically, the architecture is either violated or revised when an implementation is built, because of hardware constraints.

4.4.3.2 Operating System

To what extent should an architecture specify the type of operating system used for implementations?

Several levels of operating system may be used to implement a control system: the computer operating system itself (for example UNIX), the language-specific operating system compiled into an executable program (for example LISP or C), and possibly an operating system that appears in source code (for example the Production Management Operating System for the AMRF Vertical Workstation [Jun1, section III.2.2.4.1]).

4.4.3.3 Processes

To what extent should an architecture define the term “process”, the interaction between operating systems and processes, and the role of processes in implementations?

The term “process” has been defined in different ways in different control systems in the past. If a multitasking operating system such as UNIX is used, the operating system probably provides the definition of process. UNIX adds confusion by providing for “lightweight processes” as well as ordinary processes (there is no definition for “heavyweight process”). In a non-multitasking operating system, the control system designer can define “process” whatever way (s)he pleases. One definition of process has been that the function calls at the top level of a “main” routine in a C-language program for implementing a control system will be called processes. Fiala has suggested [Fiala1] that a software entity that implements a unit of a control system which is not further decomposed in the architecture be called a process. If a computer includes more than one cpu board (processor), whatever is executing on a cpu board may be called a process.

4.5 Methodology For Architectural Development Issues

As discussed in Section 3, the second most important element architectural definition is a methodology for architectural development. A methodology tells how to build an architecture and how to apply the architecture to create an implementation.

The issue of what language to use to state a methodology for architectural development was raised in Section 4.4.2.

4.5.1 Cyclic Development

As previously discussed, there are many ways to develop an architecture: top-down from the highest tier of architectural definition, bottom-up from the lowest tier of architectural definition, etc. One commonly used technique is that of *cyclic development*. The idea of cyclic development is that one develops an architecture, assesses the finished product (the assessment would include implementing the architecture), and uses the results of the assessment as feedback to a cycle of refining the architecture. This may be done several times.

One type of cyclic development is prototyping, wherein a vertical slice through all tiers of architectural definition is developed but only a narrow subset of the total intended capabilities of the control system is included in the slice. This results in a working control system with limited capabilities whose performance can be assessed. The lessons learned from the assessment are applied in building the full system.

Many authors explicitly encourage cyclic development. This includes, for example, [Michaloski1, page 1-1], [Quintero3, section 6], [Senehi3, section 1].

The model of an architecture described in Section 3 of this report does not have a formal role for feedback during the development process. The formal model should be augmented with an informal understanding that cyclic development is encouraged. An attempt to formalize the role of feedback in the model may be worthwhile.

4.5.2 Mapping Architectural Components Onto Hardware

However many layers of architectural definition an architecture has, as long as there are at least two, the problem of determining how to map architectural components onto hardware will always arise in building the architectural specification at the lowest tier of architectural definition. What rules can be used for making this assignment?

4.5.3 CASE Tools

What is the role of CASE tools in a methodology for architectural development?

There is no question that CASE (Computer-Aided Software Engineering) tools can be built for nearly any architecture. Given our assumption that an architecture will consist primarily of controllers arranged in a hierarchy performing pre-defined tasks, at least two software modules for a CASE tool are desirable, one for defining tasks and one for defining controllers and controller hierarchies. When an architecture has more than one tier of architectural definition, it may be desirable to have more than one CASE tool.

Many aspects of a methodology for architectural development can be built into a CASE tool. If this is done, using the CASE tool ensures that the methodology is followed. Because of this, it seems very desirable that CASE tools be built.

Building CASE tools alleviates the problem of there being no good formal languages for methodologies. When a methodology is embodied in a tool which developers can use, the tool can enforce the use of that methodology to a great extent.

4.6 Conformance Criteria Issues

The uses of conformance criteria for an architecture are discussed in Section 3. A *conformance test* is a procedure that determines if conformance criteria have been met. In the sections below, conformance testing issues are discussed.

4.6.1 Conformance Testing Methods

What sort of conformance tests could be devised?

As noted in Section 3, methods for determining conformance might include reading source code, running documents that should be computer-processable through computers, observing an implementation in action and comparing its behavior with the behavior expected from a conforming implementation, devising test cases and using them to test control systems, and requiring documentation of development activities. Conformance testing could also include establishment of an organization to do the testing.

CASE tools for building control systems according to an architecture might be subject to conformance testing by devising test cases which any CASE tool should be able to handle and checking that any CASE tool purported to be in conformance could handle these cases. Existing CASE tools for building control systems, however, usually embody the higher tiers of architectural definition of some specific architecture, and it is hard to imagine a useful CASE tool which did not embody an architecture. Thus, only if there were a widely accepted reference architecture would it be feasible to test CASE tools this way.

4.6.2 Usefulness of Conformance Testing

How important is it that conformance criteria be included in an architecture? Who would use conformance tests?

Conformance testing can be useful to the developers of an architecture in the context of evaluating the architecture. To evaluate an architecture, implementations of the architecture would have to be built. Each implementation would be a test of the architecture, provided that the implementation conforms to the architecture.

The end user of a control system may want to be assured that a component is conformant with a particular architecture to ensure that it can be used with previously installed components.

4.6.3 Testing Conformance in Development

If an architecture includes one or more methodologies for architectural development, the development process for building an implementation should use them. Using the methodologies is part of conforming to the architecture. How can this type of conformance be tested?

Determining conformance to a methodology is difficult. Unless the activities in a methodology result in lasting documents, there may be no evidence whether the methodology was followed or not. It is usually not difficult to define documents which must be created during development to provide such evidence, but it is difficult to define such documents so that developers regard them as anything but a nuisance. It is also usually not difficult - although it may be tedious - to create the required documents even if the methodology has not been followed. This is a common tactic used by developers.

To the extent a methodology for architectural development is embodied in a CASE tool, conformance to a methodology may be obtained by ensuring the tool is used.

4.6.4 Conformance Classes

A *conformance class* is a set of architectures (or implementations) distinguished by a combination of features at a tier of architectural definition. Different conformance classes may have different and incompatible choices of features or may correspond to different degrees of conformance to an architectural requirement.

In defining an architecture, there are often situations in which incompatible choices must be made. Rather than requiring that each choice result in a different architecture, it may be useful to define conformance classes for implementations using incompatible choices of architectural features.

4.6.5 Allowing Non-Conformance

Once an implementation is built, it is common that the implementation offers easy opportunities for better performance by making small changes outside the scope of the architecture, so that the implementation is no longer fully in conformance. Typically, easy changes of this sort have a high hidden cost, in that they compromise the modularity of a control system, make its behavior less understandable, make it less portable, make it harder to reuse, etc. Because such changes may have high value, it may be useful to provide a formal method of assessing the degree of conformance. To this end, it may be advisable to establish conformance classes which correspond to different degrees of conformance to an architecture.

4.7 Standards Issues

An architecture should make use of established standards. For developing standards there is an issue of suitability of the current state of the standard. The standard may not yet have a degree of maturity which the developers of an architecture need. In this case, it is possible to use the standard as much as possible and add the necessary enhancements to make it useful. When using a developing standard, there is an additional issue of when to upgrade from one version to another. Considerable cost may be involved in doing upgrades, so it is important to evaluate the stability of the version before switching to it. As always with the use of any new technology, the availability of tools for development generally trails the development of the new technology. This fact needs to be considered in deciding whether to use an emerging standard.

5 Control Architecture Issues

In this section we raise issues which apply to control architectures, rather than all architectures.

In order to discuss the issues, we must assume that certain architectural units are present in a control architecture. These architectural units are: controllers, planners, schedulers, groupings of related controllers/planners, communications systems, data systems, resources, plans and tasks. Definitions will be given in the first subsection of this section. The definitions given here are meant to provide a basic understanding only; a major portion of defining a control architecture is in generating the detailed description of architectural units at each tier of architectural definition. In issue discussions we assume that the definition of the architectural unit is as given in that subsection. How the architectural unit's definition relates to the other uses of that term or what name an architectural unit may have in a specific architecture are not issues here.

Since a primary domain of the joint architecture will be manufacturing, a number of issues will appear which are specific to manufacturing.

We conclude the control architecture issues discussion with a section that outlines the desirable characteristics of a control architecture.

5.1 Preliminary Definitions

The purpose of a control system is to achieve goals. A *goal* is a desired state of affairs. Goals include such items as manufacturing a part, moving a robot arm to a specific place, or navigating a vehicle from one point to another. A scheme developed to accomplish a specific goal is termed a *plan*. Typically, a plan consists of a number of discrete steps. A *step* is the basic unit of subdivision of the procedures section of a plan, usually specifying that a single activity (single at some conceptual level) be carried out (drill a hole, deliver a tray, machine a lot of parts, etc.). Often the steps are sequential, but this is not necessarily so. *Planning* is the activity of making plans of any sort—process plans, production plans, schedules, etc. A piece of work which achieves a specific goal - actual work, not a representation of work - is termed a *task.* A generic representation of a type of work, such as moving in a straight line from one point to another, opening a gripper, or drilling a hole, is a *work element*. An instruction from a superior controller to a subordinate controller (or from a client controller to a server controller) to carry out a task is a *command*.

Tasks are usually the result of a command, although an architecture may permit a control system to have spontaneous activity. The most obvious examples of tasks initiated by commands are the processing tasks. Other types of activity, such as fetching data, navigating through the plan or synchronizing with other plans may also be initiated by commands.

If a controller can carry out only one work element, the command does not need to name it. Otherwise, it is expected that a command will name a work element and will provide the necessary values of parameters to the work element.

The process of determining which tasks must be carried out by the control system is termed *task generation* and performing these tasks is termed *task execution*. The specification of the mechanism for task generation and execution forms a major portion of a control architecture.

In a control system, a *planner* is an agent which generates or selects plans to accomplish one or more goals, and a *controller* is the agent which directs the performance of or performs specific tasks. *Scheduling* is the assignment of specific resources and times to the steps in a plan. A *scheduler* is an agent which performs scheduling. Often, the operations of scheduling and planning are combined in one function. In our discussions however, we will not assume that this is so.

5.2 Domain

Earlier in the report, we stated that there are certain characteristics of the domain which make a distinction in the architecture. For control systems, there are several important characteristics of the domain in which the system operates which affect the architecture of the system. These are:

(1) the degree to which the environment is known in advance. The greater the environment is known in advance, the less sensory processing and adaptive capabilities the control system must have.

(2) the degree to which the environment is structured. A highly structured environment permits the control system to make certain assumptions, but may force the system to be able to handle systems of constraints.

(3) the degree of variability. An environment with rapidly varying features requires more rapid response.

5.3 Architectural Conformance

Many control architectures, since they include physical components which are not originally designed to work with the system, must come to grips with the issue of the extent to which components which do not conform to the architecture can be included in the architecture. In particular, non-conformant controllers and communications systems are frequently encountered in applications to real world situations. It is not clear how best to deal with this.

5.4 Human Interactions with the Control System

Should the architecture specify how humans interact with the control system? Which parts of the control system should the human be permitted to interact with? Which aspects of the interactions between the components of a control system should the user be permitted to alter?

Since virtually every control system has situations in which human intervention is required, some specification of the nature of human interfaces seems appropriate. The type of intervention may well depend on the architectural unit affected and the assumptions of the control system.

Interactions of humans with individual controllers are discussed in Section 5.5.9.

5.5 Controller Issues

This section discusses architecture issues pertaining to individual controllers.

5.5.1 Controller Functionality

A primary issue is: what functionality should be included in a controller? Many architectures include planning and scheduling within the controller. Others insist that controllers are merely dispatchers of tasks or performers of tasks upon command.

5.5.2 Internal Units

Related to the question of controller functionality is the issue of what (if any) internal units a controller should have. This issue is also related to the architecture granularity issue. Should there be one internal unit for each function? Which of these internal units should be permitted to be architectural units? Which should be submodules? Which internal units should communicate independently? What should be the form and content of the communications among these internal units?

As examples of functional decompositions, we note that the RCS architecture decomposes a controller into value judgment, behavior generation, world modeling, and sensory processing and then decomposes behavior generation into job assignment, planning, and execution. See Section 7.1 for a detailed discussion of each of these functions. The MSI architecture considers a controller to be primarily a task execution and task monitoring agent, with other functions being placed in separate architectural units. See Section 7.2 for a detailed description of the functional units of MSI.

5.5.3 Operational States

The *operational state* of a control system or controller (or other active component of an implementation of an architecture) is a state variable indicating its fitness for operation. Typical values for operational state are: down, idle, ready, and active.

Should controllers have operational states? To bring a control system up, deal with errors, etc., it seems essential to have operational states. What should they be and what sequence should be followed during start-up and shutdown?

Any kind of reconfiguration of a controller hierarchy, other than rebuilding the hierarchy when the control system is totally shut down, may be difficult without operational states.

5.5.4 Execution Model Issues

An *execution model* is a logical view of how the execution of a control system is carried out. Certain execution model issues which need to be settled in selecting hardware and operating systems are discussed in this section.

5.5.4.1 Blocking vs. Non-blocking I/O

Input/output (I/O) is called “blocking” if the process doing the I/O stops while I/O operations are being carried out. If the process does not stop, its I/O is called “non-blocking.” Blocking I/O may make the process too slow or make its speed unpredictable. Non-blocking I/O may allow data to be overwritten before the I/O operation is executed and may introduce data concurrency problems.

5.5.4.2 Interrupts vs. Cyclic Processing

This issue arises when multitasking is being used, and several tasks are running on the same processor. In an interrupt model, the execution of a task may be suspended if an interrupt signal is received indicating that some other task wants to execute and the other task has higher priority. In cyclic processing, a list of tasks is maintained, and each task is executed for a certain amount of time (or until it is finished). Then the next task is executed for a time. The operating system keeps cycling through the list.

5.5.4.3 Sleeping Processes with Wake-ups

In a cyclic processing control system, if it is known that a process in an executing system will have nothing to do for a while, the process can be “put to sleep” temporarily, meaning that it does not execute on its usual cycle. A flag may be set to indicate that a process is sleeping which is unset when the process should “wake up” and resume cyclic execution.

Being able to put controllers to sleep is useful in cases where a controller has several subordinates, only one of which can operate at a time. For example, if a robot has three interchangeable grippers for its one wrist, it may be useful to have a separate controller for each gripper. Rather than repeatedly starting and stopping the gripper controllers and dynamically reconfiguring the control system when the grippers are changed, all the gripper controllers could always be part of the control hierarchy, with only one awake at a time.

5.5.5 Operational Modes

An <u>*operational mode*</u> is a style of operation of a controller or control system. Operational modes might include, for example: debugging (enabled vs. disabled), autonomy (automatic, shared control, or manual), logging (enabled vs. disabled), single stepping (on vs. off).

Should controllers have operational modes? If so, what should they be, and what values of each mode should be allowed? Having modes seems desirable.

5.5.6 Standard Internal Workings

Should the internal workings of the controllers in a control system follow some standard or paradigm? For example, some versions of the RCS architecture require that each controller be a finite state machine.

The advantages of having standard internal workings are:

(1) controller templates at intermediate and low tiers of architectural definition can be constructed. It may even be possible to design a controller shell so that all controllers in a control system not at the lowest hierarchical level are the same, and they differ only in the data that drives them.

(2) CASE tools can be constructed for building controllers.

(3) humans can understand how each controller works more easily than if each one is unique.

(4) determining the execution time of each controller is easier than if non-standard internals are used.

(5) standard methods for testing controllers can be developed and used.

5.5.7 Command Queues

Should control entities have the capability to put commands received in queues?

If queues are used, there are many ways in which they can be defined and managed. First In First Out (FIFO) and Last In First Out (LIFO) are simple methods. Priority queues are a more complex management technique. Queue management tasks can be added to the capabilities of controllers. Managing queue size or overflow becomes necessary if queues are used. The use of queues may make other control system features, such as error recovery, significantly more difficult to implement.

5.5.8 Multiple Simultaneous Tasks

Should a controller have the capability to perform more than one task at a time? If so, how should the controller determine what resources are required for each task and how any shared resources should be allocated?

5.5.9 Human Interactions with Controllers

Should the architecture specify how humans interact with the controllers in the system?

Some specification of the nature of human interfaces seems very desirable, but the appropriate degree of detail of the specification is not obvious. There are several sub-issues.

5.5.9.1 User Control of Tasks

Should the user be permitted to direct a controller to perform a specific task? If so, how should a user introduce a task to the controller?

One way in which a user could directly hand a controller a task is to define standard work elements for every controller above the (vendor-specific) hardware controller. A “user_control” work element could be defined which would be executed when a user wanted to control the subordinates of the controller. A standard interface could be designed for this work element, which might list all the subordinates and their work elements. The user would select a subordinate and a work element and the interface system would help the user construct a command from the work element. Then the command would be sent to the subordinate.

5.5.9.2 Default Interface Operations

Should there be a default set of operations possible from the human interface?

Some set of operations may apply to every controller in a control system. For example, if every controller has operational modes (e.g., debug enabled/disabled, logging enabled/disabled), it would be very useful to be able to change the operational mode from the human interface. This set of common operations provides the candidates for defining a set of operations possible from every human interface.

5.5.9.3 Default Human Interface

Should a default human interface be defined?

It may be desirable to define a default human interface so that a human could interact with every controller in a control system without having to know the specific tasks the controller is able to perform and without having to learn every interface anew. The default human interface would have a standard method of performing all default operations and a method of commanding the controller to perform each of the tasks unique to the controller.

It is not clear at what tier of architectural definition the definition of the default human interface should be part of the reference architecture. An architectural specification at an upper tier of architectural definition might simply require that a default interface be defined at a lower tier.

5.5.9.4 Situation-Specific Human Interfaces

Should situation-specific human interfaces be allowed?

Many applications involve information that is best transmitted graphically to humans. Displays tailored to such information are required. It is not clear how best to provide for interactions between a control system and its situation-specific human interfaces.

5.6 Collections of Controllers

Within a control system, it is normally necessary for controllers to coordinate their activities closely to achieve system goals. For example, a robot which places a part for a milling machine must coordinate its actions with those of the milling machine and the gripper. The necessity for coordination suggests that certain groups of controllers should work together.

5.6.1 Modes of Interaction

How should controllers that need to work together do so? What should be the criteria for grouping controllers together? Should the interaction of controllers be direct (via command-and-status), indirect (via shared data), or a combination of both?

Two common models for organizing the interaction between controllers are the hierarchical model and the client-server model. In the hierarchical model, each controller supervises a number of subordinate controllers to whom it gives commands and from whom it receives status information. In the client-server model, a controller (denoted a server) offers a type of work which it can do for other controllers (for example, transporting something). The other controllers in the control system (denoted clients) can request the server controller to provide that service for them given certain parameters. Many variations on the communications between the client and the server are possible in this model, although the basic idea is the same. Many CIM projects have found that a hierarchical method works well for all activities except material handling, which performs better in a client-server model.

Is it desirable to mix hierarchical and client-server models? If so, how can this be done? There are many ways a mixed mode might be implemented.

(1) There could be a reconfiguration queue. There would always be a strict hierarchy, but the service controller would move from superior to superior.

(2) The service controller could have multiple simultaneous superiors.

(3) The service controller could be at a fixed place in the hierarchy, with service requests being posted to the database. The superior of the service controller could just keep giving "provide_service" commands to the service controller, with parameters extracted from the database.

The scope of this report does not permit an exhaustive discussion of this issue, but the preceding remarks give the reader an indication of the many other issues which are spawned from this one.

5.6.2 Control of Devices and Controllers

In any control architecture, at the lowest level of control, each controllable physical device will be controlled by a controller. However, whether a controller can control other controllers depends on the architecture. In a partly or fully non-hierarchical control architecture, those controllers which run in the client-server mode in the role of server will have no superior controller, and those that run in the role of client will not have subordinates performing the functions for which they are client. If a control system is fully client-server, no controller will have any subordinate controllers, but some controllers will control physical devices. In a hierarchical control system, controllers may control either devices, other controllers, or a combination of both.

5.6.3 Synchrony and Speed

Two controllers are said to be in *synchrony* if there is a fixed relation in time between their execution cycles. There are many ways of being in synchrony. Both controllers might report to a common superior which keeps them synchronized. One controller might execute at random times, causing the other to execute immediately afterward. Both controllers might execute cyclically with a fixed period; for example one controller might execute every ten seconds while the other executes every three seconds. If both have nominally fixed periods, phase drift may be uncontrolled (a likely occurrence if each has its own clock) or controlled (by using the same clock, for example). If phase drift is controlled, the phase angle may be set (usually to zero); for example the two controllers with 10 and 3 second cycles might be forced to start at exactly the same time every 30 seconds.

In many situations, the issue is speed, not synchrony. For example, to ensure stability, a subordinate may need to execute at least some number of times as fast as the superior (say ten, for example). If the subordinate executes fast with randomly varying execution times, (so that it executes between 15 and 40 times each time the superior executes, to continue the example), then it is not synchronous with the superior but may meet the performance requirement through speed.

Speed and synchrony requirements may be stated independently or dependently in many different ways.

What sort of synchrony, if any, should be required of a grouping of controllers? Should the same type of synchrony be required for every grouping, or should different options be allowed.

Is there a need for a system-wide clock? What is accomplished by maintaining various levels of accuracy? How can a system-wide clock be used to maintain synchrony?

5.7 Task Specification, Generation and Execution

Tasks are one of the most important aspects of a control system. Frequently, it is the nature and decomposition of tasks which determine the structure of a control hierarchy. To accomplish a task using a computerized control system:

(1) The work elements required to describe the components of the task must be defined, and the semantics of each work element must be known to the controllers which receive commands and to the planners which plan for those controllers which execute that work element.

(2) A plan must be made for which instances of work elements are to be carried out and in what order. The plan may also include information on which resources are used, the degree to which each resource is used by the task, the duration for which the resource must be used, pointers to information needed to carry out the plan (such as current resource availability), and synchronization with the current plan or related plans.

(3) Commands to carry out subtasks must be given.

Issues related to tasks are discussed in the following sections.

5.7.1 Specification of Work Elements

What are the required characteristics of a work element? In what format should work elements be specified? It is a challenge to state the semantics of a work element unambiguously. A formal specification language that provides for representing task semantics would be useful.

5.7.2 Task Decomposition

How should abstract tasks be decomposed into less abstract tasks which devices are capable of executing? Once a task has been decomposed into less abstract tasks, how should these tasks be assigned to controllers?

The decisions on these issues are implemented in the process of defining work elements and in deciding which controllers should be able to give commands referring to these work elements and which controllers should be able to accept those commands.

5.7.3 Task Execution Model

An important part of an architecture is the model of the process of task generation and execution to which the architecture subscribes. Although it is beyond the scope of this report to discuss all models, we shall briefly describe two. In the first model, there are a number of stages in task generation and execution. The generation of a task begins with a generic plan. Next this plan is specialized by assigning specific resources and specific times for task execution. Finally, during the actual execution of the plan, current information is considered in making any choices explicitly coded into the plan. In the second model, a plan may be developed while it is being executed, with only the next step being known at any time.

Important sub-issues of task execution include the coordination of executing tasks, and the method in which controllers are to receive and monitor tasks. The degree to which the environment can be expected to remain known and stable may dictate which model is selected.

5.7.3.1 Command and Status Exchanges

As previously discussed, a superior or client controller tells a subordinate or server what is to be done by sending a command. A command is a type of message. So that the superior or client may know how the work is progressing, it is usual for the subordinate or server to send messages back. The returned messages may specify, for example, that the commanded task is done, is in progress, or is not being performed because some error condition exists. We will refer to this interchange of data as the *command and status exchange* and to the specification of the messages as the *command-and-status protocol*.

Should a control architecture specify the command and status exchange between controllers? If so, how detailed should this specification be? Should it specify the semantics of the exchange, the format of the exchange, the encoding of the exchange?

The nature and extent of command and status exchanges depend on the control structure, that is, how groups of controllers function together (see Section 5.6), but almost all existing architectures have exchanges of some sort. Most hierarchical architectures will go into an error state if the exchanges break down. Some heterarchical architectures [Duffie3] include exchanges but anticipate they might break down and provide for automatic recovery.

At the interface between a controller and a piece of physical equipment, it is normal to have a command and status exchange regardless of the control architecture. The controller must send the equipment commands in a format the equipment is designed to accept, and must accept whatever status the equipment is designed to return. Often the returned status will give the controller an indication of the physical condition of the equipment and possibly the condition of the task which it has been given.

5.7.3.2 Coordination of Tasks

Implementations should be able to handle task coordination. How should this be accomplished? Should the information for coordination be in the work element, in the plan for a task (if one exists), or in some other part of the control system? Most commonly, the controllers responsible for the performance of the task coordinate by some form of messaging. Alternatively, for example, in some CIM architectures, information for coordination of tasks is stored with the part which is being manufactured.

5.8 Data

The fundamental questions which an architecture must address with respect to data are:

(1) what data should be required to be used and generated by its components,

(2) whether the specification of such data should be conceptual, logical, physical or some combination of the three,

(3) how data may be physically distributed in a system,

(4) how distributed data should be accessed and by whom it should be accessed.

In subsequent sections, we will discuss each of these issues.

5.8.1 Required Data

All control architectures have some types of data that are required by the control system. An architecture can specify this data on one or all of three levels of data abstraction. The most abstract level is the conceptual level. Data specified at this level describes the idea which the data represents. For example, an architecture may specify that a machine tool should have a physical location, without specifying the coordinates in which this location is given or the physical location of the data storage. At the next level is the logical level of definition. To continue the previous example, a machine tool could be said to have a property called “location” corresponding to the conceptual notion of location. At the physical level, the implementation of the representation of the machine tool must be fully specified with a coordinate system for the location data, the fields of the data structure or database entry describing the machine tool location, and the physical location of the database or database server. The degree of data abstraction at which an architecture specifies data may vary at each tier of architectural definition and is related to the granularity of that tier.

Required data can be either *persistent data* (data stored on a permanent medium such as files or databases) or *non-persistent data* (data stored in memory). A decision must be made as to whether to specify the persistence of some or all data. The persistence issue is independent of the data access issue, which will be discussed later.

There are several important types of information which are specified in a control architecture. We will discuss two of them here, plans and resources, and discuss other categories later as they arise in the discussion of specific architectures.

5.8.1.1 Plans

In existing control architectures, the characteristics of plans vary extremely widely, and are related to the concept of planning and control which the architecture has and to the structure of the controller which the architecture requires. One architecture may view planning as a single-stage endeavor while another architecture may have separate stages of planning. With a single-stage view of planning (as taken by RCS, for example), the term "plan" or "process plan" is sufficient. A *process plan* is a specification of the activities (possibly including alternatives) necessary to reach some goal. With a multi-stage view, it is useful to have a different term for the plan at each stage. In a three-stage view of planning (the one taken by MSI and described in Section 7.2.2.1.1, for example) the three plan types might be "process plan", "production managed plan", and "production plan". In the three-stage view, a process plan is more narrowly defined; for example, it may be required to identify resources in generic terms.

Typically, commands issued by controllers are generated by combining current information about the system (e.g. resource status) with the information generated by reading plans.

What types of plans should be included in the architecture? What types of information should a plan contain? What plan format(s) should be used?

5.8.1.2 Resource Definition

A *resource definition* is a description of a resource, usually given in a formal information modeling language. Should the architecture include formal resource definitions? In the definitions of resources in the architecture, are the ways in which the resources are used specified? Should dynamic characteristics of the resource be included with its description? Are there types of resources which share characteristics? If so, how should resources be categorized.

Some architectures, MSI for example, include a formal definition of resources. There are several general categories of resources: for example, consumable resources (machine fluid, solder, etc.), logical resources (e.g. controllers) and permanent resources (lathe, drill press, etc.). The resource definitions within these categories describe classes of resources which have common characteristics. This can be referred to as a catalog of resources. In addition, MSI specifies that the factory under consideration describe the actual resources which are on hand and their status. Controllers themselves are considered resources.

Depending on the nature of planning in a control system, resource classes or actual resources may be specified in plans.

It is recommended that a reference architecture include some amount of formal resource definition. What information should be included in a resource definition and what language is used for resource definition are important issues not addressed in depth in this report.

5.8.2 Data Handling Architecture

The data handling architecture specifies how data is accessed and which architectural units may access which data. In this discussion, the persistence of data is not assumed—data access mechanisms are similar in concept for persistent and non-persistent data.

5.8.2.1 Data Access Mechanisms

In any system, data may be stored in a single place or be physically distributed in several places throughout the system. The access of data which is stored in one place is straightforward; the data is either local (on the same physical machine) or it is accessed through the communications system for the physical machines.

If data is distributed, there are two possibilities: an implementation can be required to specify the physical location when accessing data, or there is a server system for the data which an implementation contacts to access the data without knowing its physical location. A well-known example of a server system which can hide the location of data files from an implementation is NFS [Libes2]. Systems of this type also exist for databases (e.g. IMDAS - [Barkmeyer1]), but these are less well-developed and, at present, too slow for real-time use. An example of a server system for non-persistent data is NIST’s Common Memory [Libes1], [Rybczynski1]. Depending on the server, the implementation may need to have a method of determining that the data which it currently has is up-to-date. This issue is a distributed system problem and a full discussion of it is beyond the scope of this report.

How much or how little the architecture leaves the organization and access mechanism of data to the implementation is an important architecture issue.

5.8.2.2 Data Access Permissions

There seems to be universal agreement that any component can have a certain amount of private data. Private data is usually information which is irrelevant to other components. It is an architectural choice whether to specify the private data for components and whether to specify its mode of access.

Some data will have to be shared among different components. This data will typically be distributed and will require one of the access schemes discussed in the previous section. If the architecture permits a component to keep local copies of shared data, the architecture should also specify whose responsibility it is to ensure consistency of this data with the shared copy. Whether the architecture permits more than one mode of access for shared data is an additional issue. In some architectures, access is required to be through the data access service, in others, multiple access mechanisms are permitted.

Potentially each item of data would have a scope. A sample set of scopes would be: local to function, local to process, local to group of processes, global in network. The method of implementing a scope would be determined when a system is configured. For example, a variable local to a group of processes sharing a memory board might be implemented by establishing a memory board address for the variable and having each process use the address. If a variable were used by processes running on physically separate hardware, the variable might be implemented using the communications system.

5.9 Planning, Scheduling and Resource Allocation

A key issue for control systems, which is very sensitive to the environment in which they operate, is the way in which the control system plans, schedules and allocates resources. In some domains, such as manufacturing, the construction of long-range schedules and the allocation of resources is required. In other domains, such as the navigation of autonomous vehicles, the environment is not known in advance and scheduling and resource allocation must be done in real time, as information about the environment is processed. The following sections point out important architecture issues concerning planning, scheduling and resource allocation. Some issues are more appropriate to some domains than others.

5.9.1 Process Planning

To what extent should the architecture require plans to be generated in advance, as opposed to deciding what to do in real time? If plans are generated in advance of their use, should the resource allocation for the plans and the scheduling of the plans be done at the same time as the plan generation, or can these activities be performed later using a skeletal plan which refers to resource classes? Can these modes be mixed effectively? If so, what requirements does this place on the control structure?

5.9.2 Scheduling

How should scheduling be handled? There is a wide spectrum of possibilities for scheduling. These possibilities range from operating with no schedule, in which the order of events and what gets done is a by-product of control system operation, to full scheduling, where detailed activities are planned for each controller at a scheduled time, so that all controllers not at the lowest hierarchical level are merely dispatching commands generated from a plan.

What aspects of the architecture must be adjusted for control systems which use scheduling? At least, such a control system must include a scheduler, and control information must include schedule information. Such a system would need information on the availability of resources as well, if the architecture has availability classes for resources and the resources are used in plans.

The type of scheduling required for an application may vary widely. For example:

A transfer line (or flow shop) is necessarily sequential, so little scheduling is required for the controllers of the line.

A shop in which there is no resource contention (for example, if the shop is underutilized, or if all the machines are identical, all jobs have equal priority, and each job can be done on one machine) may not need scheduling. Everything is processed as fast as possible as soon as it arrives.

[Biemans1] makes the point that a group of steps that must be executed sequentially may be treated as a unit for scheduling purposes.

In a manufacturing system, in addition to the scheduling of control tasks, other items must be scheduled. For example, maintenance tasks and material handling tasks must also be accounted for. A manufacturing architecture must specify whether these non-control tasks are scheduled or handled by other means.

5.9.3 Resource Allocation

Some control architectures need to provide for ***resource allocation***: assigning resources (temporarily or permanently) for some specific purpose. Resource allocation is critical in control systems in which resources must be shared, such as manufacturing systems[3].

In the case of control systems which are reconfigured periodically (dynamically or when shut down), controllers and the equipment they control are also treated as resources. With respect to processors and processes, there are four additional cases of resource allocation that may be considered (given here in order of increasing dynamics):

(1) Allocating processes to processors in a control system where processes do not move from processor to processor. In the rest of this report we treat this as a methodology for architectural development issue, not a resource allocation issue, since the problem is what design rules to adopt for making the allocation. This issue was introduced in Section 4.5.2, “Mapping Architectural Components Onto Hardware”.

(2) Allocating processes to processors in a control system which is reconfigured periodically (dynamically or when shut down), and reassigns processes to processors during reconfiguration.

(3) Allocating processing time to various processes when a processor has to execute several processes which do not move from processor to processor. This is an intensively researched operating system issue in computer science and many commercial computer systems deal with it.

(4) Allocating a processor for a certain amount of time to a process in a control

3. A number of other issues specific to the manufacturing domain must be dealt with by the architecture. Among these are: product specification, order processing and tracking, part and lot tracking, lot sizing (the designation of the number of parts which will be manufactured at one time), and material handling.

system where there are several processes and several processors, and the processes can move from processor to processor. This is a current research issue in parallel processing.

5.10 Communications

To what extent should the architecture deal with communications? Should the architecture specify the communications paradigm, the communications implementation? Should one communications scheme be required for all portions of the architecture, or can multiple schemes be used?

Three aspects of the communications paradigm which affect the architecture are: the number of communications entities which may receive a message transmitted by a communications entity at one time, whether a message is guaranteed to arrive at its destination by the network software, and the timeliness of message delivery by the communications system.

Regarding the first aspect, there are three possibilities. A communications system can permit a communications entity to send a given message: to only one other communications entity (termed *point to point* communications), to more than one known communications entity (termed *multicast* communications), or to more than one unknown communications entity (termed *broadcast* communications). From an architecture viewpoint, the critical distinction is that point to point and multicast communications require a communications entity to know which other entity or entities it is sending a message to, whereas broadcast communications permits the sending of messages to other communications entities whose identities are unknown to the sender.

Message delivery may be either guaranteed or non-guaranteed. The architectural impact of this is that, if non-guaranteed message delivery is used, the implementation is responsible for ensuring message transmission or for compensating for potential message loss.

Open System Interconnection (OSI) communications systems use point to point communications with guaranteed message delivery. This paradigm frees the user from retransmissions to ensure that a message is received, but forces the sender of a message to identify the receiving party for the message. An alternative is NIST’s Common Memory [Libes1], [Rybczynski1]. This paradigm has both buffered and non-buffered variants. The buffered version prevents immediate over-writing of messages but does not guarantee delivery. The NIST Common Memory paradigm has the advantage that any number of communications entities can have access to the same shared information, without the message sender knowing in advance who the recipients are.

The timeliness of message delivery by the communications system is an important consideration for real-time control systems. Communications systems can be designed with this in mind, and there are standard networking schemes which support data transmission for real-time control systems.

5.11 Checks and Safeguards

To what extent should checks and safeguards be built into an architecture?

Checks and safeguards are useful to provide reliability, fault tolerance, and error recovery.

Clearly, control systems which are dangerous when working improperly (airplane autopilots and nuclear reactor operation systems, for example) need safeguards. Are the safeguards outside the basic architecture or an integral part of it?

MSI requires a guardian interface for each controller. The guardian interface functions as the human emergency override and intervention point. The existence of this interface is mandated by the architecture.

Some architectures, the Dornier architecture, for example, distinguish between "normal" error (such as position error for a machine tool axis, which is fed to the servo law of the controller for the axis) and abnormal errors (such as when a machine tool axis trips its overtravel switch).

5.12 Error Recovery

In addition to error checking which is required for safety reasons, a control system needs to have mechanism(s) for identifying and correcting error conditions. To what extent is the ability to recover from errors in each major subsystem (e.g. communications, groups of controllers, data system) built into the architecture? What mechanisms does the architecture permit or require for handling cross-subsystem errors? What feedback mechanisms for fine tuning control system operation does the architecture permit?

5.13 Desirable Characteristics of a Control Architecture

What are the desirable characteristics of a reference architecture, and how can a reference architecture be defined to have these characteristics? For each characteristic, what is it about the architecture that provides for the characteristic?

There is general agreement in the literature on what the desirable characteristics are. The subsections below present these desirable characteristics. Some of the characteristics (e.g., understandability) can be measured on an entire architecture, while others (e.g., speed of performance) can only be measured on implementations of the architecture. Even where only the implementation can be measured, the level of performance may be indicative of the quality of the architecture, not just its implementation.

Achieving one desirable characteristic may help or hinder in achieving another. Some of the characteristics are synergistic, others antagonistic. Low cost, for example, goes well with low complexity and easy modifiability. High speed, on the other hand, is likely to conflict with low cost, understandability, low complexity, and several others.

Determining what it is about the architecture that provides for any given characteristic may be easy (modularity leads to ease of modifiability, for example) or hard.

5.13.1 High Quality

The quality of the architecture should be high.

This is an inescapably vague idea which should, nevertheless, be stated. Many other subsections in this section describe components of quality. Other components of quality might also be identified, and a method of integrating them to produce a single measure of quality might be devised.

5.13.2 Low Cost

The cost of implementing an architecture should be low for a given level of performance.

The method of keeping costs down varies according to the environment. In a research environment, the main cost is usually the salaries of the researchers, and it is commonly less expensive to buy more computing hardware to improve control system performance than it is to devote a lot of effort to optimizing performance within existing hardware. In commercial control systems it is usually more cost effective to optimize hardware performance and make efficient use of computing resources.

5.13.3 Modularity

Aspects of a control system (data flow, control flow, communications, etc.) should be kept as separate as possible, and each aspect should be divided into encapsulated parts, to the extent possible.

Increasing modularity generally helps achieve many other desirable features.

5.13.4 Low or Manageable Complexity

The less complex an architecture, the better. Many domains, however, require complex functionality from the architecture. Usually, the most that can be asked is that the architecture provide a good method of managing this complexity.

5.13.5 Fault Tolerance

A control system is fault tolerant if it will continue to work when one or more of its components is not working.

Fault tolerance may be strong or weak. A strongly fault tolerant system will continue to work at the same level of performance when there are faults. A weakly fault-tolerant system will have performance degradation proportional to the number of faults in the system. Weak fault tolerance is often called “graceful degradation”.

5.13.6 Error Detection and Recovery

It should be possible to be informed of problems quickly, and if something goes wrong, it should be possible to get the control system working again quickly.

5.13.7 Extensibility

It should be easy to add new components or capabilities to the control system without having to make major changes to the existing components and capabilities.

5.13.8 Speed and Response Time

Implementations should perform fast enough to meet the requirements of the domain.

Control systems which must keep pace with events in the environment are called *real-time* systems. Systems in which response must be generated within a fixed time interval are called *hard real-time* systems. Real-time systems which are not hard real-time systems are refered to as *soft real-time* systems. Control systems in which the rate at which events occur in the environment does not matter are not called real-time but may still have speed or response time requirements.

5.13.9 Modifiability

It should be easy to change existing components of the control system. The foremost method of achieving this is modularity. Having adequate documentation, using standard languages, and keeping complexity down also contribute to ease of modifiability.

5.13.10 Portability

It should be feasible (preferably easy) to transport an implementation of an architecture from one computing platform to another. An obvious method of helping provide portability is to use a standard high-level computer language for source code in the implementation. Compilers for the language will be available for many computing platforms. Portability may be limited by the implementation’s requirements on the operating system.

5.13.11 Predictability

It should be possible to predict what the control system will do. It is necessary both to be able to predict what will happen given a fully specified environment and set of inputs and to be able to predict limits on control system behavior given any possible inputs in any environment in which the control system is intended to work.

The need for this requirement is dependent upon the domain. For control systems in which failure is dangerous, predictability is very important. In domains where multiple “correct” choices are possible given the same set of inputs, this requirement may not be as stringent.

5.13.12 Reconfigurability

A control architecture should include methods for configuring control hierarchies or networks when the control system is fully idle.

Dynamic reconfiguration (modifying the control hierarchy while the control system is working) is very desirable for error recovery and fault tolerance; it should be possible to replace a defective controller without shutting everything down.

Dynamic reconfiguration is very desirable in some situations, even if nothing goes wrong. For example, if a shop gets a wide mix of parts, it would be desirable to reconfigure cells so that a cell can handle all operations for a part it is trying to manufacture. In implementations for use in space missions, it would be advantageous to be able to reassign robots to different working groups smoothly in mid-mission.

5.13.13 Reliability

Implementations should be reliable. A control system is reliable if it works as intended with an acceptable failure rate - what is acceptable is up to the system designers and users.

The reliability of software correlates positively with its predictability. Both can be achieved, at least on the software module level, by testing each module with every combination of allowable inputs and verifying that the output is correct.

5.13.14 Reusability of Software

It should be feasible to reuse software from one implementation of an architecture to another, since the new implementation should have the same controllers and tasks as the old one.

5.13.15 Understandability

It should be possible for a human to understand the architecture and implementations of the architecture. For very large, complex control systems this is critical. Most large systems are developed by teams of people who require a common understanding of the system to do their work correctly.

5.13.16 Compatibility with Existing and Emerging Standards

An architecture should conform to existing, well-established standards, where appropriate. With respect to evolving standards, an architecture should, depending upon the maturity of the standard, require their use insofar as it is feasible.

6 Other Control Architectures

In preparing this report, it was necessary to understand previous work which has been performed by both divisions (RSD and FASD) and in the general community concerned with control architectures. Although the authors of this report did not attempt an exhaustive literature search, many papers were analyzed. In this section we review a number of control architectures that have been described in the literature.

A brief description of each architecture is given, highlighting those aspects of the architecture to which the architecture pays the most attention. Two of the architectures described in this section, CIM-OSA and Dornier, are reviewed in more depth in Appendix E. The RCS and MSI architectures are not covered in this section, but are discussed in detail in Section 7 and Appendix C.

Some papers contain special features which may interest the reader. These are:

[Dilts1] provides a classification and comparison of types of architectures.

[Auslander1] presents principles of real-time control software.

[Dornier1], [Lumia2], [Min1], [VanHaren1] have performed comparisons of specific proposed architectures.

Several papers include glossaries of terms related to architectures: [Albus7], [Dornier2], [Leake1], [Martin1][4], [Quintero2], [Senehi1], [Senehi2], and [Wallace1].

6.1 Introduction

All the architectures which we have reviewed address some of the elements of architectural definition. Not one of the architectures which we reviewed includes all of the elements of architectural definition in its definition. Moreover, each architecture places emphasis on a unique and limited subset of the issues identified in Section 4 and Section 5 of this report. For example, the Dornier architecture, but not most others, includes requirements definition; the RCS, Dornier, CIM-OSA, and other architectures include methodologies for architectural development at various levels of formality; the MSI architecture includes general criteria for determining conformance. This lack of uniformity makes architectures difficult to compare, as the data for performing the comparison is often lacking.

A traditional approach to solving this difficulty is to devise a classification scheme which permits comparable architectures to be grouped together. This section describes two classification schemes found in the literature, and proposes a simple classification scheme which will be used to organize the architectures reviewed in this report.

4. [Martin2] through [Martin6] also have glossaries, but they are subsumed by the glossary of [Martin1].

6.1.1 Classifying Architectures

A classification scheme requires that dimensions of a control architecture be identified and that any dependencies among these dimensions be made clear. There is enormous variety in the choices of dimension which can be made. The authors present two classification schemes which proved useful in the analysis of architectures performed in this report.

In [Bohms1] nine dimensions for modeling Computer Integrated Manufacturing systems are identified.

(1) modeling level (reality, models of reality, models of models)

(2) language level (level of modeling language used)

(3) aspect (set of views, e.g. functions, information, resources)

(4) composition (global to detailed)

(5) scope (type of activity)

(6) representation (modeling language used)

(7) product life cycle (design, production, maintenance, etc.)

(8) actuality (to be vs. as is)

(9) specification level (generic to fixed - how much choice left)

Section 4 of [Bohms1] proposes decompositions of each of the nine dimensions into points or regions. For example, the modeling level dimension has three points: CIM Framework, CIM Models, CIM in Practise.

In section 2 of [Biemans1] a second, very different, set of dimensions is identified. They are not explicitly called dimensions in the paper.

(1) flexibility

(2) precision of architecture definitions

(3) generality of a CIM architecture

(4) level of architectural definition of a CIM architecture5

The two sets of dimensions just described are interesting and reveal the difficulty of establishing a comprehensive method of characterizing architectures. Each of the dimensions may be of interest in some situations. Most of the dimensions are useful in describing each tier of architectural definition, not entire architectures. For example, a single architecture may encompass several tiers of architectural definition and use several different languages in its specification.

Some programming languages, such as C++, are object-oriented, in that programming is done by defining classes and instances of objects, which have attributes and functional behaviors. “Object-oriented” is not descriptive of a type of architecture. Any

5. In this report, tier of architectural definition is used instead.

of the types of architectures described here can be implemented in an object-oriented language. Heterarchical architectures, however, are particularly suitable for implementation in an object-oriented language.

Since no method of classification is totally satisfactory, this paper will not attempt a detailed classification. Using the control method as the first preliminary dimension, Section 6.2 discusses architectures in which control is the aspect of the architecture most heavily emphasized. Section 6.3 discusses architectures in which control is a minor aspect. Table gives a summary of architectures reviewed for this report which are not discussed in detail.

Table 2: Miscellaneous Architectures

Citation	Description
[Biemans2]	Presents a hierarchical controller-based reference model for manufacturing planning and control.
[Boykin1]	Presents a very brief overview of the CAM-I CIM architecture.
[ISO1]	Presents a reference model of shop floor production.
[Judd1]	Describes a method of manufacturing system (workstations, cells, or individual lines) design using “executable” functional specifications. This might be viewed as an analysis-based approach.
[Litt1],[Jung1]	Presents the RAMP (rapid acquisition of manufactured parts) CIM architecture [Litt1], and discusses the implementation of the RAMP architecture at an established site [Jung1].
[Norcross1]	Discusses a controller which handles multiple simultaneous tasks and coordinates them via resource allocation.
[Skevington1]	Describes an architecture that includes a “metadatabase” and a “metaoperating system.” It is said to be suitable for manufacturing in small shops.
[Spector1]	Presents a “supervenience” architecture for controlling autonomous robots. The architecture provides for on-line process planning in a dynamic environment.
[Wendorf1]	Reports on a family of controllers developed in a commercial environment, one of which is a workstation controller intended to be used in a hierarchical control system with a cell controller as its superior and automation modules as subordinates. The workstation controller uses process plans (called “recipes” in the paper) and handles scheduling and resource allocation and contention. It can do multiple simultaneous tasks.
[Weston1]	Describes a system named AUTOMAIL, which is called “a flexible integration shell” for CIM and provides services for communications, information access, and task execution.

6.2 Architectures Emphasizing Control Aspects

Although there is no universally agreed on categorization, four commonly discussed types of control architecture are: centralized, hierarchical, modified hierarchical, and heterarchical [Dilts1]. We have given disproportionate representation to hierarchical architectures in this report (in comparison to the universe of papers on control systems and architectures) because MSI and RCS are hierarchical, and we intend to propose a hierarchical architecture.

In this section we cite advantages and disadvantages of the four types of control architecture. These are generalizations, not hard truths. The advantages and disadvantages cited are in comparison to the other types.

6.2.1 Centralized Control Architectures

The idea of a *centralized control* architecture is that a single controller running on a single computer controls everything directly. Centralized controllers are discussed in [Dilts1]. Usually, a centralized control architecture includes a centralized data repository with a single data access protocol.

Advantages of a centralized control architecture include:

(1) no need for communications among controllers (although communications to device drivers is still needed),

(2) ease of data handling,

(3) less difficulty with global optimization,

(4) the several advantages of having only one program to worry about (only one place to look for bugs, one status interface, one control interface etc.).

Disadvantages of a centralized control architecture include the following. The disadvantages tend to be catastrophic:

(1) vulnerability to failure,
If one little thing goes wrong, the entire system is likely to stop,

(2) graceless performance degradation,
Typically, if something goes wrong the system does not just work a little more slowly; it does not work at all.

(3) hard to extend,
If the system grows, when the computational demands become too large for the host computer, there may be no way to extend it.

(4) hard to program.
The program that is the heart of the controller becomes complex, prone to bugs, and hard to maintain.

[Johnson2] describes a centralized control system named CIMPLICITY, a product offering from GE Fanuc.

[Maimon1] presents a centralized controller for a flexible manufacturing system which is decomposed internally into a scheduler, a process sequencer, a dynamic resource allocator, and run-time services. It is served by several databases and issues commands to a number of machine controllers. The internal decomposition is described as a hierarchical decomposition by the author.

6.2.2 Hierarchical Control Architectures

The hallmark of a *hierarchical control architecture* is that controllers are arranged in a hierarchy in which controllers interact through a command-and-status protocol. While a strictly hierarchical control system explicitly disallows the exchange of commands among peer controllers, it may permit peer controllers to share information by any number of mechanisms (e.g. through information shared in a database.) Often, the command structure is a simple type of hierarchy, a tree, in which each controller has one superior and zero to many subordinates. Hierarchical systems can be run on a single computer with a single processor, on a single computer with several processors, or on several different computers.

Most of the control architectures described in the papers reviewed for this report are hierarchical control architectures.

Advantages of a hierarchical control architecture include:

(1) natural modularity,
Each controller can be treated as a software module, facilitating incremental development, making the system software easier to understand and maintain, and allowing the use of templates for controller code.

(2) fairly easy extensibility,
The system may be extended by adding controllers and computers and changing the hierarchy.

(3) somewhat graceful degradation,
If something goes wrong during system operation, in a well-designed hierarchical system, only one branch of the hierarchy needs to stop.

(4) allowance for different frequencies of operation of controllers on different levels of the hierarchy.
Typically, controllers at lower levels of the control hierarchy have higher frequencies than those at higher ones.

Disadvantages include:

(1) need for communications among controllers,
Centralized controllers do not need such communications.

(2) difficulty integrating system-wide service functions, such as material handling.
Heterarchical control (to be discussed shortly) does not have this problem.

(3) difficulty in debugging.
Errors may occur in the interactions between controllers. When control programs are distributed among processes or computers, standard debuggers, which are effective with centralized control, are not likely to help much in finding such errors.

Several hierarchical control architectures will be discussed in the following sections.

6.2.2.1 AMRF (Automated Manufacturing Research Facility)

The AMRF architecture was developed at NIST. As discussed earlier in this report, it was a predecessor of both RCS and MSI. We have included only a small sample [Jackson1], [Jones2], and [Jones5] of the several dozen papers written about the AMRF in the bibliography of this report. The lineage of RCS at NIST predates even the AMRF, a five-level hierarchical control system with sensory feedback for a robot having been described in [Barbera1] in 1977.

The AMRF architecture includes five hierarchical levels: facility, shop, cell, workstation, and equipment. Material handling is positioned as a workstation under cell. Tasks are decomposed along control hierarchy lines. Controllers are resident on several different computers and communicated via standard interfaces. Process planning is done off-line for all controllers. The IMDAS (Integrated Manufacturing Data Administration System) data system [Barkmeyer1] and a network communications system with NIST's Common Memory [Libes1] are used.

Scheduling and resource allocation are not handled.

6.2.2.2 Dornier

Under contracts from the European Space Agency, the Dornier firm produced several papers [Dornier1], [Dornier2], and others concerning control architectures. [Dornier2] proposes a reference architecture for European space automation and robotics control systems. The architecture is intended to be suitable for at least robot systems, surface roving vehicles, and dedicated automation equipment. We are not aware of any implementations of the Dornier architecture.

The Dornier architecture describes tiers of architectural definition explicitly (in other terms) and has four of them.

A three layer control hierarchy is proposed. No rationale is offered for why three layers are suitable.

Each controller has three major modules: nominal feedback functions, forward control functions, and non-nominal feedback functions. No rationale is offered for the decomposition of a controller into three modules.

Of all the architectures examined for this report, the Dornier architecture provides the most formalized methodology for architectural development. The methodology uses the "structured analysis and design technique" (SADT).

More details on the Dornier architecture are provided in Appendix E.

6.2.2.3 GISC (Generic Intelligent System Control)

GISC is being developed by the Department of Energy. It is not yet a fully defined architecture. It includes a set of software systems called GISC-Kit contributed by various DOE laboratories. As described in [Griesmeyer1], *"GISC is an approach to the construction of controllers for complex robotic systems … GISC-Kit is the library of* *software modules that the designer of a robot system controller can access for an actual implementation*". The primary domain intended of GISC is robotics for cleaning up hazardous waste at DOE sites, but the architecture itself is not limited to that.

One of the components of the GISC-Kit is a system called General Interface for Supervisor and Subsystem (GENISAS) [Griesmeyer2], which is a communications and event-handling shell service for supporting a command-and-status protocol between superiors and subordinates in a hierarchical control system. The shell knows about various types of command and status messages, but does not know the semantics of the messages (e.g., in the case of a command, it does not know what the superior is telling the subordinate to do).

6.2.2.4 Jones

Albert Jones of NIST has written extensively on CIM architectures, starting with the AMRF in [Jones2] and [Jones5] and continuing with an examination of architecture issues in [Jones1]. In recent years, he (with several colleagues) has proposed an architecture in a series of papers [Jones3], [Jones4], [Davis1], [Joshi1], which identifies the functions of controllers as adaptation, optimization, and regulation. The controllers are arranged hierarchically. "*Adaptation is responsible for generating and updating plans for executing assigned tasks. Optimization is responsible for evaluating proposed plans, and generating and updating schedules. Regulation is responsible for interfacing with subordinates, monitoring execution of assigned tasks.*" [Jones4, page 63].

6.2.3 Modified Hierarchical Control Architectures

Systems which serve many different controllers in different parts of the control hierarchy (such as material handling systems, which deliver part blanks to workstations) pose special challenges for a hierarchical architecture. Modifying a hierarchical control architecture by implementing these controlled services as independent agents without a superior is typical. This is what we mean by a modified hierarchical control architecture.

The advantages and disadvantages of a modified hierarchical control architecture are similar to the unmodified version, except that the disadvantage of not handling system-wide services well is removed, and a disadvantage of having system performance be less predictable is added.

6.2.4 Heterarchical Control Architectures

In a pure *heterarchical control architecture*, each controller has no superior and no subordinates[6]. Controllers interact by issuing requests for bids, making bids, and entering into contracts to do work.

Heterarchical architectures typically use distributed databases - each controller has its own database - but that is not a requirement.

6. Device drivers for equipment are not regarded as subordinates.

Advantages of a heterarchical control architecture include:

(1) strong natural modularity,
Each controller can be treated as a software module.

(2) very easy extensibility,
The system may be extended by adding controllers and computers.

(3) graceful degradation,
If something goes wrong with a controller during system operation, only the controller that has the problem needs to stop.

(4) allowance for different time scales.

Disadvantages include:

(1) need for heavy communications among controllers,
This is to handle all the soliciting, bidding, and contracting.

(2) very hard to predict system behavior,
Predicting what a heterarchical architecture will do (which controllers will do which tasks and when a task will be done, for example) is typically difficult. It is often not even clear if the solicit-bid-contract procedure will reach closure.

(3) very hard to optimize system behavior globally.

[Dilts1] discusses heterarchical architectures in the context of a comparison of types of architectures and seems enthusiastic about them. [Hatvany1], [Johnson1], [Shaw1], [Ting1], and [Vamos1] espouse heterarchical architectures.

A dispassionate analysis of the performance of heterarchical architectures is offered in [Upton1].

A heterarchical architecture is not necessarily focused on controllers. The focus may be on the item being worked on, rather than on the item that does the work. The domain for which this seems most appropriate is discrete parts manufacturing. The method of interaction employed is to make “intelligent parts” which know what needs to be done to them and can negotiate with controllers and enter into contracts to have it done. The controllers with which an intelligent part deals must be independent agents (able to negotiate and enter into contracts).

Neil Duffie (with co-authors) has published several papers about heterarchical architectures with intelligent parts, including [Duffie1], [Duffie2], and [Duffie3]. The papers report implementations of heterarchical architectures for discrete parts manufacturing in a research environment.

A third variety of heterarchical control is called the “data flow” model, although “claiming” model might be a better description. In this model, controllers are active agents, but there is no bidding. A special component, Module 0, is responsible for keeping track of what work has been done on a set of tasks. Module 0 broadcasts the work to be done, and each controller that can do a piece of work puts it on its queue of things to do. When the piece of work comes to the top of a controller's queue, the controller claims the piece of work and works on it; all other controllers remove it from their queues.

[Ting1] discusses controller-driven, part-driven, and data flow models.

6.2.5 Other Architectures Emphasizing Control

6.2.5.1 NGC (Next Generation Controller)

Under contract with the NGC program of the United States Air Force, the Martin Marietta Company developed a Specification for an Open System Architecture Standard (SOSAS). SOSAS is documented in six draft volumes [Martin1] through [Martin6] totaling about 1000 pages.

The SOSAS architecture is the most comprehensive architecture which was reviewed for this report, but it is very uneven. Most curiously, although control is emphasized throughout, there is no explicit commitment to any of the four kinds of control described above. The services which are provided lend themselves to a hierarchical or modified hierarchical architecture and not to a heterarchical architecture. There is no explicit support, in particular, for controllers requesting bids, making bids, or entering into contracts. Neither, on the other hand, is there any support for modeling a controller hierarchy.

SOSAS defines two distinct categories of architectural units: services and applications.

Services are defined in the first volume of the set and include operating system (called "platform"), communications, data management, presentation management, task management, geometric modeling, and basic I/O. Operating system and basic I/O are dealt with on less than a page each by referencing POSIX and OBIOS standards, respectively.

Four "standardized applications" intended to use the services are defined, one in each of the last four volumes of the set: workstation management, workstation planning, controls, and sensor/effector. The controls volume includes the definition of a neutral command language (NCL) for numerically controlled machine tools.

Volume 2 gives formal models in the EXPRESS language of information required in SOSAS-compliant systems. This volume defines three categories of information models: execution, manufacturing practice, and controller practice. Many hundred EXPRESS entities and types are defined. Rather little explanatory text accompanies the formal EXPRESS statements.

The SOSAS volumes include extremely little to describe the range of applications to which the architecture is intended to apply. Conformance is given attention in five of the six volumes, but is marked as "TBD" in many places in volumes III, IV and VI. The SOSAS provides no methodology for developing systems which comply to its architectural specifications.

6.2.5.2 Subsumption

A control architecture called the “subsumption” architecture has been developed by Rodney Brooks [Brooks1] which is quite different from any of the others discussed in this report. The focus of the architecture is on systems such as robots, which may use many controllers, but have system-wide behaviors, such as walking or hiding. The general approach is to use layers of behaviors, with upper level behaviors being built on lower level behaviors. For example, the “exploring” behavior is a level above the “moving around” behavior, and uses the “moving around” behavior.

The subsumption architecture uses sensed data from the environment directly as input to the controllers. This approach is in contrast to the approach (used in MSI and RCS) of constructing a model of the environment from the data and using this model as an input to controllers.

The atomic unit of the subsumption architecture is a process, usually implemented as an augmented finite state machine. Processes are grouped into behaviors as a molecular unit. There can be message passing, suppression, and inhibition between processes within a behavior, and there can be message passing, suppression, and inhibition between behaviors. A behavior cannot interact with a process inside another behavior.

6.3 Architectures Emphasizing Data Aspects

Some architectures say little or nothing about control but emphasize data aspects. These cannot be located along the control dimension of architecture classification, so we have made a separate class for them.

6.3.1 CIM-OSA

CIM-OSA (Computer Integrated Manufacturing - Open Systems Architecture) is an architecture being developed by the ESPRIT AMICE project [Chen1], [Jorysz1], [Jorysz2], [Klittich1], [Klittich2], [Panse1], [Shorter1]. The CIM-OSA scope is limited to CIM but looks at the whole system life-cycle, including consideration of requirements specifications at one end and system change at the other. The aim of the architecture is to provide an integrated framework to support manufacturing within an enterprise. The documentation of CIM-OSA does not define clearly what the framework is. It includes, at least, an integrated data system architecture for manufacturing enterprises.

The data system architecture provides “front end services” to users. There are four types of front end service: application (e.g., CAD or CAPP), human, machine (e.g., robot or NC machine tool), and data management. Each front end service uses a “data access protocol” to provide the requested service. The two data access protocols are “business process services” and “information services”.

The CIM-OSA architecture is a work in progress. Prototype implementations conforming to the architecture are only now being built.

CIM-OSA does not include control in the architectural specifications. There is not even any discussion of control processes as independent entities.

More details on the CIM-OSA architecture are provided in Appendix E.

6.3.2 Harhalakis

[Harhalakis1] describes an unnamed data-oriented architecture for computer integrated manufacturing which deals with data common to computer aided design, computer aided process planning, and manufacturing resource planning. This architecture "*focuses on the facility level of the enterprise and concentrates on the integration of information, rather than hardware*". The intent is to ensure consistency and integrity between data common to two or more modules. Each module is assumed to have its own database.

Implementation is accomplished by identifying the key information types used by each module and incorporating a rule base into a distributed database management system to control the flow of data between modules.

7 The RCS and MSI Architectures

Is it feasible for RSD and FASD to jointly develop any single reference architecture? If it is feasible and done, how useful will the resulting architecture be?

It should be feasible for RSD and FASD to develop a joint architecture if its scope is sufficiently limited. The broader the scope, the harder it will be to define a joint architecture.

In this section we describe the RCS and MSI architectures individually and then compare them issue-by-issue, using the issues identified in Section 4 and Section 5. An issue-by-issue analysis of MSI and RCS is in Appendix C.

7.1 The RCS Architecture

The bibliography to this report lists 40 papers about RCS by 14 primary authors. There are more papers about RCS not reviewed for the report. In this section we will use “the RCS papers” to mean the 40 papers about RCS which were reviewed.

This section provides a brief description of RCS, using the elements of architectural definition identified in Section 3 of this report to structure the section.

The definition of RCS has evolved over the years, and different people developing or using it have different views on what it should be. The description given here is intended to follow the mainstream, as defined primarily by the eleven papers by James Albus reviewed for this report. Where there are significant variations in other papers, they are cited, but we have not tried to describe all variants of RCS. A more detailed discussion of RCS and its variants may be found in Appendix C.

7.1.1 Scope and Purpose

RCS is intended as an architecture for complex, integrated machine control systems[7] which work in a changing world and keep pace with the changes in real time. The spectrum of intended RCS applications includes:

(1) high-speed servo control of machines with multiple joints or axes of motion

(2) coordinated control of several machines or large machines with several subsystems

(3) computer integrated manufacturing

(4) mining

(5) submarine navigation

(6) space station robotics

(7) land vehicle driving.

7. Most of the RCS papers also say that RCS is an architecture for “intelligent” systems. In this report, however, we do not deal with the notion of intelligence.

7.1.2 Domain Analyses

The RCS architecture does not require any formal analyses. In order to apply RCS, however, a user of RCS will find it necessary perform analyses (task decomposition, controller hierarchy structure, etc.).

7.1.3 Architectural Specifications and Methodology For Architectural Development

In most of the RCS papers, architectural specifications are not distinguished from methodology for architectural development. Some papers ([Quintero3, page 3], for example) explicitly say that “methodology” and “architecture” are interchangeable.

Nevertheless, a methodology for architectural development for building RCS systems is given in at least one RCS paper [Quintero3, section 6], which describes the activities a control systems developer should do and the types of architectural specifications that should be produced as a result.

RCS is not explicitly divided into tiers of architectural definition in the literature describing it, but it seems implicitly to have three tiers below the top level, as shown in Table 1 on page 9, so that RCS fits that table fairly well.

7.1.4 RCS Control Systems and their Environments

The RCS architecture provides for control of systems which react to events in the environment. Control systems are expected to have mechanisms for sensory input so that changes in the environment can be detected. The control system is constantly monitoring its sensory input to determine when events have occurred in the environment that it must react to. The processing of raw sensor data into abstract information about the condition of the environment is termed *situation assessment*. Once situation assessment has been performed, the control system makes decisions about what actions should be taken and plans reactively for the events it perceives. The execution of plans produces the external actions needed to cope with the environmental changes. An RCS controller continuously performs a sense-decide-act cycle.

7.1.5 Architectural Units of RCS

An RCS system interacts with the environment by sensing conditions in the environment with its sensors and performing actions in the environment with its actuators. The internal representation of selected features of the environment and the state of the RCS system is termed the of the system. The *world modeling* architectural unit governs interactions with the *world model*. In addition to world modeling and the associated world model, an RCS system includes three other architectural units. The four internal architectural units of RCS are:

(1) *sensory processing* (SP),

(2) *world modeling* (WM),

(3) *behavior generation* (BG),

(4) *value judgment* (VJ).

In early papers about RCS, behavior generation is often called *task decomposition* (TD).

The sensory processing, world modeling, and value judgment architectural units are involved in situation assessment, while the value judgment and behavior generation architectural units are involved in deciding what to do. Figure 1 [8], "RCS View of an Intelligent Machine System," illustrates these conceptual architectural units. The system includes everything above the lower horizontal dotted line. The world model is central, since other architectural units rely upon it to provide and accept current information about the environment. The remainder of the architectural units are arranged in a clockwise loop, depicting the notion that the system continually repeats a sense-decide-act cycle.

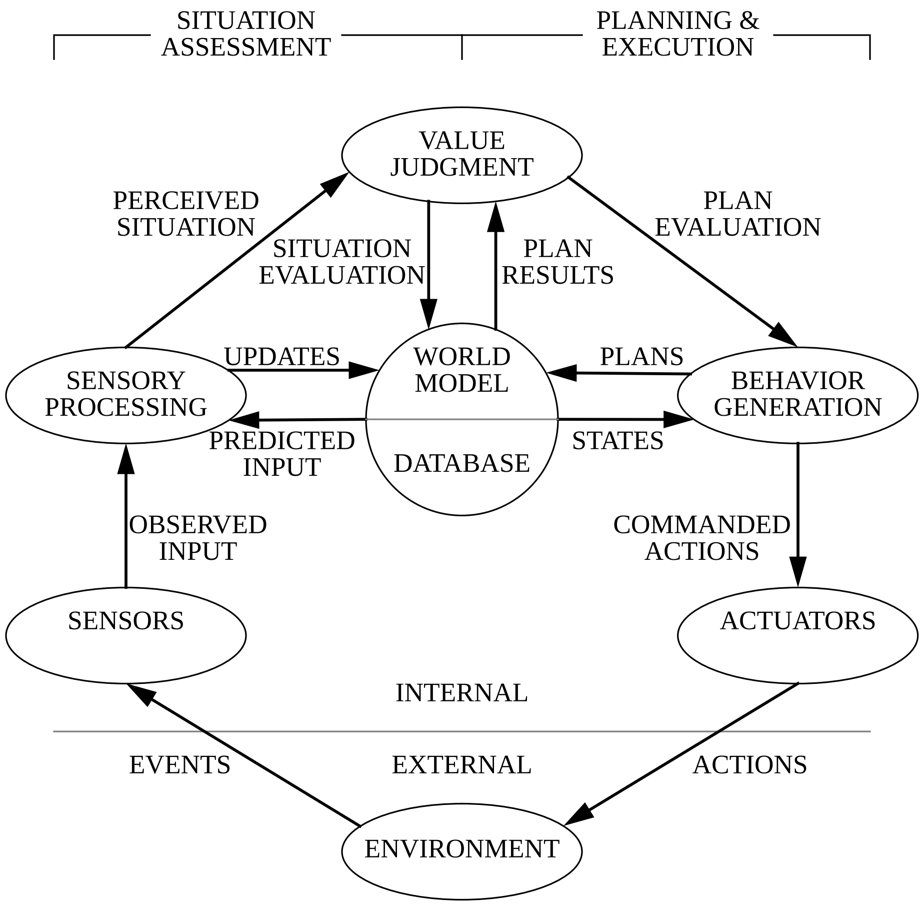


Figure 1. RCS View of an Intelligent Machine System

8. [Quintero3, Figure 2], [Albus 4, Figure 1], [Herman1, Figure 5], [Michaloski1, Figure 1]

In Figure 1, the sensory processing function system (described in more detail in Section 7.1.8) takes sensory data from sensors, interprets the data, and passes the interpreted data to world modeling.

The world modeling function keeps a description of the environment and the internal state of the system (the world). It receives information from sensory processing for updating the world model. It also predicts events and sensory data and answers questions about the world model. The world modeling function interacts with the RCS system's database. The database is usually described as a distributed, global database - in the sense that all data is available throughout the system.

The behavior generation function (described in more detail in Section 7.1.6) makes plans and carries them out by controlling the system's actuators.

The value judgment function evaluates both the observed state of the world and the predicted results of hypothesized plans. It computes costs, risks, and benefits both of observed situations and of planned activities. The value judgment function thus provides the basis for choosing one action as opposed to another, or for acting on one object as opposed to another.

In terms introduced earlier in this report, "task generation" in RCS is performed by the value judgment and behavior generation function, with input from the sensory processing module and sensors. "Task execution" is performed within the behavior generation module of RCS. The following section gives additional details on task generation and execution in RCS.

7.1.6 Hierarchical Levels in RCS

The behavior generation system in RCS is strictly hierarchical. That is, each controller responsible for behavior generation has at most one superior and zero to many subordinates, for the purposes of performing actions. Control levels are typically refered to by number (numbering from 1 at the bottom, on up) or by a label. Different applications of RCS have used different labels for these levels, but typically the lowest level is termed the servo level, next is the primitive level and above that is the elementary move (or e-move) level.

Superiors interact with subordinates by sending commands to them and receiving status messages from them. Each controller has a number of tasks that it can carry out, and these tasks are understood by the superior of the controller.

The RCS architecture decomposes system activities into hierarchical levels. The levels are characterized by the relative amount of time taken to perform activities and by the relative spatial extent of the activities. Roughly an order of magnitude change in spatial and temporal extent is expected between any two adjacent levels, with activities getting smaller and faster at lower levels of the hierarchy. Between levels, a corresponding change is also expected in the interval of time over which the system detects and remembers events. Approximate times corresponding to the RCS control levels are shown in Figure 2.[9]

9. [Albus4, Figure 4], [Albus5, Figure 22a], [Albus6, Figure 5], [Albus7, Figure 4.2], [Albus8, Figure 2], [Michaloski1, Figure 3], [Quintero3, Figure 4].

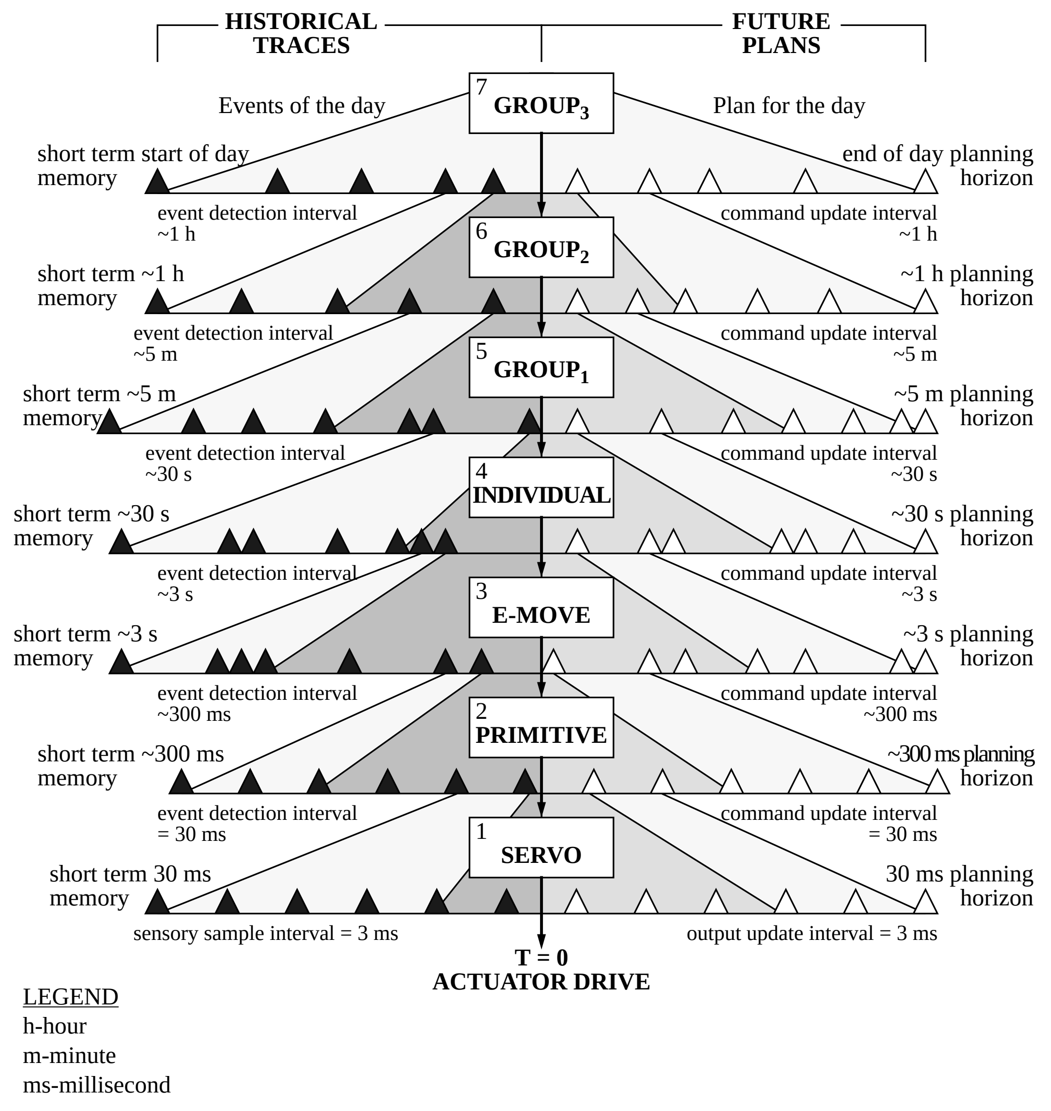
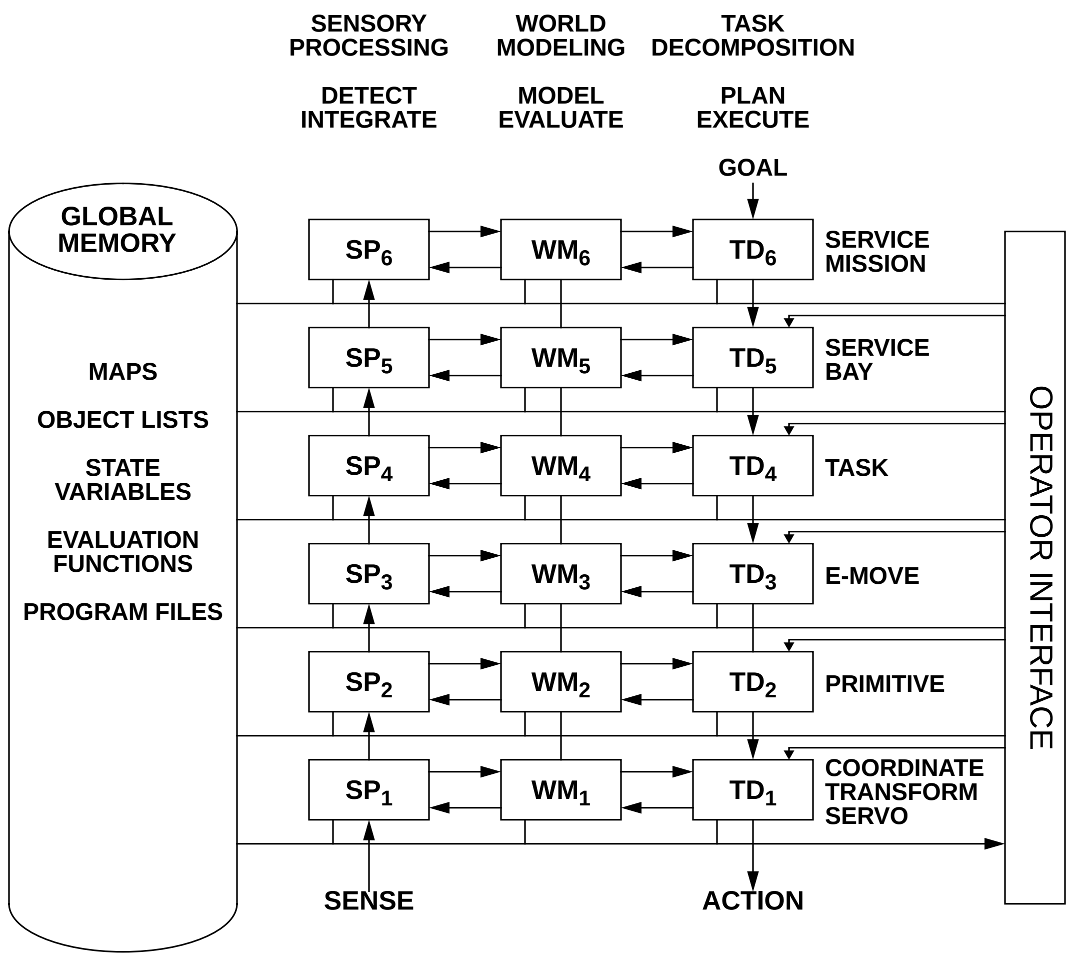

HISTORICAL TRACES
FUTURE PLANS
7 GROUP3
Events of the day
Plan for the day
short term start of day memory
end of day planning horizon
event detection interval ~1 h
command update interval ~1 h
6 GROUP2
short term ~1 h memory
~1 h planning horizon
event detection interval ~5 m
command update interval ~5 m
5 GROUP1
short term ~5 m memory
~5 m planning horizon
event detection interval ~30 s
command update interval ~30 s
4 INDIVIDUAL
short term ~30 s memory
~30 s planning horizon
event detection interval ~3 s
command update interval ~3 s
3 E-MOVE
short term ~3 s memory
~3 s planning horizon
event detection interval ~300 ms
command update interval ~300 ms
2 PRIMITIVE
short term ~300 ms memory
~300 ms planning horizon
event detection interval = 30 ms
command update interval = 30 ms
1 SERVO
short term 30 ms memory
30 ms planning horizon
sensory sample interval = 3 ms
output update interval = 3 ms
T = 0
ACTUATOR DRIVE
LEGEND
h-hour
m-minute
ms-millisecond

Figure 2. RCS Timing Diagram

At each control level, the sensory processing, world modeling, behavior generation and value judgment architectural units may exist. Figure 3[10] illustrates a six level RCS architecture appropriate for telerobotic applications. The label TD on Figure 3 and elsewhere in this section stands for *task decomposition*, which is a synonym for behavior generation.

10. [Albus1, Figure 3], [Albus5, Figure1a], [Albus6, Figure 2], [Albus7, Figure 1.1], [Albus8, Figure 1], [Fiala2, Figure 1], [Fiala4, Figure 1], [Herman1, Figure 6], [Herman2, Figures 1, 4, 6, 7, 14, 18], [Herman3, Figure 1], [Lumia1, Figure 1], [Lumia2, Figure 1], [Lumia3, Figure 1], [Szabo 3, Figure 1], [Wavering1, Figure 1]. Many versions of the figure have G instead of SP, M instead of WM, and H instead of TD.

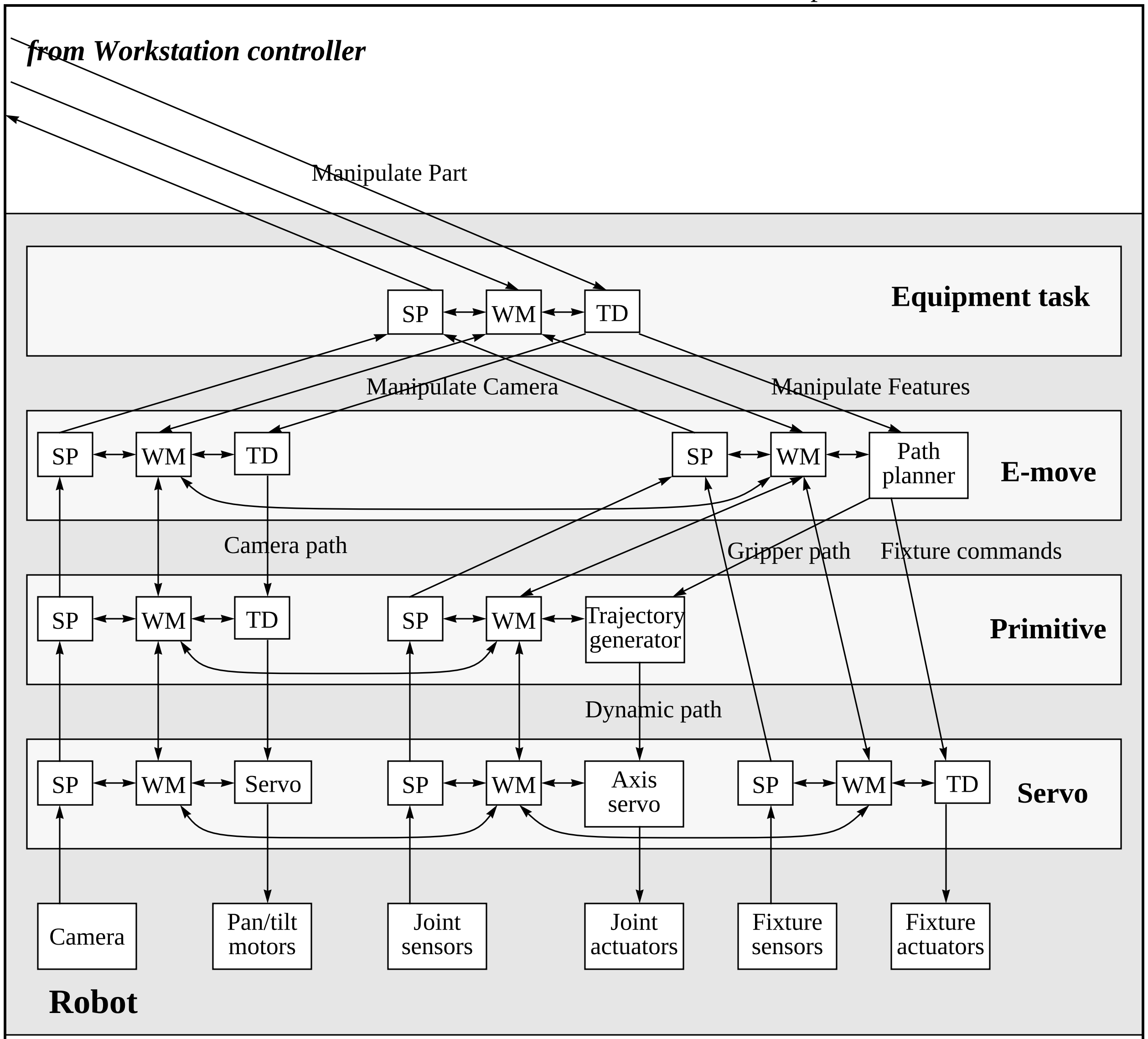
In considering Figure 3, it must be understood that each rectangle labeled SP, WM or TD (except those in the top level) normally represents several separate instances of the given function. That is, there are several TD5 behavior generators that are subordinates of the TD6 behavior generator, for each TD5 behavior generator there are several TD4 subordinates, and so on down the hierarchy. The situation can be visualized by imagining that the figure is the front view of a 3-dimensional arrangement, which, when looked at from the side, is a hierarchy. The version of the figure in [Szabo5] hints at this, and it is discussed in [Fiala2, section 1].



Legend
SP – Sensory Processing
WM – World Modeling
TD – Task Decomposition

Figure 3. RCS Control System Architecture

An example of a hierarchy in which there are several SP, WM, and TD boxes at each hierarchical level below the top is shown in Figure 4 (a portion of Figure 1 from [Albus2]). This figure shows a control system for a robot with a camera, an active fixture, and grippers. The robot is subordinate to a workstation level controller not shown on the figure. Four hierarchical control levels are shown: equipment task, e-move, primitive, and servo. As noted in the preceding paragraph, there are several SP, WM and TD boxes at each hierarchical level below the top.



Legend
SP – Sensory Processing
WM – World Modeling
TD – Task Decomposition

Figure 4. Example RCS Robot Control Hierarchy

7.1.7 Tasks and Work Elements

Methods of defining work elements and describing tasks are not strictly specified in RCS, and different sorts of specifications are used in different implementations. The most detailed suggestions for task definition are given in [Michaloski1, section 4]. In general, the RCS papers use the term “task” for what this report calls “work element”. In most RCS implementations, a work element has a name, and the effect of carrying out a work element with a given name is specified in some way. It is also usual to describe the information needed to specify an instance of a work element. In carrying out an instance of a work element, this information may be passed as parameters to a command or retrieved from a database. In all implementations, a command to perform a task may be specified by naming a work element and giving the values of zero to many parameters which characterize the work element.

7.1.8 Sensory Processing

The sensory processing function of an RCS system takes sensory data at the lowest hierarchical level, interprets the data, and passes the interpreted data to world modeling. Sensory data may need to be filtered as it arrives. Sensory data may also need to be integrated over space (for constructing a map, for example) or time (for speech recognition, for example). As indicated on Figure 3, the integration of data proceeds upwards from level to level. In the case of shape recognition in a vision system, for example, points might be detected at the lowest level and fed upwards where some of them may be integrated into lines; lines are fed upwards, and some of them may be integrated into boundaries of (geometric) faces; faces may be fed upwards and integrated into closed shells of solid objects.

The sensory processing function may be aided by receiving predictions of sensory data from the world modeling function.

Sensory processing at upper levels may perform data fusion, in which different sets of data which should be consistent (such as the distance to an object measured by optical triangulation, radar, and sonar) are reconciled, or different types of data (outline and color, perhaps) are correlated.

7.1.9 Task Definition and Decomposition

The behavior generation (or task decomposition) process is shown in Figure 5 [11]. Behavior generation is decomposed into three parts:

(1) *job assignment* (JA)

(2) *planning* (PL)

(3) *execution* (EX)

11. [Albus1, Figure 4], [Albus4, Figure 7], [Albus5, Figure 2], [Albus7, Figure 2.1], [Herman1, Figure 9], [Huang1, Figure 3], [Lumia1, Figure 2], [Lumia3, Figure 2], [Michaloski1, Figure 4], [Quintero3, Figure 5]

A task is decomposed by a job assignment manager (JA) into subtasks for several subordinates. The planner for each subordinate (PL1, PL2, and PL3 on the “Spatial Decomposition” axis in the figure) orders the subtasks in a temporal sequence (the “Temporal Decomposition” axis). Each subtask is executed by an executor (EX1, EX2, and EX3 on the figure). The same executor will execute different subtasks at different times, as indicated by the dotted circles and dotted lines on the figure. The figure shows what happens at one hierarchical level. The subtasks coming from an executor at one level become the tasks for the next level down.

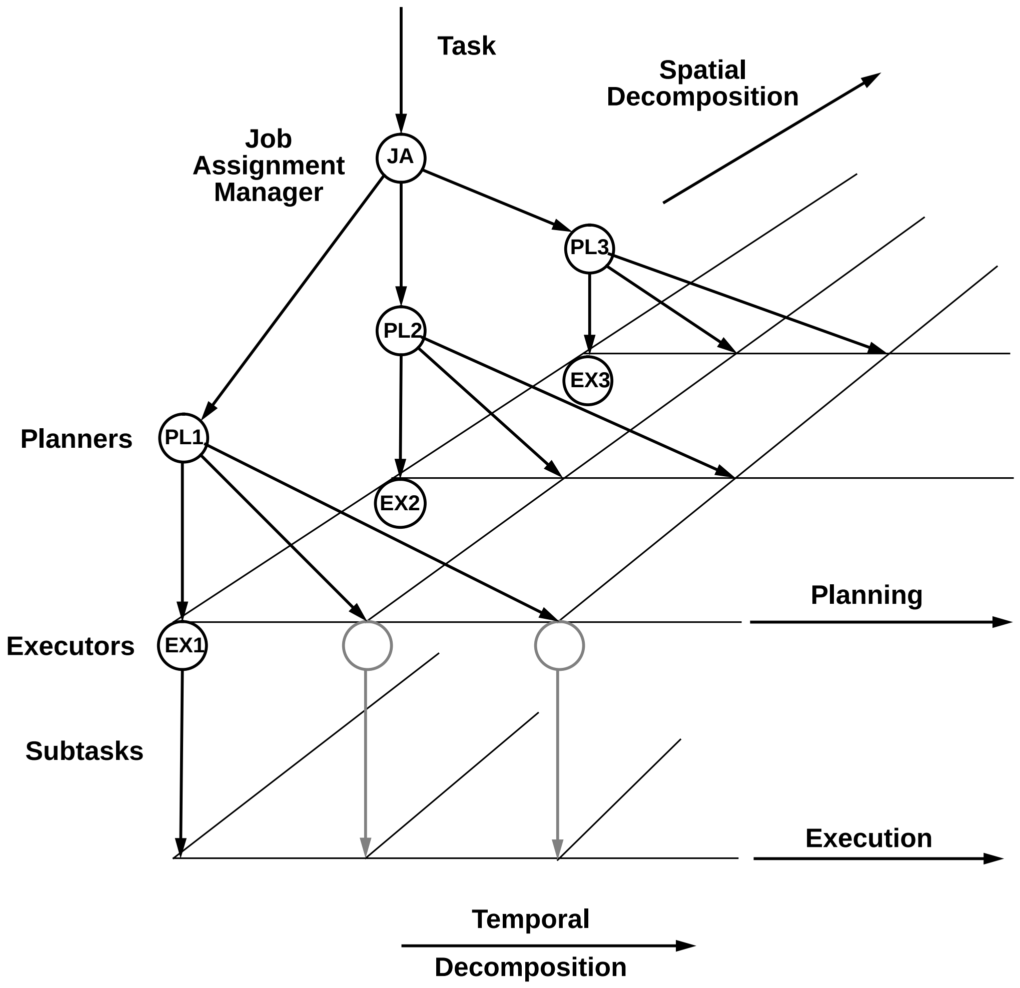


Figure 5. RCS Task Decomposition

7.1.10 Communications

RCS does not specify a standard for communications but anticipates that at lower hierarchical levels, fast communications will be required. Some RCS papers, [Quintero3] for example, state that communications must permit input and output at any time, regardless of current system activities, in order to ensure that sufficiently fast performance can be achieved. In most implementations, shared memory or some form of NIST's Common Memory [Libes1], [Rybczynski] has been used.

Standard communications protocols such as Ethernet/TCP/IP [Tanenbaum] or RS-232 [EIA] have been used in RCS implementations for interfacing processes which are not on a common bus. For mobile applications, radio frequency communications hardware is also used.

7.1.11 Error Recovery

Automatic error recovery for handling "abnormal" error conditions is discussed in [Albus5] and [Herman3]. In [Albus5] it is anticipated that if there is a subtask failure, the executor should branch immediately to a pre-planned emergency subtask while the planner selects or generates an error recovery sequence. [Herman3] reports an implementation of a subtask failure re-planning software module.

7.1.12 Conformance Criteria

Although a few RCS papers discuss the issue of conformance criteria, none of the RCS papers contain any.

7.2 The MSI Architecture

The MSI (Manufacturing System Integration) architecture is a product of the Manufacturing System Integration project which was conducted from 1990-1993 within the Factory Automated System Division. This architecture is the work of the MSI architecture committee members: Ed Barkmeyer, Steven Ray, M. Kate Senehi, Evan Wallace and Sarah Wallace.

The architecture is directly applicable to the production of discrete metal parts. Many of the concepts are more broadly applicable, but a discussion of this is not included in this summary of the architecture. The MSI architecture focuses upon the operation of a shop which receives orders and raw materials for the production of parts and in which each controller can be directly manipulated.[12] The architecture is required to be able to control a shop with any combination of physical and emulated equipment. Additionally, the architecture is required to permit the integration of systems not initially designed to work within the architecture, such as commercial products or university-produced prototype systems.

12. Other types of manufacturing organizations, such as rework organizations, which receive damaged pieces to be repaired and must construct custom plans to repair them, or shops which contain autonomous "subshops" such as tool cribs have been considered by the MSI architecture committee, and found to need adaptations of the architecture which are not fully developed.

The architecture draws upon the early work of the AMRF on hierarchical control [Albus12], [McLean1], [Simpson1] and the work of the Manufacturing Data Preparation Project [Hopp1] which focused on information required for manufacturing, particularly process plans and resources.

This section provides a brief description of the second version of the MSI architecture. Although the second version of the architecture differs somewhat from the initial version, many of the concepts from the initial architecture still apply. Documentation for the initial architecture may be found in [Senehi2].

7.2.1 Architecture Overview

The goal of the architecture is to integrate the operation of a shop which manufactures discrete metal parts. Particular emphasis is placed by the architecture on the integration of shop planning, scheduling and control functions in both nominal and error situations. The architecture does not attempt to provide enterprise integration. In particular, it does not describe information needed for business decisions, such as whether to buy or manufacture a part.[13]

The architecture approaches integration by identifying the systems in the shop which need to be integrated, examining the interactions among the systems, and proposing mechanisms to ensure that these systems function in a cohesive manner.

7.2.1.1 MSI Architectural Units

The MSI architecture identifies a number of systems which are normally part of the shop production environment. The architecture defines architectural units corresponding to each of the shop systems identified, characterizing each system by the functions which it performs. The MSI architecture avoids specifying the internal structure of any of the architectural units. This approach facilitates building implementations of the architecture which use systems which were not designed specifically to work within the architecture.

The architectural units which correspond to shop systems and their functions may be summarized as follows:[14]

(1) Part Design—which creates the designs for parts, associated fixtures and jigs,

(2) Process Planning—which creates step-by-step plans or numerical code for manufacturing a part and its associated fixtures and jigs, according to the part design,

(3) Production Planning[15]—which selects batch sizes, specific machines, and scheduled times to perform the tasks specified by a process plan,

13. It does permit the user to include technical information such as cost functions for use in determining the way in which an order is filled, or scheduling and quality parameters for the parts being generated.
14. Additional systems may, of course, be part of a manufacturing system, but these have not been considered in the formulation of the architecture.
15. Schedulers are one type of production planning system.

(4) Controllers—which perform manufacturing tasks,

(5) Order Entry—which permits entry of orders which direct a shop as to what to make and when to make it,

(6) Configuration Management—which identifies and controls shop resources and capabilities,

(7) Material Handling—which routes and delivers material throughout a shop.

7.2.1.2 Interactions of Architectural Units

Most architectural units are *loosely coupled*, that is, they share information, but their activities do not need to be coordinated except to ensure the integrity of the information they share. In this report, this type of interaction is called *indirect interaction*.

In indirect interaction, the shared information is stored in a known location (e.g., a memory location, database, file, variable), and components (of an implementation) may be given access (e.g., read, write, no access) to the information as required. Components which have access to the same information need not be known to each other, and need not acknowledge any access or change of the information by any other component.

In order to integrate architectural units which interact indirectly, the MSI architecture specifies that it is sufficient to describe the shared information at a conceptual level, and provide guidelines for the access of the information. The description of the shared information is given through a number of information models. The information models, and the guidelines for information access form the information architecture of the MSI architecture. This will be discussed in detail in Section 7.2.2.

The production planning and control architectural units are *tightly coupled*, that is, they must interact more closely than through the passive sharing of information. To understand the interactions of these architectural units and the MSI solution to integrating them, it is necessary to understand the MSI perspective on task generation and execution in a shop.

A shop’s function is to manufacture products to fill the orders which it has received. The orders are for some number of a specific product, which is described by a design. For each design, a *process plan* is formulated. The process plan gives detailed instructions on how to manufacture the product, using classes of resources. For example, a process plan might say “This step requires a three-axis milling machine,” rather than “This step requires machine XYZ001.” When an order is received for making a number of a product, an appropriate process plan is retrieved or generated, the order is broken into batches for manufacturing and for each batch, the specific resources for product production are selected and the plan and the resources are scheduled. The end result of performing these operations is a production plan which contains all necessary information for the making of the product. When the scheduled time for the start of manufacturing of the batch arrives, the controllers in the shop interpret the production plan and perform the work to manufacture the product.

In performing the work of manufacturing the product, the activities of controllers must be coordinated. This is accomplished by using two mechanisms. First, as described above, *production plans* are generated which schedule the activities of each of the controllers in the shop. Secondly, controllers are connected in a control structure that provides support for integrated start-up, shutdown, emergency stopping, and affecting the disposition of tasks generated from production plans. The MSI architecture requires that the controllers in the shop be arranged in a hierarchical control structure. Commands are transmitted form superior controllers to their subordinate controllers, and subordinate controllers send status information to their superiors. Interaction through a command-and-status mechanism is refered to as *direct interaction*.

Thus, the integration of the tightly coupled planning and control architectural units requires both indirect interaction through the process and production plan information and direct interaction through a control hierarchy. The representation of the information for process and production plans is discussed in Section 7.2.2. The control structure is discussed in Section 7.2.3.

7.2.2 Information Architecture

As previously mentioned, the MSI architecture states that for indirect interactions among architectural units, it is necessary only to describe the shared information and the information access characteristics (i.e., which components can access which information and what type of operations the components can perform). The following sections discuss each of these in turn.

7.2.2.1 Information Models

The information needed to integrate the manufacturing shop is highly interconnected. The Integrated Production Planning Information Model describes the manufacturing environment at a high level of abstraction. This model shows the relationships among product design, shop resources, plans, shop configuration, and shop status. Detailed models were made for process and production plans, resource types, orders, tools, shop status, and shop configuration. Following is a brief summary of the information models in MSI. More details of the process plan model are available in [Catron1]. Details for other models are available in [Barkmeyer2] and [Ray1]. The specification of product design is imported from the information models generated by the International Standards Organization Technical Committee 184, Subcommittee 4 (ISO TC184/SC4) [ISO3].

7.2.2.1.1 Plan Models

Process, production managed, and production plans are key vehicles by which information is shared between planning and control architectural units in the MSI architecture.

A process plan designates the steps necessary to make a part, specifying the sequence(s) of operations by which a part is made and the relative timing of these operations. As received from the engineering systems, process plans should contain a number of cost-effective alternatives which take into account the resources of the local production environment, but not the status of such resources. Process plans are directed graph structures which may express both alternative and parallel paths of part production, and sets of potential resources for part production. Process plans provide for synchronization of operations by several mechanisms and support hierarchical decomposition of operations as well. This definition differs from the traditional use of the term “process plan” in that alternatives are expressed within a single plan and the plan may specify resources by their class instead of specific instances.

A *production managed plan* gives the plan for producing a batch of parts and is derived from the process plan for making that type of part. One or more alternatives from the process plan is selected and material handling steps are placed where needed.

A production plan is constructed from the production managed plan by selecting, scheduling and planning for the allocation of the specific resources, and refining the material handling planning necessary to move the batch of parts from one resource to another. Since production managed and production plans are constructed with reference to a process plan, it is obvious that their representations are logically, if not physically, linked.

Production plans are parsed by controllers. In parsing a production plan, a controller may request information from databases, make judgments on which alternative to take based on this information, produce commands to direct subordinates to perform manufacturing tasks, or perform manufacturing tasks.

7.2.2.1.2 Resource Model

The resource model contains a physical and functional description of resources available in the shop. It contains templates for all such resources (e.g., machine tools, robots), information on shop floor configuration, and status information on shop systems. In addition to the typical physical resources and equipment expected in a resource model, the resource model includes *consumable resources* (such as coolant and solder) and *logical resources* which are pieces of information which have been created to assist the production management and control functions. Items from the resource model are used in both the process, production managed and production plans.

The models necessary for shop integration are given in the table below.

Table 3: MSI Information Models

Model	Description
Product Model	Specifies information needed to describe the parts being manufactured; includes information needed to create a solid model of the part, to describe manufacturing features, and to specify detailed information about tolerances.
Process Plan Model	Describes plans which give the steps necessary to make a single part, specifying the sequence(s) of operations by which a part is made, the relative timing of these operations, and classes of resources required.
Production Managed Plan Model	Describes plans for producing batches of parts along with routing information and is derived from the process plans for making those types of parts.
Production Plan Model	Describes fully developed plans for making batches of parts. the plans include specific resource selections, allocation and schedules for part production.
Resource Model	Contains a physical and functional description of resources available in the shop. Resources may be physical or logical.
Order Model	Describes information about orders; includes the type of the part to be manufactured, the quantity to be made and identifies information needed to record the engineering status and production status of the order.
Inventory Model	Specifies information needed about stock (e.g., part blanks), consumable machining supplies, free carriers, and completed parts no longer in-process. Such information includes type, quantity, location, etc.
Configuration Model	Describes the relationships between controllers, schedulers and network entities.
Materials Model	Describes the characteristics of raw materials, stock, and consumable machining supplies.

7.2.2.2 Data Storage and Access

The MSI architecture specifies that information which must be shared among components be placed in a data storage location which is accessible by all components which need this information. The architecture does not specify the data storage mechanism. Options include files, variables, memory locations, databases, etc. MSI permits both physically distributed and centralized storage. The access method typically depends on the storage mechanism and may be different for different data, depending upon which components need to share the data. Different access privileges to each item may be accorded to different components. In some cases, multiple components may be able to write the same data.

The architecture states that it is desirable that the physical and logical location of the data be invisible to components of an implementation of the architecture insofar as practically possible within performance constraints. The architecture also permits components to make local copies of shared information, but states that in this case, the component is responsible for maintaining consistency between the local copy and the public copy of the shared information.

At present, the architecture does not specify which systems should access each specific item of data. This omission was intentional, to give implementors of systems more freedom. Such a specification is a possible enhancement of the architecture and would aid vendors in constructing systems that could be made interoperable.

7.2.3 Control Architecture

In the MSI architecture, the control architecture provides for the integrated start-up, shutdown and maintenance of the controllers in the shop and provides a mechanism for performing operations on tasks such as starting, aborting, temporarily halting and resuming them. It is through the control architecture that errors in planning and task execution are discovered and repaired. The basic tenets are discussed in the following sections.

7.2.3.1 Levels of Control

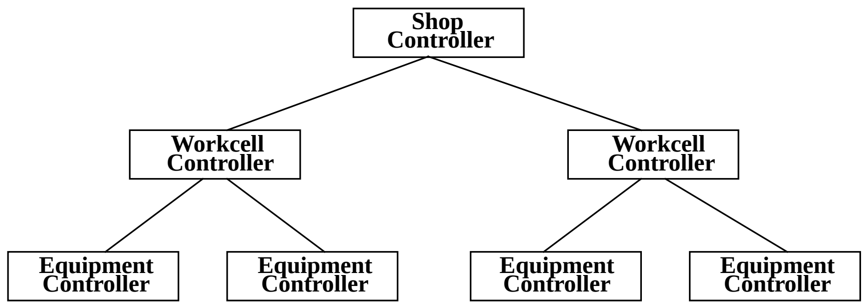
In the MSI architecture, control levels are arranged in a hierarchical tree structure. The hierarchical control structure has a single highest-level controller. Every other controller has exactly one superior controller from which it receives commands, and zero or more subordinate controllers to which it may issue commands.[16] See Figure 6 for an illustration of sample permitted control trees.

In the MSI architecture, the highest level controller is the *shop controller*. The shop controller has general responsibility for all production processes involved in filling orders. The coordination of all orders for parts, determination of global scheduling constraints and creation of routings for part delivery are done by the shop controller. However, many details required to fill these orders are the responsibility of subordinate controllers and are not visible to the shop controller.

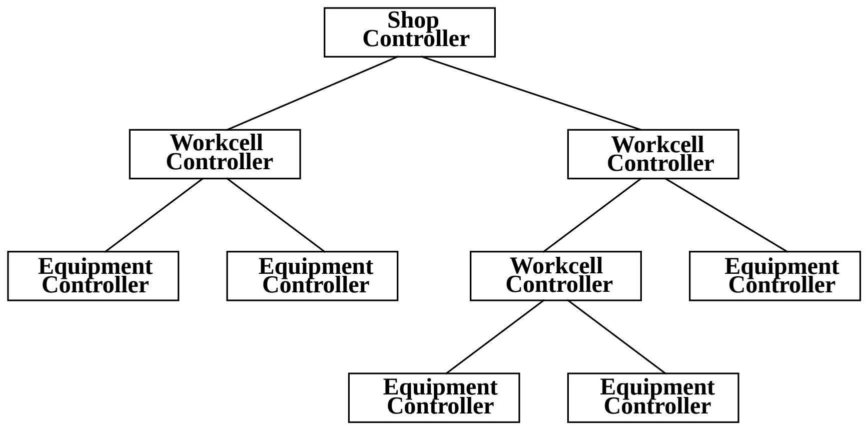
16. The reason for this constraint on the hierarchy structure is related to the mechanisms for recovering from errors. Should multiple supervisors be allowed, there world be no guarantees that any single controller in the hierarchy has a complete picture of the subordinate controller’s status, making error recovery extremely difficult.

Equipment is controlled by an *equipment controller*. By definition, an MSI equipment controller can execute only one task from its superior at a time; any internal task decomposition by an equipment controller invisible to the MSI architecture. Equipment controller tasks are items such as loading a part, opening a vise, or manipulating the spindle of a machine tool, depending on the particular equipment.

Between the shop and equipment controllers there may be any number of controllers, called *workcell controllers*, which coordinate the activities of two or more subordinate controllers, each of which is either an equipment or a workcell controller. The number of controllers between a given equipment controller and the shop controller remains unchanged regardless of the tasks being executed. This number specifies the level of control for that controller, and may be different for different equipment controllers depending on the complexity of coordination necessary. See Figure 6 for an example.



(a) This is an example of a control hierarchy, in which the number of levels between each equipment controller and the shop controller is the same.



(b) This is an example of a control hierarchy in which the number of levels between equipment controllers and the shop controller differs.

Figure 6. Examples of Valid MSI Control Hierarchies

In most cases it is expected that, once established, the control hierarchy will remain fixed as long as the shop is in operation. In some cases, it is possible to reconfigure the control hierarchy dynamically. It is intended that dynamic reconfiguration will only be used to remove a dysfunctional equipment controller or to bring new equipment on line. In any case, at any fixed time the MSI architecture specifies that there be a *single* control hierarchy originating at the shop controller.

7.2.3.2 Task Generation and Execution Process

In the MSI architecture, the execution of manufacturing tasks is a result of the parsing of production plans. For each controller involved in making a product, a production plan must be available to tell the controller what task to perform and when to perform it. Production plans are generated from corresponding process plans. Therefore, the hierarchical organization of process and production plans mirrors the hierarchy of controllers in the MSI architecture.

For each part design, process plans must be constructed for each level of the manufacturing hierarchy. Although process plans contain similar structure at all levels, distinct types of operations are performed at each level which are unique to that level. Listed below are descriptions of the operational characteristics of process plans at levels pertinent to the MSI architecture.

- Shop Level
  At the Shop Level, process plans primarily address the movement of workpieces and sequencing of different types of machining operations, such as turning, milling, etching, etc.
- Workcell Levels
  Workcell Level process plans prescribe the coordination of controllers subordinate to a given workcell, such as the use of a robot to load a machine tool table. Such plans can require extensive use of synchronization between process plans for subordinate controllers.
- Equipment Level
  Equipment Level process plans describe the most detailed level of operation that a process planner would generate. In this case, the activity called process planning in MSI terminology overlaps with what is usually termed off-line programming. The steps within such a plan provide instructions which are carried out by individual pieces of equipment. Example operations within the domain of metal cutting might include steps such as *drill hole* or *chamfer edge*. The degree of detail required in such a step depends on the capability of the controller. If the controller possesses sophisticated capabilities, higher level instructions such as those above, or even as abstract as *load part* and *fixture part*, might be sufficient. If the controller is less capable, instructions at the level of numerical code may be required.

While the specification of the task is gleaned from the process and production plans, a controller will not perform the specified task unless it is instructed to do so by its superior[17]. This permits the supervising controller to remain in control of the execution of the task by its subordinate. The ability of the supervising controller to manipulate the execution of the task by its subordinate on a gross level (such as stopping or aborting the task) aids greatly in handling scheduling and execution errors.

7.2.3.3 Error Recovery

In an error-free environment, the relationship between planning and control is straightforward. When errors occur, this relationship is greatly complicated and the ability of the control system to recover from an error is intimately related to the capabilities of the planning and controller systems. The MSI architecture explores error recovery from specific types of errors in the shop in detail, extracts requirements for the planning and control systems and devises interfaces which support error detection and recovery. These aspects of the architecture are detailed in the following sections.

7.2.3.3.1 Error Scenarios

Errors can be grouped into three different classes based upon their cause: resource error, task error, and tooling error. A resource error occurs when a piece of equipment, whose controller is part of the control hierarchy, becomes impaired (e.g. the machine tool changer jams). A task error is an error which affects a specific task only; the resource on which it is being performed is unaffected (e.g., if the robot drops the workpiece, the robot is unaffected). While a resource error usually causes a task error, task errors may occur without a resource error. A tooling error occurs when a tool is damaged (e.g., a cutter breaks) or unavailable (e.g., the tool was not delivered at the proper time). Tools differ from other resources in that they are not permanently associated with any member of the control hierarchy, but are moved from resource to resource as needed.

The MSI architecture committee examined a number of error scenarios from both the task and resource error categories. It was observed that the use of a hierarchical control system facilitates the localization of task error handling. When an error occurs in the execution of a task, if it is possible to resolve it by affecting only subtasks of the task, controllers at all levels of control above the superior controller are unaffected by the error. If localized error recovery is not achieved at this level of control, the recovery for the error is handled by the next higher control level in the hierarchy. At each level, there is potential for error resolution. Only in the event that the error cannot be resolved at any lower level is a global solution to the error required.

17. The Shop level controller is a special case. The architecture specifies that there is an entity which checks to see if a new order has arrived and which then selects and passes the appropriate production plan to the Shop controller. In an implementation, the previously described entity may be a separate component, or may be within the Shop controller.

For the error scenarios considered, methods of recovery from scheduling errors and equipment failures were examined and incorporated into the specifications for the functionality of architectural units and the interfaces of the architectural units in the control architecture. This is discussed briefly in the following sections. More details on error-handling in MSI are available in [Wallace1] and [Senehi3].

7.2.3.3.2 Planners, Controllers and Control Entities

Analysis of the error scenario reveals two capabilities which a production planner must have in order to be effective. The first and more general requirement is that the production planner must be able to do re-planning. *Re-planning* is the ability to localize the error to a subset of the tasks and only re-plan those which are affected.

If a re-planning capability does not exist, automated error recovery will be extremely limited. The production planner must schedule for the entire shop again. The availability of resources can be fed back into the scheduler, but the scheduler can not plan for the completion of partially executed production plans. Human intervention is required to avoid scrapping everything in execution when an error occurs.

The second requirement is the need for the production planner to work with a hierarchical control system. In order to localize an error at a given level of the hierarchy, the production planner must be able to plan for that level. Additionally, it must be possible for the production planner to be informed that a resource or task error has occurred at a given level, and re-planning may be necessary.

Error information which is needed to re-plan must be made available to the production planner (e.g., how many minutes late a machining task is expected to be). As a minimum, the production planner must be able to be notified by the shop controller that a resource or task error has occurred and re-planning may be necessary. It is the controller’s responsibility to notice the error and inform the production planner; the production planner does not monitor either the health of the controller or the execution of tasks. Beyond these interface requirements, the internal architecture of the production planner is not specified.

When an error occurs, a controller may apply any strategies it has available to repair the problem. If these local efforts at correcting the problem fail, the controller must hand the problem to its superior for correction. In order for a controller to participate in error recovery involving its superior, it must be able to:

(1) detect when a subordinate has failed,

(2) detect when a subordinate’s task is late,

(3) abort task execution,

(4) halt task execution and retain information to restart later,

(5) restart task execution from previous point,

(6) halt task execution and discard all information related to the task,

(7) halt task execution and regard the task as complete,

(8) estimate task completion time, and alter task execution based on new parameters (e.g., new start, completion times).

The inability of either the production planner or the controller to perform any of the indicated functions does not prevent a production planner or controller from being integrated into a control system for a shop using the architecture, but it does weaken the recovery ability of the system.

7.2.3.3.3 Control Entities

Since effective participation in the error recovery mechanism requires both a (production[18]) planner and a controller, the MSI architecture defines an architectural unit called a *control entity*, which consists of a planner and its associated controller.[19] It should be emphasized that the control entity is a logical, rather than a physical architectural unit. The planner in the control entity is required to support scheduling of plans and the allocation of resources, and may support process planning and batching. The controller must support task execution and may have any level of intelligence desired.

Since process planning systems, production systems, and controllers are not likely to be capable of fully supporting error recovery in the near future, a mechanism for external intelligent intervention is included in the MSI architecture. Throughout this document, the intelligent agent will be referred to as the guardian.

The MSI architecture requires interfaces for any control entity in the architecture and contains detailed specifications for each of the interfaces. In the following sections, the communications mechanisms for control entities, the interfaces of the control entity, and the physical distribution of a control entity will be discussed in turn.

7.2.3.3.3.1 Communications of Control Entities

All communications between control entities which are direct (see Section 7.2.1.2), are required by the MSI architecture to be via a command and status interface. Such interfaces require communications channels between components. The MSI architecture requires that the communications channels for command and status messages use a point to point, guaranteed message communications paradigm. One communications mechanism that provides such a communications service is the Manufacturing Automation Protocol (MAP) [MAP1], [MAP2], with the Manufacturing Messaging Specification (MMS) application layer [ISO1].

Since message delivery is guaranteed, messages can rely on information conveyed in previous messages. This means that messages need not contain all the information required for a complete picture of the situation, reducing the amount of data which must be transferred with each message. As a consequence of point to point communications,

18. The planner specified here is the production planner architectural unit. However, we will refer to it as planner in the remainder of the discussion.
19. The architecture permits hierarchies of planners without associated controllers. These hierarchies would only be needed for “what if” scenarios and would not need error recovery capabilities.

the communications pairs must be set up when the connections are established, and it is not possible to hide the way in which communicating control entities are physically distributed.

7.2.3.3.3.2 Control Entity Interfaces

A control entity may have as many as five types of interfaces. These direct interfaces are:

(1) a planning interface—which governs interactions of superior and subordinate planners concerning the selection, generation and scheduling of process, production managed and production plans,

(2) a controller interface—which govern interactions of superior and subordinate controllers concerning task execution,

(3) a guardian to planning interface—which governs how an intelligent agent may interact with the planner,

(4) a guardian to controller interface—which governs how an intelligent agent may interact with the controller,

(5) a planner to controller interface—which governs how the planner and the controller may interact in both ordinary and error situations.

A detailed specification of each of these interfaces is found in [Wallace1]. A conceptual view of the potential direct interfaces is shown in Figure 7 [Wallace1].

In an implementation of the architecture, which interfaces must actually be supported is determined by the physical[20] distribution of the control entity. The general rule is that, if the two interacting components are physically distributed, the exposed interface must conform to the corresponding interface specification.

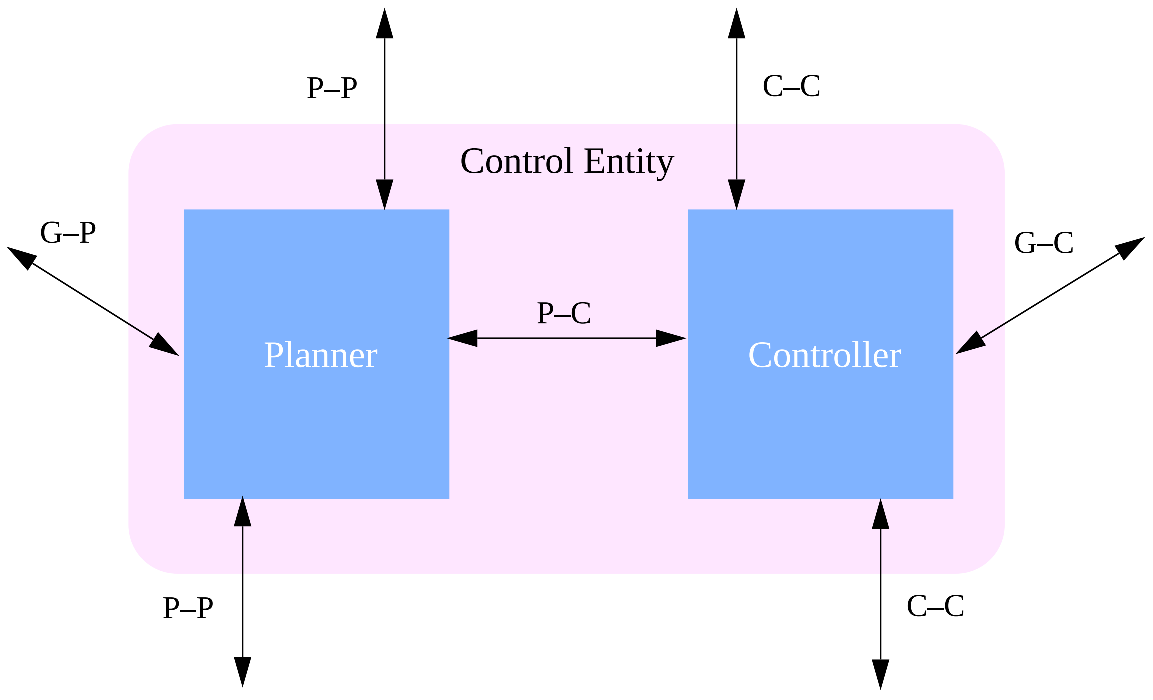
7.2.3.3.3.3 Physical Distribution of Control Entities

Permitting flexibility in the physical distribution of the control entity allows the MSI architecture to accommodate a number of common configurations for planners and controllers. Examples are:

(1) A centralized planning system may be used, provided that each controller has a logically distinct interface to the planning system and that the planning system can plan for each member of the hierarchy. In this case, the internal functioning of the planner is not made public, but the interfaces among controllers and between a controller and its planner are exposed. This configuration is shown in Figure 8.

(2) A distributed planning hierarchy which mirrors the control hierarchy may be used. In this case, the interfaces between planners, between controllers, and between each controller and its planner are public and must conform to the

20. Architectural entities are considered to be physically distributed whenever they consist of two or more (operating system) processes or have portions which execute on physically distinct processors.



<u>Legend</u>
P–Planner
C–Controller
G–Guardian

Figure 7. MSI Control Entity Interfaces

MSI interface specification. Figure 9 shows this configuration[21].

(3) The planner and controller functions may be embedded in a control entity, resulting in a single hierarchy of control entities. In this case, the interfaces between the planner portion of a control entity and the planner portion of both its superior and subordinate control entity are public, and the corresponding controller interfaces are public, but the interface between the planner and the controller of any one control entity remains private. Figure 10 shows this configuration.

In Figures 8, 9 and 10, the control entities are homogeneous (i.e., they all split or combine their component planners and controllers the same way). The architecture also allows the use of heterogeneous control entities.

21. Note that, in figures 8, 9 and 10, the interface between the planner and the controller are shown only in the highlighted areas.

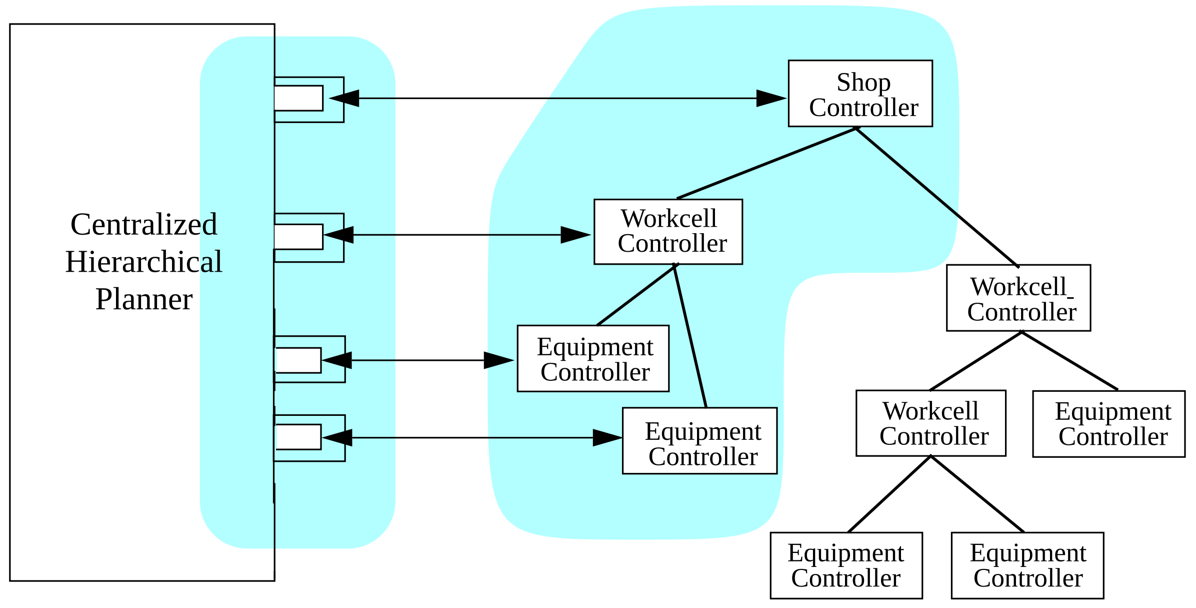


Figure 8. An MSI System with a Centralized Planner

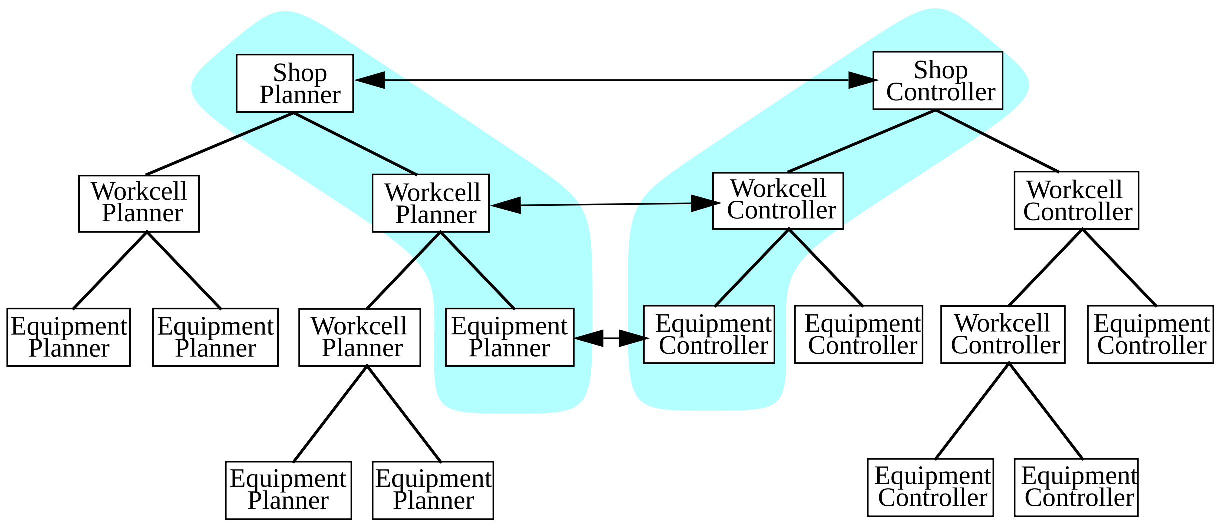


Figure 9. An MSI System with a Planning Hierarchy

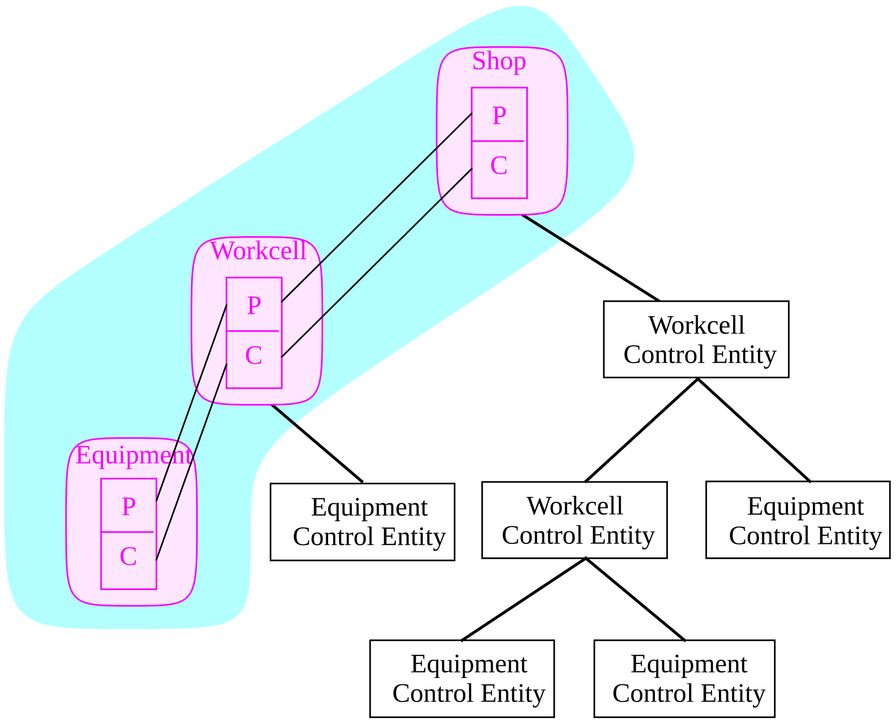


Figure 10. An MSI System with Embedded Planners

7.2.4 Conclusion

The MSI architecture provides an architecture for a shop which manufactures discrete parts that supports information integration for the major shop systems and provides specifications for an integrated production planning and control environment. The MSI architecture can be used with a centralized or a distributed planner, and other combinations of control and planning systems.

The operations of a shop are guided by the schedules generated for its current orders. This mode of operation encourages global optimality for shop production, since local schedules are constrained to accommodate the needs of the factory. Since the shop schedule is important to efficient functioning for the shop, the architecture provides for schedule maintenance via detailed sets of command and status messages for controller and planner interactions. The use of hierarchical control aids in localizing and recovering from scheduling errors.

It is anticipated that the MSI architecture will be useful both for the integration of current shops and in future research. The information models are immediately applicable to aid in shop integration, while the interface specifications provide direction for further research in automated re-planning and control.

7.3 Compatibility Assessment of MSI and RCS Architectures

Two architectures are most compatible when it is possible to take systems which were built according to the specifications of one architecture and use them, with minor modifications, in a system built conforming to the other architecture. Two such architectures are said to be *interoperable*. If this is not possible, it may still be possible to build other architecture(s) for specific purposes using features from both architectures.

We will present a discussion of the interoperability of the two architectures, then a summary of the detailed point-by-point comparison of the MSI and RCS architectures according to the issues identified in the previous sections. The full text of the point-by-point comparison is available in Appendix C. Finally, we present our conclusions concerning the degree of compatibility of the two architectures.

7.3.1 Interoperability Assessment

One way to assess the interoperability of two architectures is to compare how current implementations would be able to function together. Suppose we took an implementation of RCS aimed at machining discrete parts, and we tried to get it to work together with MSI by making an RCS controller subordinate to an MSI controller. We chose this mode of knitting the two control systems together as the one with the highest probability of success, since the MSI architecture specifies a method of including black box controllers in the hierarchy and anticipates this method of inclusion for real-time systems.

We would have to agree on a mechanism for communicating between the RCS controller and the supervising MSI controller. To do this, a front-end for the RCS controller must be made which exhibits the MSI interfaces to the superior controller and planner and communicates to the MSI control entity using a compliant communications system (such as MMS). The front end would also have to communicate with the RCS controller using the communications mechanism which RCS expects.

Such a front end must have considerable functionality. It must be able to translate command and status information between MSI and RCS formats, filling in appropriately if adequate information is not available, or dropping extra information. The front end would have to interpolate the status of the resource, controller, and task well enough to populate the expected parts of the MSI information base. Finally, the front end would have to neutralize the cycle-time difference between the RCS controller and the MSI controller by responding appropriately in real time.

An interface is therefore possible. However, the real question would be what value can be derived from such an arrangement. The real-time capabilities of the RCS controller would clearly be preserved, and the controller would be integrated into the factory’s information and control structure via the MSI architecture. However, the ability of the system to recover from errors depends on the sophistication of the front end and the abilities of the RCS controller. To participate in error recovery, the MSI architecture expects that a controller can perform the following functions:

(1) abort task execution,

(2) halt task execution, and retain information to restart later,

(3) restart task execution from previous point,

(4) halt task execution and discard all information related to the task,

(5) halt task execution and regard the task as complete,

(6) estimate task completion time,

(7) alter task execution based on new parameters (e.g. new start, completion times),

(8) provide a guardian interface for human/intelligent intervention.

The RCS controller specification says nothing about any of these functions, other than providing a human interface. Hence, it is not clear whether an RCS controller would be able to support some of the functionality MSI expects. As an example, suppose a controller cannot provide estimated task completion times. If the controller provides enough information for the front end to accurately estimate the completion time, then recovery from scheduling errors can proceed as usual in the MSI scenario. If there is no reasonably accurate estimate of the completion time, the error recovery scenario will not be activated until the supervising MSI controller notices that the task is late. If the controller can not respond by disposing of the task as the superior controller directs, the error recovery mechanism is compromised.

Similarly, the ability of the RCS controller to use and provide the information required by the MSI information services will determine the degree of integration the RCS controller achieves with the factory.

In summary, one can say that the ability of the RCS and MSI controllers to be coupled in this way depends on features of the RCS controller which are not mandated by the RCS architecture. It is unclear whether these functions could be required of RCS controllers. Conclusions on this matter require in-depth study of the functional capabilities of RCS controllers.

7.3.2 Summary of Point-by-Point Comparison

Given that the two architectures are not a priori interoperable, we proceeded with a point-by-point comparison of the two architectures to determine where the architectures are similar and where they differ. This section presents a summary of the point-by-point comparison.

7.3.2.1 Intended Domain

Many of the differences between the MSI and RCS architectures stem from differences in intended domain. RCS’s primary focus is on uses which require real-time response and in which the domain is unstructured and highly changeable. MSI’s primary focus is on timely (but not real time) control in a highly structured, relatively predictable manufacturing environment.

MSI and RCS both apply to the control of systems, but in addition, MSI has connections to maintenance, orders, and other concepts external to the system.

7.3.2.2 Conformance Criteria

The conformance criteria for the two architectures are fundamentally different. Conformance to the RCS architecture consists of obeying certain basic tenets, which involve both external behavior of the architectural unit and internal decomposition of the architectural units. Most of the tenets are loosely stated and subject to interpretation. Conformance to the MSI architecture consists of displaying architectural units which understand specific information, exhibit certain interfaces and provide specified functionality. The emphasis of MSI upon external characteristics, rather than internal structure makes the inclusion of manufacturing systems not originally designed to work with the architecture possible.

7.3.2.3 Domain Analysis

The MSI architecture provides the results of information analysis for the architecture implementor’s use in the form of information models. RCS provides sample task decompositions and data descriptions for the information required at the servo and primitive hierarchical levels of the architecture. In addition, RCS provides requirements which are aimed at ensuring that compliant controllers have the required real-time capability.

Neither MSI nor RCS requires any formal domain analysis. However, in both cases, some analysis must be done to determine a proper control hierarchy and task decompositions.

7.3.2.4 Controllers and Other Architectural Units

Both MSI and RCS have architectural units from which an implementation of an architecture is built. Primary among these units are the units which perform scheduling functions and units which perform task execution and monitoring functions. In MSI these architectural units are called planners and controllers; in RCS these modules are called planners and executors. A major difference in functionality here is that MSI assumes that plans exist a priori (although they may be adjusted at run-time) whereas in RCS, plans may be constructed at run-time. This difference is a direct result of the different environments for which the architectures are intended.

In addition to the basic task execution functions, the MSI architecture suggests that the eight additional functions listed earlier be available to permit full use of the architecture’s error recovery capabilities.

Beyond the functional requirements, the MSI architecture imposes strenuous interface requirements for the controller/scheduler interfaces. These interface requirements assume certain internal information is available, and that certain sequences of messaging be followed (although in some cases, more than one sequence is permitted). RCS has no similar interface requirements above the implementation level.

In addition to specifying the basic planner/controller architectural units, RCS defines other functions which must be present either as submodules within the controller or as architectural units. These are sensory processing, behavior generation and value judgment. These functions permit the control system to query about the world, evaluate the results of queries, make predictions based upon current information and make decisions based upon values.

The MSI architecture does not explicitly address the functions of behavior generation or value judgment, but expects these to be embedded in the planners, the controllers, or the guardian interface at the discretion of the implementor. MSI does not address sensory processing at all, whereas RCS has made elaborate provisions for it.

Both RCS and MSI permit human intervention into the control system. In MSI this intervention is tightly formalized, in RCS more implementation choice is given.

7.3.2.5 Collections of Controllers

Perhaps one of the greatest differences between MSI and RCS is the way in which “atomic” entities can be combined. MSI permits any combination of scheduler/ controller units or controller/controller and scheduler/scheduler units to be combined into a block which must exhibit appropriate interfaces. As shown in figures 8, 9, and 10, MSI permits controllers and their related planners to form separate hierarchies (in fact planning is not necessarily hierarchical). RCS, on the other hand, requires certain canonical ways of combining the functions together. It is assumed in RCS, for example, that controllers and their related planners are always linked together, so there is always a single control hierarchy.

7.3.2.6 Required Data and Data Handling

Both RCS and MSI have a requirement for information which describes aspects of the environment and internal information about the control system.

A major difference between MSI and RCS is the way in which controllers expect to get information from the environment. MSI expects that there are a great many relevant environmental variables which have to be monitored and so proposes interrupt mechanisms be available to inform the control system of important changes. Versions of RCS vary in this aspect, but some versions require that environment variables be sampled cyclically. The two mechanisms are fundamentally incompatible and a resolution must be reached in order to build an architecture which accommodates both types of controllers.

MSI expresses the data it requires in information models. Implementation of the data in memory, databases, and files is not specified, although an (accessing) scope may be defined. RCS has similar specifications for data at the servo and primitive hierarchical levels, for specific classes of machinery. RCS has only sketchy suggestions for data handling, but often says that the database should be global, and all data which is not local must be globally accessible. MSI does not require that all non-local data be globally accessible.

MSI expects all data handled by the control system to be symbolic or numeric, and says nothing about how such data should originate. RCS expects the control system to need to process sensor data, and makes sensory processing one of the major focuses of the architecture.

7.3.2.7 Communications

MSI requires that the communications mechanism for status and command messages be point to point and guarantee message delivery. The MAP and MMS communications standards provide such services.

RCS does not specify a standard for communications but anticipates that at lower hierarchical levels, fast communications will be required. In most implementations, shared memory or some form of NIST’s Common Memory has been used.

Interestingly, NIST’s Common Memory can be made to provide point to point and guaranteed message delivery, while MMS was designed to provide real-time data delivery for controllers.

7.3.2.8 Task Generation and Execution

MSI and RCS have fundamentally different notions of how tasks are generated. MSI assumes that all tasks originate as the execution of a plan which has been made in advance (although it may be dynamically altered). MSI is designed to explicitly provide for scheduling and allocating resources for tasks. Some versions of RCS assume that the plan is constructed as it is executing, based upon the current states of the controller and the task being worked on. Scheduling and resource allocation are not specifically addressed. Both architectures permit adapting the plan during execution.

Both architectures postulate the existence of work elements, which are units of work for a controller. RCS includes the semantics of these work elements within the architecture, while MSI insists that the semantics of the work elements are in the domain of the implementation.

Both architectures agree that task decomposition and the distribution of tasks among members of the control hierarchy are desirable. RCS includes a number of specific guidelines for task decomposition, MSI concludes that tasks should be decomposed whenever an intermediate degree of coordination is desirable.

7.3.2.9 Error Recovery

MSI makes error recovery one of the main facets of the architecture. MSI defines sequences of command and status messages to recover from scheduling errors, resource failures, and task failures. RCS permits adaptive control but does not address other errors explicitly.

7.3.3 Conclusions and Compatibility Assessment

In this section we distill some key features of an architecture's domain which affect the architecture from the foregoing discussion and present general conclusions regarding the compatibility of RCS and MSI.

7.3.3.1 Key Features of a Domain

Several features of the domain to which a control system is intended to apply profoundly influence the selection of appropriate components for the architecture of the control system.

7.3.3.1.1 Boolean Features

There are three features of a domain which make a qualitative, rather than a quantitative difference in an architecture. There is no middle ground in providing for these features. Either an architecture provides support or it does not. These are:

1. whether resource contention occurs,

If resource contention occurs, the architecture must make provisions for tracking resource status and availability. The architecture must also include mechanisms for different types of allocation and locking for resources. Being able to forecast availability is also very desirable in such domains.

2. whether the control system must be able to respond in hard real time (i.e. a response must be generated within a specific time slice),

If hard real-time response is required, the communications system and data access must be geared to providing service within a specified time slice. As applied to software, the hard real-time requirement means that it must be known in advance what code is to execute and an upper bound must be available on the amount of time in which it executes. In particular, the time which it takes to retrieve relevant data must have an absolute upper bound. This argues strongly for limiting the amount of data which needs to be accessed and disallowing ad hoc queries for information.

3. whether commercial systems whose internals may be unknown must be controlled by the control system.

If inclusion of systems whose internals may be unknown is a driving force in an architecture, then the architecture will avoid specifying the internals of the atomic units which it defines and will concentrate on specifying the external functions, interfaces and dynamic characteristics which the atomic unit displays.

7.3.3.1.2 Continuous Features

In addition, there are a number of features of the domain which make quantitative differences in the architecture. In the case of these features, there are differences in the degree to which the architecture supports it. These are:

1. the variability of the physical environment in which the control system must function,

The more variable the environment, the more important current information about the environment is. In such an environment, accurate updating of such information will be a priority. In some cases, this will mean that sensory processing will be a priority.

2. the degree of structure of the physical environment in which the control system must function,

In an environment which is unstructured, it may be impossible to generate even tentative plans in advance, as the formulation of each step depends on current conditions which cannot be known in advance. Such control systems typically use goals to direct the generation of appropriate plans. For highly structured environments, likely courses of action can be anticipated and encoded in plans in advance of execution.

3. the degree of integration with the organizational environment which the control system must provide,

In some applications within the domain, the control system must be integrated with other systems, such as business systems. In these cases, information interchange and access control must be implemented for the integrated systems.

4. the degree to which the control system must be reliable and fault tolerant,

An architecture must provide for reliability and fault tolerance throughout. The architecture must provide mechanisms for realizing when errors occur and for reporting and resolving them. If an architecture is constructed assuming flawless operation and data, it is difficult, if not impossible, to acquire the data to perform error recovery.

5. the degree to which the dynamic reconfiguration of the control system is supported.

An architecture may need to provide for changing the control hierarchy of a control system. At one end of the flexibility spectrum, this may have to be accomplished by halting the control system, reconfiguring it and restarting. At the other end of the spectrum, the control system could be reconfigured without affecting most executing tasks at all. If a great degree of flexibility is required, the architecture must provide for managing the internal states of controllers and be capable of managing tasks in process.

7.3.3.2 Conclusions

With regard to these key features, MSI provides for the inclusion of commercial systems in an environment which is moderately variable and highly structured (the factory floor). It provides a high degree of integration with other systems and permits dynamic reconfiguration to correct for impaired equipment. The architecture contains methods for handling resource contention. The MSI architecture specifically addresses error recovery for resource problems, scheduling difficulties and task failure. RCS provides for hard real-time control and operation in a range of environments, from the highly structured to the highly unstructured, which may be highly variable. By permitting sensor feedback and intelligent response, RCS provides for reliability and fault tolerance in task execution.

It is clear that the MSI and RCS architectures are not perfectly compatible. However, many of the differences are the result of differing domain requirements. MSI and RCS strengths and weaknesses complement each other.

8 Outline of Proposed Reference Architecture

This section discusses a proposed joint architecture which serves the purposes of (at least) the NIST Factory Automation Systems Division (FASD) and Robot Systems Division (RSD). The section presents generic recommendations for constructing any reference architecture for control, discusses an approach for constructing a joint architecture, and gives the outline for the contents of a specific proposed architecture.

It is planned that this feasibility report be followed by an effort to fully define a joint architecture, an effort involving a larger group of people from RSD and FASD. The architecture outlined here is intended as the starting point for that effort, but that group will be not be bound by the outline given here. Appendix C contains many non-generic recommendations addressed to that group.

8.1 Generic Recommendations

The authors have several recommendations to make which are independent of any specific architecture. These generic recommendations are intended to apply to developing a control system reference architecture in any domain.

8.1.1 Elements of Architectural Definition

All five elements of architectural definition (statement of scope and purpose, domain analyses, architectural specification, methodology for architectural development, and conformance criteria) should be given explicit consideration during the development of an architecture. Architectural specifications (what it is) should be distinguished clearly from methodologies for architectural development (how you build it).

It is the authors’ recommendation that all of these elements of architectural definition be addressed in a balanced fashion. Failure to address all the elements of architectural definition is a common oversight, which leads to an incomplete, inconsistent or ambiguous architecture.

8.1.2 Tiers of Architectural Definition

An architecture should be divided explicitly into tiers of architectural definition and the five elements of architectural definition should be clearly stated at each tier. Where a range of options is intended by the developers of the architecture to be available to implementors or refiners of the architecture, that should be handled explicitly at the appropriate tier of architectural definition, not implicitly by being vague or silent.

8.1.3 Formal Languages

Formal languages should be used where appropriate. Where there is doubt about what is appropriate, lean toward using a formal language; natural language can always be used as an adjunct to a formal language. Formal languages are already universally used at the lowest tier of architectural definition because computers do not deal well with natural language.

An obvious place to use formal language is in the expression of the models resulting from domain analyses. Other places where formal language may be used are in the architectural specification itself and in the conformance criteria.

8.2 Specific Recommendations

For the joint architecture, the authors have made a number of specific choices which are high-level. These choices are outlined in the remainder of this section.

8.2.1 Scope and Purpose

The purpose of the joint architecture is to serve as a guideline for developing a control system which is integrated with its environment and can perform real-time control of system hardware where required. At each more specific tier of architectural definition, the scope and purpose of the architecture is refined.

8.2.2 Tiers of Architectural Definition

The authors suggest that the architecture have four tiers of architectural definition. The lowest tier consists of the implementations of the architecture, and will not be discussed. The three upper tiers may be characterized as follows:

(1) a domain-independent, application-independent tier (tier one),

(2) a domain-specific, application-independent tier (tier two),

(3) a domain-specific, application-specific tier (tier three).

The structure of the tiers of definition of the architecture is shown graphically in Figure 11.

Note that while the first tier of architectural definition gives many of the guidelines necessary to construct a control system, it is necessary to specify domain-specific and application-specific items before constructing an implementation. In practice, many architectures are specified only to this second tier of architectural definition, and the implementor of the architecture must supply all the missing information as (s)he sees fit.

In Section 8.3, we discuss tier one and tier two of the joint architecture, addressing each of the elements of architectural definition at each tier. These tiers are not fully defined. Additional items of architectural specification are desirable at both tiers but have been omitted, not from lack of ideas, but because none of the alternatives for these items is clearly best. Tier three is not addressed in this document, as defining this tier requires the selection of meaningful applications within the domain. At this point in the development of the architecture, it is not clear what these applications should be.

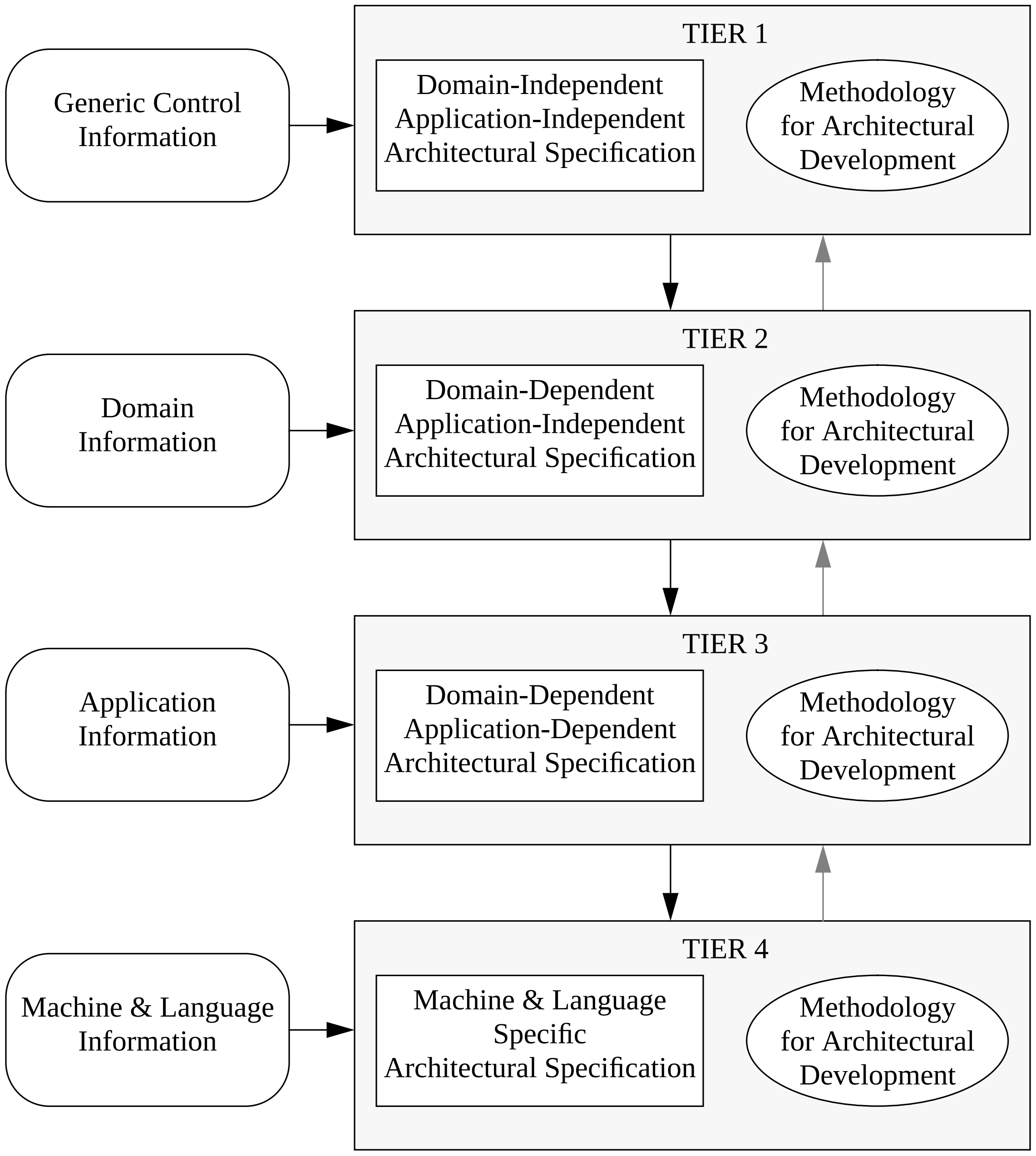

TIER 1
Generic Control Information
Domain-Independent Application-Independent Architectural Specification
Methodology for Architectural Development
TIER 2
Domain Information
Domain-Dependent Application-Independent Architectural Specification
Methodology for Architectural Development
TIER 3
Application Information
Domain-Dependent Application-Dependent Architectural Specification
Methodology for Architectural Development
TIER 4
Machine & Language Information
Machine & Language Specific Architectural Specification
Methodology for Architectural Development

Figure 11. Proposed Architecture - Tiers of Architectural Definition

8.2.3 Methodologies for Architectural Development

For the joint architecture, a methodology for architectural development for transitioning from one tier of architectural definition to another will not be required as part of conformance to tiers 1 and 2 of the architecture. However, when a methodology is formulated, it may be included as advisory and required at tier 3 of the architecture. Specific methods of specializing the architecture for a particular application (the transition from tier 2 to tier 3 and from tier 3 to tier 4) are very likely to be useful and may be required.

8.3 Outline of Proposed Architecture

The following sections give a discussion of the authors’ recommendations for the construction of a joint RSD/FASD architecture at the first and second tiers of architectural definition.

8.3.1 First Tier of Architectural Definition

This tier gives a domain-independent, application-independent architectural specification. As required for each tier, scope and purpose, domain analyses, architectural specification, methodology for architectural development, and conformance criteria are all discussed.

8.3.1.1 Scope and Purpose

The domain definition at this tier is intentionally broad. It is assumed that there is a need for a control system, and that the system being controlled must interact with its environment and react to unpredicted changes in the environment. No further characteristics are assumed.

The control architecture proposed here is intended to give guidelines for the construction of a general control system in this very broad domain. This architecture is specifically intended to be applicable to control systems for factories, robots, autonomous vehicles, construction machines, and mining machines.

8.3.1.2 Domain Analysis

At this tier, it is important to create model(s) for the generation of plans, the transformation of plans to tasks, and the assignment of tasks to resources. At this tier, it is necessary to specify resources, plans and tasks generically, as explicit information about the domain is not present. The authors recommend that an analysis be made of the types of representations appropriate for each of the planning paradigms specified.

It is also recommended that the control function be analyzed to reveal the necessary functions. There is considerable applicable work to draw on in this area.

Finally, it is recommended that hierarchical task decomposition be a mandated strategy and that a methodology of task decomposition (which is advisory only) be adopted.

8.3.1.3 Architectural Specification

The architectural specification at this tier should contain the following:

(1) definition of basic architectural units and functional characterization,

(2) rules for interactions of basic architectural units,

(3) rules for combinations of basic architectural units,

(4) rules for interaction of the control system with humans,

(5) rules for stored data and access,

(6) identification of and explicit models for relevant domain-independent, application-independent information,

(7) selection of paradigms for communications mechanisms.

In the following sections, we address each of these topics in turn.

8.3.1.3.1 Atomic Units

The authors recommend that the basic architectural units of the joint architecture be atomic units of the finest granularity possible and that the internals of architectural units be invisible to the architecture. It is also recommended that atomic units have states, and that these be capable of being externalized to aid in start-up, shutdown, reconfiguration, and error recovery. Associated with each atomic unit is a set of functionality. In addition to the atomic units, the architecture may include a number of architectural units with the functionality of several atomic units combined together. Architectural units with functionality that is a subset of the functionality of an atomic unit should not be permitted.

8.3.1.3.2 Interactions among Architectural Units

The authors recommend that the interface description of each architectural unit specify the information, functionality, and dynamics of the interactions of the unit with other architectural units. The interface should provide for error recovery and safety considerations. Given that this is a domain-independent architectural tier, and all the relevant information may not be known, it is not immediately clear the extent to which this can be done.

8.3.1.3.3 Combinations of Architectural Units

Attention must be paid to the way in which the architectural units, whether atomic or not, can be combined. The architecture should define the interfaces and states of units composed of more than one architectural unit, as well as interfaces and states of atomic units.

Note that, with the above definition of architectural units, it is no longer intuitively clear what a ‘controller’ is. Certainly a controller should contain the task execution function, as this is the most basic control function. However, by specifying how architectural units can be combined, the joint architecture can identify a number of permitted configurations which can be considered controllers in that architecture.

It is recommended that the architecture mandate that the task execution architectural units be arranged in a strict hierarchy at any point in time, and that there be a command and status interface between superiors and subordinates. Modifications to this rule can be made for specific services at more detailed tiers of architectural definition, if required.

Whether the other architectural units can have separate hierarchies, or whether links between them must follow the ones established by the task execution architectural units, must be addressed in the joint architecture. The authors’ recommendation is that the formation of separate hierarchies which mirror that of the task execution hierarchy be permitted. It is tempting to disallow fundamentally different hierarchies, owing to the complication which allowing different hierarchies brings to the architecture, but different hierarchies may be necessary. Sensory processing, in particular, may need to have a hierarchy different from the control hierarchy.

In fact, the joint architecture should consider that a complex control system, which is composed of more than one controller, may have a number of different types of controllers. The authors strongly suggest that the joint architecture consider at least three types of controllers: those capable of real-time control, those without this restriction, and coordinating controllers, whose job it is to provide the coupling between the two types.

Within each type of controller there should probably be conformance classes at tiers 3 and 4, based upon incompatible choices of various sorts. As examples we give the following incompatible pairs: one type of controller may poll for information, while another may be interrupt driven; one type of controller may queue commands while another may disallow queueing; one type of controller may allow multiple simultaneous tasks, and another may not. In the formulation of the joint architecture, the focus should be on whether it is possible to include both types of controllers in the same control system, rather than on which choice is better.

8.3.1.3.4 Human Interaction with the Control System

It is recommended that a set of generic operations be constructed for the interface of human or other intelligence to each of the architectural units. This interface should be extensible at lower tiers of architectural definition. The interface should identify the type of intervention possible with that unit. For example, a human operator should be able to cause task execution to stop in an emergency. At lower tiers of architectural definition, it will be necessary to decide when the interaction should be blocking or non-blocking. The specification of the generic interface operations is not intended to preclude having specialized, task-specific human interfaces at lower tiers of architectural definition. In some cases, the introduction of specialized interfaces will introduce a lack of interoperability.

Human intervention will also be possible for both the information storage and access systems and the communications system. Explicit rules for this interaction seem unnecessary at this tier of architectural definition.

8.3.1.3.5 Stored Data and Access

The architecture should place a minimum of restrictions on where data is stored physically, but should indicate the scope of data (i.e., who has permission to access it). The access of stored data need not use the same communications paradigm as the command and status links.

8.3.1.3.6 Domain-Independent Information

A number of domain-independent models must be formulated at this tier. Primary of these are models for expressing whatever plans are used by the controllers. It is recommended that different controllers in the control hierarchy be permitted to have different types of plans.

Catalogs of work elements should not be included in this tier of architectural definition as they are domain-dependent at least and possibly even application-dependent.

Resources can only be described generally at this tier, but resource classes may be defined. Controllers, planners, and other architectural units should be considered resources and be in a separate class. Models of the characteristics of tasks, controllers, and the relationships among tasks, controllers and plans should be generated.

8.3.1.3.7 Communications Paradigms

In a single architecture, different communications mechanisms may be required for different purposes. For example, one mechanism may be used to exchange command and status information, while another may be used to get information from a shared memory location. Note that even for a single purpose it is not necessary to have a single communications mechanism. For example, it is entirely possible for some communications to take place using a point to point mechanism, while the rest of the communications are through another mechanism, such as a shared bus. In fact, the only time that a communications system would be global is when communications may take place between two parties where it was not known in advance what the two parties would be. Such a situation occurs when synchronizing task execution. The joint architecture should specify a global mechanism as previously described, and should offer the implementor freedom in choosing the communications mechanism between architectural units.

8.3.1.4 Methodology For Architectural Development

A formal methodology for architectural development for going from this tier of architectural definition to the next is not proposed. As stated earlier, it is planned that the transition will be handled by a working group with members from RSD and FASD.

8.3.1.5 Conformance Criteria

At this tier, conformance should consist of complying with the general recommendations, possessing the stated architectural units, supporting the defined interfaces, and exhibiting the required information and process models. Again, note that an architecture at this tier is not implementable without many additional assumptions on the part of the implementor.

8.3.2 Second Tier of Architectural Definition

Given the dependence of an architecture upon the features of its domain, it is clear that defining the domain of the joint architecture will have a profound effect upon the end result. The authors recommend that the domain of the architecture at this tier be discrete parts manufacturing. The authors believe that a good architecture in this domain will use features of existing RSD and FASD architectures. A manufacturing architecture will need hard real-time control, which RSD architectures handle well. Manufacturing architectures also require a high degree of integration with their environment, which MSI supports well.

8.3.2.1 Scope and Purpose

The manufacturing control architecture proposed here is intended to provide guidelines for the construction of a control system which provides for real-time control of manufacturing equipment and integration of the control system in the discrete parts manufacturing environment.

Although the focus of the architecture is on the planning and production phases of the operation of a shop, the scope should include some information from design, management, business, and maintenance to ensure integration with these functions at a later date.

8.3.2.2 Domain Analysis

At this tier of architectural definition, detailed models of the shop are required. Models which should be included are models of shop configuration, shop physical layout, shop networks, orders, resources, schedules, tools, processes, raw materials, and consumable materials.

8.3.2.3 Architectural Specification

At this tier, the architectural specification should include the additional information models specified above. The interfaces of the architectural units should be expanded to include domain-specific information that needs to be exchanged and possibly additional functionality demanded by integration with other systems.

A catalog of work elements should be defined.

For manufacturing, it is definitely necessary to have the higher-level controllers be able to function on a predefined schedule, as a shop is generally required to predict when products will be ready.

It is highly recommended that the joint architecture propose a method for including material handling systems.

8.3.2.4 Methodology For Architectural Development

A methodology for architectural development should be developed at this tier for helping build control systems for specific discrete parts manufacturing uses. Once the methodology is agreed upon, it would be useful to build a CASE tool embodying the methodology and architectural specifications to assist in building realizations of the architecture at the next tier of architectural definition.

8.3.2.5 Conformance Criteria

Conformance criteria at this tier are entirely similar to those of tier 1.

9 Conclusion

9.1 Summary

This report began with the examination of architectural definitions, with an emphasis on the definitions of control architectures. It collected a number of architecture issues which it is critical for a control architecture to address and assessed the MSI and RCS architectures against each issue. Finally, the report discussed the feasibility of building a reference architecture for machine control systems integration and provided guidance to the committee which is to construct the joint architecture.

9.1.1 Architecture Defined

Architecture was defined in Section 1 of this report as giving the design and structure of a system.

The elements of architectural definition were identified in Section 3 as:

(1) statement of scope and purpose,

(2) domain analyses,

(3) architectural specification,

(4) methodology for architectural development, and

(5) conformance criteria.

In that section, also, the notion of tiers of architectural definition was presented and explained.

9.1.2 Issues Discussed

Many architecture issues were discussed in Section 4 and Section 5. Section 4 focused on generic architecture issues, while Section 5 focused on issues specific to control architectures. The identification of the elements of architectural definition, and the statement of the issues served as a framework for writing much of the rest of the report. The authors believe this framework should be useful in the field of control architecture research, independent of any consideration of MSI, RCS, or the proposed joint architecture.

A few additional issues that have arisen in RCS discussions but were too narrowly focused for Section 4 or Section 5 are given in Appendix F.

9.1.3 Architectures Presented

The RCS and MSI architectures were presented in Section 7.1 and Section 7.2, respectively, with many details expanded in Appendix C. Section 6 presented several other architectures, two of which are examined more closely in Appendix E.

9.1.4 RCS and MSI Compared

An assessment of the compatibility of RCS and MSI was made in Section 7.3. The assessment was based on a detailed comparison of the two architectures given in Appendix C. The authors concluded that while MSI and RCS are not perfectly compatible (and two control systems built conforming to the different architectures will certainly not be interoperable), MSI and RCS strengths and weaknesses complement each other, so that an architecture combining their strengths would be feasible and desirable.

9.1.5 New Architecture Outlined

The outline of the contents of a proposed control architecture for (at least) RSD and FASD was given in Section 8. The proposed architecture has four tiers of architectural definition. The contents of tiers one (domain and application independent) and two (focused on the domain of discrete parts manufacturing but independent of a specific application) of the proposed architecture were outlined in moderate detail in that section. The third (application-specific) tier and the bottom tier (a description of a specific implementation) were not discussed in that section.

A manufacturing architecture needs both integration with the environment and real-time control. These two goals work against each other, but hierarchical control is ideally suited to resolving this conflict by permitting upper tiers to handle integration, while lower tiers guarantee hard real-time control. The key to composing an adequate architecture is determining which aspects of the architecture can be local, and which are required to be global. The larger heterogeneous architectural units that satisfy local goals can then be integrated together to compose a complete architecture.

9.2 Observations on Control System Architectures

9.2.1 No “Best” Architecture

The authors conclude that there is no “best” architecture for control systems. The conditions under which control systems must operate vary widely, leading to correspondingly wide variation in the functionality required of control systems. No one control system architecture is best in all cases, or even in a large proportion of cases.

An architecture which includes only the upper tiers of architectural definition will be broadly applicable but will lack technical specifics needed to make implementations. The higher tiers of architecture must be implicitly or explicitly specialized to a domain and then an application before an implementation can be built. Depending on the tier, an architecture may be relevant to a large number of situations.

9.2.2 Research Needed

Several different existing architectures show promise, and the field is open for the development of new architectures. It is desirable, therefore, to continue research on control architectures along many paths.

Hierarchical architectures, such as RCS, MSI, and the new architecture proposed in this report, have merit and should be pursued. Heterarchical architectures and the subsumption architecture have much to recommend them, as well, and seem likely to be superior in at least some situations. Research into these other architectures should also be supported.

In the final analysis, the performance of a control architecture of an automated system is a key determinant of its usefulness. A general shortcoming of current research on control architectures is that aspects of system design are claimed to have been selected to enhance some aspect of system performance, but it is not clearly demonstrated (sometimes not at all) how the design aspect leads to enhanced performance. Furthermore, little research has been done on how to measure the performance of control architectures in various situations and few claims of architectural effectiveness have been established. The authors recommend that more attention be paid to the measurement of the effectiveness of control architectures, so that comparisons of architectures can readily be made.

9.3 RSD/FASD Joint Architecture

The authors stress that completing the proposed architecture will require a great deal of work. The proposed architecture goes beyond RCS and MSI. Completing it requires the specification of many architectural components and their interactions on several tiers of architectural definition and requires consideration of the relationship between corresponding components at different tiers of architectural definition. However, the end result will be an architecture which truly fills both the need for hard real-time control and information integration, each at the appropriate tier.

References

[Albus1] Albus, James S.; Blidberg, D. Richard; *A Control System Architecture for Multiple Autonomous Vehicles*; Proceedings of the Fifth International Symposium on Unmanned, Untethered Submersible Technology; Merrimack, NH; June 1987; 25 pages

[Albus2] Albus, James S.; *RCS: A Reference Model Architecture for Intelligent Control*; IEEE Journal on Computer Architectures for Intelligent Machines; May 1992; pp. 56 - 59

[Albus3] Albus, James S.; Quintero, Richard; Lumia, Ronald; Herman, Martin; Kilmer, Roger D.; *A Reference Model Architecture for ARTICS*; Manufacturing Review; Vol. 4, No. 3; September 1991; pp. 182 - 193

[Albus4] Albus, James S.; *A Theory of Intelligent Systems*; Control and Dynamic Systems; Vol. 45; 1991; pp. 197 - 248

[Albus5] Albus, James S.; McCain, Harry G.; Lumia, Ronald; *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*; NIST Technical Note 1235, 1989 Edition; National Institute of Standards and Technology; April 1989; 76 pages

[Albus6] Albus, James S.; *RCS: A Reference Model Architecture for Intelligent Machine Systems*; unpublished; January 1993; 21 pages

[Albus7] Albus, James S.; Quintero, Richard; Huang, Hui-Min; Roche, Martin; *Mining Automation Real-Time Control System Architecture Standard Reference Model (MASREM)*; NIST Technical Note 1261, Volume 1; National Institute of Standards and Technology; May 1989; 56 pages

[Albus8] Albus, James S.; Lumia, Ronald; Fiala, John; Wavering, Albert J.; *NASREM The NASA/NBS Standard Reference Model for Telerobot Control System Architecture*; proceedings of 20th International Symposium on Industrial Robots; Tokyo, Japan; October 1989; 10 pages

[Albus9] Albus, James S.; Quintero, Richard; Lumia, Ronald; Herman, Martin; Kilmer, Roger O.; Goodwin, Kenneth; *Concept for a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)*; NIST Technical Note 1277; National Institute of Standards and Technology; April 1990; 27 pages

[Albus10] Albus, James S.; Juberts, Maris; Szabo, Sandor; *RCS: A Reference Model Architecture for Intelligent Vehicle and Highway Systems*; Proceedings of the 25th Silver Jubilee Symposium on Automotive Technology and Automation; Florence, Italy; June 1992; 8 pages

[Albus11] Albus, James S.; Quintero, Richard; T*oward a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)*; Proceedings of the IEEE Third International Symposium on Robotics and Manufacturing; Burnaby, B. C.; Canada; July 1990; 8 pages

[Albus12] Albus, James S.; Barbera, Anthony J.; Nagel, R.; *Theory and Practice of Hierarchical Control*; Proceedings of the 23rd IEEE Computer Society International Conference; September 1981

[Auslander1] Auslander, David M.; Tham, Cheng Haam; *Real Time Control Software for Manufacturing Systems*; Control and Dynamic Systems; Vol. 46; pp. 1 - 36

[Barbera1] Barbera, Anthony J.; *An Architecture for a Robot Hierarchical Control System*; NBS Special Publication 500-23; National Bureau of Standards; December 1977

[Barkmeyer1] Barkmeyer, Edward J.; *Distributed Data Interfaces -- The Lessons of IMDAS*; proceedings of CIM Databases ‘88 Conference; Boston, Massachusetts; March 1988

[Barkmeyer2] Barkmeyer, Edward J.; Ray, Steven; Senehi, M. Kate; Wallace, Evan; Wallace, Sarah; *Manufacturing Systems Integration Information Models for Production Management*; National Institute of Standards and Technology Interagency Report, 1992, (forthcoming). (Available from the National Technical Information Service, Springfield, VA 22161.)

[Biemans1] Biemans, Frank P. M.; Vissers, Chris A.; *A Systems Theoretic View of Computer Integrated Manufacturing*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 390 - 410

[Biemans2] Biemans, Frank P. M.; Vissers, Chris A.; *Reference Model for Manufacturing Planning and Control Systems*; Journal of Manufacturing Systems; Vol. 8, No. 1; 1989; pp. 35 - 46

[Bohms1] Bohms, Michel; Tolman, Frits; *RIA: Reference Model for Industrial Automation*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 114 - 132

[Boykin1] Boykin, Robert E., III; *CAM-I CIM Reference Model*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 35 - 41

[Brooks1] Brooks, Rodney A.; *Elephants Don’t Play Chess*; Robotics and Autonomous Systems; No. 6; 1990; pp. 3 - 15

[Catron1] Catron, Bryan; Ray, Steven R.; *ALPS - A Language for Process Specification*; International Journal of Computer Integrated Manufacturing; Vol. 4, No. 2; 1991; pp 105 -113

[Chen1] Chen, D.; Vallespir, B.; Doumeingts, G; *An Integrated CIM Architecture - A Proposal*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 153 - 165

[Davis1] Davis, Wayne; Jones, Albert T.; Saleh, Abdol; *Generic Architecture for Intelligent Control Systems*; Computer Integrated Manufacturing Systems; Vol. 5, No. 2; May 1992; pp. 105 - 113

[Dewhurst1] Dewhurst, Stephen C.; Stark, Kathy T.; *Programming in C++*; Prentice Hall; Englewood Cliffs, New Jersey; 1989

[Dilts1] Dilts, D. M.; Boyd, N. P.; Whorms, H. H.; *The Evolution of Control Architectures for Automated Manufacturing Systems*; Journal of Manufacturing Systems; Vol. 10., No. 1; 1991; pp. 79 - 93

[Dornier1] Dornier GmbH; E*valuation of Standards for Robot Control System Architectures - Final Report Executive Summary*; European Space Agency Contract Report; August 1990; 51 pages

[Dornier2] Dornier GmbH; *Baseline A&R Control Development Methodology Definition Report*; study managed by P. Putz; European Space Agency Contract Report; October 1991; 106 pages plus appendices

[Duffie1] Duffie, Neil A.; Piper, Rex S.; *Nonhierarchical Control of Manufacturing Systems*; Journal of Manufacturing Systems; Vol. 5, No. 2; 1986; pp. 137 - 139

[Duffie2] Duffie, Neil A.; Piper, Rex S.; *Nonhierarchical Control of a Flexible Manufacturing Cell*; Robotics and Computer-Integrated Manufacturing Systems; Vol. 3, No. 2; 1987; pp. 175 - 179

[Duffie3] Duffie, Neil A.; Chitturi, Ramesh; Mou, Jong-I; *Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities*; Journal of Manufacturing Systems; Vol. 7, No. 4; 1988; pp. 315 - 327

[EIA] ANSI/EIA/TIA/232-E *The Interface between Data Terminal Equipment and Data Circuit Terminating Equipment Employing Serial Data Binary Interchange;* July 1991. (Available from Global Engineering Documents, 15 Inverness Way, E. Englewood, Colorado 80112-5704.)

[Feldman1] Feldman, Philip; Huang, Hiu-Min; Hira, Ron; *A Submarine Maneuvering System Demonstration Using a Generic Real-Time Control System (RCS) Reference Model*; proceedings of Summer Computer Simulation Conference; Reno, Nevada; July 1992; 5 pages

[Fiala1] Fiala, John; *memorandum on Jeff Becker's evaluation of NASREM*; unpublished; May 1990; 8 pages

[Fiala2] Fiala, John; *Note on NASREM Implementation*; NISTIR 89-4215; National Institute of Standards and Technology; December 1989; 17 pages

[Fiala3] Fiala, John; Lumia, Ronald; *An Approach to Telerobot Computing Architecture*; NISTIR 4357; National Institute of Standards and Technology; June 1990; 21 pages

[Fiala4] Fiala, John; *Manipulator Servo Level Task Decomposition*; NIST Technical Note 1255; National Institute of Standards and Technology; October 1988; 37 pages

[Fowler1] Fowler, James; draft STEP Part 22; *Step Data Access Interface Specification*; January 1992

[Griesmeyer1] Griesmeyer, J. Michael; et al; *Generic Intelligent System Control (GISC)*; Version 1.0; Sandia National Laboratories report SAND92-2159; October 1992.

[Griesmeyer2] Griesmeyer, J. Michael; *General Interface for Supervisor and Subsystem (GENISAS)*; Version 1.0; Sandia National Laboratories; October 1992

[Harhalakis1] Harhalakis, G.; Semakula, M. E.; Johri, A.; *Architecture of a Facility Level CIM System*; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 430 - 445

[Hatvany1] Hatvany, J.; *Intelligence and Cooperation in Heterarchic Manufacturing Systems*; Robotics and Computer-Integrated Manufacturing; Vol. 2, No. 2; 1985; pp. 101 - 104

[Herman1] Herman, Martin; Albus, James S.; Hong, Tsai-Hong; *Intelligent Control for Multiple Autonomous Undersea Vehicles*; Neural Networks for Control; MIT Press; 1990; 19 pages

[Herman2] Herman, Martin; Albus, James S.; Hong, Tsai-Hong; *Real-Time Vision for Autonomous and Teleoperated Control of Unmanned Vehicles*; Active Perception and Robot Vision; Springer-Verlag; 1991; 36 pages

[Herman3] Herman, Martin; Albus, James S.; *Real-time Hierarchical Planning for Multiple Mobile Robots*; Proceedings of DARPA Knowledge-Based Planning Workshop; Austin, Texas; December 1987; pp. 22-1 to 22-10

[Hopp1] Hopp, T. *Proceedings of the Manufacturing Data Preparation (MDP) Workshop*, unpublished report, June 1988. (Available from the Factory Automation Systems Division, National Institute of Standards and Technology, Building 220, Room A127, Gaithersburg, MD 20899.)

[Horst1] Horst, John Albert; Barbera, Anthony J.; *An Intelligent Control System for a Cutting Operation of a Continuous Mining Machine*; NISTIR 5142; National Institute of Standards and Technology; February 1993; 49 pages

[Huang1] Huang, Hui-Min; Quintero, Richard; Albus, James S.; *A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real-Time Control for Coal Mining Operations*; Advances in Control & Dynamic Systems; Academic Press; Vol. 46; July 1991; pp. 173-254

[ICAM1] ICAM; *Information Modelling Manual IDEF Extended (IDEF1X)*; ICAM Project Report (Priority 6201); Air Force Systems Command; Wright-Patterson Air Force Base, OH; December 1985

[ISO1] ISO; *International Standard ISO/TR 10314-1 Industrial automation - Shop floor production - Part 1: Reference model for standardization and a methodology for identification of requirements*; 1990; 23 pages

[ISO2] ISO 9506, *Industrial Automation Systems Manufacturing Message Specification, Part 1: Service Definition*. (Available from the International Organization for Standardization, Geneva, Switzerland.)

[ISO3]	ISO 10303, *Product Data Representation and Exchange, Part 1: Overview and Fundamental Principles,* ISO TC184/SC4/Editing: Document N11 (Working Draft) (Available from the IGES/PDES/STEP Administration Office, National Institute of Standards and Technology, Building 220, Room A127, Gaithersburg, MD 20899.)
[ISO4]	ISO TC184/SC5/WG1: Document N-282 Version 3.0, *Framework for Enterprise Modelling*; May 1993 (Working Draft) (Available from National Electrical Manufacturers Association, 2101 L Street, N.W. Washington, D.C. 20037.)
[Jackson1]	Jackson, Richard H. F.; Jones, Albert T.; *An Architecture for Decision Making in the Factory of the Future*; Interfaces; Vol. 16, No. 6; November-December 1986; pp. 15 - 28
[Jayaraman1]	Jayaraman, Sundaresan; *Design and Development of an Architecture for Computer-Integrated Manufacturing in the Apparel Industry Part I: Basic Concepts and Methodology Selection*; Textile Research Journal; Vol. 60, No. 5; May 1990; pp. 247 - 254
[Johnson1]	Johnson, Matt; Kirkley, James R.; *Towards a Distributed Control Architecture for CIM*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 166 - 176
[Johnson2]	Johnson, T. L.; Baker, Albert D.; *Trends in Shop Floor Control: Modularity, Hierarchy, and Decentralization*; IEEE; 1990; pp. 45 - 51
[Jones1]	Jones, Albert T.; Barkmeyer, Edward J.; Davis, Wayne; *Issues in the Design and Implementation of a System Architecture for Computer Integrated Manufacturing*; International Journal of Computer Integrated Manufacturing; Vol. 2, No. 2; 1989; pp. 65 - 76
[Jones2]	Jones, Albert T.; McLean, Charles R.; *A Proposed Hierarchical Control Model for Automated Manufacturing Systems*; Journal of Manufacturing Systems; Vol. 5, No. 1; 1986; pp 15 -25
[Jones3]	Jones, Albert T.; Barkmeyer, Edward J.; *Toward a Global Architecture for Computer Integrated Manufacturing*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 1 - 20.
[Jones4]	Jones, Albert T.; Saleh, Abdol; *A Multi-level/Multi-layer architecture for Intelligent Shopfloor Control*; International Journal of Computer Integrated Manufacturing; Vol. 3, No. 1; 1992; pp. 60 - 70
[Jones5]	Jones, Albert T.; McLean, Charles R.; *A Production Control Module for the AMRF*; Proceedings of the 1985 ASME Computers in Engineering Conference; August 1985

[Jorysz1]	Jorysz, H. R.; Vernadat, F. B.; *CIM-OSA Part 1: Total Enterprise Modelling and Function-View*; International Journal of Computer Integrated Manufacturing; Vol. 3, No. 3 and 4; 1990; pp. 144 - 156
[Jorysz2]	Jorysz, H. R.; Vernadat, F. B.; *CIM-OSA Part 2: Information View*; International Journal of Computer Integrated Manufacturing; Vol. 3, No. 3 and 4; 1990; pp. 157 - 167
[Joshi1]	Joshi, S.; Wysk R.; Jones, Albert T.; *A Scaleable Architecture for CIM Shop Floor Control*; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 21 - 34
[Judd1]	Judd, Robert P.; et al; *Manufacturing System Design Methodology: Execute the Specification*; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 133 - 152
[Jun1]	Jun, Jau-Shi; *The Vertical Machining Workstation Systems*; NISTIR 88-3890; National Institute of Standards and Technology; November 1988; 65 pages
[Jung1]	Jung, David W.; *Implementation of the RAMP Architecture at an Established Site*; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 266 - 286
[Klittich1]	Klittich, Manfred; *CIM-OSA Part 3: CIM-OSA Integrating Infrastructure the Operational Basis for Integrating Manufacturing Systems*; International Journal of Computer Integrated Manufacturing; Vol. 3, No. 3 and 4; 1990; pp. 168 - 180
[Klittich2]	Klittich, Manfred; *From CIM to CIM-OSA a Step Ahead in System Integration*; publication unknown
[Kramer1]	Kramer, Thomas R.; *EXPRESS Schema for NASREM*; unpublished; National Institute of Standards and Technology; March 1992; 16 pages
[Leake1]	Leake, Stephen A.; Kilmer, Roger D.; *The NBS Real-time Control System User's Reference Manual*; NIST Technical Note 1250; National Institute of Standards and Technology; October 1988
[Libes1]	Libes, Don; *NIST Network Common Memory User Manual*; NISTIR 90-4233; National Institute of Standards and Technology; February 1990
[Libes2]	Libes, Don; Ressler, Sandy; *Life with UNIX: A Guide for Everyone*; Prentice Hall; Englewood, New Jersey; 1989; pp. 291-293
[Litt1]	Litt, Eric E.; *The Development of a CIM Architecture for the RAMP Program*; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 251 - 265
[Lumia1]	Lumia, Ronald; Fiala, John; Wavering, Albert J.; *The NASREM Robot Control System Standard*; Robotics and Computer-Integrated Manufacturing; Vol. 6, No. 4; 1989; pp. 303 - 308

[Lumia2] Lumia, Ronald; *NASREM as a Functional Architecture for the Design of Robot Control Systems*; Proceedings of NATO's Advanced Research Workshop on Sensors; Maratea, Italy; August 1989; 11 pages

[Lumia3] Lumia, Ronald; Fiala, John; Wavering, Albert J.; *NASREM: Robot Control System and Testbed*; proceedings of the Second International Symposium on Robotics and Manufacturing Research, Education, and Applications; Albuquerque, New Mexico; November 1988; 14 pages

[Maimon1] Maimon, Oded Z.; *Real-time Operational Control of Flexible Manufacturing Systems*; Journal of Manufacturing Systems; Vol. 6, No. 2; 1987; pp. 125 - 136

[MAP1] *Manufacturing Automation Protocol Version 3.0*; August 1988. (Available from North American MAP/TOP Users Group, ITRC, P.O. Box 1157, Ann Arbor, MI 48106.)

[MAP2] *Technical and Office Protocols Version 3.0*; August 1988. (Available from North American MAP/TOP Users Group, ITRC, P.O. Box 1157, Ann Arbor, MI 48106.)

[Martin1] Martin Marietta; *System*; Draft Volume I of Next Generation Workstation/ Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS); Document No. NGC-0001-13-000-SYS; March 1992; 185 pages plus appendices

[Martin2] Martin Marietta; *NGC Data*; Draft Volume II of Next Generation Workstation/ Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS); Document No. NGC-0001-16-000-NGC DATA; March 1992; 243 pages plus appendices

[Martin3] Martin Marietta; *Workstation Management Standardized Application (WMSA)*; Draft Volume III of Next Generation Workstation/Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS); Document No. NGC-0001-14-000-WMSA; March 1992; 36 pages plus appendices

[Martin4] Martin Marietta; *Workstation Planning Standardized Application*; Draft Volume IV of Next Generation Workstation/Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS); Document No. NGC-0001-14-000-WPSA; March 1992; 89 pages plus appendices

[Martin5] Martin Marietta; *Controls Standardized Application (CSA)*; Draft Volume V of Next Generation Workstation/Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS); Document No. NGC-0001-14-000-CSA; March 1992; 73 pages plus appendices

[Martin6] Martin Marietta; *Sensor/Effector Standardized Application (SESA)*; Draft Volume VI of Next Generation Workstation/Machine Controller (NGC) Specification for an Open System Architecture Standard (SOSAS); Document No. NGC-0001-14-000-SESA; March 1992; 20 pages plus appendices

[Mayer1] Mayer, R.J., ed.; *IDEF0 Function Modeling: A Reconstruction of the Original Air Force Report*; Knowledge Based Systems, Inc.; College Station, TX; 1990. (Available from Knowledge Based Systems, Inc. 1408 University Drive East, College Station, TX 77840-2335).

[Mayer2] Mayer, R.J., ed.; *IDEF1 Information Modeling: A Reconstruction of the Original Air Force Report*; Knowledge Based Systems, Inc.; College Station, TX; 1990 (Available from Knowledge Based Systems, Inc. 1408 University Drive East, College Station, TX 77840-2335).

[Mayer3] Mayer, R.J.; Painter, M.; *IDEF Family of Methods*; Technical Report; Knowledge Based Systems, Inc.; College Station, TX; 1991 (Available from Knowledge Based Systems, Inc. 1408 University Drive East, College Station, TX 77840-2335).

[McLean1] McLean, C. R.; *Interface Concepts for Plug-Compatible Production Management Systems*; Proceedings of the IFIP WG5.7 Working Conference on Information Flow in Automated Manufacturing Systems; Gaithersburg, MD; August 1987. Reprinted in Computers in Industry; Vol. 9; pp. 307-318; 1987.

[Menzel1] Menzel, C.P.; Mayer, R.J.; Edwards, D.; *IDEF3 Process Descriptions and Their Semantics*; Kuziak, A., and Dagli, C., editors; Knowledge Base Systems in Designing and Manufacturing; Chapman Publishing; 1990.

[Michaloski1] Michaloski, John; *Handbook for Real-Time Intelligent System Design* Version 2.0; draft prepared for RSD; National Institute of Standards and Technology; 1992

[Michaloski2] Michaloski, John; Wheatley, Thomas; *System Factors in Real-Time Hierarchical Control*; NISTIR 4386; National Institute of Standards and Technology; August 1990; 17 pages

[Michaloski3] Michaloski, John; Wheatley, Thomas; *Design Principles for a Real-Time Robot Control System*; proceedings of IEEE International Conference on Systems Engineering; August 1990; Pittsburgh, PA; 4 pages

[Min1] Min, Moonkee (Daniel); *A Survey of the Literature on Computer Integrated Manufacturing Architectures*; draft prepared for the MSI project; National Institute of Standards and Technology; 1991; 21 pages

[Murphy1] Murphy, Karl N.; *Real-Time Control System Modifications for a Deburring Robot User Reference Manual*; NBSIR 88-3832; National Institute of Standards and Technology; August 1988; 45 pages

[Norcross1] Norcross, Richard J.; *A Control Structure for Multi-Tasking Workstations*; proceedings of the 1988 IEEE International Conference on Robotics and Automation; Philadelphia, PA; April 1988; pp. 1133 - 1135

[Panse1] Panse, Richard; *CIM-OSA - A Vendor Independent CIM Architecture*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 177 - 196

[Plonsky1] Plonsky, Leo; *Manufacturing Systems Strategic Plan*; Department of Defense Manufacturing Systems Committee; March 1993; 75 pages plus appendices

[Quintero1] Quintero, Richard; *RCS Methodology Issues Log*; unpublished; National Institute of Standards and Technology; May 1991; 16 pages

[Quintero2] Quintero, Richard; *The RCS Methodology: A Task Oriented Method for Developing Intelligent Real-Time Control Systems Software*; National Institute of Standards and Technology; unpublished working draft; May 1991

[Quintero3] Quintero, Richard; Barbera, Anthony J.; *A Real-Time Control System Methodology for Developing Intelligent Control Systems*; NISTIR 4936; National Institute of Standards and Technology; October 1992; 62 pages plus appendices

[Ray1] Ray, Steven R.; Wallace, Sarah; *A Production Management Information Model for Discrete Manufacturing*; submitted for publication to Production Planning and Control; September 1992; 27 pages

[Ross1] Ross, Douglas T.; Structured Analysis (SA), A Language for Communicating Ideas; *IEEE Transactions on Software Engineering*; Volume SE-3, No. 1 January 1977; pp. 16-34.

[Rybczynski] Rybczynski, S.; et. al.; *AMRF Network Communications*; NISTIR 88-3816; National Institute of Standards and Technology; June 1988

[Senehi1] Senehi, M. K.; Wallace, Sarah; Barkmeyer, Edward J.; Ray, Steven R.; Wallace, Evan K.; *Control Entity Interface Document*; NISTIR 4626; National Institute of Standards and Technology; June 1991; 39 pages

[Senehi2] Senehi, M. K.; Barkmeyer, Edward J.; Luce, Mark E.; Ray, Steven R.; Wallace, Evan K.; Wallace, Sarah; *Manufacturing Systems Integration Initial Architecture Document*; NISTIR 4682; National Institute of Standards and Technology; September 1991; 62 pages

[Senehi3] Senehi, M.K.; Wallace, Sarah; Luce, Mark E.; *An Architecture for Manufacturing Systems Integration*; Proceedings of ASME Manufacturing International Conference; Dallas, Texas; April 1992; 15 pages

[Shaw1] Shaw, Michael J.; *Dynamic Scheduling in Cellular Manufacturing Systems: A Framework for Networked Decision Making*; Journal of Manufacturing Systems; Vol. 7, No. 2; 1988; pp. 83 - 94

[Shorter1] Shorter, D. N.; *Progress Towards Standards for CIM Architectural Frameworks*; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 216 - 231

[Simpson1] Simpson, J.; Hocken R.; Albus, J.; *The Automated Manufacturing Research Facility*; Journal of Manufacturing Systems; Vol. 1; Number 1, 1982

[Skevington1] Skevington, Craig; Hsu, Cheng; *Manufacturing Architecture for Integrated Systems*; Robots and Computer Integrated Manufacturing; Vol. 4, No. 3/4; 1988; pp. 619 - 623

[Spector1] Spector, Lee; *Supervenience in Dynamic-World Planning*; University of Maryland Computer Science Technical Report CS-TR-2899, UMIACS-TR-92-55; May 1992; 159 pages

[Spiby1] Spiby, Philip; draft STEP Part 11 *EXPRESS Language Reference Manual*; April 1991; 160 pages

[Stallings1] Stallings, William; *Handbook of Computer-Communications Standards*; V. 2 Local Network Standards; Macmillan Publishing Company; 1987; p. 144.

[Szabo1] Szabo, Sandor; *memo on Evaluation of current RCS Methodology*; National Institute of Standards and Technology; October 1992; 3 pages

[Szabo2] Szabo, Sandor; Murphy, Karl N.; Scott, Harry A.; Legowik, Steven A.; Bostelman, Roger V.; *Control System Architecture for Unmanned Land Vehicles*; in proceedings of Association for Unmanned Vehicle Systems; Huntsville, Alabama; June 1992; 8 pages

[Szabo3] Szabo, Sandor; Scott, Harry A.; Kilmer, Roger D.; *Control System Architecture for Unmanned Ground Vehicles*; in proceedings of Association for Unmanned Vehicle Systems; Dayton, OH; July-August 1990; pp. 258 - 266

[Szabo4] Szabo, Sandor; Scott, Harry A.; Murphy, Karl N.; Legowik, Steven A.; *Control System Architecture for a Remotely Operated Land Vehicle*; proceedings of 5th IEEE International Symposium on Intelligent Control; Philadelphia, PA; September 1990; 8 pages

[Szabo5] Szabo, Sandor; Scott, Harry A.; Kilmer, Roger D.; *Control System Architecture for the TEAM Program*; proceedings of the Second International Symposium on Robotics and Manufacturing Research, Education, and Applications; Albuquerque, New Mexico; November 1988; 9 pages

[Tanenbaum] Tanenbaum, Andrew S.; Computer Networks-Second Edition; Prentice Hall; Englewood Cliffs, New Jersey; 1988.

[Ting1] Ting, James J.; *A Cooperative Shop-Floor Control Model for Computer-Integrated Manufacturing*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 446 - 465.

[Upton1] Upton, D. M.; Barash, M. M.; Matheson, A. M.; *Architectures and Auctions in Manufacturing*; Journal of Computer Integrated Manufacturing; Vol. 4, No. 1; 1991; pp. 23 - 33

[Vamos1] Vamos, Tibor; *Cooperative Systems Based on Non-Cooperative People*; Control Systems Magazine; August 1983; pp. 9 - 14

[VanHaren1] VanHaren, Clyde R.; Williams, Theodore J.; *A Reference Model for Computer Integrated Manufacturing from the View Point of Industrial Automation*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 42 - 62

[Verheijen1] Verheijen, G.M.A.; VanBekkum, J.; *NIAM: An Information Analysis Method*; Information Systems Design Methodologies: A Comparative Review; North-Holland; 1982; pp. 538-589

[Wallace1] Wallace, Sarah; Senehi, M. K.; Barkmeyer, Edward J.; Ray, Steven R.; Wallace, Evan K.; *Manufacturing Systems Integration Control Entity Interface Specification*; NISTIR 5272; National Institute of Standards and Technology; September 1993; 114 pages

[Wavering1] Wavering, Albert J.; *Manipulator Primitive Level Task Decomposition*; NIST Technical Note 1256; National Institute of Standards and Technology; October 1988; 49 pages

[Wavering2] Wavering, Albert J.; Fiala, John C.; *The Real-Time Control System of the Horizontal Workstation Robot*; NBSIR 88-3692; National Institute of Standards and Technology; December 1987; 127 pages

[Wendorf1] Wendorf, Roli G.; Biemans, Frank P. M.; *Structured Development of a Generic Workstation Controller*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 411 - 429

[Weston1] Weston, R. H.; et al; *Highly Extendable CIM Systems Based on an Integration Platform*; Proceedings of CIMCON ‘90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 80 - 94

Appendix A - Glossary and Acronyms

A.1 Glossary

analysis
: an examination of the components of some complex system and how they relate to one another.

application
: a subset of a domain for an architecture.

architectural specification
: a prescription of what the pieces (software, languages, execution models, controller models, communications models, computer hardware, machinery, etc.) of an architecture are, how they are connected (logically and physically), and how they interact.

architectural unit
: an atomic unit or molecular unit that is recognized by an architecture.

architecture
: the design and structure of a system. Typically, an architecture consists of a set of components, together with specifications of how the components work together within the system, and how they may interact with the environment outside of the system.

aspect
: a cross-cutting view of an architecture from some specialized viewpoint, such as information, communications, or control flow.

atomic unit
: an architectural unit of an architecture which the architecture does not break down further into simpler architectural units.

black box
: a subsystem which is described only in terms of its inputs, outputs, and functionality, but whose internal architecture is unspecified.

broadcast communications
: a communications system style in which a communications entity sends messages out over the communications medium without addressing the message to one or more specific communications entities.

centralized control
: a control method in which single controller (usually running on a single computer) controls everything directly.

command
: an instruction from a superior controller to a subordinate controller (or from a client controller to a server controller) to carry out a task.

command and status exchange

an exchange of messages between a superior (or client) controller and a subordinate (or server) controller in which the superior tells the subordinate what is to be done by sending a command and the subordinate sends a status message back.

command-and-status protocol

a specification of the messages which two interacting controllers exchange and the rules by which they exchange them. There are two types of messages: those which are commands and those which give the status of the execution of the commands.

component

an implementation of an architectural unit of an architecture.

conformance class

a set of architectures (or implementations) distinguished by a combination of features at a tier of architectural definition. Different conformance classes may have different and incompatible choices of features or may correspond to different degrees of conformance to an architectural requirement.

conceptual data model

a description of a set of information, always giving relationships among the members of the set, often including the data type of the members of the set, and often including some of the semantic content of the information.

conformance criteria

criteria which specify how an architectural unit at one tier of an architecture conforms to the architectural specifications of a higher tier, or how a process for building part of an architecture conforms to the development methodology given by the architecture for building that part.

conformance test

a procedure that determines if conformance criteria have been met.

controller

the agent which directs the performance of or performs specific tasks.

cyclic development

development (of a control system, controller architecture, etc.), by doing an initial implementation, assessing the finished product, and using the results of the assessment as feedback for refining the system. The assessment and refinement may be repeated several times.

domain

the class of situations for which an architecture is intended to be used.

domain analyses

analyses of the target domain of an architecture. Commonly used forms of domain analysis are functional analysis, information analysis, and dynamic analysis.

dynamic analysis
an analysis of the characteristics of the functions and information in a domain which vary over time during control system operation. It provides qualitative and quantitative information about the sequence, duration, and frequency of change in the functions and information of the domain.

dynamic aspects
aspects of a control system which describe how the information and functioning of the system vary over time.

dynamic reconfiguration
modifying the control hierarchy of a hierarchical control system while the system is working.

element of architectural definition
a part of the definition of an architecture. The elements of architectural definition are: statement of scope and purpose, domain analyses, architectural specification, methodology for architectural development, and conformance criteria.

execution model
a logical view of how the execution of a control system is carried out.

functional analysis
an analysis of all the activities within the scope of an architecture which a conforming control system is supposed to be able to perform.

functional aspects
aspects of a control system architecture which describe what a system conforming to the architecture does.

goal
a state of affairs intended to be brought about. Goals are such items as manufacturing a part, moving a robot arm to a specific place, or navigating a vehicle from one point to another.

granularity (of a tier of architectural definition of an architecture)
the size of the atomic units which the architectural specification of that tier addresses.

hard real-time (control system)
a control system in which a response must be generated within a fixed time interval.

heterarchical control architecture
a type of control system architecture in which each controller has no superior and no subordinates, and controllers interact by issuing requests for bids, making bids, and entering into contracts to do work.

hierarchical control architecture
a type of control system architecture in which controllers are arranged in a hierarchy, and controllers interact through a command-and-status protocol.

implementation
the realization of an architecture in hardware and software.

information analysis
an analysis of all the information within the scope of an architecture needed for a conforming control system to function properly.

information aspects
aspects of a control system architecture which describe the information required for the operation of a system conforming to the architecture.

information modeling language
a formal language intended to be useful for representing information. Examples are EXPRESS, NIAM, and IDEF1X.

interoperable (architectures)
two architectures such that a control system built according to the specifications of one architecture can be used (possibly with minor modifications) in a control system built conforming to the other architecture.

life cycle
the stages in the life of a system or product.

methodology for architectural development
a set of procedures for applying an architecture.

molecular unit
a combination of atomic units or smaller molecular units recognized by an architecture.

multicast communications
a communications system style in which a communications entity can send a given message to several other known communications entities.

non-persistent data
data which is stored temporarily and which is lost when the system containing it is reset.

operational mode
a style of operation of a controller or control system. Operational modes might include, for example: debugging (enabled vs. disabled), autonomy (automatic, shared control, or manual), logging (enabled vs. disabled), single stepping (on vs. off).

operational state
the fitness for operating of a controller or control system. Operational states might include, for example: down, idle, ready, active.

organizational extent (of an architecture)
the set of related activities of an organization covered by the architecture.

persistent data
data stored on a permanent medium such as files or databases.

plan

a scheme developed to accomplish a specific goal.

planner

an agent which generates or selects plans to accomplish one or more goals.

planning

the activity of making plans. The plans may be process plans, production plans, schedules, etc.

point to point communications

a communications system style in which a communications entity can send a given message only to one other communications entity, i.e. communications occurs between pairs of communications entities.

process

The term is commonly used in several senses. See discussion in Section 4.4.3.3.

process plan

a specification of the activities (possibly including alternatives) necessary to reach some goal. A process plan serves as a template, or recipe. Process plans may be distinguished from production plans and schedules, both of which are derived from process plans.

real-time

the condition that a system must keep pace with events in the environment.

reference architecture

a generic architecture for a specific domain.

resource allocation

assigning resources (temporarily or permanently) for some specific purpose.

resource definition

a description of a resource, usually given in a formal information modeling language.

scheduler

an agent which performs scheduling.

scheduling

the assignment of specific resources and times.

scope

see statement of scope.

soft real-time

requiring real-time response, but not within a specific time interval.

statement of purpose

a statement that identifies what the objectives of an architecture are within the given scope.

statement of scope
: a description of the range of areas to which an architecture is intended to be applied.

step (of a plan)
: the basic unit of subdivision of the procedures section of a plan, usually specifying that a single activity (single at some conceptual level) be carried out (drill a hole, deliver a tray, machine a lot of parts, etc.).

submodule
: an internal unit of an atomic unit of an architecture.

synchrony
: a fixed relation in time between the execution cycles of two controllers.

task
: a piece of work which achieves a specific goal - actual work, not a representation of work.

tier of architectural definition
: a grouping of architectural units of similar concreteness.

work element
: a generic representation of a type of work, such as moving in a straight line from one point to another, opening a gripper, or drilling a hole.

A.2 Acronyms

AcSL	Activity Scripting Language
ALPS	A Language for Process Specification
AMICE	European CIM Architecture
AMRF	Automated Manufacturing Research Facility
ARTICS	Architecture for Real-Time Intelligent Control Systems
BG	Behavior Generation
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CAM-I	Computer-Aided Manufacturing - International
CAPP	Computer-Aided Process Planning
CASE	Computer-Aided Software Engineering
CEI	Control Entity Interface
CIM	Computer-Integrated Manufacturing
CIM-OSA	Computer Integrated Manufacturing - Open Systems Architecture
DOE	Department Of Energy
DOS	Disk Operating System
ESPRIT	European Strategic Program for Research and Development in Information Technology
EX	EXecution
FASD	Factory Automation Systems Division
FIFO	First In First Out
GENISAS	GENeral Interface for Supervisor And Subsystem
GISC	Generic Intelligent System Control
IBM	International Business Machines
ICAM	Integrated Computer-Aided Manufacturing
IDEF0	ICAM DEFinition 0 (activity modeling technique)
IDEF1X	ICAM DEFinition 1 eXtended (data modeling technique)
IDEF2	ICAM DEFinition 2 (simulation modeling technique)
IDEF3	ICAM DEFinition 3 (process description capture)
IMDAS	Integrated Manufacturing Data Administration System
I/O	Input/Output
ISO	International Organization for Standardization
IVHS	Intelligent Vehicle-Highway System
JA	Job Assignment
LIFO	Last In First Out
MAP	Manufacturing Automation Protocol
MEL	Manufacturing Engineering Laboratory
MMS	Manufacturing Messaging Specification
MSI	Manufacturing Systems Integration
NASA	National Aeronautics and Space Administration
NASREM	NASA/NBS Standard REference Model for Telerobot Control System Architecture
NBS	National Bureau of Standards
NC	Numerically Controlled (or Numerical Control)

NCL	Neutral Command Language
NFS	Network File System
NGC	Next Generation Controller
NIAM	Nijssen Information Analysis Methodology
NIST	National Institute of Standards and Technology
OLPS	Off-Line Programming System
OSI	Open System Interconnection
PC	Personal Computer
PL	PLanning
RAMP	Rapid Acquisition of Manufactured Parts
RCS	Real-time Control System
RSD	Robot Systems Division
SADT	Structured Analysis and Design Technique
SDAI	STEP Data Access Interface
SOSAS	Specification for an Open System Architecture Standard
SP	Sensory Processing
STEP	STandard for the Exchange of Product Model Data
TBD	To Be Done
TD	Task Decomposition
USAF	United States Air Force
VJ	Value Judgment
WM	World Modeling

Appendix B - Summary of General Architecture and Control Issues

The following Appendix lists for quick reference the general architecture and control issues discussed in the document without any accompanying discussion.

B.1 General Architecture Issues

Which of the five basic elements of architectural definition be addressed? How much emphasis should be placed on each one of them?

What dimensions of scope does the architecture address? For what domain is the architecture intended? How much of the life-cycle of a control system should be covered by the architecture? What aspects of an organization are covered by the architecture?

How far from theory to implementation should a reference architecture go? How should that continuum be divided?

What types of analysis should be performed on a domain in order to construct or apply an architecture? What methodology should be used in performing the analyses?

What tiers of architectural definition should an architecture specify? What language or languages are suitable for defining architectures at different tiers?

What degree of granularity is best at each tier of architectural definition? In an architecture with several tiers of architectural definition, is it reasonable to have granularity become finer at lower tiers of architectural definition?

Should a methodology of how to develop an architecture be specified? If so, what should this be? Should a methodology on how to implement an architecture be specified?

What architectural units should be specified? Should these units have standard internals? Should they be black boxes? Which should be atomic units?

What types of interactions should be permitted between architectural units?

To what extent should an architecture specify the type of operating system used for implementations?

To what extent should an architecture define the term “process”, the interaction between operating systems and processes, and the role of processes in implementations?

How should architectural components be mapped onto hardware and software? What rules can be used for making this assignment?

How important is it that conformance criteria be included in an architecture? What sort of conformance tests could be devised? Who would use conformance tests?

Should the architecture allow for non-conformance of components? Should the architecture define conformance classes with different degrees of conformance? Should the architecture define different conformance classes for different, incompatible choices of features?

What established standards can be used? What developing standards can be used? When should the latest version of a developing standard be used?

B.2 Control Architecture Issues

Should the architecture specify how humans interact with the control system? Which parts of the control system should the human be permitted to interact with? Which aspects of the interactions between the components of a control system should the user be permitted to alter?

What functionality should be included in a controller? What (if any) submodules should a controller have? Should there be one submodule for each function? Which of these submodules should be permitted to be independent communicating modules? What should be the form and content of the communications among these submodules?

Should controllers have operational states? What should they be and what sequence should be followed during start-up and shutdown?

Should controllers use blocking or non-blocking I/O? Should controllers use interrupts or cyclic processing? Should sleeping processes with wake-ups be used?

Should controllers have operational modes? If so, what should they be?

Should the internal workings of the controllers in a control system follow some standard or paradigm?

Should control entities have the capability to put commands received in queues?

Should a controller have the capability to carry on more than one task at a time? If so, how should the controller determine what resources are required for each task and how any shared resources should be allocated?

Should the user be permitted to direct that a controller perform a specific task? If so, how should a user introduce a task to the controller?

Should there be a default set of operations possible from the human interface?

Should a default human interface be defined?

Should situation-specific human interfaces be allowed?

How should controllers that need to work together do so? What should be the criteria for grouping controllers together? Should the interaction of controllers be direct (via command-and-status), indirect (via shared data), or a combination of both?

Is it desirable to mix hierarchical and client-server models? How can this be done, if so?

Should a controller be able to control both other controllers and equipment, or just one or the other?

What sort of synchrony, if any, should be required of a grouping of controllers? Should the same type of synchrony be required for every grouping, or should different options be maintained?

What response requirements do implementations of the architecture need to satisfy?

What is the need for a system-wide clock? What is accomplished by maintaining various levels of accuracy? How can a system-wide clock be used to maintain synchrony?

To what model(s) of task generation and execution does the architecture subscribe?

Should a control architecture specify the command and status exchange between controllers? If so, how detailed should this specification be? Should it specify the semantics of the exchange, the format of the exchange, the encoding of the exchange?

How should task coordination be accomplished? Should the information for coordination be in the work element, in the plan for a task (if one exists), or in some other part of the control system?

At what level of abstraction should information be modeled by the architecture? What information should be specified by the architecture? What data should be required to be used and generated by architectural components? Should the specification of such data be conceptual, logical, physical or some combination of the three? Should data storage and/or data access mechanisms be specified by the architecture? How should data be physically distributed in a system? How should data be accessed and by whom should it be accessed?

What types of plans should be included in the architecture? What types of information should a plan contain? What plan format(s) should be used?

Should the architecture include formal resource definitions? In the definitions of resources in the architecture, are the ways in which the resources are used specified? Should dynamic characteristics of the resource be included with its description? Are there types of resources which share characteristics? If so, how should they be categorized.

To what extent should the architecture require plans to be generated in advance, as opposed to deciding what to do in real time? If plans are generated in advance of their use, should the resource allocation for the plans and the scheduling of the plans be done at the same time as the plan generation, or can these activities be performed later using a skeletal plan which refers to resource classes? Can these modes be mixed effectively? If so, what requirements does this place on the control structure?

How should scheduling be handled?

Is support needed for resource allocation? If so, what types are provided for?

To what extent should the architecture deal with communications? Should the architecture specify the communications paradigm, the communications implementation? Should one communications scheme be required for all portions of the architecture, or can multiple schemes be used? What types of communications should the architecture allow? How should communications between architectural units be handled?

To what extent should checks and safeguards be built into an architecture? Are the safeguards outside the basic architecture or an integral part of it? To what extent is the ability to recover from errors in each major subsystem (e.g. communications, groups of controllers, data system) built into the architecture? What mechanisms does the architecture permit or require for handling cross-system errors? What feedback mechanisms for fine tuning control system operation does the architecture permit?

Appendix C - Issue Analysis and Comparison of RCS and MSI

This appendix gives an issue-by-issue analysis of the RCS (Real-Time Control System) architecture developed in the Robot Systems Division, and the MSI (Manufacturing Systems Integration) architecture developed in the Factory Automation Systems Division, using the issues raised in Section 4 and Section 5 of this report. The subsections of this section are in exactly the order in which the issues are presented in Section 4 and Section 5. Page references are included to make it easy to refer back to the issues.

The analysis is preceded by introductory comments about the two architectures and the papers describing them. The two architectures differ greatly in the numbers of papers written about them. Both MSI and RCS are founded on the work in manufacturing control systems performed in the Automated Manufacturing Research Facility (AMRF) at NIST. RCS has been under development for over a decade under the auspices of a number of projects. There are dozens of papers about or involving RCS. In contrast, the MSI architecture has been under development for only three years. Papers describing the initial version the architecture have been published, but papers describing the current version of the architecture are in the process of being written. Thus, the RCS discussions include many citations to RCS papers, while the MSI discussions include few.

For each issue we present where RCS and MSI stand, and we give comments on the compatibility of the two positions. The RCS positions are high-level positions, not the positions taken in individual applications of RCS. In the comments, we use the term "joint architecture" to refer to the architecture the two Divisions plan to build to subsume or supercede RCS and MSI. Some of the comments include suggestions regarding the joint architecture.

For several issues, RCS or MSI is silent on the issue. In these cases RCS and MSI are described as being compatible; the authors are aware that this is a very weak form of compatibility. In particular, where RCS is silent at a high level on an issue addressed by MSI, individual RCS implementations are likely to have either done nothing (i.e. implemented silence) or developed a point solution, and either is almost certainly incompatible with MSI.

The level of the outline at which RCS and MSI are compared varies in this appendix. If an issue has independent subissues, RCS and MSI are broken out at the subissue level. If the subissues of an issue are closely related, RCS and MSI are broken out at the issue level.

This appendix has two primary intents: first, to provide the details that support the more general discussion of the compatibility of RCS and MSI that appears in Section 7, second, to provide guidance for the joint architecture described in Section 8. Two secondary intents are to provide an issue-by-issue index into the contents of the RCS papers and to provide a detailed description of the MSI architecture.

C.1 RCS Introduction

Where different RCS papers represent different positions on an issue, we have tried to include all positions. In this appendix, by “the RCS papers” we mean the 40 papers about RCS which were reviewed for this report and are covered in Appendix D.

C.1.1 Characterization of the RCS Papers

The RCS papers may be grouped as follows. The definition of the groups and the assignment of a given paper to a group are not clear-cut, and the reader should not imagine that this the only way it might be done.

(1) pre-NASREM RCS - theory and implementations
[Leake1], [Murphy1], [Wavering2]

(2) NASREM & NASREM-like RCS - theory, issues, implementations
[Albus1], [Albus5], [Albus7], [Albus8], [Albus10], [Fiala1], [Fiala2], [Fiala3], [Fiala4], [Herman1], [Herman2], [Herman3], [Kramer1], [Lumia1], [Lumia2], [Lumia3], [Michaloski3], [Szabo2], [Szabo3], [Szabo4], [Szabo5], [Wavering1]

(3) Post-NASREM RCS - (NASREM with value judgment)
[Albus2], [Albus4], [Albus6], [Huang1], [Michaloski1],

(4) ARTICS RCS
[Albus3], [Albus9], [Albus11]

(5) Barbera RCS
[Feldman1], [Horst1], [Quintero3]

(6) Other
[Michaloski2], [Quintero1], [Quintero2], [Szabo1]

The NASREM papers are hard to classify as application-dependent or application-independent because:

(1) some emphasize NASREM as an architecture for the specific application for which it was designed, space station robotics - [Albus5], for example,

(2) some (without any change in the upper tiers of architectural definition of the architecture) emphasize the application-independence of the NASREM architecture, while acknowledging its original intent - [Lumia1], [Lumia2], [Lumia3], for example, and

(3) some emphasize the applicability for a different specific application of an architecture which is identical to NASREM at the upper tiers of architectural definition but which changes at the application-specific tier - [Albus7], which is about coal mining, for example.

The ARTICS (Architecture for Real-Time Intelligent Control Systems) papers are appeals to the community concerned with real-time machine control to get together and agree on an open systems reference model. In these papers, RCS is presented as a proposed starting point or strawman for the reference model.

Several of the papers listed above ([Fiala3], for example) discuss implementation details (mostly details of control systems for robots) and assume an RCS architecture, but barely mention RCS. Many other such papers exist which have not been reviewed for this report. The authors felt it was important to read a sampling of such papers but not necessary to read all of them.

C.1.2 Preliminary Observations about RCS

RCS does not fit fully with the definition of an architecture adopted in this report. Rather than distinguishing between architectural specifications and methodologies for architectural development, RCS lumps them together. Rather than having firm rules for building control systems, RCS has soft rules often called “tenets”, “principles”, or “canons”.

There are no explicit tiers of architectural definition in RCS. As already discussed, however, RCS fits a model with three layers below the top.

In RCS, a conscious effort has been made to make the architecture broadly applicable. As stated in [Albus2 page 59]:

> *The evolution of the RCS concept has been driven by an effort to include in a single reference model architecture the best properties and capabilities of most, if not all, the intelligent control systems currently known in the literature, from subsumption to Soar, from blackboards to object-oriented programming.*
>
> *A reference model architecture … is a canonical form, not a system design specification. A canonical form does not force anything. It simply structures the problem so that designers can understand what they are attempting to do.*

As a result of making the RCS architecture broadly applicable, it is often hard to determine from the RCS papers what is acceptable and what is not under RCS.

C.2 MSI Introduction

The definitive MSI architecture paper has not been written. Hence, all statements about the architecture written here represent the author’s perspective on the current version of MSI.

C.2.1 MSI Architectural Definition

The MSI architecture is most directly applicable to the domain of discrete parts manufacturing. Some concepts may extend beyond this domain, but which concepts are portable is not made explicit. The goal of the architecture is to enable flexible integration of manufacturing systems. As a consequence, the MSI architecture focuses on defining expected functionality and interfaces of the architectural units, rather than their internals. With this type of specification, it is possible to include black box components which were not originally designed to be part of an MSI system.

MSI has three tiers of architectural definition. The first is concerned with broad architecture issues such as what control structures to use, communications paradigm to use, and acceptable paradigms for data access; the second is composed of the detailed specification of the interaction between architectural components and the information models; the third tier is an implementation of the architecture. The MSI architecture intentionally leaves quite a few implementation choices. Many of these choices are explicitly delineated in the specification.

At the first tier of architectural definition, MSI does not explicitly state how an architecture should conform to it. At the second tier, MSI requires that components be able to interpret the relevant portion of the information model and requires that controllers and schedulers have the interfaces specified in the architecture. In the model, it has not been made explicit which portions are applicable, therefore the implementation has considerable leeway in the interpretation of this requirement. The interfaces include the specification of information which must be exchanged, and dynamic behavior. The expected functional aspects of each component are outlined, but the component complies if it exhibits the interface, even if does not actually perform all the functions.

C.2.2 Preliminary Observations about MSI

The MSI architecture is designed to promote the effective integration of a manufacturing system. As a consequence, great attention has been given to the functional and information requirements for integration. The MSI architecture does not address issues of hard real-time control of equipment. The architecture is designed so that controllers which are capable of providing such control can be interfaced to an MSI controller and included in the control hierarchy.

The MSI architecture committee views architectural development as a cyclic process in which each implementation provides feedback to be included in the next version of the architecture. The MSI architecture presented here includes the result of the first implementation.

C.3 Balance Among Elements of Architectural Definition

This issue is discussed in Section 4.1 on page 13.

C.3.1 RCS

Discussions of the components of RCS given previously may be summarized as follows:

(1) Scope: RCS papers omit two of the three dimensions of scope (see Section 4.2 on page 13). The discussion of domain is generally vague.

(2) Domain Analyses: RCS papers report a lot of functional analyses and some dynamic analyses (in natural language) but no information analyses above the computer language level. RCS papers report no use of formal domain analysis methodologies.

(3) Architectural Specification: RCS papers place heavy emphasis on architectural specifications and methodology for architectural development. These two components are not differentiated, however, and there is almost no explicit recognition of the need for tiers of architectural definition.

(4) Methodology For Architectural Development: See previous bullet.

(5) Conformance Criteria: The idea of conformance criteria is discussed briefly in a few RCS papers, but no actual criteria are given in any papers.

The authors agree with the notion embodied in RCS that architectural specifications and methodology for architectural development are the two most important components of an architecture, but conclude that the components of RCS are not well-balanced.

C.3.2 MSI

At the highest tier of architectural definition in MSI, the emphasis was placed on the domain analysis and the middle tier of architectural specification. The MSI architecture specifically constrained implementations as little as possible while still achieving integration.

C.3.3 Comments

Any joint architecture should be balanced consciously. Scope and conformance criteria could use more attention than given by RCS or MSI.

C.4 Scope Issues

This issue is discussed in Section 4.2 on page 13.

C.4.1 RCS

The intended scope of RCS and the domain of RCS may be characterized along the dimensions identified earlier: domain, life cycle, organizational extent, and tiers of architectural definition.

At the top (application independent) tier, the scope and purpose of RCS are presented informally and generally in several RCS papers. The texts of these papers generally provide examples of situations to which the architecture is intended to be applicable, but not a precise definition of scope. The introductions to the ARTICS papers, for example, say ARTICS should focus on “real-time, sensory-interactive, intelligent machine control systems” (or very similar phrases), without being more specific [Albus3], [Albus9], [Albus11][22].

22. [Albus11] is a straightforward condensation of [Albus9]. From this point on in the appendix, we omit references to [Albus11]. Anywhere there is a reference to [Albus9], it may be assumed that [Albus11] says roughly the same thing.

C.4.1.1 Domain

The application-independent RCS papers generally say RCS is suitable for a broad domain in which discrete operations are performed. Specific applications RSD has tried are listed in Section 7.1.1. RCS papers do not specifically exclude any applications except for continuous processes like oil refining.

C.4.1.2 Life Cycle

RCS has not been described in the context of commercial products, but rather in the context of performing tasks. Life cycle considerations are not explicit in any RCS papers. The part of the life cycle described in Section 4.2.2 that seems implicit in RCS is design, manufacturing (of a prototype), and the testing and operation portions of use.

C.4.1.3 Organizational Extent

The RCS papers generally cover only operations (shop floor operations in the case of manufacturing or shipboard navigation operations in the case of submarines, for example). Other parts of the organization that are closely related and might be considered (the design engineering department for manufacturing or the maintenance and repair department for a submarine, for example) are generally not considered.

C.4.1.4 Tiers of Architectural Definition

Only a few RCS papers deal with the issue of tiers of architectural definition, and those papers do not use that term (or even the identical concept) [Albus8, abstract], [Fiala1, section “Specification” at end], [Lumia2, section 1], [Michaloski1, section 2.1]. Most RCS papers, however, are consistent with an architecture containing three tiers of architectural definition below the top, as exemplified in Table 1 on page 9 of this report. [Albus8, abstract] splits the lowest tier into two: software architecture and hardware architecture.

C.4.2 MSI

The MSI architecture explicitly applies to discrete parts manufacturing.

MSI is primarily concerned with the operation aspect of the life cycle described in Section 4.2.2, although stubs are provided for design, and maintenance.

With respect to organizational extent, MSI covers planning, scheduling, and production. Rudimentary business considerations are within the scope of MSI. For example, the concept of order is included and the notion of the cost of part production, and inventory is in place. Such concepts provide the connection between the business and technical sides of manufacturing but do not adequately deal with the business aspect of manufacturing.

The MSI architecture is intended to support the closed-loop control of soft real-time applications where the computation time of the computer systems involved is negligible in comparison to the response time expected in the manufacturing environment. The MSI architecture assumes that for hard real-time systems, a control system designed for this purpose which exhibited the required MSI interfaces would be employed.

C.4.3 Comments

On the domain axis, MSI is limited to manufacturing while RCS is intended to include manufacturing and many other uses. RCS explicitly handles hard real-time control, while MSI explicitly does not.

On the organizational extent axis, both MSI and RCS concentrate on operations and say little or nothing about other parts of the organization. MSI does more than RCS.

MSI addresses the life cycle axis by providing hooks where a life cycle model developed elsewhere may be included.

In summary, the two architectures have somewhat different scopes, but there is a large area of overlap. It would not be difficult to define a slightly larger scope for a joint architecture that includes both. At the top tier of architectural definition, the scope would be broad and cover what RCS and MSI currently cover. At an intermediate tier of architectural definition, above the application-dependent tier, it may be useful to define two separate architectures, both of which conform to the top tier, such that the scope of one includes the applications currently handled by RCS, and the scope of the other includes the applications currently handled by MSI. A complete architecture for manufacturing must include both the integration with the information and scheduling systems of the factory, and provisions for hard and soft real-time control of the machine tools and robots within it.

C.5 Domain Analysis Issues

This issue is discussed in Section 4.3 on page 14.

C.5.1 RCS

C.5.1.1 Aspects Covered

C.5.1.1.1 Functional Aspects

The majority of the RCS papers cover functional aspects in great detail (so we do not cite them here). The details are summarized in Section 7.1.

C.5.1.1.2 Information Aspects

Information Descriptions

At the middle tier of architectural definition, several papers (as cited in Section C.16.1) recommend the use of particular types of data, such as maps, frames [Michaloski1], and lists. At the lowest tier, each implementation, of necessity, handles the information it needs. A few papers about RCS implementations (as cited in Section C.16.1.1.1) give details about the information required in the implementation.

Data Handling

This is discussed in Section C.16.2.1.

C.5.1.1.3 Dynamic Aspects

Dynamic Reconfiguration

Several high-level RCS papers say that RCS systems should be dynamically reconfigurable, in the sense that the controller hierarchy should be modifiable by adding or removing controllers while the system is operating [Albus4, page 203], [Albus5, section 1.3], [Albus6, page 4], [Michaloski1, sections 2.3.4, 3.2.6], [Quintero3, section 3.2]. None of these, however, describes any formal procedure for accomplishing dynamic reconfiguration.

None of the RCS papers describing implementations of RCS describes any implementation of dynamic reconfiguration. Two papers describing implementation details say that dynamic creation of processes (one method of implementing dynamic reconfiguration) is to be avoided [Michaloski1, section 3] [Michaloski2, section 2.0].

In [Wavering2, section II.2.3], three gripper controllers were kept in the hierarchy at all times, although only one gripper could be used at a time. The control system had the ability to let processes sleep when not in active use, to avoid wasting computational resources. Although that paper did not discuss dynamic configuration directly, the method used for dealing with three grippers avoided dynamic reconfiguration in a situation where it could have been used.

Start-up and Shutdown

Two of the papers about RCS applications give detailed explanations of how to start a specific control system up and shut it down [Leake1, sections 6.1, 6.11], [Murphy1, appendix B]. There are no discussions in any of the RCS papers of a general approach to start-up and shutdown.

Hard Real-time Performance

One major thrust of RCS (my first name is Real-Time) is the achievement of real-time performance. As may be expected, many RCS papers deal explicitly with the issue [Albus1 throughout], [Albus3, section “Real-time and Temporal Reasoning”], [Albus8, section “Software Development Environment], [Albus9, section 3.4], [Fiala2, throughout], [Fiala4, sections 6, 8.3], [Herman1, section 9], [Herman2, sections 4.2 - 4.5], [Herman3, section 6], [Horst1, section 3.10 and other], [Leake1, section 3.1], [Michaloski1, sections 3.3.3, 5.2], [Michaloski2, most], [Michaloski3, most], [Quintero3, sections 3.8.6, 3.9.1] [Wavering1, section 9], [Wavering2, section IV]. In addition, many other papers talk about measuring “performance” without explicitly saying that one very important measure of performance is whether the control system can run in real time, even though it is apparent that this is assumed.

C.5.1.2 Analysis Methodology

In the “Summary of the RCS Methodology Steps” given in [Quintero3, section 6], steps are suggested in natural language for gathering domain knowledge and developing the problem description, but no formal activity analysis methods (such as IDEF0 or SADT) or methods for functional analysis are mentioned in any of the RCS papers.

Little is said in the RCS papers about techniques for analyzing the information aspects of a control system.

C.5.2 MSI

In formulating the MSI architecture, the MSI committee performed formal information analyses. The results of these modeling efforts are a number of information models in NIAM concerning the information required by control, planning and scheduling, and other systems. These information models will be discussed elsewhere in this report.

An activity analysis of the process of converting orders to production plans was also made. This information was not explicitly modeled in a formal activity modeling language, but has affected the information models and other aspects of the architecture.

Informal analyses of the function and dynamics of a number of processes in the manufacturing environment were made. Functional aspects of each architectural unit were described, although the degree of detail varies. One of the more detailed analyses made was of the way in which a scheduler, human, and controller interact in order to implement error handling. The results of this analysis are expressed in the specification for the interfaces of controllers and schedulers. This specification reflects conclusions formed by the construction of numerous scenarios of normal and error situations. Dynamic aspects of the exchange of command and status information are included.

The dynamics of information access has also been included for some of the information (such as plans). A sequence for start-up, shutdown, and reconfiguration of the control hierarchy is described, as are mechanisms for triggering reconsideration of important pieces of information, such as resource availabilities.

C.5.3 Comments

RCS and MSI both include functional aspects. Neither architecture provides formal functional analysis techniques. Many other sections of this appendix discuss specific functional aspects.

MSI does much more than RCS in formalizing the information aspects of a control system. These formal techniques are essential for making MSI work and could be expected to provided significant improvements in RCS implementations. RCS is compatible with the MSI formalisms, so MSI’s formal techniques for information could be included in a joint architecture in a straightforward way.

Dynamic reconfiguration is included in principle in RCS and MSI. There is no theory of how to accomplish dynamic reconfiguration in RCS, and none of the RCS papers report doing it in practice. There is some dynamic reconfiguration theory in MSI, and specific provisions for how to implement it. Although MSI allows for dynamically including or excluding specific controllers, this is intended to accommodate error recovery and maintenance operations. An architecture that included specific methods for dynamic reconfiguration and covered more types than MSI could be defined in principle, but the authors believe that actually defining such an architecture would be quite difficult. Covering dynamic reconfiguration in a joint architecture should be regarded as desirable but optional.

C.6 Architectural Specification Issues

C.6.1 Granularity

This issue is discussed in Section 4.4.1 on page 16. It is suggested the reader review that section before continuing with this one.

C.6.1.1 RCS

The degree of granularity that should be part of the definition of RCS does not seem to be resolved.

Although none of the RCS papers use the term “granularity”, four of them discuss atomic units [Fiala1, section 1], [Fiala2 all], [Quintero1, issue ME0002], [Quintero3, sections 1.3, 3.6]. Several others either state the intended size of “modules” whose interface with other modules is intended to be part of the architecture or provide examples of the size of modules or atomic units. Recalling that an RCS controller includes sensory processing (SP), world modeling (WM), behavior generation (BG), and value judgment (VJ), and that behavior generation may be decomposed into job assignment (JA), planning (PL) and execution (EX), the RCS papers suggest (by statement or example) three different sizes of atomic unit:

(1) controller is atomic unit: [Horst1], [Quintero3, sections 1.3, 3.6]

(2) SP, WM, BG, and VJ are atomic units: [Albus4, page 202], [Albus5, section 4], [Albus7, section 4]

(3) JA, PL, and EX are atomic units: [Albus8, figure 4], [Fiala4], [Lumia2], [Wavering1]

Finally, [Fiala1, section 1] suggests explicitly that JA, PL, and EX are atomic units which decompose BG, but WM and SP should not be decomposed. The same notion is implicit in most of the RCS papers which have JA, PL, and EX as atomic units

C.6.1.2 MSI

The MSI architecture has definitions of atomic units for planners and controllers. These atomic units can be composed in a variety of ways which expose (or do not expose) some of the defined interfaces. For example, if a unit is constructed which performs both the planner and controller functions, the combined control entity does not expose the planner-controller interface. Thus the planner-controller interface does not need to conform to the MSI specification, but the planner/controller must still exhibit the required interfaces to its superior planner, superior controller, subordinate planner, subordinate controller, planner-guardian and controller-guardian. Conglomerations of any number of the planner and controller atomic units are permitted, so long as the exposed interfaces are preserved.

C.6.1.3 Comments

Any joint architecture must be clear about what its atomic units are and how they can combine and interact with other components, what its molecular units are and how they can combine and interact with other components, etc. This issue will require significant discussion in the formulation of a joint architecture.

C.6.2 Architecture Definition Languages

This issue is discussed in Section 4.4.2 on page 17.

C.6.2.1 RCS

Upper Tiers of Architectural Definition

Natural language (English) is used in all RCS papers for describing the upper tiers of architectural definition, except [Kramer1], which defines a portion of the NASREM architecture in EXPRESS. No use is made of IDEF0, IDEF1, NIAM, or SADT. A little use of DeMarco notation appears in [Michaloski2]. There are no formal models of information in RCS implementations above the bottom tier of architectural definition.

In natural language, several different types of definitions are possible. In the case of nouns, one type of definition is given to provide a good general idea of what the meaning is; this may include brief descriptions, synonyms, and examples; dictionary definitions are usually of this type. Another type of definition is given for the purpose of classification - being able to determine whether something is or is not an instance of the particular noun. The second kind of definition usually requires enumeration of specific attributes and behavioral characteristics of things which are instances of the noun. Natural language definitions in the RCS papers are almost entirely of the first type. As a result, RCS is very loosely defined, and it is often hard to say whether some aspect of a particular control system conforms to RCS or does not.

Bottom Tier of Architectural Definition

To the extent that data structure definitions in a computer language can be termed formal information models, there are information models at the lowest tier of RCS implementations. Implementations of RCS have used C, C++, FORTH, SMACRO, Ada, and other languages.

C.6.2.2 MSI

MSI is expressed in natural language, NIAM and EXPRESS. Implementations of MSI have been built using C, C++ and LISP.

C.6.2.3 Comments

Any joint architecture should use formal languages, where appropriate, to avoid the ambiguities of natural language.

C.6.3 Issues at the Lowest Tier of Architectural Definition

C.6.3.1 Hardware

This issue is discussed in Section 4.4.3.1 on page 19.

C.6.3.1.1 RCS

The upper tiers of architectural definition of RCS do not specify computer or communications hardware or even any performance characteristics of the hardware, other than being able to perform in real time.

Computer Hardware

RCS implementations have been developed using many different computers, including VME backplane systems and 680x0 computers, Sun workstations, IBM PC’s (and PC clones), Apple MacIntosh, Silicon Graphics Iris, and others. VME bus is used extensively.

Communications Hardware

Standard communications hardware, such as ethernet or RS232 cables and processors are reported in the RCS papers for when interfacing components need to communicate and are not part of the same process or using shared memory on a bus. For mobility applications, radio frequency communications hardware is also used.

C.6.3.1.2 MSI

The MSI architecture is hardware independent. However, the architecture assumes that the facilities for shared file access and for communications between processes exist. If data systems are available (for databases, memories, whatever storage mechanism is chosen by the implementation) which can retrieve information from physical storage based upon the location of that information in the conceptual schema, information access can be made independent of physical location. The implementation is free to place some of the information in local memory, some in databases and some in files. Due to the way in which the Control Entity interface works, however, plans must be maintained in a database (so that one entity could be changing part of the plan while another entity is accessing and possibly changing another part of it), if full functionality is to be achieved.

The MSI architecture requires that command and status communications between controllers (and schedulers) be performed using a point to point, guaranteed delivery message service, such as provided by the MAP and MMS communications standards.

C.6.3.1.3 Comments

RCS and MSI seem incompatible here, because MSI requires a particular logical communications protocol while RCS does not and MSI requires that plans be maintained in a database while RCS does not.

C.6.3.2 Operating System

This issue is discussed in Section 4.4.3.2 on page 19.

C.6.3.2.1 RCS

RCS does not require the use of any specific operating system. RCS has been implemented using several different operating systems, including pSOS, VxWorks, GRAMPS, DOS, Lynx OS, and UNIX. The support of hard real-time processing requires that the operating system permit the user to assume control of the multitasking to permit processes undivided use of the computer resources for a specified amount of time. While the RCS papers do not report any operating systems built in source code for a particular application, they do mention the constraints which hard real-time processing puts on the operating system.

C.6.3.2.2 MSI

The MSI architecture does not specify an operating system.

C.6.3.2.3 Comments

RCS and MSI are compatible here. Note that, in a distributed control system, all controllers need not run on the same processor and therefore, need not use the same operating system.

C.6.3.3 Processes

This issue is discussed in Section 4.4.3.3 on page 19.

C.6.3.3.1 RCS

The definition of what a process is varies among RCS implementations. The issue of how a process should be defined is addressed directly in [Fiala1, section 2], [Fiala2, section 4], [Michaloski1, sections 3.2.1, 5.2], [Michaloski2, most], [Michaloski3, most], [Quintero1, issue ME0007], and [Wavering2, section II.2.2]. In addition, many other RCS papers refer in passing to processes. All of the notions of process described in section 4.4.2.3 may be found in some RCS implementation.

C.6.3.3.2 MSI

The definition of process is not an issue for the MSI architecture, the implementation is given complete freedom in determining how many processes a component may be made of. However, since the communications model is point to point, the communications system is aware of the distribution. Therefore one specific process must do the communications for each MSI planner or controller.

C.6.3.3.3 Comments

In any joint architecture, terms like “process”, if they are used, must be carefully and unambiguously defined. The salient characteristics of a process must be specified - such as (and this is only an example, not a recommendation): a process and only a process can serve as a “point” in a point to point communications model. The issue of what a process is, and what restrictions should be made on processes will need to be resolved when a joint architecture is formulated.

C.7 Methodology For Architectural Development Issues

C.7.1 Cyclic Development

This issue is discussed in Section 4.5.1 on page 20.

C.7.1.1 RCS

[Quintero3, section 6] explicitly recommends that control system development be done cyclically. Other RCS papers are not explicit, but several seem implicitly to endorse cyclic development.

C.7.1.2 MSI

The MSI architecture committee believes that architectural development is a cyclic process. The MSI architecture is currently undergoing its second design, implement and test cycle.

C.7.1.3 Comments

RCS and MSI are compatible here.

C.7.2 Mapping Architectural Components onto Hardware

This issue is discussed in Section 4.5.2 on page 20.

C.7.2.1 RCS

Several RCS papers describe the assignment of control or computation processes to processors during control system design. [Albus8, section “Software Development Environment”], [Fiala3, section 3], [Horst1, figures 7, 8 give example maps], [Lumia2, section 4], [Michaloski2, section 4]. The method for making the assignments is usually “estimate what will work, implement it, test, and redo until it works”.

This issue is closely related to the resource allocation issue discussed in Section 5.9.3 and Section C.17.4.

C.7.2.2 MSI

It is up to the implementation to divide the control entity (a controller-scheduler combination) into one or more processes. If a control entity is implemented as more than one process (in the computer science sense of the term), the scheduler-controller interface must be followed. If this is not split, the interface is permitted to not conform to the specification. Thus, certain choices may restrict an implementor's options. If the scheduler and controller are to be implemented as separate processes in a non-multitasking environment, this requires at least two machines per control entity.

The MSI architecture makes few assumptions about where data is stored. Private data is permitted and may be local or remote. Shared data is supposed to reside in a logically global database. However, it is possible for the data to be stored in a "globally accessible" place which is accessible by only a limited number of architectural units.

C.7.2.3 Comments

RCS and MSI are compatible here. It is worth noting that the assignment of controllers to hardware in RCS is usually a fixed part of an RCS implementation (it would be difficult to assure hard real-time performance otherwise), while the assignment of controllers to hardware in MSI (which is designed for soft real-time operations) is done in an easily changeable configuration file which is part of the data for an implementation.

C.7.3 CASE Tools

This issue is discussed in Section 4.5.3 on page 20.

C.7.3.1 RCS

RCS-Specific CASE Tools

In the RCS papers, Computer-Aided Software Engineering (CASE) tools designed explicitly for building RCS systems are reported only in [Horst1, section 3.18]. That is a reference to a CASE tool for designing "Barbera RCS" systems which has been developed at the Advanced Technology Research company. In addition, one of the authors has worked on a prototype RCS CASE tool, but this has not been reported.

The use of CASE tools is explicitly suggested in [Albus3, section "ARTICS Vision"], [Albus9, section 2 and appendix A], and [Szabo1].

Development Environment

Some RCS papers, [Lumia2, section 4], for example, mention that it is useful to have a control system development environment with software tools and hardware systems which differ from the software and hardware configuration that the completed control system is to have. In the development environment, the systems are generally set up to allow systems programmers to monitor what the control system is doing and to help with debugging computer code for the system. Often the operational hardware and software are optimized for hard real-time performance and do not permit monitoring or debugging.

Simulation and Animation

Several RCS papers cite existing or suggested simulation and animation systems for simulating the hardware being controlled and/or the environment in which the hardware operates and for graphically depicting the simulations and the operation of the system in action ([Albus5, section 7], [Albus3, section “Simulation and Animation”], [Albus9, section 3.11], [Feldman1, sections 5, 7, 8], [Horst1, sections 2.4, 2.5, 2.6, 4.1.2, 4.1.3], [Quintero1, issue ME0010], [Quintero3, section 6 - table 1 item 6]). Simulation and animation are useful both for control system development and for system demonstrations.

C.7.3.2 MSI

MSI does not address CASE tools.

C.7.3.3 Comments

RCS and MSI are compatible here. There is no requirement for CASE tools, and if they are used, they do not have to be regarded as part of the architecture.

C.8 Conformance Criteria

This issue is discussed in Section 4.6 on page 21.

C.8.1 RCS

The notion of conformance criteria is discussed in [Albus3, section “Testing and Validation”], [Albus9, section 5.4], and [Szabo1]. No conformance criteria are reported in the RCS papers as having been defined for RCS. The extent to which the specific applications described in some papers conform to the more abstract architectural specifications given in other papers is not clear.

C.8.2 MSI

MSI does not address conformance testing methods, usefulness of conformance testing, or testing conformance in development. MSI does not have different conformance classes.

MSI addresses allowing non-conformance in controllers and schedulers. An entity can be included if it contains the functionality of one or more architectural units in MSI or can be made to appear as if it contained the functionality of one or more architectural units. For example, a commercial controller which has the functionality of both an MSI planner and controller may be packaged to include the interfaces of both a planner and a controller to the exterior, and the interface between the planner and controller remains private. Another example is that of the centralized planner. As long as it can be made to exhibit interfaces to the appropriate controllers, this can be made to fit into the architecture. It would not be possible, however, to include a controller which did not perform all of the expected functions, without writing software to perform the missing ones. It should be noted that the fullest use of the error-handling facilities of the architecture occurs when the architecture is followed down to the lowest possible hierarchical level. The “higher up” the black box extends, the more impaired the error-handling capabilities.

C.8.3 Comments

Any joint architecture should include conformance criteria that are at least sufficient to allow one to decide whether a specific control system is an implementation of the architecture.

C.9 Standards Issues

This issue is discussed in Section 4.7 on page 22.

C.9.1 RCS

The upper tiers of architectural definition of the RCS architecture do not specify that standards must be used at lower tiers, but most RCS papers that describe hardware and software used in RCS systems mention the use of one or more standard items (such as RS232 for communicating or a programming language such as C).

The ARTICS papers ([Albus3], [Albus9]) suggest that the long-run aim is that ARTICS should be a set of standards.

C.9.2 MSI

The MSI architecture suggests compliance with standards where appropriate. Presently available standards which are appropriate include SDAI (STEP Data Access Interface) [Fowler1] for remote data access, EXPRESS [Spiby1] for information modeling and MAP [MAP1] for networking.

The MAP standard enforces a connection-oriented, point to point, guaranteed messaging paradigm. The MAP standard is only required for command and status messages. Variations on the MAP stack, including the use of Ethernet instead of a broadband communications medium is permitted as long as the implementation uses the MMS protocol. Communications with the database does not have to use MAP. Internal inter-process communications are unconstrained.

A major aim of the MSI architecture development is to provide input to the appropriate standards committees.

C.9.3 Comments

RCS and MSI are compatible with regard to standards.

C.10 Domain

This issue is discussed in Section 5.2 on page 25.

C.10.1 RCS

Most RCS papers describe the intended application domain in some fashion. Those which describe applications of RCS generally describe the domain fairly specifically. The application-independent RCS papers, however, do very little to describe the range of situations to which RCS is intended to apply. Usually the range is characterized by a short phrase such as “intelligent real-time control systems”. These papers also do not describe the secondary characteristics of applications that make them suitable or unsuitable for RCS [Albus2, beginning and section “Applications”], [Albus3, section “Introduction”], [Albus6, section “Introduction”], [Albus8, section “NASREM: The Conceptual Architecture”], [Albus9, section 2], [Lumia1, section 1], [Lumia2, sections 2, 3], [Michaloski1, section 1.2], [Quintero3, sections 1, 2 - introduction].

The one common characteristic of the uses described in the RCS papers is that they are all hard real-time and have to deal to some degree with an environment that can change unpredictably, so that sensors and sensory feedback to control are required.

C.10.2 MSI

The intended domain of MSI is discrete parts manufacturing, which is a highly structured environment. The architecture applies only to situations which do not need for the response time of the control system to be guaranteed and in which software response time is not a critical factor. This limits the architecture to upper levels of control.

The information models are specific to discrete parts manufacturing. The information architecture is generic. The communications paradigm is generic, but the suggestion that it be MAP is manufacturing-specific. The scheduling/execution interaction is manufacturing-specific, but could be generalized.

C.10.3 Comments

MSI is much more narrowly focused than RCS.

If a joint architecture is built, the builders should try to identify the secondary characteristics of the domain of the architecture to determine whether the architecture will fit the specific situations.

C.11 Architectural Conformance

This issue is discussed in Section 5.3 on page 25.

C.11.1 RCS

Dealing with black box controllers in an RCS system is discussed in [Michaloski2, section 2.1], [Michaloski3, sections 2,3], and [Quintero3, sections 2.3.3, 6]. In several RCS implementations, at the lowest level of an RCS controller hierarchy, the “actuator signals” that are sent are instructions to a non-RCS controller in a non-RCS format used by the controller.

Of course, if an architecture places requirements only on the interfaces of controllers, all controllers in a control system implementing the architecture are black boxes from the viewpoint of the architecture. Often, however, RCS implementations place requirements on the internals of controllers as well as on their interfaces - such as by requiring that every controller be a finite state machine. Implementations may also place limitations on the nature of a controller (such as: it must be able to run on a processor that fits into a VME bus) that force architectural conformance.

C.11.2 MSI

The MSI architecture defines atomic units, namely planners and controllers. Among planners and controllers, only exposed interfaces need to conform to the architecture. This gives many ways in which to include black box systems. For example, a centralized scheduler could be included by exhibiting a planning interface for each controller, while allowing the internals of the planner to remain private. Black box controllers are permitted, provided they exhibit the required interfaces. The user should be warned that, wherever black boxes are inserted, the error recovery capability of the architecture may be diminished.

C.11.3 Comments

Any joint architecture should make clear the requirements for conforming to the architecture. It should clearly identify the atomic units of the architecture. Whether atomic units should be black boxes or should have some of the internals specified must be resolved. Whether to permit the inclusion of software or hardware modules whose internals are unknown is an additional issue. It should be noted that the ability to permit black box modules to be included in an architecture is required if an architecture is to permit the inclusion of commercial systems.

C.12 Human Interactions with the Control System

This issue is discussed in Section 5.4 on page 25.

C.12.1 RCS

This will be covered in Section C.13.9, since interactions with a control system as a whole are very closely related to interactions with individual controllers.

C.12.2 MSI

MSI expects humans to be involved in the generation of the process and production plans.

Humans (or other intelligent agents) are expected to interact with the control system in the event that there are errors which can’t be automatically fixed. The guardian/planner and guardian/controller specify the corrective actions which a human monitoring agent may take through the command hierarchy. Of course, human agents may affect the control system by altering the database or performing mechanical operations, such as rebooting machines. A major conclusion of the architecture is that, until process and production planning can be automated, automatic error-resolution for most non-scheduling errors can not occur.

C.12.3 Comments

See Section C.13.9.

C.13 Controller Issues

C.13.1 Controller Functionality

This issue is discussed in Section 5.5.1 on page 26.

C.13.1.1 RCS

Most of the RCS papers deal with controller functionality. How RCS deals with controller functionality was described in Section 7.1 of this report.

C.13.1.2 MSI

In the MSI architecture, the item called a controller performs manufacturing tasks. The controller is also responsible for monitoring the performance of the manufacturing task and reporting when errors occur. The controller executes either a plan or a work element. In the case of all controllers except the shop level, it fetches the plan upon command of its superior and parses the plan using any parameters which the superior may pass down to it.

C.13.1.3 Comments

The RCS and MSI views of controller functionality are similar but generally incompatible in the details. RCS itself has many different variations of controller functionality.

Reaching agreement on controller functionality will be a challenge in developing a joint architecture.

C.13.2 Internal Units

This issue is discussed in Section 5.5.2 on page 26.

C.13.2.1 RCS

This is covered fully in the discussion of granularity in Section C.6.1.

C.13.2.2 MSI

In the case of MSI, the controller is an internal unit of an architectural construct called a control entity. A control entity is a unit which contains a controller and its related planner.

C.13.2.3 Comments

See Section C.6.1.

C.13.3 Operational States

This issue is discussed in Section 5.5.3 on page 26.

C.13.3.1 RCS

None of the RCS papers mention operational states.

C.13.3.2 MSI

MSI has operational states for planners (available, active) and controllers. These states determine what can be done with messages when they arrive. The state diagram for controllers is complex. The action of the controller is not completely determined by the state however, but includes information in its plan, in the database (e.g. resource information) and in parameters passed down from the superior. The controller states are: available, active, pausing, paused, terminating, terminated, e-stopped. If the command affects an existing task, the disposition of a command is also affected by the task state of the associated task.

C.13.3.3 Comments

RCS and MSI are compatible regarding operational states, since RCS says nothing.

Defining operational states seems very desirable in a joint architecture.

C.13.4 Execution Model Issues

This issue is discussed in Section 5.5.4 on page 26.

C.13.4.1 RCS

An architecture may require certain characteristics of the execution of controllers. Because RCS systems are hard real-time control systems, some RCS papers suggest placing requirements on the execution model for controllers. An often-stated set of requirements is that in the execution of a controller:

(1) non-blocking (rather than blocking) I/O should be used

(2) cyclic processing (rather than interrupts) should be used

(3) it should be possible to put inactive controllers to sleep

Those requirements are not adopted by all RCS implementations. RCS papers discussing these issues include: [Horst1, section 3.7], [Leake1, section 3.1], [Michaloski2, sections 2, 3], [Michaloski3, section 3], [Quintero1, issue ME0007], [Quintero3, section 3.8]

[Fiala2, section 4] and [Wavering2, section 2.3] provide examples of control systems using sleeping controllers with wake-ups.

C.13.4.2 MSI

A control system conforming to the MSI architecture functions in a data-driven interrupt mode. On the thesis that it would be impossible to examine every piece of data in one processing cycle, MSI stipulates that a re-examination of pertinent data be done after any of the following events:

(1) a new order arrives,

(2) a resource availability changes,

(3) an error in execution occurs,

(4) a plan is changed.

Control entities are informed of these events through commands. It is not specified in the architecture how an event of type (1) is noticed. A type (2) event must be input by human intervention. A type (3) event can be noticed by either a human or a control entity. A type (4) event may be initiated by a planner.

C.13.4.3 Comments

RCS and MSI appear to be incompatible regarding the execution model for the control system.

A joint architecture might be defined that does not address the execution model at the top tier of architectural definition. It would seem desirable to allow different execution models at different hierarchical levels, to take account of the different speeds of execution and different data requirements.

C.13.5 Operational Modes

This issue is discussed in Section 5.5.5 on page 27.

C.13.5.1 RCS

Operational Modes are discussed in [Albus3, section “Level of Automation Flexibility”], [Albus5, section 2.5], [Albus7, section 2.5], [Albus9, section 3.3], [Fiala4, section 3.2], [Herman2, section 1], [Leake1, section 4.2], [Lumia2, section 3], and [Szabo3, section “High Level Review of TEAM Design]. In general, the modes which are discussed are modes of human interaction with the control system. The modes described in [Albus3], for example, include: teleoperation and remote control, computer-aided advisory control, traded control, shared control, human override, human-supervised control, autonomous control, sensory interactive control, and mixed mode control. Types of modes other than human interaction modes are rarely discussed, but certainly many RCS implementations include at least a “debugging” mode.

Unlike the Dornier architecture, the RCS architecture does not suggest having a tier of architectural definition to deal with the requirements introduced by specifying a mode of human interaction. Several of the RCS papers imply, to the contrary, that several alternate modes should be available within an implementation.

C.13.5.2 MSI

The architecture does not require that controllers have different modes. However, in the implementation, the controllers ran as either real or emulated, based on whether the hierarchy was really controlling equipment. The emulated mode allowed the user to speed up or slow down time for demonstration and testing purposes.

C.13.5.3 Comments

RCS and MSI seem compatible regarding operational modes.

It seems desirable to cover operational modes in the joint architecture.

C.13.6 Standard Internal Workings

This issue is discussed in Section 5.5.6 on page 27.

C.13.6.1 RCS

Many of the RCS papers say that each controller (or each WM, SP, VJ or BG controller component - or each JA, PL, or EX component of BG) should be a finite state machine or should be a sandwich consisting of a pre-process, a finite state machine, and a post-process. RCS papers taking this position include: [Albus5, section 4], [Albus7, section 4], [Horst1, section 3.5], [Leake1, sections 2, 3.1], [Quintero3, section 3.8] [Wavering2, section II.2.2]. Wavering points out that: "*Decision processing is usually composed of a two-level hierarchy of state tables. The top state table is used to call the appropriate state table for the current input command*." This seems to be the usual way state tables are used in RCS implementations. Logically, of course, a hierarchy of state tables of the sort described may be thought of as a single state table.

Requiring that each controller be a finite state machine seems implicit in [Huang1, section 3.4] and [Michaloski2, section 1], as well, even though it is not explicitly stated. It seems likely that several other RCS papers refer to implementations in which the controllers are finite state machines, but the paper just does not mention it. Some RCS implementations may not use finite state machines.

The RCS papers do not suggest any other type of standard internal working.

C.13.6.2 MSI

If the internals of the controller are exposed, they must conform functionally to the planner/controller division mandated by the architecture and must have the five interfaces specified.

C.13.6.3 Comments

RCS and MSI are not compatible in this respect. As discussed previously in the Conformance section, conformance in MSI is solely a matter of exhibiting the correct interfaces. All units are black boxes. Some versions of RCS specify the internals of the atomic units. The two approaches are incompatible. There are several possible resolutions of this issue. First a choice between the two could be made. Secondly, the architecture could specify when internals were required, and when they were optional. Third, the joint architecture could avoid requiring standard internal workings at the top tier of architectural definition but could provide an application-independent tier of architectural definition in which a requirement of standard internal workings could be stated.

C.13.7 Command Queues

This issue is discussed in Section 5.5.7 on page 28.

C.13.7.1 RCS

Having a controller maintain queues of incoming commands is discussed only in [Lumia3 section 4] and [Wavering1 section 2] of the RCS papers. [Lumia3] states that for the primitive hierarchical level of an RCS control system in the particular application described (robot motion control): "*An input command queue is necessary at the Prim level so that smooth transitions between consecutive path segments may be planned*." In other words, future commands are needed in order to plan what to do now.

C.13.7.2 MSI

An MSI controller cannot put commands in queues. The superior will send the command at the correct time for it to be performed. Messages, however, may be queued if the rate of transmission of the subordinate overloads the higher level controller's ability to process the messages.

C.13.7.3 Comments

MSI is incompatible with the two versions of RCS that require that incoming commands be queued, but most RCS papers are silent on such queues. Note that the goal of queuing commands, that knowledge of future anticipated actions is required, is achieved in MSI by requiring plans to be generated in advance. For a joint architecture, there may be a number of different ways of accommodating the need to anticipate possible future actions.

A joint architecture should address the issue of command queues. Allowing two varieties of controller (with and without queueing capability) might be a workable solution.

C.13.8 Multiple Simultaneous Tasks

This issue is discussed in Section 5.5.8 on page 28.

C.13.8.1 RCS

None of the RCS papers discuss this issue or report controllers that deal with multiple simultaneous tasks.

C.13.8.2 MSI

The MSI architecture permits all controllers except for equipment controllers to have multiple simultaneous tasks. An equipment controller is constrained to do only one “MSI” task at a time. If the task is decomposed further, this decomposition remains private and is not seen by the rest of the MSI architecture.

C.13.8.3 Comments

RCS and MSI are compatible regarding multiple simultaneous tasks since RCS is silent on the issue.

A joint architecture should address the issue of multiple simultaneous tasks.

C.13.9 Human Interactions with Controllers

This issue is discussed in Section 5.5.9 on page 28.

C.13.9.1 RCS

This section covers both human interactions with (individual) controllers and human interactions with the control system (as a whole). These issues are discussed in [Albus3, section “Human and Computer Interface Flexibility”], [Albus5, section 2.5], [Albus7, section 2.5], [Albus8, section “Operator Interface”], [Albus9, section 3.2], [Albus10, section “An IVHS Vehicle Control Architecture”], [Fiala4, section 3.2], [Lumia2, section 3], [Huang1, section 2.10], [Michaloski1, sections 2.3.11, 3.3.4, 5.4] [Quintero3, section 3.7], [Szabo4, scattered], [Wavering1, sections 3.2, 8.6, 8.7], [Wavering2, section IV]. This issue relates closely to “Operational Modes”, discussed in section C.4.4.5, since most operational modes are modes of human interaction with the control system as a whole.

The ARTICS papers, [Albus3] and [Albus9], simply argue that there should be a wide range of human interfaces to controllers and control systems.

The NASREM theory papers, [Albus5] and [Albus7], most extensively, say there should be both control and monitoring interfaces. Monitoring interfaces are to be available for every component.

Most other application-independent RCS papers require that a human interface to every component or controller be included in any application. Application-specific RCS papers which raise the issue generally report that human interfaces to individual software components have been built for monitoring and debugging, at least.

C.13.9.2 MSI

Humans may interact with either the planner or controller function or with both. An interface may be either passive or active. A passive interface may only view the information being sent by the control entity and set the parameters of the reports. Active interfaces (there can only be one of these for each planner or controller) may modify the administrative aspects of the monitored agent or tasks assigned to it. Interactions are limited to those specified in the control entity interface. With the controller, the human may do such things as emergency stop the system, tell the system that a subordinate is impaired, terminate, abort or defer tasks, or set task reporting and notification parameters. With the scheduler, the human may do such things as emergency stop, or tell the control system that a subordinate is impaired and intervene in the contracting process in limited ways. A process is not required to have a human interface. However, in order to handle certain conditions, you must have one.

Purely application-specific interfaces are not allowed. However, the user can make error and status codes and perform other sorts of (specified) modifications to the code. It is not possible for the user to add another function to the interface.

C.13.9.3 Comments

RCS and MSI are incompatible with regard to the human interface to controllers. Many RCS implementations involve having a human in exclusive or shared control of a task that requires a task-specific human interface, like a joystick or steering wheel. MSI does not anticipate human control or shared control as part of the normal functioning of the control system and does not permit task-specific interfaces.

For monitoring and debugging interfaces, RCS and MSI are not fully compatible, but it would probably be easy to agree on how to handle them in a joint architecture.

C.14 Collections of Controllers

C.14.1 Modes of Interaction

This issue is discussed in Section 5.6.1 on page 29.

C.14.1.1 RCS

RCS requires that control systems be strictly hierarchical. We do not include citations to all the RCS papers that say that here, since there are so many. Each controller (except the one at the top of the hierarchy) has exactly one superior and zero to many subordinates. Dynamic reconfiguration of the hierarchy is intended to be available, as discussed in Section C.5.1.1.3. No use of negotiation for tasks is reported in the RCS papers, and the problem of dealing with a material handling system serving several workstations is not discussed.

Controllers interact via a command-and-status protocol. Citations to the use of a command-and-status protocol for interactions between controllers are given in Section C.15.3.2.1.

A little theoretical discussion of non-hierarchical interactions is given in [Horst1, section 3.3] and [Quintero3, section 3.2].

C.14.1.2 MSI

The MSI architecture mandates that controllers be structured in a hierarchy. The hierarchy must have a single top-level controller called the shop controller. It is distinguished from other controllers by being driven off of orders being given to the shop from external or internal sources. The lowest hierarchical level of controller specified by the MSI architecture is the equipment controller. This entity may or may not control actual equipment. It may internally be composed of any number of cooperating controllers, but this is not visible to the higher levels of the hierarchy. This means that, below the equipment level, only the types of errors which are recognized by the MSI architecture can be percolated up, although certainly error codes may be added which convey additional information. Between the shop and the equipment controllers, any number of levels of "workcell controllers" coordinate the actions of other controllers.

The hierarchy is flexible in the sense that a controller (or a part of a controller such as the MSI planner or the MSI controller) can be replaced by another planner/controller if it becomes impaired. This is accomplished by specific messages outlined in the control entity interface. Commands to ignore or accept a new subordinate must be given by the guardian (intelligent) interface. Other than replacing subordinates, the hierarchy does not change during control system operation. Complete specification of the hierarchy is modeled in the information models for the system and is required information for an implementation.

C.14.1.3 Comments

RCS and MSI are compatible with regard to how controllers interact.

C.14.2 Control of Devices and Controllers

This issue is discussed in Section 5.6.2 on page 30.

C.14.2.1 RCS

The RCS papers do not take a position on this issue.

C.14.2.2 MSI

In the MSI architecture, it is permitted for a controller to supervise both another controller and a device.

C.14.2.3 Comments

RCS and MSI are compatible regarding control of devices and controllers.

C.14.3 Synchrony and Speed

This issue is discussed in Section 5.6.3 on page 30.

C.14.3.1 RCS

RCS is, of course, designed for working in real time, so it is required that all individual controllers, and the control system as a whole, run fast enough to keep up with events in the environment in which the system is operating.

Some RCS implementations provide that all controllers must keep to a fixed cycle rate. The RCS handbook [Michaloski2], however, suggests that the lowest levels of a control hierarchy may need to be synchronous, but the higher levels may be asynchronous.

C.14.3.2 MSI

MSI does not specify any sort of synchrony for controllers. It considers that this may be required for hard real-time response, but is not appropriate for the higher levels of control. MSI includes a system clock, which is for purposes of making up schedules in which times mean (approximately) the same thing to each control entity. This clock is explicitly *not* for functions such as determining message sequencing, or matching times with a precision of greater than one second.

Controller actions are coordinated according to the required sequences of command messages, and the timing of this is controlled by the production plan’s schedule.

C.14.3.3 Comments

MSI is incompatible with regard to synchrony with the branch of RCS that requires controllers to keep to a fixed cycle rate.

C.15 Task Specification, Generation and Execution

C.15.1 Specification of Work Elements

This issue is discussed in Section 5.7.1 on page 31.

C.15.1.1 RCS

RCS methods for specifying work elements (often called tasks in the RCS papers) are discussed in [Albus4, section IIIC] and [Michaloski1, sections 2.3.9, 4.1, 4.2, 4.3.6, 6.4]. In these papers, a work element is defined using a frame representation. Each work element has zero to many parameters (or attributes) which are assigned values for specific instances of the work element. One of the parameters identifies the goal of the task.

Although none of the RCS papers describing implementations use frames to define work elements, the work elements in implementations are conceptually like frames and generally do take parameters.

In RCS the semantics of the work element (an understanding of the nature of the task and how the parameters of the work element relate to the task) is an integral part of the architecture for the specific application.

C.15.1.2 MSI

In the MSI architecture, activity is achieved by either issuing a command (for non-shop controllers) or by receiving an order (shop controller). The plans in MSI contain complete descriptions of what to do, with branches for various contingencies and types of concurrency among the branches and may require that the control entity fetch up-to-date information from the database to make appropriate decisions. The plan may be updated in real time using feedback from the monitoring of processing. The plan must exist in advance; it cannot be generated totally reactively. Task execution consists primarily of parsing and implementing plans.

MSI requires work elements but does not specify a catalog of them. Such a catalog is regarded as beyond the scope of MSI. Work elements are typically strings with semantics known only to the controller. For example, a work element string may specify an NC code program for a controller.

C.15.1.3 Comments

RCS and MSI are incompatible with regard to the specification of work elements. Some versions of RCS require the implementation to understand the semantics of the work element. In these implementations, the controller must understand what goal the work element was intending to perform. This knowledge is required if the controller must construct either its own plan or that of a subordinate. In such versions, the semantics of the work elements are part of information required for control system execution. In other versions of RCS and MSI, the plans are constructed in advance, and the semantics of the work element need not be known. At present, there are no semantics attached to work elements in the MSI information models. This incompatibility can be easily addressed by adding semantic information to the model.

C.15.2 Task Decomposition

This issue is discussed in Section 5.7.2 on page 32.

C.15.2.1 RCS

The RCS architectural specifications and methodology for task decomposition are described in Section 7.1.3.

RCS papers discussing task decomposition generally include: [Albus6, section 4], [Albus8, section “Task Decomposition”], [Horst1, sections 2.2, 3.1], [Huang1, sections 2.3, 3.3], [Lumia1, sections 3, 4], [Lumia3, sections 3, 4], [Michaloski1, section 4.3], [Michaloski2, section 1], [Quintero3, sections 2.3.4, 3.1], [Szabo3, section “RCS Hierarchical Task Decomposition”], [Wavering2, section II.2.1, V, VI].

RCS papers focusing on temporal and spatial task decomposition include: [Albus2, section “level definition criteria”], [Albus4, sections IIIB, IIIE], [Albus5, section 2.1], [Albus7, section 2.1], [Albus6, section 3], [Albus8, section “Task Decomposition”] [Lumia1, section 2.1], [Michaloski1, sections 2.3.2, 2.3.3], [Quintero1, issues MM0001, TD0001], [Quintero3, section 2.3.3].

C.15.2.2 MSI

The results of task decomposition in MSI are represented by a hierarchy of plans. The decomposition of tasks is constrained to follow the established control hierarchy. The structure of the control hierarchy is based upon an examination of many, many samples of how task decomposition would be natural for this set of machinery. Typically, a layer of control is added when there are two or more other controllers whose activities need coordinating. For example, in a cell with a robot and a machining station, it makes sense to have a controller to coordinate the activities, instead of trying to write one controller with the ability to control both machines. This case is extremely clear because the robot and the workcell are essentially independent. In cases where there is a considerable overlap between the controllers being supervised, it is not clear whether having a superior is advantageous. An example of this would be using separate controllers for separate joints of a robot finger; it just doesn't work well [Fiala3]. Of course, there is always a trade-off between clarity of system design and efficiency. Introducing an extra controller adds complexity and reduces efficiency. In MSI it is possible to have a "trivial" plan for any number of hierarchical levels. This gives a mechanism to bypass intervening levels of control without substantial task decomposition if required.

C.15.2.3 Comments

RCS and MSI are incompatible in regard to task decomposition.

RCS defines and decomposes work elements in addition to defining a controller hierarchy for carrying out the tasks specified by the work elements. The work element decomposition and the controller hierarchy are separate but closely linked.

In MSI, only the controller hierarchy is defined, and this is represented as configuration data for an implementation. There is no work element decomposition in the MSI architecture. Task decomposition (in the context of a controller hierarchy configuration) must be done by the process planner in order to generate process plans, but generating process plans is outside the scope of the architecture.

It is unclear whether a joint architecture should include methods for task decomposition.

C.15.3 Task Execution Model

This issue is discussed in Section 5.7.3 on page 32.

C.15.3.1 General Approach to Task Execution

C.15.3.1.1 RCS

In RCS, tasks are executed as the result of commands being carried out. A command may be created in an RCS controller by instantiating a step from a plan. The step identifies a work element. In one way or another (usually by putting information in a database or passing parameters in a command), the information required to define a command explicitly is put in place. Then the command is issued to the appropriate subordinate. The command itself may be as simple as GO! (or the logical equivalent), if the subordinate can do only one thing and does not require any parameters to be passed to it. Alternatively, the command may be quite long, naming a work element and giving the values of many parameters.

None of the RCS papers report making resource allocations during the process of generating a command from a plan.

Command formats are discussed in [Albus4, section IIIC], [Albus5, section 6], [Albus7, section 6], and [Murphy1, section IV.2].

C.15.3.1.2 MSI

MSI has a very definite model for developing tasks, consisting of the sequence of process planning, production planning, scheduling, and execution of production plans. Process planning is developing a description of the processing of a part. A process plan references classes of machines and does not specify times. A process plan may have several alternatives which need to be pruned later in the planning process. In production planning, lot and batch determination is made and a preliminary selection of which type of resource will be used. Finally, the plan is scheduled, a time for the execution of each step is made and an actual resource is scheduled to perform the work. In the MSI architecture, production plans are computer-interpretable by the control entity (planner/ controller).

The MSI architecture expects each controller to monitor its own progress. If it is going to be late or if a problem has occurred, the controller is expected to inform its planner or guardian through a standardized set of messages. If the problem cannot be resolved at the level at which it is spotted, it will be percolated up the hierarchy to higher levels which presumably have broader world knowledge. For details see [Wallace1].

C.15.3.1.3 Comments

The RCS and MSI approaches to task execution are largely incompatible. In a joint architecture, it is unclear whether one strategy should be chosen, or whether the two philosophies can be blended. This will be a major issue.

C.15.3.2 Command and Status Exchanges

This issue is discussed in Section 5.7.3.1 on page 32.

C.15.3.2.1 RCS

As already described, a superior controller tells a subordinate controller what to do by issuing a command. RCS also requires that each subordinate controller pass status messages back to its superior informing the superior about the performance of tasks and the operational condition of the subordinate. This command-status exchange is described or exemplified in [Albus1, section 4], [Albus3, section “Overview …”], [Albus4, page 202], [Albus5, section 1.3], [Albus7, section 1.3], [Fiala4, section 2], [Feldman1, section 4], [Horst1, footnote 12 and scattered], [Huang1, figure 6], [Leake1, section 3.1], [Lumia1, section 4], [Lumia3, figure 4], [Michaloski1, sections 2.3.10, 5.2], [Michaloski2, section 1], [Michaloski3, figure 1], [Quintero3, figures 9, 12], [Szabo3, page 265], [Szabo4, figure 3] [Wavering1, scattered], [Wavering2, section 2.1, sections V and VI].

C.15.3.2.2 MSI

The MSI architecture specifies command and status exchanges between the following architectural units: controllers and planners, subordinate and superior controllers, subordinate and superior planners, controller and guardian, planner and guardian. Each of these interfaces is specified with a fixed set of messages with a fixed set of parameters. Values of some of the parameters are enumerated sets to which the user can add (e.g. error-codes). Where the user has an option, this is indicated in the MSI specification. Although there is a suggested order in which the messages should be sent to construct a bidding and error recovery, the user is free to use the messages in other ways, if desired.

C.15.3.3 Comments

The RCS and MSI approaches to command and status exchanges are compatible, though MSI has more specific requirements. It is unclear whether MSI’s specific requirements can be met by RCS controllers. Resolution of this requires further study.

C.15.3.4 Coordination of Tasks

This issue is discussed in Section 5.7.3.2 on page 33.

C.15.3.4.1 RCS

In versions of RCS which include the requirement that controllers run synchronously, much coordination of tasks may be accomplished by building coordination into process plans. No devices such as semaphores are required in most instances. Some RCS papers, however, do mention the use of semaphores.

C.15.3.4.2 MSI

Since the execution of tasks is tied to plans, plans form the basic method for coordinating tasks. In some cases, semaphores and rendezvous are needed to synchronize tasks. The information requirements for such system-wide synchronization is provided by the plan. The semantics of the semaphore are given in the information models, the form of the semaphore (file, variable, database element) is left for the implementation to determine.

C.15.3.4.3 Comments

The RCS and MSI approaches to coordination of tasks are similar but do not seem fully compatible.

C.16 Data

C.16.1 Required Data

This issue is discussed in Section 5.8.1 on page 33.

C.16.1.1 General Approach

C.16.1.1.1 RCS

Some RCS papers contain informal descriptions of required information. A few of these describe the general sorts of information required and include specific examples [Albus2, section “Functionality of RCS levels”], [Albus4 section IV “The World Model”], [Albus5 section 2.3.1], [Albus6, section 5], [Albus7 section 2.3.1], [Albus8 section “World Modeling”], [Albus10, scattered throughout].

Several of the papers cited above propose the extensive use of maps for representing information about the world, and maps are often mentioned briefly in other RCS papers. In addition to maps, lists and state variables are often given as types of things found in the world model.

In no case does any RCS paper mention a formal information model for a world model or any of the components of a world model in any of the commonly used information modeling languages (IDEF1, NIAM, EXPRESS). For the servo and primitive hierarchical levels, data which is required has been described in English [Wavering1, throughout], [Fiala4, throughout]. Such a description constitutes a first step toward formal modelling of the information. To the extent that data structure definitions in a computer language can be termed formal information models, there are information models at the lowest tier of architectural definition of RCS implementations. These computer-language models have not been included in the RCS papers, other than “task data” in [Wavering2, section VII].

C.16.1.1.2 MSI

MSI focuses on the data which is required in a manufacturing system. The models include both persistent and non-persistent data. It is not explicit in the architecture which data must be persistent and which need not be; this is left to the discretion of the implementors. Obviously, to provide a standard which would make manufacturing systems interchangeable, this would have to be specified.

Information models also do not have temporal constraint information - for example, that certain fields must be populated at certain times, but are optional at others. This information was discussed for the data in the MSI information models, but not written down as part of the models.

The MSI architecture defines a unified production management model, which shows the relationship of resources, tooling, product description, orders, maintenance, scheduling, material handling (containers, logical and physical constructs) and process plans. The MSI architecture defines detailed models for resources, tooling, and process plans, and defines types of logical and physical unit used in part production and assembly. The MSI architecture also defines a configuration model which describes the MSI architectural entities, their interfaces and states, MSI servers (clock, database server, common memory server), the computer hosts, and the OSI network entities. The model shows the relationships among the planned configuration(s) and actual (running) configurations. It is anticipated that there be only one running configuration at a time, but that there may be many planned configurations. More detailed information may be found in [Ray1], [Barkmeyer2].

C.16.1.1.3 Comments

Information integration is critical in certain applications, such as manufacturing, where many systems other than control systems must work together. A joint architecture for manufacturing must include some provisions for information integration. Other domains must include information about the environment in which the control system operates. It is unclear whether formal information models are required in this case, but they may be desirable, in order to make explicit the assumptions being made about the environment.

Architectures assume certain functional, dynamic and information characteristics of the control system. These should be explicitly modeled in any joint architecture to prevent confusion and express unambiguously what is required and what is left for the implementation to decide.

C.16.1.2 Plans

This issue is discussed in Section 5.8.1.1 on page 34.

C.16.1.2.1 RCS

The format of process plans is discussed in [Albus5, section 6], [Albus7, section 6], [Herman1, section 7.1], [Herman3, section 7], [Horst1, appendix C], [Huang1, section 3.4], [Leake1, section 3.1, figures 12-5, 12-7], [Quintero1, issue ME0011], [Quintero3, section 4.2].

RCS does not recognize different types of plans (such as distinguishing between process plans and production plans).

For writing plans and other purposes, one branch of RCS (including what is usually called “Barbera” RCS, after Tony Barbera, one of the developers of RCS) requires that a controller be viewed as a finite state machine. In this branch, a plan is necessarily a state table or a control law algorithm. State tables are described in the RCS papers in at least three formats:

(1) an actual table with rows and columns,

(2) a state transition diagram,

(3) a “case” statement written in a computer language, such as C or Forth.

C.16.1.2.2 MSI

In MSI, every controller operates off of plans. Plans are generated in advance, but may have parameters which are passed to it in real time and require information which is up-to-date. For example, information about the status of resources may be required. The shop controller operates off of plans, but it is order rather than command driven. When an order arrives, appropriate process plan(s) are retrieved and scheduled. It is not specified by the architecture whether this functionality is within the shop controller itself or whether it resides in a separate process. The specific plan format which MSI mandates is ALPS (A Language for Process Specification) [Catron1].

MSI uses hierarchies of plans. This implies that for plans which are not at the shop level, all related plans have a common parent. ALPS uses this fact to ensure proper transmission of synchronization and parameter information.

ALPS has many features of a programming language: constants, variables, variable binding, interprocess synchronization, concurrency specification, nodes for fetching information, specifying (sets of) resources (both permanent and consumable) and error exit from plan. ALPS supports sequential, parallel, and concurrent plan segments. It supports synchronization between different branches in the same plan and in different plans in the same plan tree. The types of synchronization are: lock/unlock, signal/await, delay, rendezvous. Resource allocation can be either exclusive or non-exclusive.

Enhancements are needed to ALPS in the following areas:

(1) A general purpose symbol manipulation language is needed for expressions involving variables.

(2) Time representation should be enhanced to better represent minimum/maximum elapsed intervals, overlapping interval times, etc.

(3) Resource allocation expression needs to be broadened to permit more sophisticated forms of resource sharing.

C.16.1.2.3 Comments

RCS and MSI are inconsistent with regard to the representation of plans as well as the general approach to using plans.

Any joint architecture should provide for the types and format of plans. Once again we find that the type of plan which is appropriate depends on the area of application of the architecture and (possibly) the internals of the controller.

C.16.1.3 Resource Definition

This issue is discussed in Section 5.8.1.2 on page 34.

C.16.1.3.1 RCS

None of the RCS papers do any formal resource modeling. [Michaloski1, section 3.1] presents a “RCS Resource Taxonomy” which includes objects, agents (actor or sensor), and tools (physical or computational).

C.16.1.3.2 MSI

Resources are defined in the MSI resource model in broad categories. It was decided that developing a detailed taxonomy of resources was not fruitful; there are several classifications about.

A brief overview of the structure of resource model is as follows. Resources can be physical or logical. Logical units are such things as kits and lots. Physical resources are further broken down into permanent and moveable resources. Permanent resources are such items as machine tools, automatic guided vehicles, humans, etc. Moveable resources are such items as tools, workpieces, kits. Consumable resources include such items as machine fluid, solder, etc. In the system each permanent resource has a unique universal resource code, a status (enumeration of unknown, down, impaired, unstaffed, reserved, needs_help, ready and busy), a schedule, and a capabilities list. For the process planner, the capabilities of the resource are the key item in selecting a class of resource for a task. For the production planner and scheduler, resource status and schedule are key in determining resource availability.

C.16.1.3.3 Comments

RCS and MSI are compatible regarding resource definition since RCS is silent on the issue.

C.16.2 Data Handling Architecture

This issue is discussed in Section 5.8.2 on page 35.

C.16.2.1 RCS

Many RCS papers refer to a “global database” very briefly, without providing details of what is expected. Those papers that do provide details usually specify that there should be a distributed global database, meaning that data may reside in many different computers, but all the non-private data in those computers should be accessible by any part of the control system [Albus1, section 2.3], [Albus5, section 2.3], [Albus7, section 2.3], [Quintero1, issue WM0001], [Quintero3, sections 2.3.6, 3.8.5]. An apparently non-distributed global database is described in [Leake1, sections 3.3, 10.2].

There is not complete agreement on the use of a global database. [Fiala2, section 3] discusses the issue.

Implementations of RCS often put several processors on a bus along with a shared memory board on which is stored various kinds of information required by more than one processor, and each processor that needs the information knows the address on the memory board where to find it ([Albus1, figure 8], [Fiala3, figure3], [Michaloski2, figure 10], for example). Implementations also use NIST’s Common Memory which hides communications details from the user and provides a place where various processes can read or write into a mailbox.

RCS implementations generally seek to ensure that all the processes that need a specific kind of data can get it and can get it fast enough. None of the papers about recent (NASREM and after) implementations of RCS describe any implementation of a global database of the sort envisioned in the theoretical papers. There are no reports of the use of commercial database systems in the papers that describe RCS implementations.

C.16.2.2 MSI

The MSI architecture provides a scope for some items in the information models. The assumption of the architecture is that there is information which must be globally available (to every control entity in the system), and information which may need to be accessed by a group of control entities, and information which is truly private. The CEI specification assumes certain scoping decisions. For example, for plans the assumption is that a controller-planner pair can access the appropriate plans, but the plan can be in an entirely private place. Information which needs to be passed to superiors and subordinates is exchanged through message parameters. In contrast, to implement the synchronization constructs required by the ALPS model, the data specified for that type of synchronization must be available to all the parties which need it. Note: the parties involved in a synchronization scheme cannot be determined in advance, when the configuration is set up. They are determined dynamically according to the plans being executed. The author’s view is that every item in the information model should have a scope, or a notation that the scope of the item is left for the implementation to determine.

As discussed above, there are various sorts of permissions for data access. There are also various degrees of transparency for data access. For example, it may be physically transparent where the data resides, but conceptually, you may need to know what model the information is in. A higher level of transparency is not to need to know where the data is, either conceptually or physically. The MSI perspective is that the optimum is for implementations to be able to access information through the conceptual schema. This is typically not achievable with the technology available today because for each additional degree of transparency, there is usually a performance penalty.

C.16.2.3 Comments

RCS and MSI are incompatible with regard to data handling.

A joint architecture should provide for data handling. It is unlikely that a single model of data access permissions will be adequate. Multiple data access mechanisms are also required.

C.17 Planning, Scheduling, and Resource Allocation

This issue is discussed in Section 5.9 on page 36.

C.17.1 General Approach

C.17.1.1 RCS

RCS does not provide for the explicit separation of process planning, scheduling, and resource allocation.

C.17.1.2 MSI

As previously discussed in Section C.16.1, the MSI architecture has a specific model for the transformation of process plans to executable production plans. Process planning, scheduling, and resource allocation are separate processes which are carried out in advance of execution, although the scheduling and resource allocation can be dynamically altered during execution.

C.17.1.3 Comments

The RCS and MSI approaches are incompatible. In formulating joint architecture(s) the production process (which includes planning, scheduling, execution processes) should be made explicit, so that the assumptions hidden in the two different approaches can be identified.

C.17.2 Process Planning

This issue is discussed in Section 5.9.1 on page 36.

C.17.2.1 RCS

The format of RCS process plans was discussed in Section C.16.1.2.

Discussions of the activity of process planning in RCS systems are given in [Albus4, page 213], [Fiala1, section 4], [Fiala4, section 6], [Herman1, section 7], [Herman3, section 6], [Lumia3, sections 3, 4], [Quintero3, section 5.3.2.1], [Wavering1, section 6].

RCS allows that plans may be either devised ahead of time and called up as needed (off-line planning) or devised as needed (on-line planning). Most implementations reported in the RCS papers do off-line planning, but on-line planners are also used, particularly for systems in uncontrolled environments and for robot paths. Statements about off-line vs. on-line may be found in [Albus5, section 6], [Albus7, section 6], [Fiala1, section 4], [Herman1, section 7], [Herman3, section 6], [Horst1, section 3.7], [Huang1, section 3.3.2]. Examples of on-line planning are given in [Albus1, section 5], [Herman1, section 7], [Herman3, section 6], [Wavering1, section 6].

C.17.2.2 MSI

MSI views process planning as usually an “off-line” activity. Typically the process plans are generated using classes of machines and are made for lot sizes of one. Process planning is usually manual at present, but it is anticipated that in the future this activity will be automated. An envisioned extension of MSI is the extension of error recovery to include dynamic modifications to process plans. This is typically necessary when a part has been partially processed and the processing must be modified to permit completion of the part, or when resources which were originally going to be used are not available.

C.17.2.3 Comments

RCS and MSI are somewhat incompatible regarding how process planning is done, since MSI views it as off-line, whereas several RCS implementations do it on-line.

It is desirable that a joint architecture permit both off-line and on-line process planning.

C.17.3 Scheduling

This issue is discussed in Section 5.9.2 on page 36.

C.17.3.1 RCS

Some of the ideas of scheduling as presented in Section 5.9.2 of this report are discussed briefly in [Albus6 sections 2, 4], [Albus8, section “Timing”], [Michaloski2, section 1], and [Quintero3, section 4], but of those, only [Albus6] even uses the word “scheduling”. There is no section of any RCS paper focused on the idea of scheduling. The concept of scheduling used in MSI is not addressed anywhere in the RCS papers.

C.17.3.2 MSI

Scheduling is viewed as the last step performed before execution. The most current resource availability is used in generating the schedule. However, there is an intimate tie to the allocation of resources for the task and therefore scheduling and planning are often regarded as a unit.

When the superior planner/subordinate planner interfaces or the planner/controller interfaces are exposed, the MSI architecture requires a bidding mechanism to be supported for scheduling tasks.

C.17.3.3 Comments

RCS and MSI are incompatible regarding scheduling. Even though RCS is silent on scheduling per se, RCS systems do decide when each controller does what.

This will be a difficult area for the builders of a joint architecture.

C.17.4 Resource Allocation

This issue is discussed in Section 5.9.3 on page 37.

C.17.4.1 RCS

The RCS papers mention the allocation of resources other than processing resources (see section 5.9.3 of this report) briefly or not at all. Those that do include [Albus4, section “Hierarchical vs. Horizontal”], [Michaloski1, section 3.3.9], [Quintero1, issue TD0002], and [Quintero3 sections 3.2, 3.3]. In most RCS implementations reported in the RCS papers, resources are shared not at all, or only a very few resources must be shared, and the treatment of sharing these resources is handled on a case-by-case basis.

Some RCS papers use the term “resource allocation” to refer to allocating processes to computer hardware during system design. That issue is discussed in Section C.7.2.

C.17.4.2 MSI

MSI allows several types of resource allocation. Resources may be exclusively allocated, or they may be shared. An example of an exclusive use of a resource would be the use of a robot gripper to carry a part; the gripper is unavailable for any other task as long as the grippers are filled. An example of shared use is the use of a tray. More than one part may fit on it. The balance of the tray is an available resource even though part of it is allocated. Resources may also need to be allocated over periods of time, even though they may be idle. An example of this occurs when a part may need to be processed quickly, and a resource must be prohibited from accepting new tasks until the critical one is complete. In MSI, this type of allocation of equipment is said to be a non-preemptible resource allocation.

C.17.4.3 Comments

RCS and MSI are compatible regarding resource allocation since RCS has little to say about it.

In defining a joint architecture, it may be desirable to have an application independent tier of architectural definition below the top tier where an architecture for applications requiring resource sharing and allocation may be differentiated from an architecture for applications that do not.

C.18 Communications

This issue is discussed in Section 5.10 on page 38.

C.18.1 RCS

The RCS theory papers do not require any particular type of communications, but often ([Albus5, section 4], [Albus7, section 4] for example) state that communications must be fast enough to get messages from the output of one controller to the input of another within one clock cycle of the receiving controller. The communications model for RCS systems often includes the use of buffers which have one writer and many readers.

A model of communications is discussed in [Michaloski1, section 5.3]. A detailed report of an implementation of communications is given in [Michaloski1, section 6.3]. Parts of a communications model are scattered in [Quintero3, section 3.8]. Discussion of other detailed communications issues is given in [Michaloski2, most] and [Michaloski3, section 3]. Several of the papers just cited state that communications must be non-blocking in order to ensure that hard real-time performance can be achieved.

Communications hardware is discussed in Section C.6.3.1 of this report.

Other references to communications are made in [Fiala2, sections 3,4], [Leake1, sections 3.1, 3.3, 4.2, 8], [Michaloski1, sections 2.2.7, 2.3.10], [Murphy1, section VII], [Szabo3, section "Communications System Characteristics"], [Szabo4, section "Communication Design Details"], [Wavering2, sections II.2.4, IX].

C.18.2 MSI

The MSI architecture mandates that the command and status links among controllers be via a point to point, guaranteed message delivery paradigm and conform to the MSI control entity interface specification.

The MSI architecture assumes there is a mechanism for exchanging data among any two control entities. In present implementations of MSI, this has been accomplished using either NIST’s Common Memory or files (using NFS).

Additionally, the MSI architecture assumes that there is an (unspecified) communications mechanism to support the client/server interfaces to the database and clock.

C.18.3 Comments

RCS and MSI do not seem to be compatible regarding communications. This issue will require a lot of attention by the builders of a joint architecture.

C.19 Checks and Safeguards

This issue is discussed in Section 5.11 on page 39.

C.19.1 RCS

Brief discussions of checks and safeguards may be found in [Albus6, section 3], [Fiala1, section “Specification”], [Huang1, section 3.3.3], [Szabo2, section “Current System Description”], and [Szabo4, section “Remote Control”].

Explanations of error conditions are given in [Leake1, section 10, appendix A] and [Murphy1, appendix B.5].

A brief discussion of a safety system is given in [Albus5, section 2.6] and [Albus7, section 2.6], which are NASREM papers, but the safety system described there is not included in any of the system diagrams in the same papers, and none of the NASREM implementations describes implementing it as described.

[Huang1, section 5.3.2] discusses “vehicle guidance error control”. This discussion deals with “normal” error (excessive yaw in the heading of a mining machine, for example).

C.19.2 MSI

To provide for safety of operation, the MSI architecture provides the guardian interface for human/intelligent access to the control entity. This interface provides for immediate aborting of all tasks, and controller shutdown. It provides for stopping tasks which are in progress in a variety of ways: aborting, temporarily halting, calling a task completed.

Each plan node is tagged with a “checkpoint” attribute which indicates when the task can be interrupted and at which points the resource being used can be preempted. This aids the user in assessing the results of prematurely ending a task.

C.19.3 Comments

A joint architecture should provide explicitly for checks and safeguards.

C.20 Error Recovery

This issue is discussed in Section 5.12 on page 39.

C.20.1 RCS

Automatic error recovery for handling “abnormal” error conditions is discussed in [Albus4, section IIIB] and [Herman3, section 6]. In [Albus4] it is anticipated that if there is a subtask failure, the executor should branch immediately to a preplanned emergency subtask while the planner selects or generates an error recovery sequence. [Herman3] reports an implementation of a subtask failure re-planning software module.

C.20.2 MSI

MSI provides for recovery from scheduling perturbations, task interruptions, and equipment failure. In general, scheduling failure can be fully fixed, while equipment or task failure may require human intervention and/or redoing the process plan.

Through the CEI, tasks can be aborted, terminated (stopped and deleted), deferred (stopped with provisions for later resuming), the plan can be checked for modifications, and the task can be continued from a previous halt. In addition to commands for individual tasks, commands are provided to pause, terminate, or continue all of a controller’s current tasks.

The CEI provides for dynamic hierarchy reconfiguration by allowing the operator to drop or add a subordinate from the configuration.

C.20.3 Comments

RCS and MSI are compatible regarding error recovery. Major features of MSI were created expressly to provide for error recovery. RCS says little about error recovery, but what there is seems compatible with MSI.

The builders of a joint architecture should consider error recovery explicitly.

C.21 Desirable Characteristics of a Control Architecture

This issue is discussed in Section 5.13 on page 39.

C.21.1 RCS

The desirable characteristics of a control architecture identified in earlier sections of this report are the same as those identified in the RCS papers. Extensive discussions of these are given in [Albus3, section “Requirements”] and [Albus9, section 3]. Other discussions may be found in [Leake1, section 2.2], [Michaloski1, sections 2 - introduction, 6 - introduction], and [Quintero3, section 2.1].

C.21.2 MSI

MSI directly supports the following desirable characteristics:

(1) modularity
(2) manageable complexity
(3) fault tolerance
(4) error detection and recovery
(5) extensibility
(6) modifiability
(7) portability
(8) reconfigurability
(9) reliability
(10) reusability of software (generic controller, scheduler)
(11) understandability
(12) compatibility with existing and emerging standards.

C.21.3 Comments

RCS and MSI agree regarding the desirable characteristics of an architecture.

The builders of a joint architecture should keep these characteristics in mind.

Appendix D - Annotated Bibliography

This appendix gives a brief synopsis of a number of papers reviewed for this report. Complete citations to most of these papers are given in the list of references following the main text of this report. Only those papers which are not listed in the references have complete citations here. Text set in italics is a quote from the paper. Some parts of the synopses are in outline form.

Five classes of related papers are identified immediately after the author identifier: **AMRF**, **CIM-OSA**, **MSI**, **NGC**, and **RCS**. Not all papers fall in one of these classes.

[Albus1] **RCS** ***A Control System Architecture for Multiple Autonomous Vehicles***

This paper describes the application of RCS to a pair of independently controlled underwater vehicles. The paper describes what it was intended to do, not a completed project. It was planned to use RCS for high-level control of two EAVE vehicles *"to control the overall vehicle systems to perform a set of demonstration mission scenarios which require the two vehicles to cooperate in accomplishing a task"*. The two scenarios are (i) cooperative search and approach and (ii) cooperative search and map. A non-RCS architecture was also to be implemented so a comparison could be made.

The University of New Hampshire provided the autonomous undersea vehicles, equipped with knowledge based systems for control.

Section 3 of the paper (subsections 2.1 - 2.4) describes *"the MAUV system architecture"*, which is largely RCS, including task decomposition, planners, executor, world modeling, sensory processing, global memory. This paper differs from some other RCS papers in including a planner manager. Also, emphasis is placed on the evaluation function of the world modeler, which is to provide value driven decision logic.

Section 4 describes the levels of the control hierarchy. In this RCS hierarchy, level 4 processes single vehicle tasks, and level 5 processes tasks involving more than one vehicle. Level 6 was used to decompose a mission into a set of tasks for multiple vehicles.

Section 5 describes *"Real Time Planning in the MAUV Hierarchy"*, but it is not described how independent real-time planners can implement coordinated action or what the communications requirements for the implementation are. The two vehicles do not appear in the figures to have any method of communicating in any way while underwater. The two submarines may have to rendezvous to communicate world model updates.

[Albus2] **RCS** ***RCS: A Reference Model Architecture for Intelligent Control***

This paper describes RCS apart from any specific application, although a list of applications is given on the last page.

RCS is described as (p. 59) *"a canonical form, not a design specification"*.

The paper describes four versions of RCS:

(1) RCS1 - state-table-based,

(2) RCS2 - state-table-based with image processing added,

(3) RCS3 - task decomposition (TD) is decomposed to JA-PL-EX. NASREM is an RCS3.

(4) RCS4 - adds value judgment and *“sophisticated multilevel tracking filter interaction between sensory processing and world modeling*” (p. 57).

Figure 1 shows RCS3 as applied to a machining workstation. This includes the bottom five of a proposed 7-level hierarchy: shop, cell, workstation, equipment, e-move, primitive, servo. All seven levels are described briefly (p. 58), including what the world model might contain at each level.

Brief mention is made of temporal and spatial decomposition.

Communications (p. 57) *“are typically implemented through common memory or message-passing*” at low levels, through a backplane bus at middle levels, and via bus gateways and LAN’s at high levels.

[Albus3] **RCS** ***A Reference Model Architecture for ARTICS***

“This article advocates the development of a reference model open-system Architecture for Real-Time Intelligent Control Systems (ARTICS) as a means to accelerate the pace of technological development in automation and robotics” (page 182).

As described in the second section *“ARTICS Vision”*, ARTICS is identical to RCS, which is described in the third section. However, the paper is an invitation to other parties than RSD to participate in defining ARTICS. *“RCS is suggested here as a starting point for discussion of ARTICS concepts*”. The target architecture should be developed by cooperative efforts of industry, academia, and government.

As envisioned, the reference model would include hardware components, software components, communications protocols, and application development tools. It is envisioned to be an evolving specification.

The paper says the ARTICS architecture should:

(1) provide extensibility in functionality,

(2) provide extensibility in temporal range,

(3) provide flexibility in human and computer interfaces,

(4) support varying levels of complexity and combinations of autonomy and human control,

(5) provide for real-time control system hardware and software,

(6) support distributed control systems,

(7) be fault-tolerant,

(8) provide application-independence,

(9) provide software portability and interoperability,

(10) include CASE tools and a selection of simulation and animation tools.

Figure 4 in this paper shows a classification of *"automation flexibility"* using three dimensions:

(1) autonomy,

(2) processing and communications power,

(3) sensory interaction.

ARTICS is intended to handle any level of automation. Nine varieties are listed:

(1) teleoperation and remote control,

(2) computer-aided advisory control,

(3) traded control,

(4) shared (simultaneous) control,

(5) human override,

(6) human-supervised control,

(7) autonomous control,

(8) sensory interactive control,

(9) mixed mode control.

The paper calls for additional research on real-time reasoning (fast enough to keep up) and temporal reasoning (reasoning about time).

The paper calls for coordinating existing research and lists 23 relevant research areas.

[Albus4] **RCS** ***A Theory of Intelligent Systems***

This paper presents a theory of intelligent systems *"largely based on the Real-time Control System (RCS) that has been implemented [at NIST]"*. The paper uses examples from machining and autonomous navigation paradigms.

Section II gives *"the elements of intelligence"*:

(1) actuators,

(2) sensors,

(3) sensory processing,

(4) world model,

(5) values,

(6) task decomposition.

Section III gives *"the system architecture of intelligence"*. In this, commands and status are strictly hierarchical, but sensory processing is a layered graph, and data sharing may be horizontal. A seven-layer hierarchy with temporal and spatial decomposition is given in Figure 4. Task frames, command frames, and task decomposition are discussed.

Section IV discusses *"the world model"*. At every node at every level of the control hierarchy there is a world model and a knowledge database. *"WM modules provide memory, communication, and switching services that make the world model behave like a knowledge database in response to queries and updates from the BG, WM, SP, and VJ modules. … Together the WM and KD [knowledge database] modules make up the world model."* The world model makes heavy use of maps and map overlays as well as entities. Entities are objects which the intelligent system knows about. In this document, they are technically defined as an element from an infinite enumerated set. The information for an entity can be expressed in a frame representation, giving the *"important"* properties. The entity database is hierarchical. Each entity has both a parent and children. An intelligent system also must have a representation for events. Events may also be represented as frames with such attributes as: kind, type, modality, time, interval, position etc.

A second Section IV (this section number is used twice) deals with sensory processing. Sensory processing consists of gathering sensory input and integrating it into more abstract information units. Subsections are:

A. Measurement of Surfaces,
B. Recognition and Detection,
C. The Context of Perception,
D. Sensory Processing SP Modules,
E. World Model Update,
F. The Mechanisms of Attention,
G. The Sensory Processing Hierarchy,
H. Gestalt Effects,
I. Flywheeling, Hysteresis and Illusion.

The paper assumes the perceptual hierarchy corresponds to the control hierarchy.

Section V discusses value judgments.

Error recovery is built into the planning at each hierarchical level. If an error is encountered which cannot be handled at that level, the error is propagated up to the next level. Errors are recognized by comparing the expected value of the world model to the actual one recognized by sensors (at that level).

[Albus5] **RCS** ***NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)***

This is the fullest description of the NASREM architecture. The document "*is to be used as a reference document for the functional specification, and a guideline for the development of the control system architecture, of the IOC Flight Telerobot Servicer*". (a component of a NASA space station). The document does not state that the architecture is intended for any other application, although other documents about NASREM do, e.g. [Lumia1, page 303].

"The NASREM control system architecture is a three legged hierarchy of computing modules, serviced by a communications system and a global memory, and interfaced to operator and programmer workstations" (p. 5). The three legs are task decomposition (H modules), sensory processing (G modules), and world modeling (M modules); these are described as having the usual RCS functions.

It is stated that the architecture is composed of a set of standard modules and interfaces. However, the document also discusses communications paradigms and data access paradigms.

The overall architecture is described in section 2. Except for section 2.5, which covers the operator interface and makes occasional reference to space station things, the section is application independent. Section 2.1 covers task decomposition briefly. World modeling is covered in section 2.2, global memory in 2.3, sensory processing in 2.4, and safety system in 2.6.

The safety system has access to all information in the world model of the control system and also maintains its own world model, which is updated redundantly. The safety system is required to prevent the machine system from entering forbidden volumes in physical space and state space.

Section 3, *"Levels in the Control Hierarchy"*, is very application-specific. The description of the levels given in section 1 (pp. 1 - 2), however, is general:

"The [NASREM] architecture is hierarchically structured into multiple layers, as shown in Figure 1a, such that a different fundamental mathematical transformation is performed at each layer. At layer one, coordinates are transformed and outputs are servoed. At layer two, mechanical dynamics are computed. At level three obstacles are observed and avoided. At level four, tasks on objects are transformed into movements of end effectors. At level five tasks on groups of objects are sequenced and scheduled. At level six objects are batched into groups, resources are assigned to worksites and parts and tools are routed and scheduled between worksites."

Section 4 describes communications. For communications, *"it is conceptually useful to think of passing variables through a global memory*". *"One possible implementation"* would use a state clock which sets a time interval for the system during which variables are locked from update. Each variable consists of an input and an output box. The output box has a compute flag on it which signals whether data may be moved into the output variable. When this flag is not set, the communications system moves this data.

At the beginning of section 4 the intended level of granularity is given (p. 4): *"The H, M, and B modules at all levels of the NASREM architecture can be viewed as state machines which periodically read input variables, compute some function of their input and state, write output variables, and go to a new state.*"

Section 5 describes the *"Detailed Structure of the H Modules"*. This includes the usual RCS decomposition in JA, PL, and EX. There is only one JA per H module. JA is responsible for parsing a task command into spatially or logically distinct jobs to be performed by physically distinct planner/executor functions. PL decomposes job commands into *"a temporal sequence of planned subtasks."* Planners construct plans over the permitted time horizon, and perform cost-benefit analysis of the plan. EX is responsible for carrying out the plan generated by the planner. Queues of subtasks are permitted. Executors can request world model information and report status to higher levels (which component of the upper level task decomposition module is not discussed).

Section 6 describes *"Tasks and Plans". "A task is an activity which begins with a start-event and is directed toward a goal. A goal is an activity which terminates a task.*" Several different descriptions of what a plan might be are given. It is provided that *"A plan can be represented in a number of different notations.*" Gannt Notation is described in section 6.1, State-Graph Notation in section 6.2 Plans may also be represented as Petri nets or finite-state-automata grammars. Plans may involve scheduling of parallel subtasks on different machines, *"conditional branching, and probabilistic decision rules. Plans may also include provisions for error correction activities and lack of progress toward a goal.*" Planning may be done off-line or in real time.

Section 7 describes the possible computer hardware and timing of *"An Example Implementation".*

Sections 4 - 7 are application-independent. Section 8, *"A Functional Description of Control Levels"* describes how NASREM might be applied in specific instances of space robotics.

Information flow is described as follows:

(1) Command and status are hierarchical.

(2) Other data flows horizontally between modules in the same level of the hierarchy.

(3) Information flow other than command and status between levels is not prohibited. The requirement is that all information reside in global memory variables.

It is specified that an operator interface be provided at each level of the hierarchy. Through the operator interface, it is possible to:

(1) monitor a process,

(2) insert information,

(3) take control of the task,

(4) make judgments about sensory processing and world modeling.

The operator interface is constrained by synchronization and data integrity constraints. Specific operator operations are defined for each control interface level.

[Albus6] **RCS** ***RCS: A Reference Model Architecture for Intelligent Machine Systems***

Section 1 describes the evolution of RCS.

"In RCS-1 the emphasis was on combining commands with sensory feedback so as to compute the proper response to every combination of goals and states…RCS-1 was implemented as a set of state-machines arranged in a hierarchy of control levels."

"The new feature of RCS-2 was the inclusion of … a number of sensory processing algorithms …"

"The principle new features introduced in RCS-3 are the World Model and the operator interface."

"The principle new feature in RCS-4 is the explicit representation of the Value Judgment (VJ) system."

Section 2 gives *"A Machining Workstation Example"* of RCS-3 for a workstation containing a machine tool, part buffer, and robot with vision system. A seven-level control hierarchy is described. The characteristic time, the nature of commands, and the world model contents are described at each level.

Section 3 describes the *"organization and timing in the RCS hierarchy."*

"Levels in the RCS command hierarchy are defined by temporal and spatial decomposition of goals and tasks into levels of resolution, as well as by spatial and temporal integration of sensory data."

Section 4 describes task decomposition, including task frames and planning.

Section 5 covers world modeling. *"The world model is an intelligent system's internal representation of the external world … The world model also includes knowledge about the intelligent system itself."*

In the knowledge database of the world model, *"Knowledge about space is represented in maps. Knowledge about entities, events, and states is represented in lists and frames. … Information in the world model knowledge database may be organized as state variables, system parameters, maps and entity frames."* Examples of these four types of data at five hierarchical levels are given.

Section 6 covers sensory processing. *"The function of sensory processing is to extract information about entities, events, states, and relationships in the external world, so as [to] keep the world model accurate and up to date."* The sensory processing system does map updates, recognition, and detection. *"Each SP module at each level consists of five types of operations:*

(1) Coordinate transformation,

(2) Comparison,

(3) Temporal Integration,

(4) Spatial Integration, and

(5) *Recognition/ Detection."*

Section 7 covers value judgments. *"Value judgments evaluate the cost, risks, and benefits of plans and actions, and the desirability, attractiveness, and uncertainty of objects and events."*

Concluding remarks include: *"RCS is not a system design, nor is it a specification of how to implement specific systems. … A reference model architecture is a canonical form, not a system design specification."*

[Albus7] **RCS** ***Mining Automation Real-Time Control System Architecture Standard Reference Model (MASREM)***

This paper describes an adaptation of NASREM to coal mining. Much of the material is identical to that found in [Albus5], which is the primary description of NASREM. The sections which are mostly the same are 2, 4, 5, and 6.

The differences between this paper and [Albus5] are in the application-specific sections. Sections 1 and 3 of this paper describe a 7-level control hierarchy for coal mining. The hierarchy is shown in figures 1.2 and 3.1.

A detailed glossary is included.

[Albus8] **RCS** ***NASREM The NASA/NBS Standard Reference Model for Telerobot Control System Architecture***

This paper gives a history of the development of RCS and gives related references.

The first half of this paper describes NASREM and is essentially a summary of [Albus5].

The abstract notes *"There are five major elements required for the development of an intelligent robot system. Four of these are architectures:*

(1) Conceptual architecture,

(2) Functional architecture,

(3) Software architecture,

(4) Hardware architecture.

The fifth element is the software development environment."

The second half of the paper describes the servo level of an experimental implementation built at NIST.

Care is taken to create an interface which supports the standard algorithms for robotic manipulation. This was done by taking each algorithm for servo control, splitting it into the (functional) parts which belong to the task decomposition, world modeling, and sensory processing modules and deriving the interfaces which will support these algorithms.

A similar analysis has been done at the primitive level, and is planned for other levels.

[Albus9] **RCS** ***Concept for a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)***

The content of this paper (advocating the development of ARTICS) is the same as that of [Albus3]. Most of the text here is the same as or similar to the text in that paper.

This paper refers to RCS briefly, but does not have a discussion of it of the sort included in [Albus3]. All three figures in this paper appear in [Albus3].

The paper includes a few details of a survey about the need for ARTICS and an appendix listing potential reference model components which is not in [Albus3].

[Albus10] **RCS** ***RCS: A Reference Model Architecture for Intelligent Vehicle and Highway Systems***

The first section describes RCS briefly, showing the architecture for the Army Robotics Testbed vehicle as an example.

The bottom layer consists of actuators and sensors. At the top of the hierarchy are goals from a higher level controller to groups of vehicles. The control system is designed for closed loop control. That is, sensory feedback interacts with goal decomposition at each level to correct for disturbances.

The second section gives a 7-level RCS architecture for an intelligent vehicle and describes briefly for each level what the level does, what sort of data is in the world model at that level, and what sensory processes go on at that level. The levels of the hierarchy, from the bottom up, are:

(1) actuator servos,

(2) steering dynamics,

(3) steering and attention coordination,

(4) vehicle path planning,

(5) road segment planning,

(6) trip segment planning,

(7) trip destination planning.

The third section gives a 7-level highway control system architecture with the same sorts of information as the second section. The hierarchy is:

(1) high bandwidth communications: to modulate communications transmitted for individual vehicles,

(2) signal control: produces signals to control traffic lights and generate the desired messages displays,

(3) lane control,

(4) road section,

(5) segment control: of a particular road segment for a limited duration,

(6) region control: of a specific region of the highway, limited in duration,

(7) statistical input regarding traffic statistics from level 6 modules.

[Albus11] **<u>RCS</u>** ***Toward a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)***

This is a straightforward condensed version of [Albus9].

[Auslander1] ***Real Time Control Software for Manufacturing Systems***

Defines the term “real time” well.

“Real time software is characterized by the following criteria:

1. *Delivery of the result at the right time is critical to correct system operation (rather than a convenience).*
2. *[The] software sequence must be responsive to events in the physical world (i.e., outside the computer).*
3. *The operator (if there is one) must be able to interact with the software in a substantive way, without interfering with control activities.*
4. *A number of semi-independent activities must be coordinated.”*

The paper uses the example of controlling a saw to describe real-time systems.

This paper is an excellent introductory-level (for technical people) presentation of real-time control problems and control systems. It discusses control hierarchies and has an excellent discussion of synchronous vs. asynchronous control. Varieties of multitasking are discussed. The paper discusses communications issues: synchronization, shared memory, and networks. Programming languages and CASE tools are described briefly.

[Biemans1] ***A Systems Theoretic View of Computer Integrated Manufacturing***

This paper discusses properties of CIM architectures. It gives an example of comparing the relative merits of two architectures and offers general rules for evaluating architectures, such as:

1. Avoid unnecessary complexity.
2. Define architectures unambiguously.
3. Specify architectures generically.
4. Develop specific systems from abstract architectures (only two levels of architectural definition are discussed: functional or task description architectures and physical implementation).
5. Use decomposition.
6. Separate independent concerns.
7. Define and analyze interaction of components.
8. Analyze how variation in components affects the performance of the organization as a whole.
9. Describe relevant information flows.

The thesis is that *"good"* CIM architectures specify a production organization unambiguously at a high level of abstraction and in generic terms, as a configuration of components. The architecture should allow us to understand how variations in each of the components affect the system as a whole.

The authors emphasize that a statement of the purpose that the CIM architecture serves is a highly necessary component of a successful CIM architecture which is often omitted.

A systematic approach for designing a reference model is described. The steps are as follows:

(1) Describe the target organization with respect to its interactions with its environment.

(2) Identify components of the organization.

(3) Separate tasks into as orthogonal concerns as possible.

(4) Describe interactions between separate tasks.

The article goes on to demonstrate the construction of a reference model for a production organization. The model developed recognizes task planning, management, and execution functions as distinct. Specific functions for each hierarchy level for the executor are given. The manager's function is to adjust the task of the executor based upon changing conditions. Subfunctions of the manager include that of the master planner (interfacing with business concerns such as product portfolio, production capacity or production costs), product and process development (development and selection of production plans), execution supervision, and monitoring.

The reference model includes the capability to reconfigure the tasks, resources, physical layout, etc. of the organization. This is a more demanding requirement than dynamic reconfiguration, which is also expected, since dynamic reconfiguration does not include defining new tasks.

This paper is more inclusive than most of the architecture papers reviewed, in that it includes the ideas in the previous paragraph, while most other do not.

The paper is well-written, cogent, and convincing.

[Biemans2] ***Reference Model for Manufacturing Planning and Control Systems***

This paper describes in detail the Manufacturing Planning and Control System (MPCS) which is a top-down design and an abstract specification of an integrated manufacturing planning and control system. This architecture is said to span from enterprise to actuator.

"A MPCS is decomposed into a 'factory controller' concerned with the negotiation task and a 'shop' concerned with the processing task. Figure 4 illustrates the decomposition. Factory controller and shop are distinct components that are simultaneously operational and that interact by exchanging messages."

The philosophy is to specify the behavior of the components, their purpose, and a structure that defines which components may cooperate. Tasks give the goals of actions, and the structure shows how each component contributes to the organization. Defining *"behavior"* includes defining the inputs and outputs of the module and specifying the command and status messages possible between components and their temporal ordering.

An MPCS system:

> *"1. aims at earning a target amount of money and gaining a target market by buying and selling certain types of products in certain target numbers."*
>
> *"2. negotiates the exchange of products for money and exchanges products for money with customers and suppliers.*"

Figure 5 shows the *"Reference Model for MPCS at a Glance.*" The model has nine levels: company, factory, shop, workcell, workstation, automation module, equipment, device, sensor or actuator.

The paper states that consistency is the primary design principle for human understanding of the system and underlies all principles of quality of the system. Resultant quality characteristics are: separation of concerns, generality, and propriety. Propriety is the statement that inessential tasks should not be introduced.

In the view of this author, the fact that certain information should be shared is required by the architecture, but not how it should be shared. Each component has multiple layers of decomposition, but it appears that only adjacent layers know about the subordinate layer's internal structure.

The paper contains a detailed list of the functions of the shop controller. Functions include:

(1) controlling the amount of inventory of some key parts to be able to dispatch products within the procurement lead time,

(2) managing time lags to acquire stocks,

(3) managing stocks with respect to cost and risk of under- and over-estimating the cost and utility of maintaining stocks.

This functionality is typically embodied in a manufacturing resources planning system, but manufacturing resources planning systems do not usually take into account demand forecasts and limits in processing capabilities, nor do they aggregate demands for individual products to reduce variance in forecasts of demand.

The workcell controller is the portion of the controller which is responsible for executing commands from the shop controller. It determines who should perform part processing and how this processing should be scheduled.

Workcells are formed on the basis of how frequently they need to exchange parts--that is on the flow of parts through the factory. This contrasts with the MSI and RCS view, where grouping is based the resource’s involvement in decompositions of common tasks. Whether these are in fact equivalent is unclear.

This architecture clearly limits the information to the module which needs it. No discussion is given to shared information.

The proposed model is just the control hierarchy. The paper does not deal with other issues: tasks, concurrency, resource allocation, etc.

[Bohms1] ***RIA: Reference Model for Industrial Automation***

This is an excellent meta-level paper for CIM architecture.

The paper points out the need to model CIM modeling itself. It presents two key components a CIM reference Architecture:

(1) a blueprint for applying CIM in discrete parts manufacturing,

(2) a language for expressing the architecture (called a base model).

The authors see the industrial enterprise as a real-system/information system combination. They contend that due to the unavailability of standards, information exchange required for the implementation of just in time (JIT) technology may be impossible.

The descriptive framework (section 3) is particularly strong. Nine dimensions for modeling CIM are identified:

(1) modeling level (one of: reality, models of reality, models of models),

(2) language level (level of modeling language used),

(3) aspect (set of views, e.g. functions, information, resources),

(4) composition (global to detailed),

(5) scope (type of activity),

(6) representation (modeling language used),

(7) product life cycle (design, production, maintenance, etc.),

(8) actuality (to be vs. as is),

(9) specification level (generic to fixed - how much choice left).

Section 4 proposes decompositions of each of the nine dimensions into points or regions. For example, the modeling level dimension has three points: CIM Framework, CIM Models, CIM in Practise.

Section 5 presents initial ideas for a CIM reference architecture, with the beginnings of a language for expressing it. The authors conclude “*It is our intention to develop a complete CIM reference architecture expressed in a CIM Base Model that covers all dimensions*.”

[Boykin1] ***CAM-I CIM Reference Model***

This paper presents an extremely brief overview of the CAM-I CIM reference model. The application scope was oriented to discrete parts, the life-cycle scope was product R&D, marketing, production, and field support, and the organizational scope was organization-wide. Attachment A (a single figure) shows a very high-level view of the architecture.

[Brooks1] ***Elephants Don't Play Chess***

Sections 1 and 2 of this paper discuss approaches to the control of robots.

Section 3 discusses the *"physical grounding hypothesis"* and presents the *"subsumption"* architecture. The subsumption architecture *"emphasizes ongoing physical interaction with the environment.*"

"The [subsumption] behavior language groups multiple processes (each of which usually turns out to be implemented as a single AFSM [augmented finite state machine]) into behaviors. There can be message passing, suppression, and inhibition between processes within a behavior, and there can be message passing, suppression, and inhibition between behaviors. Behaviors act as abstraction barriers; one behavior cannot reach inside another."

Section 4 describes the performance of seven robots constructed using subsumption architecture. The robots have behaviors such as moving around without bumping into things, hiding, and seeking and taking soda cans.

[Chen1] **<u>CIM-OSA</u>** ***An Integrated CIM Architecture - A Proposal***

This paper argues that an architecture should include both an architecture design specification and a method for building instances.

Proposes the *"GRAI Integrated Method (GIM)"*, which combines with the CIM-OSA architecture.

The paper is extremely general.

[Davis1] ***Generic Architecture for Intelligent Control Systems***

The section on *"Functional Description of the Generic Control Module"* gives the four major functions of a controller as: assessment, optimization, execution, and monitoring. Error recovery is considered in the section *"What happens when a deviation occurs?"*.

This paper explicitly builds on [Jones4].

[Dilts1] ***The Evolution of Control Architectures for Automated Manufacturing Systems***

This paper gives a comparison of architectures for automated manufacturing systems. It discusses robot control architectures for manufacturing briefly.

Issues discussed include:

(1) reliability,

(2) fault tolerance,

(3) error recovery,

(4) modifiability,

(5) extensibility,

(6) reconfigurability,

(7) adaptability,

(8) resource allocation.

The paper does not discuss data handling as separable from control. It assumes data flow is similar to control flow.

The paper discusses four main architectural approaches:

(1) centralized,

(2) proper hierarchical,

(3) modified hierarchical,

(4) heterarchical.

The relative merits of each type for handling the issues listed above are discussed, including a table of them.

The paper seems enthusiastic about heterarchical architecture but admits that there are unsolved issues (interprocess communications, bandwidth, unavailability of commercial software) and does not discuss the problems of scheduling and figuring out what a heterarchical system will do.

[Dornier1] ***Evaluation of Standards for Robot Control System Architectures - Final Report Executive Summary***

The intent of the reported study is to: “*Identity and evaluate existing control architecture concepts w.r.t. their suitability as a reference model for European space Automation and Robotics (A&R) control systems*.” (p. 8).

The need for a unified robotics control architecture arises from:

1. Complex controllers will have a lifetime of up to 30 years.
2. Sophisticated control and intelligence can’t be provided on board at this time.
3. Control systems will be developed in a multi-vendor environment.

Their idea of a reference architecture includes:

1. unambiguous and unified vocabulary,
2. modular partitioning of development tasks,
3. reusable modules which are easy to interface to,
4. ability to incorporate emerging technologies in the architecture.

Includes some discussion (in sec. 4) of what a functional reference model is:

“A functional model of a control system is a description of the functional as well as information architecture of the control system reflecting the breakdown and interrelation of all subfunctions necessary to hierarchically decompose a global goal (task input) down to a level of elementary executions.” (p.23).

Four different viewpoints of the functional aspects of an architecture are given:

(1) function - what has to be done,

(2) application - why it has to be done,

(3) operational - by whom and where it must be done,

(4) implementation - how it is done.

The paper evaluates NASREM and alternative architectures including a representative industrial robot controller (sec. 3.3.1), ESPRIT 623 architecture (sec 3.3.2), IMAS (sec. 3.3.3), and Intelligent Robot Control (sec. 3.3.4).

NASREM strengths: complete, well-documented, hierarchical structure strong, task decomposition well defined.

NASREM weaknesses: higher levels not yet elaborated, structure of global data system is unclear, ambiguous information architecture, concept and structure of safety system is unclear.

Industrial Robots: They found that industrial robots were special purpose, proprietary, undocumented, and unextendable.

ESPRIT 623 strengths: generally applicable, integrated error recovery planned for, good growth potential.

ESPRIT 623 weaknesses: unclear general architecture, missing execution functions, missing sensor data processing functions, missing project documentation, specific to automatic operation.

IMAS strengths: good hierarchical structure, forward control, nominal feedback very well described, world modeling well defined, re-planning is planned for.

IMAS weaknesses: information architecture not documented, lack of overall architecture documentation, generality of architecture is questionable, non-nominal feedback not well separated from the rest of the architecture.

The paper reaches the conclusion that no existing architecture is suitable for the intended use and proposes a new architecture. The new architecture is summarized in this paper and presented in detail in [Dornier2] (see next entry).

The paper proposes documentation requirements for reference models briefly in section 4.4 (page 32).

[Dornier2] ***Baseline A&R Control Development Methodology Definition Report***

This (long and detailed) paper proposes a reference architecture for European space Automation and Robotics (A&R) control systems. The architecture is intended to be suitable for at least robot systems, surface roving vehicles, and dedicated automation equipment.

A three layer hierarchy (A&R Mission, Task, and Action) is proposed (figure 5.3.2-1, page 52). The use of a hierarchy is justified in 5.1.2.1, but no rationale is offered for why three layers are suitable, rather than four, six, or a variable number depending upon the application.

Each controller has three major modules: nominal feedback functions, forward control functions, and non-nominal feedback functions (figure 5.4.1-1, page 53). No rationale is offered for this decomposition in this paper, but other papers in the same series are referenced.

Forward Control Functions:

(1) partitioning incoming commands into jobs for subsystems,

(2) decomposing each job command into a logical sequence of subtasks, with appropriate timing constraints,

(3) identifying execution functions and directing subdominants to the servo level.

Nominal Feedback Functions:

(1) sensor data measurement functions (at the lowest levels),

(2) (logical) sensor data processing functions (filtered and processed sensor data),

(3) process oriented sensor data processing functions (transforming sensor readings into quantities meaningful to the application process),

(4) control oriented data processing functions (providing processed information based on a knowledge of control algorithms).

Non-nominal Feedback functions

(1) monitoring functions for both Nominal Feedback and Forward Control Feedback functions,

(2) failure diagnosis functions,

(3) failure identification functions,

(4) failure recovery functions.

The interconnections between layers and modules are so numerous that control system behavior and performance may be hard to predict. The ability of non-nominal feedback to do planning and give *"directives"* to forward control looks particularly like a troublemaker.

A highly-structured methodology for architectural development is presented. The methodology uses the Structured Analysis and Design Technique (SADT). The steps and documents required by the methodology are shown in figures 3.3-1, 3.3-2, and 3.3-3 (pages 17 and 19). The documents include:

(1) Activity Script,

(2) Application Architecture,

(3) Operations Architecture,

(4) Logical Model of Control System,

(5) Functional Reference Model (FRM),

(6) Application Reference Model (ARM),

(7) Operations Reference Model (ORM).

The Activity Script is prepared using an Activity Analysis Methodology (ActAM) and written in an Activity Scripting Language (AcSL) which has not been rigorously formalized. The control development methodology (CDM) is discussed in detail in section 8.

Appendix 2 gives the definitions of many tasks and some activity scripts, as examples.

State information flows both up and down the hierarchy.

The paper does not deal much with database, communications, resource allocation, or reconfiguration issues. There is no discussion of world modeling.

A detailed glossary is included. In general, definitions and the use of defined terms are handled quite well.

[Duffie1] ***Nonhierarchical Control of Manufacturing Systems***

This paper is a brief work in progress report on a heterarchical control system for a work cell, including a milling machine, a vision system, a robot, some part buffer stations, a simulated mill, and a simulated wash station.

Issues which must be addressed in a nonhierarchical architecture are:

(1) performance of commercially available communications networks in regard to the capacity and response time in supporting such intensive messaging,

(2) optimality of totally distributed scheduling,

(3) real-time optimization of the system,

(4) deadlock avoidance, detection and resolution.

[Duffie2] ***Nonhierarchical Control of a Flexible Manufacturing Cell***

This paper is a brief report on a heterarchical control system for a work cell, including a milling machine, a vision system, a robot, some part buffer stations, a simulated mill, and a simulated wash station. The paper is much like [Duffie1].

The paper describes the implementation of centralized, hierarchical and heterarchical prototypes and gives a comparison of their performance.

Results are said to show heterarchical advantages of "*increased fault-tolerance, inherent adaptability and reconfigurability, decreased complexity, and reduced software development cost.*" With regard to optimization, the paper says "*the fundamental objective of maintaining local autonomy contradicts objectives of optimizing over-all system performance,*" and says optimization techniques are needed. The paper reports that "*network communications and multi-tasking operating systems were required for the heterarchical control systems and hence are quite complex.*"

The control system is said to use negotiation, but is not said to include bid and contract procedures.

Information is kept locally rather than globally.

[Duffie3] ***Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities***

The paper gives a list of design principles for producing a system of cooperating autonomous entities (i.e. a heterarchical system) with a high level of fault-tolerance.

Types of system entities:

(1) material handling robot entity software,

(2) part processing entity software,

(3) pallet entities,

(4) human entity software,

(5) manufactured part entity software.

The human entity is intended to get the system working again when it breaks or otherwise fails to work right.

Control is accomplished by "*dynamically negotiated transactions*" among entities. Each of these entities communicates with each other, asking for services. Contracts are not cemented until the last possible moment. The requestor queries all parties, then evaluates responses and reserves a space in the schedule. Information about the part is kept with the part entity. Information about the machine tool is kept with the machine tool entity. The paper claims that this eliminates the need for complex databases which can have access and consistency problems.

Software entities consist of a communicator and a controller. The controller software implements the control logic for each entity. It is not clear from the article where the control logic comes from, if it is predetermined or parsed in from some sort of plan. However, it does state that this software is responsible for selecting the machine code which must be run. The communications aspect is handled by a general-purpose distributed operating system which is independent of the manufacturing software and is generic. The communications code of the entity can receive asynchronous messages while the controller is executing its required steps. The communications code is event-driven. Examples of messages are reservation transactions, fault announcements or controller status messages.

To provide for system fault-tolerance, an entity is not required to respond to any message it receives and each entity should assume that transmitted messages will not be responded to by other entities. This ensures that there are no master-slave relationships between the entities.

"In heterarchical systems there is one generic type of system-level fault: a failure of entities to establish, maintain, and terminate the relationships required to achieve the goals of the system. A heterarchically controlled system inherently tolerates faults in its entities because of its high level of local autonomy. Within individual entities, high local reliability can be achieved using conventional hardware and software design techniques. This ensures that local faults will be minimized."

The paper gives the example of fault recovery from the state "*wait for message from processing entity indicating processing complete*." The entity will check to see when it has been too long since the message arrived. When it realizes this, it transmits (broadcasts) a fault message. It recovers from the fault by continuing to wait for the desired message, and awaits advice from another entity. Sample responses include: from human, *"continue"* or "*go to output station"* assuming some manual intervention has occurred. This does not prohibit other responses for other actions from being used.

The communications network is used by entities to dynamically negotiate transactions with other entities for purposes of real-time scheduling of part processing, part transportation etc. This dynamic resource allocation makes the control system tolerant of faults such as unavailable machines.

A prototype system has been constructed at the University of Wisconsin at Madison including a machining cell (with a machining center, a robot, some part buffer stations, and several simulated stations), and an assembly cell with a pallet transportation system and three robots, all connected by a local area network.

The entities have keyboard and voice recognition/generation capabilities. Humans rove through the system and interact with the systems as a peer. The human is not required to interact with the systems and they are not required to acknowledge or act on the advice of the human. The distributed operating system for IBM-PCs was called MULTI and allows concurrency for 250 Pascal programs on networked PCs.

The conclusion to the paper notes: "*Local decisions made by entities are not globally optimal, and real-time optimization entities may need to be developed that can collect information in a heterarchical system and influence the operation of other entities without taking control. Deadlock avoidance is also an important issue."*

[Feldman1] **RCS** ***A Submarine Maneuvering System Demonstration Using a Generic Real-Time Control System (RCS) Reference Model***

This paper gives a very brief overview of RCS and focuses primarily on the simulation and animation techniques used in the submarine project.

The paper highlights the usefulness of having a simulation environment in which to test control systems (although the paper itself discusses this only briefly in section 5).

The system is designed to perform navigation in a hostile environment while compensating for naturally occurring temperature and salinity perturbations. This application requires a high degree of autonomy.

This implementation of the architecture uses a clock to regulate the execution cycle. Each controller is built using a generic controller template and generic functions. For example, this application has depth control and Dive/Rise control. Each controller is a closed loop control unit. Shared memory or Common Memory is used to move data between the modules.

Simulation permits the asking of *"what if"* questions. Parallel to the controller structure, a simulation hierarchy has been constructed. These modules use a communications framework similar to that of the controllers.

To date, only the helm, depth and propulsion control system have been simulated. The following were simulated: actuators, submarine dynamics, ice mapping, environment (via fractals), and sonar.

[Fiala1] **RCS** ***memorandum on Jeff Becker's evaluation of NASREM***

Jeff Becker of Martin Marietta had evaluated NASREM. Fiala comments on Becker's evaluation and discusses several NASREM issues.

The notions of atomic unit and process, as presented in [Fiala2] are discussed briefly.

The need for levels of architectural definition for NASREM is highlighted at the end of section 1, although the term is not used.

Section 2 discusses the relationship between the NASREM architecture and software architecture.

Section 3 discusses global memory.

Section 4 discusses alternative approaches for organizing job assignment, planning, and execution modules in NASREM.

[Fiala2] **RCS** ***Note on NASREM Implementation***

This describes a NASREM implementation done in the RSD Intelligent Controls Group. The architecture is viewed as appropriate only for the software parts of the control system, as the hardware cannot be expected to have the requisite flexibility.

This implementation makes use of processes which can be activated or inactivated to produce behavior similar to dynamically changing the control hierarchy: where a controller can be controlling one of two grippers, for example. Inactivation is an important real-time concept, as it allows for one process to be inactivated, so that it does not consume processing cycles, and another process can use that resource.

In this implementation, there is a single Task level for an entire robot. For each major equipment subsystem, there is an elemental-move level. For each separate piece of equipment that must be controlled, there is a separate servo level.

When the control loop is open, the Primitive level is permitted to interface directly with the sensors. Parallelism of the architecture is exploited. World modeling activities are going on simultaneously with sensory processing and task decomposition activities. The activities at each level also continue in parallel. In this implementation, using the innate parallelism of the processes motivates the task decomposition.

Section 2 emphasizes hierarchical decomposition, including when hierarchical levels might be omitted.

Section 3 discusses the idea of atomic units. Tasks have been decomposed into atomic units which can be executed in parallel. Typically, the atomic units are executed on a single processor. There is a trade-off between communications overhead and decomposition to use parallelism. This trade-off is used to determine the atomic units. Four criteria for when to put separate activities into separate atomic units are given: parallelizability, asynchronicity, communications overhead, and function decomposability.

Section 4 discusses properties of atomic units and defines what a process is. Example Ada code for how a process might be implemented is given.

Processes have the following characteristics:

(1) continuous cyclic execution,

(2) read-compute-write execution cycle,

(3) concurrency,

(4) interfaces through global data system,

(5) inactivation.

These implementations use ADA, which provides explicit support for tasking. Using this support, it is possible to encapsulate all task-local variables and to make an application relatively portable. In changing the process distribution of an application, it is necessary only to rewrite the “main” function for each processor in the system.

[Fiala3] **RCS** ***An Approach to Telerobot Computing Architecture***

This is a technical paper that assumes an RCS NASREM architecture without ever mentioning NASREM. The focus of the paper is a computing architecture for controlling a telerobot capable of 10 micro second around the time loop.

The document gives a basic design for a telerobot. It focuses on the software and hardware design with examples taken from the servo level. It discusses the hand-controller subsystem for the operator interface and the robot manipulator. The paper discusses the teleoperation mode. In this mode the two systems communicate by common memory.

The control system for components needs to be distributed in order to have sufficient processing power for the complex algorithms. However, care must be taken only to distribute control which is truly local. Inappropriate distribution of control results in high data transfer rates.

In this application, all of the processors reside on a single backplane. Different execution speeds can be performed on different processors, resulting in an ability to control the performance of the control system through variation in the distribution and timing of the software on the hardware.

A detailed analysis is presented of an implementation of a Martin Marietta control algorithm, including timing considerations and the assignment of computing processes to processing hardware.

[Fiala4] **RCS** ***Manipulator Servo Level Task Decomposition***

This paper gives a brief outline of RCS in section 1. That section also describes the particular application, control of robotic manipulators (electric-powered manipulators with serial joints and unbranched kinematics), which the rest of the paper covers.

Section 2 gives the detailed architecture of the servo level control for the application.

Sections 3 and 4 discuss the servo level job assignment interfaces and operation. Job assignment modules have interfaces to the upper task decomposition hierarchy, operator control, the servo level world modeling hierarchy, and the planning module.

Sections 5 and 6 discuss the servo level planning interfaces and operation. The planning module has interfaces to world modeling and job assignment at the same level.

Section 7 discusses the execution module interfaces. The execution module has interfaces to world modeling, from planning, and to motor control.

Section 8, *"Execution Operation"*, discusses the *"manipulator control problem"*, presents a number of manipulator control schemes, and discusses timing considerations.

This architecture has a many-read, many-write memory structure. Local copies of data are not maintained due to the difficulty of assuring consistency.

For each interface, variables (and their types) are given which contain the critical information for that interface. Examples include: position, velocity, acceleration, jerk variables (which are continuous), and state variables (for which there is an enumeration of permitted values).

The bibliography is described as forming *"a comprehensive review of the basic concepts of manipulator servo control.*"

The paper includes an index.

This paper is a good example of a fine-granularity architecture.

[Griesmeyer1] ***Generic Intelligent System Control (GISC)***

This is an incomplete draft of a description of the GISC concept and software that has been put into a *"GISC-Kit"* software toolkit.

"GISC is an approach to the construction of controllers for complex robotic systems."

"GISC-Kit is the library of software modules that the designer of a robot system controller can access for an actual implementation."

GISC is not a complete architecture. It places emphasis on communications, but also includes some informal principles of building control systems.

[Griesmeyer2] ***General Interface for Supervisor and Subsystem (GENISAS)***

"GENISAS is a GISC-Kit package that provides general communication software interface capabilities (such as command processing and event handling) between the supervisory control system … and subsystems".

Chapter headings are: GENISAS Basics, Getting Started, Tools, Subsystem Servers, Clients, Supervisors, PassThroughClientServer, Advanced Topics.

The bulk of this paper (about 200 pages) is Appendix B, the GENISAS Reference Manual, which describes the C++ classes and functions that comprise GENISAS. The appendix is in the form of heavily commented source code.

[Harhalakis1] ***Architecture of a Facility Level CIM System***

This paper describes a method of maintaining the consistency of some of the common data in the databases of three otherwise separate systems (CAD, CAPP, and MRP). The paper does not deal with any other architecture issues.

[Hatvany1] ***Intelligence and Cooperation in Heterarchic Manufacturing Systems***

This is a brief paper in support of heterarchical systems (multiple, autonomous, cooperating systems). The paper does not offer much justification for this support.

The author advocates heterarchies with two conditions:

(1) Participants in the heterarchy must conform to certain rules, in order to obtain certain privileges.

(2) The design, structuring and enforcement of a system of dual goals for the distributed subsystems must be included. One set must be concerned with the goals for local optimization, the other set must be concerned with global optimization.

The paper says heterarchical systems need to be well thought out or they are anarchic but does not suggest methods for building heterarchical systems.

[Herman1] <u>**RCS**</u> ***Intelligent Control for Multiple Autonomous Undersea Vehicles***

This paper presents the control architecture for the Multiple Autonomous Undersea Vehicles (MAUV) project.

The paper discusses the usual RCS elements of intelligence in section 2. Figure 5 illustrates them. The MAUV control architecture, as described in this paper, is standard NASREM RCS, except that a planner manager is included.

State transition graphs are discussed. Figure 7 presents one.

Some novel features include the control of two semi-independent vehicles. The paper introduces the idea of cooperation by identical planning.

The paper has seven pages in section 8 on world modeling; map representation and updating, mainly.

[Herman2] **RCS** ***Real-Time Vision for Autonomous and Teleoperated Control of Unmanned Vehicles***

About two-thirds of the paper is given to a discussion of vision processing, which is outside the scope of this annotated bibliography.

The paper describes standard RCS briefly; see figure 1.

Four different types of teleoperation and autonomy are presented on page 2.

Sections 2.1 and 2.2 briefly discuss task decomposition and world modeling at various hierarchical levels.

[Herman3] **RCS** ***Real-time Hierarchical Planning for Multiple Mobile Robots***

This paper describes the MAUV project with two underwater vehicles. The hardware is not described. Unusual features of this clear and well-written paper are the description of cooperative behavior of autonomous vehicles (section 5) and the use of real-time planning (section 6), including cyclic re-planning and planning updates. It is also unusual that the communications system is one of the subsystems which must be explicitly commanded.

The architecture is described in section 2, with reference to Figure 1 (a block diagram of the architecture). Although the architecture described is clearly a variant of RCS, the term RCS is never used. The architecture includes the usual SP, WM, and TD in six hierarchical layers:

(1) Mission: converts mission to commands for groups of vehicles,

(2) Group: converts group commands to commands for individuals,

(3) Vehicle task: converts task commands into moves and actions for vehicle,

(4) E-move: converts moves and actions to intermediate poses,

(5) Primitive: finds smooth trajectories to achieve [a sequence of] poses,

(6) Servo: converts trajectories into signals for actuators and other physical components.

Most of the paper is given to describing the task decomposition and how planning is done for this system. The paper contains a detailed specification of each of the levels in the decomposition, down to specific parameters and functions.

In this version of RCS, the world modeling component simulates possible future states of the world. *"The goal of the sensory processing function is to identify patterns, events, objects and to filter and integrate sensory information over space and time."*

Plans for this application are represented as graphs with nodes representing actions and arcs representing events. Execution is the process of carrying out a plan. Monitoring consists of querying the world model for that event, then, if it has occurred, following the appropriate arc. Plans are chosen by A* search of the planning space.

Cooperative vehicle behavior is achieved by ensuring that both subs have identical software and world models at the mission and group level. When one sub gets major new news, like detection of a new mine or hostile object, it must communicate this to the other sub to keep the world models consistent.

The paper contains a detailed description of the planner manager and planner for the group and mission level. The planner manger consists of two modules, job assignment (which divides the task into jobs and sends each of these jobs to different planners to schedule), and the plan coordination module (which coordinates planning by generating constraints for the subordinate planners). If a planner cannot generate a plan within the given constraints, the coordinator is responsible for generating new constraints. Communications and sensors are subject to planning (section 4.3).

The system replans regularly based on all the world modeling conditions. The cyclic re-planning time is determined by the planning reaction time. At the time when a plan is needed, the best available plan generated so far is selected. The new plan replaces the plan currently being executed.

For failure situations, there is an explicit Subtask Failure Re-planning module. This replans for failures at the next lower level. So far this has only been implemented for the level to handle imminent collision between the sub and the environment.

For each subtask command, a plan schema is used to provide all possible sequences of actions which define that command. An interpreter traverses the plan schema. At each node, it queries the world model to determine which arc to follow.

A node of a plan schema has two components: an alternative action component and the context subroutine component. The alternative action component contains a function that generates all possible alternative actions that can be considered when that node is reached. Alternative actions are represented as operators on state space. The context subroutine sets certain variables for the alternative action module.

Possible types of arcs in a plan are world arcs, which switch on world conditions, constrained by plan time and execution time predicates. The second type of arc is an else arc, which also has plan time and execution time predicates.

[Horst1] **RCS** ***An Intelligent Control System for a Cutting Operation of a Continuous Mining Machine***

This paper focuses on a variant of RCS, called *"Barbera RCS"* (BRCS). The paper emphasizes tasks and controllers and says very little about sensory processing or world modeling.

Section 1 briefly introduces the subject of *"large-scale intelligent control systems"* and cites previous work on the subject.

Section 2.1 describes the scenario for coal mining (using a specific type of continuous mining machine and method of mining called *"room and pillar"* mining).

Section 2.2 presents the task tree developed to deal with the job.

Section 2.3 presents an example of a state machine for the application.

Section 2.4 describes the controller hierarchy for the application.

Sections 2.5 and 2.6 discuss the simulation and animation used in the application.

Section 3 presents *"principles of the BRCS methodology"*. It includes subsections with the following titles, which summarize the principles:

(1) problem analysis through task decomposition,

(2) controllers encapsulate tasks,

(3) hierarchical with strict chain of command,

(4) rule-based,

(5) finite state machine model,

(6) generic controllers,

(7) determinism,

(8) data integrity through multiple buffering,

(9) handshaking between controllers by command numbers,

(10) real-time execution through cyclic processing,

(11) straight-through execution of controllers; no internal looping,

(12) multiprocessing inherent,

(13) [has an issue discussion],

(14) problem domain understanding critical,

(15) generic processing pattern in all controllers,

(16) controllers small enough for human understandability.

Section 4.1 presents the details of several C language files used to implement the control system and briefly describes the simulation of controllers, actuators and sensors.

Section 4.2 describes system hardware briefly.

Appendices describe (A) an overview of the underground coal mining environment, (B) a continuous mining machine, (C) a C language state table, and (D) generic controller templates in C.

[Huang1] **RCS** ***A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real-Time Control for Coal Mining Operations***

This paper describes standard RCS4 applied to coal mining.

The RCS overview is given in section 2. It includes the usual SP, WM, TD decomposition with TD decomposing to PL, JA, and EX. Includes value judgment (VJ) behind WM. Task decomposition methodology is discussed in section 3.

The job is to control a coal mining operation using Joy 14CM Continuous Miner coal mining machines in a pillar and room mining scheme. Figures 8 and 9 show the controller hierarchy. A task decomposition of the job is shown in figure 16, and each task is described in the text in section 5. State transition diagrams for the plans used by the control system are shown figures 17 to 29 in section 6.

A discussion of user interfaces at the various levels is given in section 7.

The paper includes almost nothing about the nature of the database.

Interesting technical points include:

1. Peer-to-peer data passing via a common superior in sec. 3.2.9,
2. Teleoperation is a mode - Compare this to Dornier where teleoperation is part of the *"operations"* refinement of the architecture.

[ISO1] ***International Standard ISO/TR 10314-1 Industrial automation - Shop floor production - Part 1: Reference model for standardization and a methodology for identification of requirements***

Despite the title, this is not a standard, it is a technical report.

"… it is not possible, in view of the current state of the art of modelling for manufacturing, to draw up an international Standard which would be complete and precise, and which would not be too restrictive in this rapidly changing field. This Technical Report is intended as a guideline …"

The intent of the document appears to be to have a model of discrete parts manufacturing that will help in identifying areas where standards are needed for shop floor production.

Section 1.2 gives a list of 12 *"manufacturing functions"* in the organizational scope of an architecture:

(1) corporate management,

(2) finance,

(3) marketing and sales,

(4) research and development,

(5) product design and production engineering,
(6) production management,
(7) procurement,
(8) shipping,
(9) waste material treatment,
(10) resource management,
(11) maintenance management,
(12) shop floor production.

Interestingly, the list omits post-production activities such as customer support.

Section 2 gives terminology, but the terms are very loosely defined.

Section 4 discusses the objectives of manufacturing standardization.

Section 5 gives the proposed reference model. In it, a shop is hierarchically arranged in four levels: section/area, cell, station, and equipment. A *"Generic Activity Model"* is described identifying four *"Subjects"* (control information, data, material, and resources) and four *"Actions"* (transform, transport, verify, and store).

Section 6 describes a methodology for extracting areas of standards.

[Jackson1] **AMRF** ***An Architecture for Decision Making in the Factory of the Future***

Most of this paper is a description of the AMRF: design philosophy, physical layout, controller architecture, and equipment. Problems of sequencing and scheduling in a discrete parts shop are discussed on the last few pages.

[Jayaraman1] ***Design and Development of an Architecture for Computer-Integrated Manufacturing in the Apparel Industry Part I: Basic Concepts and Methodology Selection***

In this paper, which focuses on architecture for apparel manufacture, *architecture* is taken to be detailed function descriptions and information flows. An apparel manufacturing architecture, as shown in Figure 2, consists of three types of models: dynamics models, information models, and functional models. The paper suggests using the IDEF0, IDEF1, and IDEF2 modeling languages, respectively, for these three layers. The three languages are described briefly.

The organizational scope, shown in Figure 1, is broad, including all functions of an entire apparel enterprise.

"Computer-aided manufacturing (CAM) is the effective use of computer technology in the management, control, and operation of a manufacturing facility through direct or indirect computer interface with the physical and human resources of the company."

"Computer-integrated manufacturing (CIM) involves integrating computers into the various operations of an enterprise to produce the right product at the right price and the right time."

The model provides a framework for both the technical and economic implications of automating a process. The functional and information architecture are closely related and form the foundation on which the analysis, the dynamics model, is constructed. Analysis of the present system serves as a basis for a future architecture which satisfies the requirements set forth for the new system.

The methodology suggested is to analyze functions in terms of necessary inputs, controls, mechanisms and outputs. A top level description of the functional view of running an organization is given. The dynamics model consists of the behavior of the system's information, function and resources which vary over time. Models may be formulated at varying levels of abstraction based on the purpose of the model.

The architecture proposed in this paper differs from other architectures by including the dynamics model. Unfortunately, only a couple paragraphs are devoted to explaining the model.

[Johnson1] ***Towards a Distributed Control Architecture for CIM***

This paper says that distributed (heterarchical) control is better than hierarchical control. The paper gives a brief, simplified description of heterarchical architecture and suggests using object-oriented methods to construct interoperable modules for CIM. A half page is devoted to what the scope of a CIM architecture should be. The paper describes five *"basic abstractions"*:

(1) production resources,

(2) production orders,

(3) production schedules,

(4) production operations,

(5) production activities.

[Johnson2] ***Trends in Shop Floor Control: Modularity, Hierarchy, and Decentralization***

This paper describes briefly three types of architecture: centralized (exemplified by the GE Fanuc CIMPLICITY system), hierarchical (exemplified by NASREM), and heterarchical.

[Jones1] **I*ssues in the design and implementation of a System Architecture for Computer Integrated Manufacturing***

This paper focuses on having separate architectures for production management (which includes manufacturing data preparation), data, and communications. The paper discusses hierarchical control, but not in depth. It goes into more detail on data modeling, database design, data administration, and communications.

The paper says "*there is little hope that a single architecture for production management will emerge which can serve as a reference model for all CIM applications.*" It suggests development of measures and tools to compare different designs.

The paper does not deal with application areas other than manufacturing.

[Jones2] **AMRF** ***A Proposed Hierarchical Control Model for Automated Manufacturing Systems***

This paper describes the AMRF. It includes the usual:

(1) controllers arranged *“in a hierarchy in which the control processes are isolated by function and communicate via standard interfaces,”*

(2) tasks decomposed along control hierarchy lines,

(3) *“Implemented in a distributed computing environment*,”

(4) five level control hierarchy: facility, shop, cell, workstation, equipment,

(5) material handling controlled as a workstation under cell,

(6) command-status controller interface protocol,

(7) process planning is done off-line for all controllers,

(8) IMDAS data system,

(9) network communications system with common memory.

The paper indicates that each controller would have a scheduler, but no global scheduling or coordination of schedules across controllers is provided. Resource management and dynamic reconfiguration are mentioned briefly as taking place at the shop and cell levels.

[Jones3] ***Toward a Global Architecture for Computer Integrated Manufacturing***

This paper recommends *“separate architectures for production management, information management, and data communications.*”

For production management, (section 2) a hierarchy of control modules is proposed. *“Each module in this hierarchical structure performs three major control functions: adaptation, optimization, and regulation. The adaptation function generates a run-time production plan. The regulation function provides the interface between a module and its immediate subordinates. … It releases jobs to subordinates, monitors subordinate feedback on those jobs, and guides subordinate error recovery. The optimization function … evaluates proposed production plans from the adaptation function. It generates a list of tasks … for subordinates… It resolves … any conflicts and problems with the current schedule identified by the regulation function.*”

For information management (section 3), alternative approaches are presented and evaluated. The three main functions of data administration are given as: query processing, transaction management, and data manipulation. *“It is our view that separating the query processing and transaction management functions from the data manipulation functions, producing a layered hybrid architecture, is essential for effective distributed data management in CIM systems.*”

For data communications (section 4), three fundamental ideas are presented:

(1) Use a common connection service specification for all communications

between programs in a CIM complex.

(2) Transparently interconnect the physical networks.

(3) Optimize communications subnetworks to meet CIM complex needs. It is recommended that the OSI model be used in a network with a single spine.

[Jones4] ***A Multi-level/Multi-layer architecture for Intelligent Shopfloor Control***

Section 2 reviews hierarchical and heterarchical control.

Section 3 describes *"multi-layer"* and *"multi-level"* control.

Section 4 gives a proposed approach for shop floor control.

"Each module in this stratified structure performs three major control functions: adaptation, optimization, and regulation. Adaptation is responsible for generating and updating plans for executing assigned tasks. Optimization is responsible for evaluating proposed plans, and generating and updating schedules. Regulation is responsible for interfacing with subordinates, monitoring execution of assigned tasks."

Section 5 discusses integration. It recommends using a command-and-status protocol between superiors and subordinates and separating control, communications, and data handling.

[Jones5] **AMRF** ***A Production Control Module for the AMRF***

Presents the AMRF control system, a five-layer hierarchy (facility, shop, cell, workstation, and equipment) with standard controller modules. Each controller has a production manager, a queue manager, and a dispatch manager. A command-and-status protocol is used, as described in section 5.2.

A process plan format is described in section 5.1.1. World models are described in section 5.1.2. A distributed global database is mentioned.

[Jorysz1] **CIM-OSA** ***CIM-OSA Part 1: Total Enterprise Modelling and Function-View***

Gives an overview of the CIM-OSA project and approach. CIM-OSA is focused on *"discrete manufacturing enterprises"*.

Figure 1 shows the *"CIM-OSA cube"*, entitled *"Overview of CIM-OSA Architectural Framework"*. The three dimensions are:

(1) stepwise generation (discrete values: function, information, resource, organisation),

(2) stepwise instantiation (discrete values: generic, partial, particular),

(3) stepwise derivation (discrete values: requirements definition, design specification, implementation description).

CIM-OSA takes a very broad view of the scope of a CIM architecture. The life-cycle includes:

(1) system requirements specification,

(2) system design,

(3) system description for build and release,

(4) system operation,

(5) system change.

CIM-OSA aims to go all the way to executable code.

Figure 5 uses a NIAM-like simple graphical information modeling language to show a model of *“Function-view Concepts.”*

The paper says CIM-OSA is not yet fully developed: *“The next phase is … to produce partial models for the reference architecture and to build prototypes…”*

[Jorysz2] **CIM-OSA** ***CIM-OSA Part 2: Information View***

The paper focuses on data in CIM-OSA.

The paper uses a NIAM-like information model to describe *“information-view concepts”* in Figure 1.

For design specification CIM-OSA uses the *“entity relationship attribute”* (ERA) approach. A graphical notation for ERA exists and may be translated into *“relational diagrams”* automatically.

SQL is also used for database access.

Normalization rules for translation to relational database structures have been built. However, Section 3.6 says they might move to object-oriented data models.

[Joshi1] ***A Scaleable Architecture for CIM Shop Floor Control***

This paper reports on a three-level hierarchical architecture for a shop floor control system (SCFS).

[Joshi2] Joshi, Jagdish; Desrochers, Alan; ***Performance Analysis of Network and Database Transactions in a CIM System***; Proceedings of the 1991 IEEE International Conference on Robotics and Automation; Sacramento, CA; April 1991.

“A petri-net based integrated model for the performance analysis of network and database transactions generated by manufacturing clients for the computer resources.”

“Approach is described to investigate the relevant system integration issues between the logical and the physical access of the information in the manufacturing system. The … approach proposes an integrated conceptual framework for modelling and performance analysis of data flow and communication flow. This framework will provide some answers to the impact of relative time scales on the coupling and de-coupling of the various CIM systems and identify the time-scales where integration becomes important.”

It is important to understand the interaction between data and network flow in a CIM system. Key questions are:

(1) What is the response time of database and network transmission and

utilization of resources in the system?

(2) How does changing the mix of database events and network events affect the behavior of the system in terms of the response time?

(3) What is the impact of changing the processing rate of different processors located at either the local node or the remote node on the system performance?

A CIM system can be modeled as a discrete event dynamic system (Petri nets). This can be hybridized using statistics to form generalized stochastic Petri nets to model distributed system behavior for real-time control.

"It is assumed that all resources are dedicated to a single control mission. Tasks are pre-assigned to a node and remain unchanged throughout the control mission." Other approaches include predicate/transition nets and tasks modeled as causal net.

[Judd1] ***Manufacturing System Design Methodology: Execute the Specification***

Reports on a commercial tool for designing manufacturing systems (workstations, cells, or individual lines) called XSpec, (executable specification). Specifications are executed on a tool called XFaST (executable factory simulation tool).

The tool allows the logical design of physical and control elements which may be interconnected via *"pins"* and *"connectors"* so that they exchange messages along *"paths"*. The design may then be operated in a simulation.

[Jun1] **AMRF** ***The Vertical Machining Workstation Systems***

An example of an operating system written in source code for running controllers which are finite state machine modules is given in section III.

[Jung1] ***Implementation of the RAMP Architecture at an Established Site***

This paper describes how the CIM architecture developed in the RAMP (Rapid Acquisition of Manufactured Parts) project at the RAMP Test and Integration Facility was being modified and applied at an existing manufacturing site, the Cherry Point Naval Aviation Depot.

The paper provides a useful example of altering and specializing a CIM architecture.

Section 3.3 discusses the use of IDEF0 and Yourdon-Demarco modeling methods, and notes the inadequacy of IDEF0 for some of their purposes.

[Klittich1] **CIM-OSA** ***CIM-OSA Part 3: CIM-OSA Integrating Infrastructure the Operational Basis for Integrating Manufacturing Systems***

This paper describes information services in CIM-OSA.

Figure 5 shows the integration of data. The basic idea is that a system which wants data uses a *"front end service"* to get it. The front end service uses a *"data access protocol"* to provide the requested service. There are four types of front end services: application (e.g. CAD or CAPP), human, machine (e.g. robot or NC machine tool), and data management. The two data access protocols are: *“business process services”* and *“information services.”*

It seems intended that any data which is not strictly local will be globally accessible.

It is recognized that communications services are needed, but communications is not discussed at length.

Figure 11 contrasts CIM-OSA data integration with *“standard”* methods.

The paper does not deal with control issues.

[Klittich2] **CIM-OSA** ***From CIM to CIM-OSA a Step Ahead in System Integration***

This paper discusses CIM-OSA. It says the draft specification was to be released at the end of 1991.

This papers names several existing ESPRIT projects that are *“CIM-OSA oriented”* and deal with similar issues as CIM-OSA. These are: MULTICON (Multilevel Shop Floor Control) which schedules controllers, and DSDIC (Design Support for Distributed Industrial Control Systems).

“Release 1 will cover the architectural elements for Information Integration.” The Release 1 architecture is shown in Figure 3. Release 1 also includes a Communication Service. The information architecture looks like an AMRF IMDAS.

In the Release 1 architecture, each application has a database divided into private and public parts (shown in Figure 4). To deal with public data, each application has two data service modules:

(1) an application front end module connected to the application,

(2) a data management module connected to the public part of the applications’ database.

The two modules are connected to a system-wide data agent.

It appears that all non-private data is intended to be globally accessible. There is no provision for direct data exchange between applications.

[Kramer1] **RCS** ***EXPRESS Schema for NASREM***

This is an EXPRESS schema which models the NASREM architecture. It covers individual controllers, a hierarchy of controllers, and tasks. The controller model includes the notion of a *“prototype”* controller, for which there may be several instances in a hierarchy.

[Leake1] **RCS** ***The NBS Real-time Control System User’s Reference Manual***

This is a users manual for a programmable implementation of RCS. The implementation itself is called RCS in the document.

The implementation is for controlling robots. An interface to a Unimate Puma 760 robot is part of the software provided by NBS. The implementation requires specific computer hardware (Intel MULTIBUS with Intel 86/30 computer boards), operating system (FORTH), and language (SMACRO, FORTH, and 8086 assembly language).

The architecture of the implementation is described in Chapter 3, although the controller hierarchy (consisting of four levels: task, path, prim, and joint) does not appear until the beginning of Chapter 10 in Figure 10-1. The building blocks of a hierarchy are called *"functionally bounded modules"* rather than controllers.

The commands available at each level are presented in chapter 10.

Plans are embodied in state tables.

Chapter 4 describes hardware and software components of the application.

Chapter 5 gives installation procedures.

Chapter 6 presents *"Basic Operations"* (starting the system, moving from one board to another, locating source code, loading code, executing tasks and routines, saving and rebooting the system, editing a block of code, example dialogue, using printing utilities, using tape utilities, and shutting down the system).

Chapter 7 describes SMACRO.

Chapter 8 discusses communications.

Chapter 9 describes the *"Robot Sensor Language (RSL)"*. *"RSL is a high-level, task description language designed for programming robotic tasks in which the control system, RCS, uses sensors to control the robot.*"

Additional chapters and appendices give extensions, examples, debugging techniques, user word summary, etc.

The paper includes a glossary and index.

[Litt1] ***The Development of a CIM Architecture for the RAMP Program***

The paper focuses on how the RAMP architecture was designed and what it is.

The RAMP Generic Model has seven *"Top Level Components"*:

1. production and inventory control
    a. capacity requirements planning,
    b. production control,
    c. order entry,
    d. material inventory management,
2. manufacturing
    a. a. schedule manufacturing cell,
    b. manage maintenance,
    c. coordinate/monitor manufacturing cell,
    d. manage indirect inventory,
    e. workstation control,

    f. transportation control,
3. manufacturing engineering
    a. create process plans,
    b. evaluate problem cause,
    c. generate RAMP PDES,
4. quality
    a.
    b. generate quality reports,
    c. coordinate disposition of quarantined part,
    d. assemble part pedigree,
    e. generate part quality record,
    f. resource certification,
    g. information management,
    h. communications,
    i. control.

This document goes beyond most others in its attention to peripheral but necessary activities.

Control of shop floor activities falls under item 7 above. Control is arranged in a five-level hierarchy using command and status messages. Little detail of the implementation is provided.

[Lumia1] <u>**RCS**</u> ***The NASREM Robot Control System Standard***

Section 2 gives a brief summary of the NASREM architecture. The introduction and subsection 2.1 describe the six-level *"hierarchy"* of layers of SP, WM, and TD (figure 1) as well as the spatial and temporal decomposition of tasks (figure 2). Subsection 2.2 discusses world modeling, 2.3 sensory processing, 2.4 operator interfaces.

Section 3 covers servo level task decomposition: *"Servo is responsible for controlling small dynamic motions of the manipulator"*.

Section 4 deals with primitive level task decomposition.

[Lumia2] <u>**RCS**</u> ***NASREM as a Functional Architecture for the Design of Robot Control Systems***

Section 1 discusses the general issue of control architectures.

Section 2 briefly discusses three proposed control architectures: Saridis's *"intelligent control"*, Brooks's *"subsumption"*, and Shafer's *"CODGER"*.

Section 3 briefly describes NASREM.

Section 4 describes the implementation at NIST of the NASREM servo level for a robotic manipulator.

Section 5 revisits the comparison of architectures.

[Lumia3] <u>**RCS**</u> ***NASREM: Robot Control System and Testbed***

Section 1 describes how NASREM might be used as a testbed to compare different algorithms experimentally.

Section 2 is a brief summary of NASREM as outlined in more detail in [Albus5].

Section 3 describes a servo (lowest) level task decomposition for NASREM as applied to a robotic manipulator.

Section 4 describes a primitive (next to lowest) level task decomposition for the same application.

[Malhotra1] Malhotra, Rajeev; Jayaraman, Sundareson; ***Design and Development of an Architecture for Computer-Integrated Manufacturing in the Apparel Industry. Part II: The Function Model.*** Textile Research Journal; Vol. 60, No. 6; June 1990; pp. 351 - 360.

This paper gives steps for modeling:

(1) attaining thorough understanding of the manufacturing enterprise and the focus of the modeling effort,

(2) establishing the purpose of the model,

(3) determining the context - the boundaries of the domain of the model,

(4) establishing perspective from which the domain is to be viewed for modeling purposes.

For the model they propose, they have divided the functions into three categories based upon temporal resolution:

(1) strategic decision-making (corporate management),

(2) tactical decision-making (production planning, inventory management, quality control),

(3) operational decision-making (day-to-day decisions in the factory - control).

The article then describes the process of creating an as-is functional model for the tactical end of a specific site and the to-be model created from it based on input from domain experts.

The paper describes the *"to be"* model and a selected function in that model extensively. Enterprise functions are generic. The information model is under development.

[Maimon1] ***Real-time Operational Control of Flexible Manufacturing Systems***

This paper deals with controlling short-term (a few days) activities of an FMS with a moderate number of machines. Figure 1 shows an FMS Controller Scheme which has an integrated architecture with central planning. The architecture includes databases, a scheduler, resource allocation, controllers, etc. The architecture is described as *"a generic hierarchical control system,"* but there seem to be no levels of the control hierarchy; it looks like central control.

Mathematical models are described for *"production surplus state"*, *"resource state"*, *"capacity"*, and *"penalty function"*.

The process sequencer in the control system is an expert system with rules and an inference engine.

[Martin1] **NGC** ***System***

This is the first of six draft volumes totalling about 1000 pages describing the SOSAS. This first volume has four sections.

Section 1, the introduction, gives an *"NGC Architecture Overview"* in section 1.3. SOSAS has two distinct categories of architectural elements: services and applications. Section 3.2 (136 pages) of this volume describes services, volumes 3 through 6 describe applications, and volume 2 describes data.

Section 2 gives references, including many normative references to other standards.

Section 3 gives systems specification in 3.1, a description of NGC services in section 3.2:

(1) platform,

(2) communications,

(3) data management,

(4) presentation management,

(5) task management,

(6) geometric modeling, and

(7) basic I/O.

in sections 3.2.1.1 through 3.2.1.7. Application requirements are given in section 3.3, integration and configuration environment in section 3.4, and conformance in section 3.5.

Section 4 briefly (4 pages) describes a *"Test and Validation Plan."*

A detailed glossary is included which includes almost all the terms from the glossaries in the other five volumes. Most terms that appear in the glossaries of two volumes are defined the same way in both.

[Martin2] **NGC** ***NGC Data***

This paper gives formal information models in EXPRESS of information required in the NGC. Many hundred EXPRESS entities and types are defined.

Section 1.2 describes the models generally. There are three categories of models: execution, manufacturing practice, and controller practice.

Section 3 (223 pages) gives the EXPRESS models. Rather little explanatory text accompanies the formal EXPRESS statements.

A glossary is included.

[Martin3] **NGC** ***Workstation Management Standardized Application (WMSA)***

Workstation management functions are given in section 3.4.2 as:

(1) handle NML message,
(2) perform workstation start-up sequence,
(3) perform external communications,
(4) control workstation execution,
(5) handle exceptions,
(6) perform shutdown sequence,
(7) configure workstation,
(8) control mode and state,
(9) perform safety,
(10) manage health,
(11) control diagnostics,
(12) schedule task,
(13) determine resource availability,
(14) resource request,
(15) manage application configuration,
(16) provide data logging,
(17) generate OBIOSs calls.

Section 3.4.3 gives system modes as:

(1) start-up,
(2) normal production,
(3) failure recovery,
(4) maintenance,
(5) shutdown.

A glossary is included.

[Martin4] **NGC** ***Workstation Planning Standardized Application***

Section 3.4.2 gives nineteen functions of workstation planning:

(1) handle NML messages,
(2) set initial equipment states,
(3) generate control plan from task goals,
(4) generate control plan from path goals,
(5) refine workstation plan,
(6) refine task plan,
(7) refine path plan,

(8) replan,

(9) generate and execute control plan from task goals,

(10) generate and execute control plan from path goals,

(11) refine and execute workstation plan,

(12) refine and execute task plan,

(13) refine and execute path plan,

(14) perform self diagnostics,

(15) manage configuration,

(16) implement operating modes,

(17) manage state transitions,

(18) handle exceptions,

(19) provide data logging.

Section 3.4.3 describes modes and states.

A glossary is included.

[Martin5] **NGC** ***Controls Standardized Application (CSA)***

The document is not specific about the type of machine a CSA controller is intended to control, but a 3-axis machining center fits well.

Section 3.4.2 describes the general non-motion-control behavior of a CSA controller.

Section 3.4.3 gives modes and states of a CSA controller.

Section 3.4.4 describes the motions a CSA controller must be able to produce, plus other general requirements related to machine functionality.

Section 3.4.5.2 provides NCL commands that include the functions of RS-274-D and go well beyond into new controller functionality.

A glossary is included.

[Martin6] **NGC** ***Sensor/Effector Standardized Application (SESA)***

This paper describes the application-independent functions of a sensor/effector required to fit into a SOSAS compliant system.

A glossary is included.

[Michaloski1] **RCS** ***Handbook for Real-Time Intelligent System Design***

This is a detailed handbook (in draft) aimed at people developing RCS systems.

The handbook describes RCS (implicitly on page 2-2) as including a three-level architectural specification in which the top level of abstraction is called the *"RCS Architectural Model"*, the middle level is called *"architectural design"* and the bottom level is called a *"detailed design"*.

Section 2.2 presents *"elements of intelligent control"*.

Section 2.3 presents the *"RCS Architectural Model"* with the usual features. Also includes (section 2.3.7) the sensory processing integration hierarchy seen only in other RCS papers about systems with active vision.

Section 3 gives:

1. Resource Taxonomy
   a. object,
   b. agent,
   c. tool,
2. System Terminology
   a. process,
   b. hierarchy,
   c. node,
   d. component,
   e. level,
   f. branch,
3. Architectural Analysis Criteria
   a. multiprocessor parallel architecture,
   b. distributed functionality,
   c. timing guidelines,
   d. operator interface observations,
   e. functional decomposition guidelines,
   f. top-down vs. bottom-up decomposition,
   g. object decomposition guidelines,
   h. multiple threads of control,
   i. functional coupling of processes,
   j. architectural validation with scenarios),
4. An RCS Architectural Example.

Section 4 presents RCS Task Analysis: Task Modeling, Task Frames, and RCS Task Analysis Methodology.

Section 5 gives RCS Detail Design Principles:

(1) Well-defined Interfaces,

(2) RCS Process Model Principles,

(3) RCS Communication Model Principles,

(4) RCS Human Interface Design Requirements,

(5) RCS Testing and Integration Design Principles.

Section 6 gives RCS Generic Controller Concepts:

(1) RCS Generic Controller Module,

(2) RCS Virtual Machine,

(3) RCS Communication API,

(4) Task Frames.

The paper says (page 2-10, 3-7) that dynamic reconfiguration is included in RCS without giving details of how it is to be accomplished.

[Michaloski2] <u>RCS</u> ***System Factors in Real-Time Hierarchical Control***

This paper is narrowly focused on the communications and timing details of using several processors to implement a controller hierarchy of the RCS sort with several processes to run. The applicability of the paper is not limited to RCS. It is clearly written.

[Michaloski3] <u>RCS</u> ***Design Principles for a Real-Time Robot Control System***

This paper describes a *“generic communication and control process”* (GCCP) which is a template (but is called an *“algorithm”*) for the software of an RCS Module. The basic architecture for which the GCCP is intended is the NASREM architecture. Timing considerations are discussed in detail. Two versions of the GCCP template are given in figures 2 and 3.

The paper includes the design principle that *“every process is statically allocated to memory*.”

The paper provides a proof that under certain conditions, *“every real-time [GCCP] process executes deterministically bounded by fixed response time and no processes deadlock*.”

[Min1] <u>MSI</u> ***A Survey of the Literature on Computer Integrated Manufacturing Architectures***

This paper focuses on CIM architectures, as indicated by the title. It does not look at non-manufacturing robotics applications.

The paper lists 14 management functions (without defining them): goal setting, planning, organizing, staffing, command, coordination, scheduling, execution, directing, actuating, reporting, budgeting, monitoring, control. Table 1 lists these according to papers that use them.

Table 6 lists 28 CIM architectures (including five not covered in Table 7) classified according to: controller structure, scope, genericity, and aspect.

Table 7 has a brief summary of 27 architectures (including four not covered in Table 6).

The paper has several other tables.

The paper does not have an issues analysis.

The paper includes 10 pages of references.

[Murphy1] **RCS** ***Real-Time Control System Modifications for a Deburring Robot User Reference Manual***

As the title implies, this is a users manual for the NIST AMRF Cleaning and Deburring Workstation, as it was in 1988. It is given as a series of modifications to the RCS programmable implementation described in [Leake1], and is written in the terms of [Leake1]. Hence, it is not much understandable to anyone not familiar with [Leake1].

There are sections on basic operations, the workstation interface, and changes to: RSL, task, path, prim, and communication level. An appendix gives operating instructions.

[Norcross1] **AMRF** ***A Control Structure for Multi-Tasking Workstations***

This paper reports on a special-purpose (non-RCS) controller built for the AMRF Cleaning and Deburring Workstation. As stated in the Summary, "*This paper outlines a controller structure based on computer operating system principles. Specifically, the structure uses Job Control Blocks, an active queue, critical sections, hierarchical task structure, inter-process communications, and resource allocation to implement an execution engine which segments tasks and provides for control of multiple independent tasks and coordination of multiple actors.*"

[Pan1] Pan, Jeff; Tenenbaum, J.; Glicksman, J.; ***A Framework for Knowledge-Based Computer-Integrated Manufacturing***; IEEE Transactions on Semiconductor Manufacturing; vol. 2, no. 2; May 1989

This paper gives a "*Presentation of an evolving framework for applying knowledge systems in a manufacturing environment.*" The approach is generic. "*It consists of a set of object-oriented tools for modelling key elements of the environment (processes, equipment, facilities and operational procedures), and a complementary set of application-specific shells that use the models to perform common manufacturing tasks such as monitoring diagnosis, control, simulation, and scheduling.*"

One aspect of this approach is creating an overview model for all of the data needed for the manufacturing tasks and making it available for sharing in a form which can be understood by all the systems which need it. This form would describe processes, for example, as a set of specified preconditions, post-conditions, and key parameters. Another aspect is that all modules for generic manufacturing tasks such as scheduling, diagnosis, monitoring and control should be decomposed into a task-specific shell that can be reused in multiple domains. Third, domain-specific knowledge required by different tasks should be combined into one unified model that can be shared by all modules.

Tools which they are building address generic issues in software and knowledge engineering. They have built a tool Manufacturing Knowledge System (MKS), which enables the approach cited above.

The point is to let the system engineers built these systems themselves with the assistance of these tools using a graphical interface. The approach assumes an ALPS-like language for process description. For example, basic types of nodes available are processing, decision and testing/measurement.

[Panse1] **CIM-OSA** ***CIM-OSA - A Vendor Independent CIM Architecture***

This paper gives a well-written but high-level description of CIM-OSA. The CIM-OSA architecture is intended to support *"real time control of all enterprise processes."*

"CIM-OSA contains two major parts. … The first … contains concepts for generating information technology representations of enterprise models. … The second … contains concepts for an Integrating Infrastructure within an Information Technology Environment." The paper goes on to describe the two parts in sections 2.1 *"The Modelling Framework"* and 2.2 *"The Integrating Infrastructure"*.

The paper describes the dimensions of the *"CIM-OSA Cube"* in some detail.

[Quintero1] **RCS** ***RCS Methodology Issues Log***

This paper contains a useful listing and discussion of several RCS issues.

[Quintero2] **RCS** ***The RCS Methodology: A Task Oriented Method for Developing Intelligent Real-Time Control Systems Software***

This paper contains useful discussions of *"The Concept of a Generic Controller Module"* and an *"Overview of the Task Oriented Analysis and Design (TOA&D) Methodology"* for building systems according to the RCS architecture.

This draft appears to be a precursor of [Quintero3].

The paper includes a glossary.

[Quintero3] **RCS** ***A Real-Time Control System Methodology for Developing Intelligent Control Systems***

The RCS architecture, as defined in most other RCS papers, gives an architectural specification but does not prescribe a complete methodology for architectural development for building control systems that conform to the specification. Other papers also do not cover many details that need to be handled in building an implementation. This paper provides a fuller methodology and suggests how to handle many implementation details.

The reasons given for developing the methodologies are:

(1) *"improving human understanding of a design,"*

(2) *"managing software complexity,"*

(3) *"providing for robust, verifiable, efficient, coordinated, real-time performance,"*

(4) *"provide for extensibility, portability and software reuse."*

The methodology approach given is composed of:

1. integration rules,
2. information models,
3. software execution models,
4. software engineering implementation techniques.

The RCS architectural specification is summarized in section 2.3.

Section 3 presents *"RCS method tenets"*:

1. Use task oriented decomposition.
2. Use strict hierarchical organization.
3. Organize the control hierarchy around tasks top-down and equipment bottom-up.
4.
   a. Spatial and temporal resolution of adjacent hierarchical levels should differ by an order of magnitude.
   b. Have ten or fewer decisions per plan.
5. A supervisor should have 7 plus or minus 2 subordinates.
6. Each node (controller) has SP/WM/BG (Sensory Processing, World Modeling, Behavior Generation) functions.
7. Allow a human interface at each node.
8.
   a. Controller modules are finite state machines.
   b. Controller modules communicate through common memory.
   c. Use cyclic sampling, not interrupts, for context switching.
   d. Use non-blocking I/O.
   e. Both input and output should be buffered.
   f. Implement global memory using a one writer many readers paradigm.
   g. Match the control cycle time to the application demands.
   h. Use controller templates.
9.
   a. Design for concurrent processing.
   b. Measure execution time performance.
   c. Allocate sufficient computing resources.
10. Use synchronous control at lowest levels, asynchronous at highest.

Section 4 discusses plans. Plans may be either *"path plans"* or *"rule plans"*. Rule plans may be represented in state graphs, state tables, or computer code (an example is shown in Appendix C).

Section 5 discusses implementation issues:

(1) grouping Job Assignment, Planning and Executor functions,

(2) behavior generation decomposition,

(3) using controller templates,

(4) using a main program template to run several controllers,

(5) required operating system services.

Methodology is mentioned throughout the paper and is the sole topic of Section 6. Table 1 provides a *"Summary of the RCS Methodology Steps."*

The paper does not deal with scheduling, with making the transition from process plans to production plans, or with resource allocation.

[Ray1] **<u>MSI</u>** ***A Production Management Information Model for Discrete Manufacturing***

This is an MSI paper aimed at one of the three main MSI objectives, identifying and defining information models.

It presents information models for: shop orders, plans and nodes, resources (maintained, logical, and material handling). The models are given in NIAM and explained in text.

Although not the main focus of the paper, there is a description of the intent that manufacturing planning be done in three stages: process plan, production-managed plan, and production plan. The process plan is done first and is used as the basis for building the other plans, which become more specific as scheduling and resource allocation planning are applied for actual production.

[Senehi1] **<u>MSI</u>** ***Control Entity Interface Document***

This MSI paper presents a summary of the MSI architecture and the interface specification for controllers in the MSI architecture. Its primary purpose is to document controller interfaces. A detailed description of the matching version of the architecture is given in [Senehi2].

The aim of the MSI architecture is to integrate planning, scheduling, and control in a manufacturing environment. Toward this end, a specification of the interaction of controllers was made. Controllers are considered to be black boxes, the internals of which are unknown. Only the interfaces and the functionality of the controller must satisfy the black box specification.

In interactions among controllers, there are two types of functions: administrative and task. Administrative functions deal with the health of the constituent controllers, start-up and shutdown of the hierarchy, and dynamic reconfiguration of the hierarchy. Task functions deal with the disposition of tasks, such as starting, stopping, and monitoring them.

Controllers, or control entities (CEs) as they are called in the document, are arranged in a strict hierarchy. Task control information flows parallel to the existing administrative hierarchy, but is dynamic based upon the current task. Task control is seen as a client-server interaction where superiors perform the role of task clients and subordinates are task servers. A guardian interface is provided for human interaction with each controller. All of the formal interfaces are specified as command and status interfaces.

To support the administration functions, an elaborate state machine is described that contains 12 states, 5 of which are stable and 7 of which are transitional. The document describes all the state transitions.

All commands are fully described in the document. There are 10 administrative commands and 7 task commands.

The guardian interface includes a console interface for a human. It may be used to reconfigure the control system (by adding or subtracting subordinates of a CE) or to change CE mode.

Communications are via NIST's Common Memory.

The document includes a glossary.

This document is superceded by [Wallace1].

[Senehi2] **MSI** ***Initial Architecture Document***

This is an MSI paper presenting a discussion of the first version of the MSI architecture and architecture issues.

MSI aims to integrate planning, scheduling and control in a manufacturing environment. MSI identifies six types of systems which must be integrated in a shop: process planning, production planning, controllers, order entry, configuration management, and material handling. The architecture has two main thrusts: information integration of the previously listed systems, and the integration of control and planning.

The architecture assumes independent control, data, and communications paths.

To address information integration, the architecture describes a number of conceptual models which must be constructed to describe the information required. In addition, data storage and access are discussed. The architecture anticipates that systems may keep any private data that they want, but data which must be shared among two or more systems is global. In addition, systems may make local copies of global data (for example, to improve performance). In this case, however, it is required that the system be responsible for insuring that the global and local data are kept consistent with each other. The paper includes some discussion of database issues.

The paper presents and discusses a five-interface model with separate administrative hierarchy, client server network, and guardian (intelligent system or human) interfaces. For details see [Senehi1].

The paper includes good discussions of several issues:

(1) configuration and dynamic reconfiguration,

(2) hierarchical task decomposition,

(3) hierarchical control,

(4) integration of black boxes,

(5) levels of control,

(6) reactive vs. predictive control,

(7) distributed operations,

(8) error recovery,

(9) human interface,

(10) resource allocation,

(11) resource sharing.

The paper includes a glossary.

The information models corresponding to this version of the document are discussed in [Ray1] and [Barkmeyer2]. The corresponding interface definitions are in [Wallace1].

[Senehi3] **MSI** ***An Architecture for Manufacturing Systems Integration***

Section 1 introduces the MSI project.

Section 2 gives *"the MSI Vision of CIM"*, which is the authors' view of factory integration.

Section 3 outlines the MSI architecture, as it was at the time the paper was written (mid 1991). Information exchange is covered in subsection 3.1, information models (facility, process and production plan, product, orders, inventory, tooling, work-in-process, and materials) in subsection 3.2, systems (part design, process planning, production planning, control, order entry, configuration management, and material handling) in subsection 3.3, integrating a discrete parts shop in subsections 3.4 and 3.5, and hierarchical control in subsection 3.6. Discussion of error recovery from scheduling errors is included.

Although this is the most current available write-up of the MSI architecture, the MSI architecture has changed slightly since this paper was written.

[Shaw1] ***Dynamic Scheduling in Cellular Manufacturing Systems: A Framework for Networked Decision Making***

This paper discusses using bidding (in a LAN communications environment) for scheduling a group of manufacturing cells (a group of physically close machines). The paper maintains that bidding can be used for dynamic reconfiguration but does not describe how this might be done.

The paper describes an AI system built by the author for cell-level scheduling. The system has three hierarchical levels: strategy, meta-planning, and planning.

The paper contains the results of a Monte Carlo simulation of a dynamic bidding scheme, showing that it performs better than its centralized counterpart, primarily based on the fact that the scheduling decision is achieved by cells collectively based on purely local information stored within each cell. The author claims that in the centralized control system, large amounts of time were needed to keep the system information up to date.

[Shaw2] Shaw, Michael J.; Whinston, Andrew B. A; ***Distributed Knowledge-Based Approach to Flexible Automation: The Contract Net Framework***; The International Journal of Flexible Manufacturing Systems; Vol. 1; Kluwer Academic Publishers; 1988; pp. 85 - 104

This paper gives a detailed discussion of contract net bidding. It discusses the concepts of goal decomposition, task distribution, task execution, and task synthesis. The paper compares task sharing with result sharing strategies for heterogeneous systems.

This architecture strategy is based upon the distributed AI approach. It uses petri nets augmented with production rules as the controller representation.

[Shorter1] **CIM-OSA** ***Progress Towards Standards for CIM Architectural Frameworks***

This paper discusses activities within ISO TC184/SC5/WG1 and European standards-making groups (including CIM-OSA) on Reference Models for Shop Floor Production Standards.

Control is a major part of a Factory Automation Model in WG1, as shown in Figure 3, but that Model *"is no longer an explicit part of the Reference Model."* The current Shop Floor Production Model has four levels (section/area, cell, station, and equipment), but what they do is not called control (although it sounds like control). In the terminology of the model, section/area supervises, cell co-ordinates, station commands, and equipment executes. It is not specified whether the intent is for control to be hierarchical or heterarchical.

The CIM-OSA cube is described.

[Skevington1] ***Manufacturing Architecture for Integrated Systems***

This paper gives an abstract discussion of CIM architectures.

A few elements of a CIM architecture are proposed: a *"meta database"*, a *"meta operating system"*, and a *"distributed resource manager"*.

"A manufacturing system architecture is the composite model of the virtual system configurations seen by each of the users."

[Spector1] ***Supervenience in Dynamic-World Planning***

This is a Ph.D. dissertation. It discusses the differences between planning in a static world and planning in a dynamic world. It also deals with a number of philosophical concepts related to control architectures.

The paper distinguishes *"teleological goals"* (trying to change the state of the world) from *"teleoepistemic goals"* (trying to learn something).

The paper introduces the notion of *"supervenience"* and a *"supervenience architecture"*.

"Domain A supervenes on domain B just in case representations in domain A depend on representations in domain B; that is, just in case, in matters of interest to both domains, the representations in domain B take precedence (because domain B is "closer to the world"). The motivation for the use of supervenience rather than just "reduction" of A-representations to B-representations, is that A and B might use different languages (with neither a subset of the other), different rules of inference, etc."

A symbolic logic view of supervenience is presented.

The paper describes an implementation of the architecture (called APE - Abstraction Partitioned Evaluator) for the problem of a robot called *"homebot"* performing household chores. A LISP simulation with a moderately complex model of the robot and its physical environment was used.

The author's vocabulary and range of expression are broader than those used in any other paper reviewed for this bibliography.

[Stecke1] Stecke, K. E.; ***Design, Planning, Scheduling, and Control Problems of Flexible Manufacturing Systems***; Annals of Operations Research 3; 1985; pp. 3 - 12

This paper gives a brief overview of problems in flexible manufacturing system (FMS) design, planning, scheduling, and control (as the title says). It is arranged in an easily scanned outline form.

The paper focuses on problems of designing a specific FMS, which is expected to stay in place for some time. Architecture issues are not tackled.

A few of the problems listed are:

Design (13 total)
- a. Determine the range of part types to be produced.
- b. Specify the type, then capacity of the material handling system.

Planning (5 total)
- a. Partition the machines of each type into machine groups.
- b. Determine the production ratios.

Scheduling (3 total)
- a. Appropriate scheduling methods have to be developed.

Control (4 total)
- a. Determine maintenance policies.

[Szabo1] **<u>RCS</u>** ***Evaluation of current RCS Methodology***

This memo makes the points:

(1) RSD already has several methodology documents.

(2) The term *"RCS Methodology"* is too inclusive, since it covers several ideas of various levels of acceptance.

(3) Current methodology documents do not describe a formal process.

(4) There is no plan for putting a methodology in place.

(5) There are several conformance issues once a methodology is in place.

The memo recommends:

(1) Produce a methodology like any engineering product.

(2) Develop a classification scheme for methodology concepts. Then decide which can be agreed on.

(3) Have conformance classes for implementations.

(4) Define implementation domains according to requirements.

The author believes a formal RCS specification for *“design methodology”* can be developed quickly within RSD. He contrasts *“design methodology”* with *“implementation methodology”*. It appears he intends that a design specification is part of a *“design methodology”*.

[Szabo2] **RCS** ***Control System Architecture for Unmanned Land Vehicles***

This is a brief overview of the architecture of a control system developed at NIST for controlling an unmanned ground vehicle running on a road. RCS methodology and the specific architecture for this system are presented briefly.

Rather than perception being part of sensory processing, it is one of two main tasks of the system, the other being mobility.

The architecture and system diagram for the mobility controller are given in figures 2 and 3.

A task called *“retro”* is described, in which the vehicle travels in the opposite direction along the path it has been following.

[Szabo3] **RCS** ***Control System Architecture for Unmanned Ground Vehicles***

This paper describes how NASREM could be applied to the U.S. Army TEAM project for controlling multiple unmanned ground vehicles.

The second section summarizes the NASREM architecture. The third section describes the TEAM application, including a mission scenario. The fourth section describes RCS hierarchical task decomposition as applied to the TEAM application. The fifth section covers a few communications issues.

[Szabo4] **RCS** ***Control System Architecture for a Remotely Operated Land Vehicle***

This paper gives details of the TEAM controller architecture.

The *“Introduction”* section briefly describes the TEAM program, the application scenario (two Robotic Combat Vehicles operating under remote control), and RCS architecture.

The *“Control Architecture”* section gives detailed controller hierarchies for TEAM Robotic Combat Vehicles and their superiors.

The *“Mobility Design Details”*, *“Remote Control”*, and *“Retro-traverse”* sections describe how remote control and automatic back-tracking are handled in the application.

The *“Communications Design Details”* section presents what its title says.

The *“Implementation”* section describes the computer and communications hardware and some software used in the control system.

[Szabo5] **RCS** ***Control System Architecture for the TEAM Program***

This is nearly identical to [Szabo3].

[Ting1] ***A Cooperative Shop-Floor Control Model for Computer-Integrated Manufacturing***

This paper presents a heterarchical system for shop floor control, including resource allocation and a bidding procedure. The shop floor model includes: a communications network, work modules, a management module, a transportation network, hardware storage modules, and a software module.

The model uses *"pull control with deferred commitment*."

Local optimality is claimed.

The conclusions include: *"The performance issue of the control model in a large scale FMS, however, is not addressed in this paper*," and *"Another limitation of this model is its inability of providing global production solutions beyond the work in process inventory control*."

Figure 1 shows an unusual view of the context of shop-floor control.

[Upton1] ***Architectures and Auctions in Manufacturing***

This is a well-written article discussing the problem of predicting what a small portion of a heterarchical system will do under some simple assumptions about bidding-contracting and queuing procedures.

The authors sound sympathetic to heterarchical systems, but conclude that *"the queue-theoretical analysis of even the simplest auction-based system is difficult*," and *"There are a large number of technological issues which need to be resolved before large manufacturing systems may be controlled in this manner*."

The author hints at hybrid heterarchical and hierarchical architectures not discussed in this paper.

[Vamos1] ***Cooperative Systems Based on Non-Cooperative People***

This is a semi-popular style article endorsing heterarchical systems. Contrary to the title, the autonomous units in the system are assumed to be cooperative.

The article makes the point that controlling large systems and predicting their behavior becomes non-linearly more difficult with the size of the system. The article suggests that, faced with the problem being impossible, it is more reasonable to deal with the problem using a network of cooperating autonomous agents than a hierarchy, since this seems robust.

The article acknowledges that there are *"modeling problems"* with heterarchical systems, such as determining stability.

[VanHaren1] ***A Reference Model for Computer Integrated Manufacturing from the View Point of Industrial Automation***

This paper mentions other reference models. Table I lists eleven *"reference models for CIM"* and gives citations. Table II lists *"characteristics of CIM reference models"*, but is very terse.

The paper describes, in general terms, a CIM Reference Model developed by the International Purdue Workshop on Industrial Computer Systems.

"In Figure 9 [page 56] the CIM model is represented in the implementation hierarchy view by an eleven layer structure." The layers are described on pages 57 and 58.

The paper says that more details are given in a book about the model.

[Wallace1] **MSI** ***Control Entity Interface Document***

This is an MSI paper presenting a summary of the MSI architecture and a detailed specification of the interfaces in the architecture. This paper supersedes the 1991 NISTIR of the same name, whose principal author was Senehi.

A *"control entity"* includes one or both of a *"planner"* and a *"job controller*." Control entities which have job controllers are always arranged hierarchically with zero (at the top) or one (everywhere but the top) superior and zero to many subordinates. Planning may be done centrally, hierarchically, or in mixed mode.

Control hierarchies may go to different levels on different branches. The top level is called *"shop"*, the bottom *"equipment"*, and everything in between *"workcell"*. There is no preset number of levels. There is no presumption about the speed of any controller, the physical extent of what the controller controls, or the number of its subordinates.

There is a communications paradigm for communicating between entities. The communications media carry messages between entities. Messages are in two types: *"request/response pair"* and *"unconfirmed message"*.

The architecture uses three kinds of plans: process plans (generic recipes), production-managed plans (created from process plans), and production plans (scheduled activities with resources allocated). Production plans are what gets carried out. The paper does not deal with the creation or definition of process plans or production-managed plans. The paper deals with creating production plans from existing production-managed plans.

Production plans and the steps of which they are composed have states. Planners in separate control entities deal with each other by a negotiation procedure whose details are given in the paper.

Error recovery for scheduling errors is possible using the messages defined in the specification. Error detection and recovery capability for resource and execution errors is included via a guardian interface and a watchdog interface.

The guardian interface includes a console interface for a human. It may be used to reconfigure the control system and for several other purposes.

The paper devotes a chapter to each of the 5 types of interface in the architecture: planning to planning, job control to job control, planning to job control, guardian to planning, and guardian to job control.

Much of the paper is devoted to tables giving the explicit semantics of various messages and responses. There are also several state transition diagrams.

The paper includes a sample error recovery scenario which has been worked out in detail.

The paper includes a glossary. Almost all terms in the glossary are specific to the architecture, rather than being commonly used terms.

The paper defines *"data objects"* of several types but does not discuss other database issues. Although database issues are important in MSI effort, they are handled in other papers.

[Wavering1] **RCS** ***Manipulator Primitive Level Task Decomposition***

This paper describes how the primitive (as defined in NASREM) level task decomposition function works in the case of a trajectory module in a hierarchical manipulator control system.

No specific robot is described to which this control system is intended to be applied.

The primitive level task decomposition module is divided into the usual three parts defined in RCS: job assignment, planning and execution. These three parts have defined interfaces to each other. Several of these parts also each have their own interfaces to external modules, including: prim job assignment to world modeling, prim planning to world modeling, prim execution to world modeling, e-move to prim job assignment, and prim execution to servo task decomposition.

A lot of technical issues associated with trajectory planning and execution are discussed.

Timing and processing power requirements are discussed.

The controller under discussion is capable of queueing commands.

The paper notes on page 5 that *"there is no direct correspondence between sensory processing levels and task decomposition levels."*

The paper contains an extensive bibliography.

[Wavering2] **RCS** ***The Real-Time Control System of the Horizontal Workstation Robot***

This is a user and system programmer's manual for the control system for the T3 robot in the NIST AMRF Horizontal Workstation. The control system methods of this paper are those described in [Leake1].

Chapter 2 describes the functions the robot performed in the workstation and the overall architecture of controlling the robot and its ancillary equipment, which included three different grippers (usable one at a time), a vision system (not described in this report) and an active pedestal vise. Section 2 of the chapter gives an *"Overview of the RCS-II Architecture"* including:

2.1 *"Hierarchical Task Decomposition"* - includes a controller hierarchy diagram,
2.2 *"Generic Control Process Structure"* - uses pre-process, decision processing, post-process paradigm where the decision processing uses

two levels of state table; the top one just selects the correct state table for the particular command,

2.3 *"Cyclic Execution,"*

2.4 *"Common Memory"*.

Chapter 3 shows the computer and other electric or electronic hardware configuration.

Chapter 4 tells how to start the system up, run it, and shut it down.

Chapter 5 describes tasks and command messages for the prim, e-move, subtask, and task level of the hierarchy. Chapter 6 does the same for other control system processes.

Chapter 7 describes the low-level (at the memory block level!) data structures used by the system.

Chapter 8 gives an example of entering data for a new part.

Chapter 9 describes external communications, which included communications with two controllers in the hierarchy (vision and active pedestal) as well as the (superior) workstation controller and the (completely external) AMRF database.

[Wendorf1] ***Structured Development of a Generic Workstation Controller***

This paper describes a workstation controller with four interface types: command and status to superior, command and status to subordinates, load recipes, and exchange parts.

A language called LOTOS is used for defining controller interfaces and representing control processes. LOTOS has a generic concept of interactions, temporal ordering principles, mechanisms for process abstraction and abstract data types. It can represent when processes are able to interact. It was developed by ISO for the specification of computer protocols.

With respect to the activities a controller can carry out (which are expected to be scheduled), the paper distinguishes between individual steps, like grind or drill, and sets of steps which have to be executed sequentially (which it calls *"operations"*). The sets are useful because they are the units which are scheduled. This is a nice idea for simplifying the job of scheduling.

In the implementation, C++ code for a controller is generated automatically from LOTOS statements.

Internally, a controller may be broken down into three modules: a command receiver and dispatcher and part exchanger, a world modeler, and a status receiver and dispatcher. No rationale is offered for this decomposition. The modules may be represented by separate LOTOS processes.

No communications protocol is discussed. All controllers in the implementation described run on the same board, so shared memory is used.

The implementation includes something defined as a *"queue"*, but only one item at a time may be put into the queue.

[Weston1] ***Highly Extendable CIM Systems Based on an Integration Platform***

This paper presents an approach to CIM integration that is not quite an architecture. It is intended to cover almost the entire life cycle of a product: *inception, design, manufacture, sales, re-engineering, and field support/maintenance*. It is intended to be an immediate solution to the real-world CIM integration problem.

CIM is described as having three architectures: communication, application, and information. A facility called AUTOMAIL, being developed in a research environment, is described which addresses all three architectures:

"The AUTOMAIL connection architecture provides an application process with a uniform communication interface … The action architecture provides applications with a uniform message set … the AUTOMAIL information architecture provides each application with a standard interface, currently SQL, into the available information resources."

A second, prototype, system called SEFIMA, similar in purpose to AUTOMAIL, is being developed in an industrial environment.

The system descriptions are brief.

[Yoder1] Yoder, James R.; ***Toward a New CIM Architecture for Sandia Laboratories***; Proceedings of CIMCON '90, NIST Special Publication 785; National Institute of Standards and Technology; May 1990; pp. 326 - 333

This paper provides an example of the kinds of things one laboratory wants to incorporate in a toolkit for CIM. It includes a brief discussion of how the scope of the architecture was determined.

Appendix E - Detailed Comments on Other Architectures

E.1 CIM-OSA

CIM-OSA (Computer Integrated Manufacturing - Open Systems Architecture) is an architecture being developed by the ESPRIT (European Strategic Program for Research and development in Information Technology) AMICE (European CIM Architecture) project [Chen1], [Jorysz1], [Jorysz2], [Klittich1], [Klittich2], [Panse1], [Shorter1]. The aim of the architecture is to provide an integrated framework to support manufacturing within an enterprise. The documentation of CIM-OSA does not define clearly what the framework is. It includes, at least, an integrated data system architecture for manufacturing enterprises.

The CIM-OSA architecture was discussed briefly in Section 6.3.1. This appendix gives further details.

The CIM-OSA scope (as shown in [Jorysz1, Figure 4], for example) is limited to CIM but looks at the whole system life-cycle, including consideration of requirements specifications at one end and system change at the other.

The CIM-OSA architecture has three levels of architectural definition: generic, partial, and particular, but the CIM-OSA documentation does not give many details about the intended nature of each level.

The data system architecture provides “front end services” to users. There are four types of front end service: application (e.g., CAD or CAPP), human, machine (e.g., robot or NC machine tool), and data management. The front end service uses a “data access protocol” to provide the requested service. The two data access protocols are “business process services” and “information services”.

Although the long-term data system is intended to be as just described, Release 1 of CIM-OSA includes a different data system architecture [Klittich2], which is conceptually similar to the IMDAS (Integrated Manufacturing Data Administration System) architecture used in the AMRF [Barkmeyer1]. In the Release 1 architecture, each application has a database divided into private and public parts. To deal with public data, each application has two data service modules: (i) an application front end module connected to the application and (ii) a data management module connected to the public part of the application’s database. The two modules are connected to a system-wide data agent.

It appears that all non-private data is intended to be globally accessible. There is no provision for direct data exchange between applications.

No specific communications architecture is prescribed, although the documentation mentions OSI.

The CIM-OSA documentation puts emphasis on the “CIM-OSA cube”, which is shown in Figure 11, a reproduction of figure 1 from [Jorysz1, page 147]. The three dimensions of the cube are stepwise generation (values: function, information, resource, organization), stepwise instantiation (values: generic, partial, particular), and stepwise derivation (values: requirements definition, design specification, implementation descriptions).

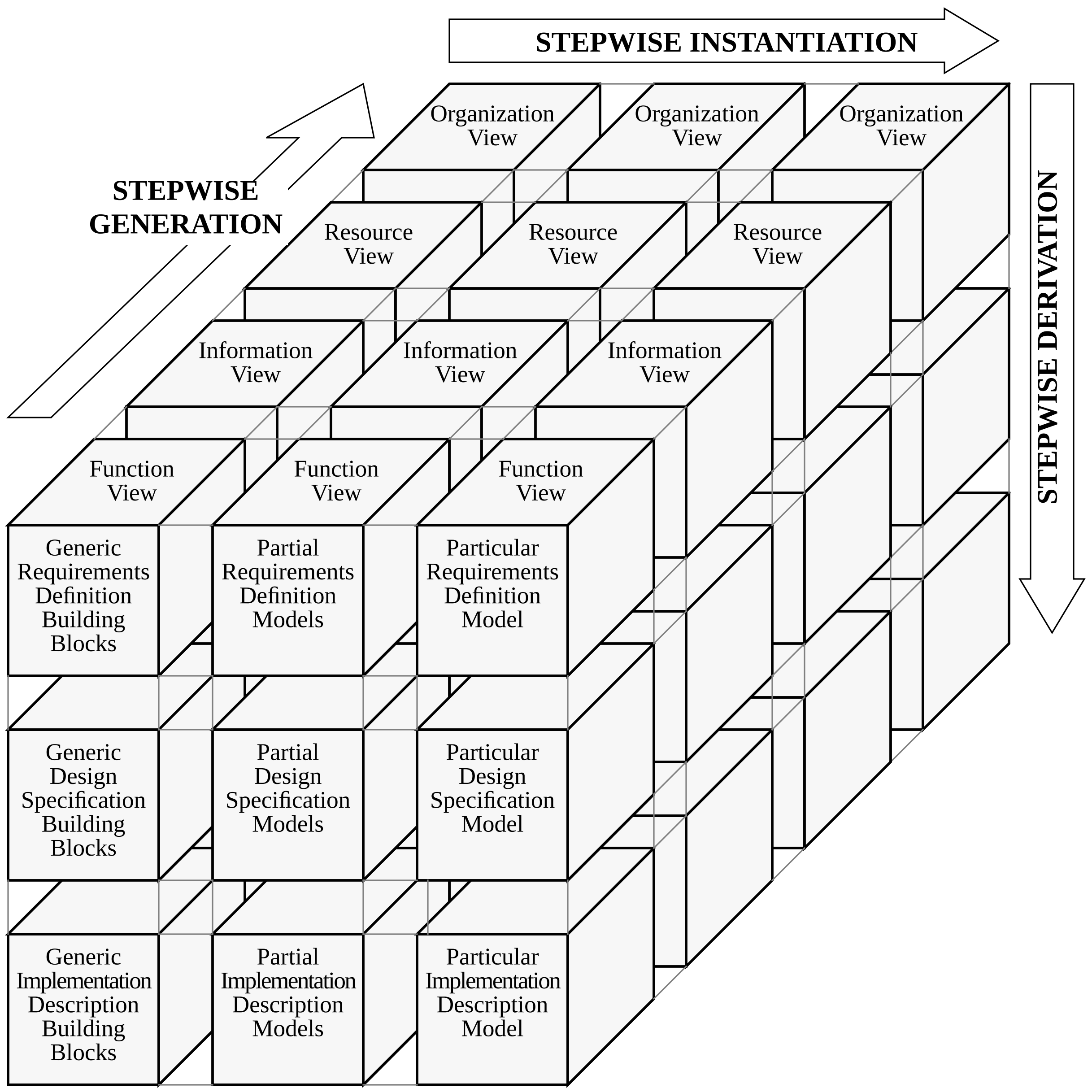

STEPWISE INSTANTIATION
STEPWISE GENERATION
STEPWISE DERIVATION
Organization View
Organization View
Organization View
Resource View
Resource View
Resource View
Information View
Information View
Information View
Function View
Function View
Function View
Generic Requirements Definition Building Blocks
Partial Requirements Definition Models
Particular Requirements Definition Model
Generic Design Specification Building Blocks
Partial Design Specification Models
Particular Design Specification Model
Generic Implementation Description Building Blocks
Partial Implementation Description Models
Particular Implementation Description Model

Figure 11. CIM-OSA Architecture: Overview

CIM-OSA takes a very broad view of the scope of a CIM architecture. It intends to cover the entire life cycle of a system, including:

(1) system requirements specification,

(2) system design,

(3) system description for build and release,

(4) system operation,

(5) system change.

CIM-OSA intends to go so far as to provide for automatically generating executable code from the requirements specification.

The CIM-OSA architecture is a work in progress. Prototype applications conforming to the architecture are only now being built.

Surprisingly, CIM-OSA does not include control in the architecture specifications. There is not even any discussion of control processes as independent entities.

The CIM-OSA cube does embody some methodology for architectural development. The stated intent for developing systems is to go from left to right on the cube (generic to partial to particular) and top to bottom (requirements definition to design specification to implementation description). The third (front to back) axis, labelled stepwise generation, is not intended to be ordered, although the arrow on the figure makes it look ordered. It is not stated how many of the 36 mini-cubes that make up the CIM-OSA cube are intended to be populated in the process of developing a control system.

CIM-OSA does not propose any specific languages or techniques for expressing an architectural design or building an architecture.

The CIM-OSA example is useful for considering the scope of an architecture and provides the skeleton of a methodology for architectural development, but it has little to say directly about control.

E.2 Dornier

The Dornier architecture was discussed briefly in Section 6.2.2.2. This section provides additional details and background on the architecture.

Under contracts from the European Space Agency, the Dornier firm produced several papers [Dornier1], [Dornier2], and others concerning control architectures. One of these [Dornier2] proposes a reference architecture for European space automation and robotics control systems. The architecture is intended to be suitable for at least robot systems, surface roving vehicles, and dedicated automation equipment. The paper makes a detailed presentation of the architecture. The same architecture is summarized more briefly in the latter part of [Dornier1]. The authors are not aware of any implementations of the Dornier architecture.

The Dornier architecture has four levels of architectural definition.

A three layer control hierarchy is proposed [Dornier2, page 52]. From the top down, the layers are named A&R Mission Planning and Control, Task Planning and Control, and Action Planning and Control. The use of a hierarchy is justified [Dornier2, pages 45-46], but no rationale is offered for why three layers are suitable, rather than four, six, or a variable number depending upon the application.

Each controller has three major modules: nominal feedback functions, forward control functions, and non-nominal feedback functions, as shown in Figure 12, which reproduces Figure 5.4.1-1 from [Dornier2, page 53]. For a single controller, six types of data flow between modules, four types of data flow to external systems, seven types of data flow with superiors, and seven types of data flow with subordinates are specified.

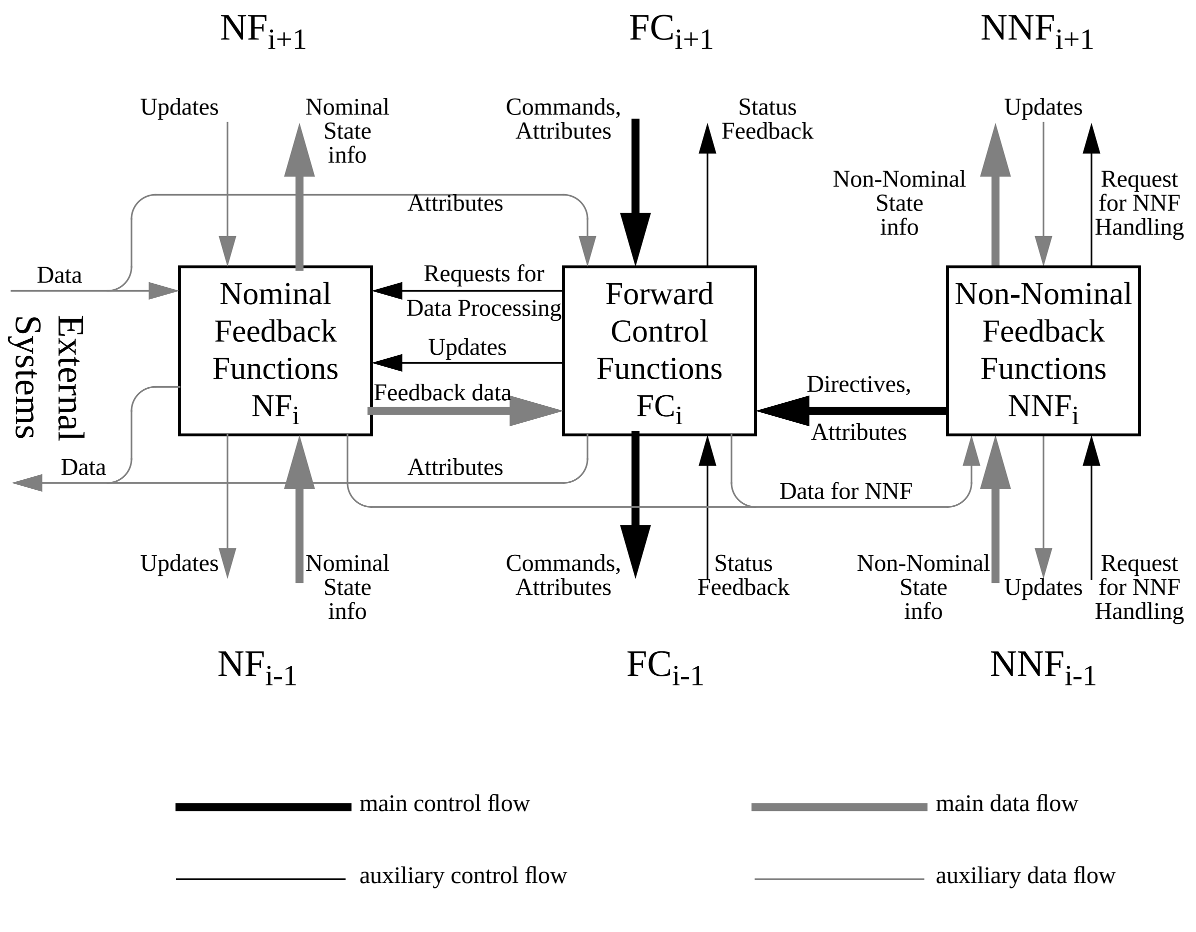


Figure 12. Dornier Architecture: A Controller

No rationale is offered for the decomposition of a controller into three modules in this paper, but it is clear a great deal of thought went into the decomposition. Perhaps other papers in the same series (which are referenced in [Dornier2]) present a justification. The interconnections between layers and modules are so numerous that control system behavior and performance may be hard to predict. The ability of the non-nominal feedback module to do planning and give "directives" to the forward control module looks particularly worrisome.

The Dornier architecture provides the most formalized methodology for architectural development of all the architectures examined for this report. The methodology uses the "structured analysis and design technique" (SADT). The steps and documents required by the methodology are shown in overview in figures 3.3-1 [Dornier2, page 17] and in more detail in figures 3.3-2, and 3.3-3 [Dornier2, page 19]. The control development methodology is discussed in detail in [Dornier2, pages 95-106].

Figure 13 is a modified version of figure 3.3-1 from [Dornier2]. As shown in the figure, the documents to be produced in the course of developing a specific implementation include: activity script, application architecture, operations architecture, and logical model of control system. The functional reference model mentioned in the figure is the application-independent reference model described in section 5 of [Dornier2] and briefly described above. The physical model of the control system shown at the bottom of the figure is the actual hardware and software of the control system.

Activity scripts are prepared using an activity analysis methodology (ActAM) and written in an activity scripting language (AcSL) which has not been rigorously formalized. Examples of scripts are given in an appendix of [Dornier2].

As described in [Dornier2], the Dornier proposal does not deal much with database, communications, resource allocation, scheduling, or dynamic reconfiguration issues. There is no discussion of world modeling.

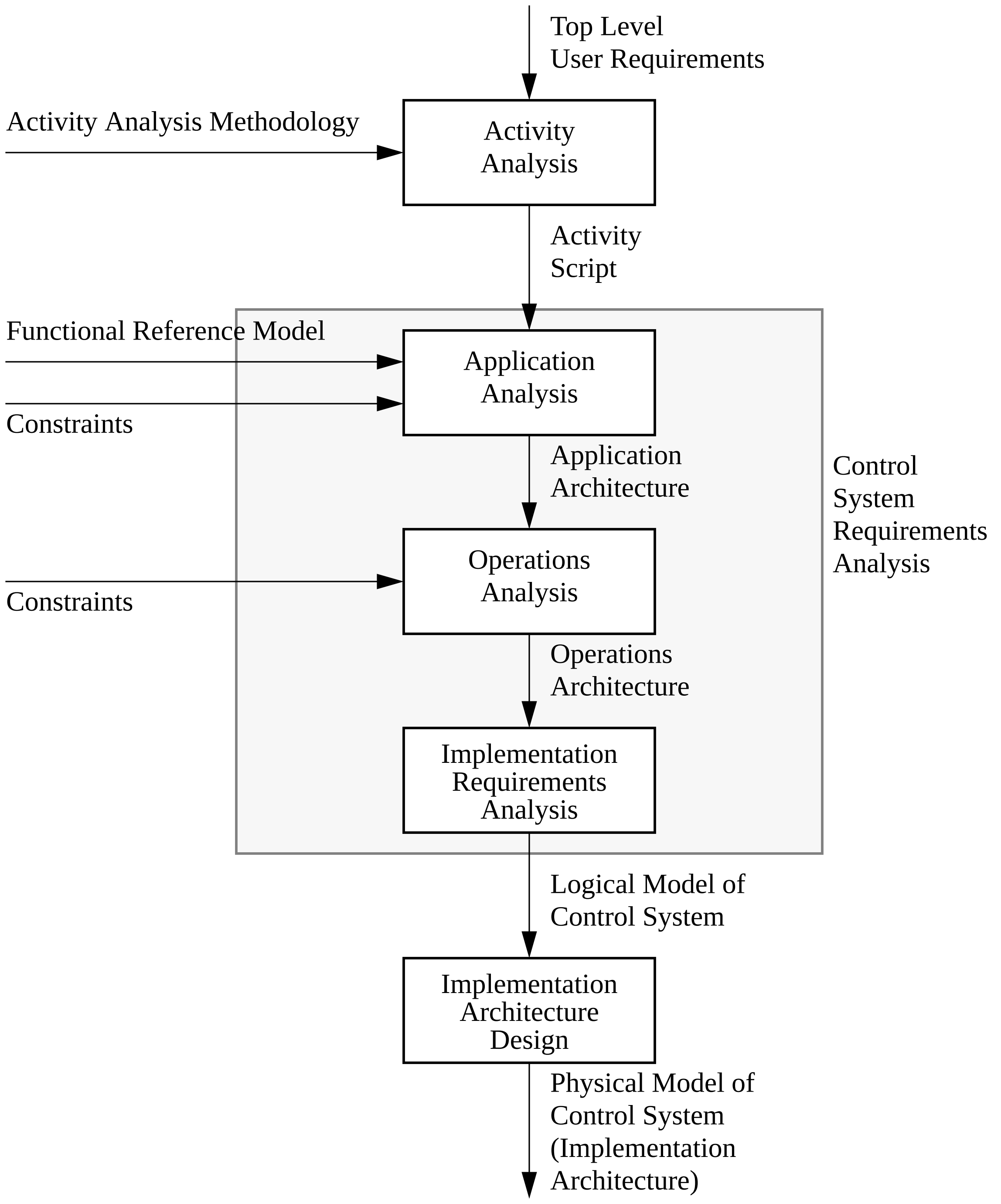


Figure 13. Dornier Methodology for Architectural Development

Appendix F - Additional Hierarchical Controller Issues

In the course of preparing this report, several issues were considered that were too specific to hierarchical controller architectures to raise in sections 4 or 5. Most of these have arisen in RCS discussions. They are listed in this appendix with little discussion.

F.1 Hierarchical Levels

F.1.1 Significance of Hierarchical Levels

Levels in a hierarchy may be given a great deal of significance or little, depending upon the specification of what constitutes a level. There are at least three different ways of making this specification.

In the first method, a controller hierarchy is built from the top down (the way a function call hierarchy in a computer program is normally built) without considering classes of controllers with similar characteristics. The level of any controller may be determined after the fact by just counting the length of the control chain from the top of the hierarchy to the controller in question. The controller at the top of the hierarchy is the only controller in level 1, its subordinates constitute level 2, the subordinates of level 2 constitute level 3, and so on. In this case, the level of a controller tells little, and the controllers in a level cannot be expected to have similar characteristics. If a level is given a name (e.g. “shop level”, “cell level”, etc.), the name may well not reveal anything useful about the level.

In the second method, a controller hierarchy is built from the bottom up, and all the controllers in the lowest level control equipment (e.g., robot, machining center, vise), so that the name “equipment level” may be meaningfully applied to the lowest level. The equipment is grouped by some method (usually by grouping equipment whose actions must be coordinated), and a superior controller is defined for each group. The set of controllers superior to equipment controllers then constitutes the next level up. These controllers are grouped under superior controllers, and so on until there is a level with only one controller. That is the top level. The length of the chain between an equipment controller and the top level may vary.

In the third method, the architecture may specify that the controller hierarchy of any implementation must include a fixed number of named levels and that there must be certain similarities among all the controllers in a given level. For example, there may be shop, task, primitive, and servo levels, and all controllers at the primitive (or any other) level may be required to have the same cycle time. Doing this might make it feasible, for instance, to run all the controllers in a level on the same processor.

Some issues arise only if the third method - where controllers in a level are required to have similarities - is used. For example: can a controller have a subordinate in the same level or from more than one level down? Under the first or second method, the statement of this issue does not even make sense.

F.1.2 Number of Hierarchical Levels

Should an architecture require a fixed number of hierarchical levels? If so, how many levels should there be?

If the architecture allows a variable number of levels, what basis should be used to determine when a new level should be added?

F.1.3 Characteristics of a Hierarchical Level

If controllers are assigned to levels based on having similar characteristics, what should these characteristics be - cycle time, nature of tasks handled, or what? If controllers are similar with respect to one characteristic, is it reasonable to expect them to be similar with respect to other characteristics?

F.2 Decomposition

F.2.1 Temporal Decomposition

What is the importance of having lower hierarchical level controllers run faster? Should this be an architectural requirement?

RCS specifies an order of magnitude factor in temporal resolution (e.g. planning horizon) from level to level. Having lower levels run fast is often critical for hard real-time operation. In some implementations of Barbera RCS, all of the controllers execute at the same rate but their goal completion rates are faster at lower levels.

For many control law algorithms, having lower levels run faster is required for stability.

F.2.2 Spatial Decomposition

What is the importance of having lower level controllers deal with smaller physical areas? Should this be an architectural requirement?

RCS specifies an order of magnitude in spatial resolution (e.g. maps) from level to level.

F.3 Subordinates Per Superior

Should a range of numbers of subordinates per superior be specified? RCS specifies 5 to 9, for example. Limiting the number of subordinates is useful for managing complexity and making the control system easier to understand, particularly if the subordinates perform many different functions.

F.4 Controller and Task Hierarchy Design

What are the best methods of building controller hierarchies and task hierarchies? How do the two jobs interact?

Without offering a full discussion of this issue, we note that there is one well-accepted way to get started with task and hierarchy design.

In building many implementations, the lowest level of the control hierarchy and the lowest level of tasks are defined around the equipment, because it is usually desired to use existing types of equipment, it is usually straightforward to identify what these types are, and existing equipment usually comes with a specialized controller. Thus, starting the definition of controller hierarchies and tasks at the equipment level is generally effective. There is one controller for each piece of equipment, and the atomic tasks that an equipment controller can do are the things the equipment can do.

Appendix G - Index to Glossary Terms

This appendix gives the page numbers on which terms in the Glossary are defined or discussed. It is not an index to all uses of the terms.

Appendix H - Index to Citations

This appendix gives the page numbers on which each of the References is cited. All citations are included here.