SANDIA REPORT
SAND97-2076 • UC-706
Unlimited Release
Printed August 1997

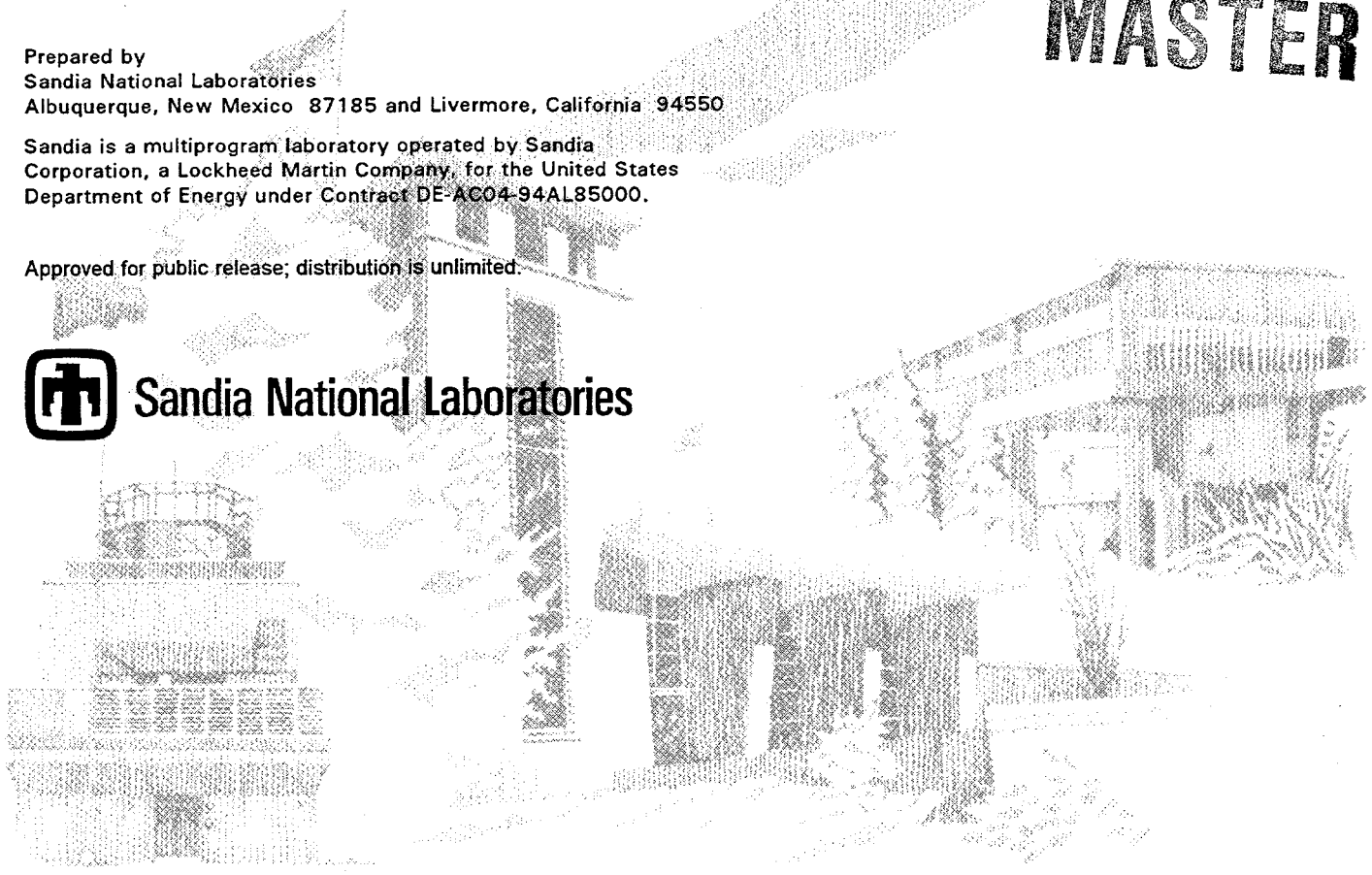
Final Report: An Enabling Architecture for Information Driven Manufacturing

J. Michael Griesmeyer

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A07
Microfiche copy: A01

SAND 97-2076
Unlimited Release
Printed Auigust 1997

Distribution
Category UC-706

Final Report: An Enabling Architecture for Information Driven Manufacturing

J. Michael Griesmeyer
Advanced Engineering and Manufacturing Software Development
Intelligent Systems and Robotics Center
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1010

Abstract

This document is the final report for the LDRD: An Enabling Architecture for Information Driven Manufacturing. The project was motivated by the need to bring quality products to market quickly and to remain efficient and profitable with small lot sizes, intermittent production and short product life cycles. The emphasis is on integration of the product realization process and the information required to drive it. Enterprise level information was not addressed except in so far as the enterprise must provide appropriate information to the production equipment to specify what to produce, and the equipment must return enough information to record what was produced. A production script approach was developed in which the production script specifies all of the information required to produce a quality product. A task sequencer that decomposes the script into process steps which are dispatched to capable Standard Manufacturing Modules. The plug and play interface to these modules allows rapid introduction of new modules into the production system and speeds up the product realization cycle. The results of applying this approach to the Agile Manufacturing Prototyping System are described.

This page intentionally left blank

Contents

List of Figures

Final Report: An Enabling Architecture for Information Driven Manufacturing

1 Introduction

This document is the final report for the LDRD: An Enabling Architecture for Information Driven Manufacturing. The project was motivated by the need to bring quality products to market quickly and to remain efficient and profitable with small lot sizes, intermittent production and short product life cycles. The emphasis is on integration of the product realization process and the information required to drive it. Enterprise level information was not addressed except in that the enterprise must provide appropriate information to the production equipment to specify what to produce, and that the equipment must return enough information to record what was produced.

We envisioned an information driven agile product realization process. The design starts with specifications of functionality, manufacturability, assembly requirements and life cycle constraints. Integrated Product and Process Development is used to concurrently engineer both the product an the processes required to produce it. Virtual prototyping is used to test the design. Virtual manufacturing validation is used to test the production system, and to monitor the actual production and verify product quality. The production system uses product and process specifications to produce the product right the first time.

We developed a Production Script Approach to Information Driven Manufacturing to address this vision. The main features of the approach are shown in Figure 1-1. The pro-

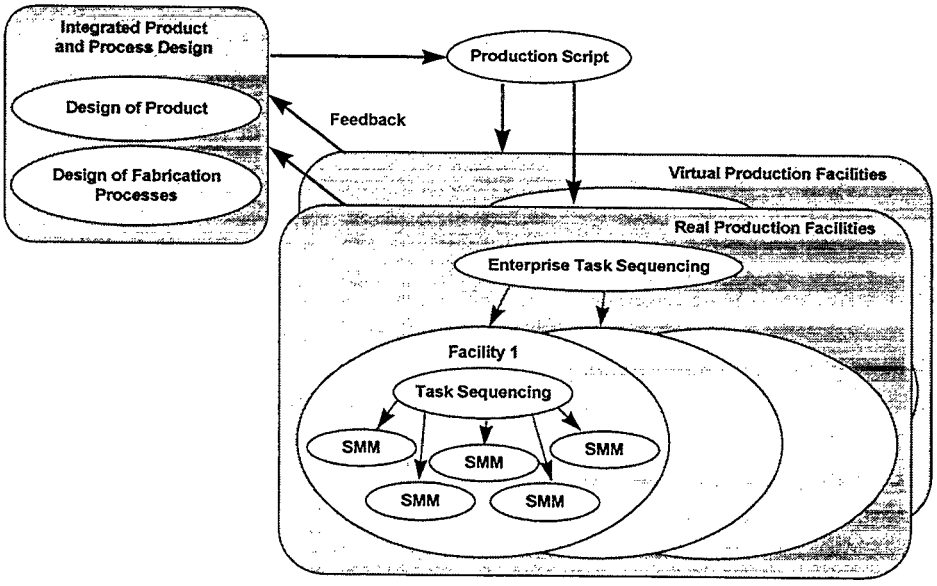


FIGURE 1-1 Production Script Approach to Information Driven Manufacturing

duction script is a hierarchical sequence of manufacturing process steps that is the output

of the product and process design efforts. A task sequencer dispatches the individual production process steps to modular components of the production system which can also be hierarchical. The production script can be used to drive both the real production system and simulated production system thus allowing virtual prototyping of production.

The basic elements of the approach are:

- a production script which contains all of the information required to produce a quality product and specifies what must be recorded to verify the quality of the product.
- descriptions of production system modules that indicate their capabilities.
- standardized interfaces through which production modules interact with other system components.
- task sequencing based upon match subsystem capabilities to the requirements of the manufacturing process steps.
- support of production system simulation.

The document provides a summary of these basic elements, their application to the Agile Manufacturing Prototyping System (AMPS)[1], and follow on work. Appendices provide detailed definitions of the generic interface to modular production subsystems called Standard Manufacturing Modules (SMMs), the production script grammar and the SMM description grammar. The appendices are working documents of intermediate results that illustrate the status of the specifications at the end of the project.

During the first two years of the project, the information models for the production script, equipment description and equipment interface were developed together with the associated grammars. A preliminary task sequencer driven by a simplified script was developed to test the basic concepts and apply them to AMPS. The last year was used to refine the task sequencer to use the complete script grammar, provide subsystem intelligence required to allow the SMMs to execute larger script components, and develop ideas for follow on work.

The next section provides some background information for the project.

2 Background

Years of experience developing prototype robotic systems in the Intelligent Systems and Robotic Center (ISRC) at Sandia National Laboratories lead to the development of the Generic Intelligent Systems Control (GISC) approach to system development[2]. The Robotic Technology Development Program (RTDP) of the U.S. Department of Energy adopted the GISC approach for the development of robot control system software in September, 1990, to stimulate rapid prototyping of robot systems and facilitate multi-laboratory teaming.

A basic premise of the GISC approach is that sophisticated robot system performance can be obtained by coordinating the operation of a collection of subsystems, each with complementary capabilities. This approach is analogous to the functioning of a team of highly

skilled individuals. Each team member has particular skills which, under the supervision of a team leader, integrates with the skills of the other team members to provide a highly effective group with problem solving abilities beyond those of the individual members of the team. As problems change, the team may be modified by the addition or substitution of specific members to add needed skills, but the basic structure remains intact as well as many of the original team members.

Key elements of the GISC approach are: delegation of responsibility to appropriate subsystems, well defined interfaces between the subsystems, orchestration of subsystem operations to perform complex tasks, and reusable subsystems. The collected software tools that were used in many of the RTDP projects became known as GISC-Kit.

Many of the GISC applications were for relatively unstructured environments leading to a large variety of interfaces between subsystems. However, for more structured environment such as the automated chemistry laboratories developed in the Contaminant Analysis Automation (CAA) program of RTDP[3], standardized interfaces for generic subsystems and standardized supervisory control seemed promising.

At about the same time as the initiation of the CAA project, the interest in agile product realization was starting at Sandia National Laboratories. Manufacturing process centered tools started to be developed. These tools were intended to provide the process developer for a product with manufacturing process specific advisors to assist in the design of the production process. The first of these tools were developed in the FASTCAST[4] and SMARTWELD[5] projects.

The output of the process design is a certified production process specification. If the specification could be standardized to permit automated interpretation, it was hoped that standardized interfaces to production equipment and standardized supervisory control would permit automation of much of the production system in a "plug and play" fashion. Further, if the production system could run in both virtual and real modes, considerable improvement in the product realization process should result. This LDRD project was intended to explore these possibilities.

3 Production Scripts

The first task in developing the production scripts was to enumerate the types of information required to completely specify the production process in order to obtain the desired quality. Grammars could then be developed to express the information. The grammars could be textual or graphical in nature. Textual grammars allow platform neutral representations of the scripts but are difficult to develop and the text programs using the grammar are hard to write.

The following types of information are required in the production script:

Task Sequencing

Must specify the sequence of tasks/operations.

Step Specification Hierarchy/Script Decomposition

Must allow hierarchical representations to permit multiple levels of detail. This is required for both visualization of processes and assignment to modular production system components (or agents) for execution. Decomposition is illustrated in Figure 3-1.

Step Specification and Control

Must specify the operation to be performed at each step in the manufacturing process together with the information required to control the process.

Temporal Constraints

Must specify any temporal constraints such as timing between processes. For example, the allowable minimum and maximum time between application of adhesive and mating of parts to be joined may be required.

Agent Constraints/Certification

Must specify the constraints on the selection of agent to perform the step. Constraints may be the training required for an operator of a piece of equipment or standards with which the equipment itself must be in compliance.

Environmental Constraints

Must specify any environmental constraints such as temperature and humidity that may effect desired quality.

Material Stream

Must specify the physical inputs and outputs of the process step.

Script Tuning

As the script is decomposed, constraints at the higher levels may need to be mapped to specification and control of steps at lower levels in the decomposition.

Step Setup

Must specify any setup requirements for a step. Upon decomposition of a step into smaller steps, the setup requirements may be resolved into separate step.

Step Records

Must specify what needs to be recorded in order to verify step execution.

Equipment Program Description

Must specify enough information about a program to be downloaded to a piece of equipment to allow comparison against equipment capabilities. This is required to permit capability based dispatching of process step.

A simplified script grammar was constructed to test the basic approach to task sequencing and dispatching of process steps to plug and play equipment according to equipment capabilities. The approach to plug and play equipment is described in Section 4.

An extended grammar for the production scripts was developed that attempted to address each of the kinds of information described above. A compiler was developed for the grammar using lex and yacc[6]. The technical description of the grammar is given in Appendix C.

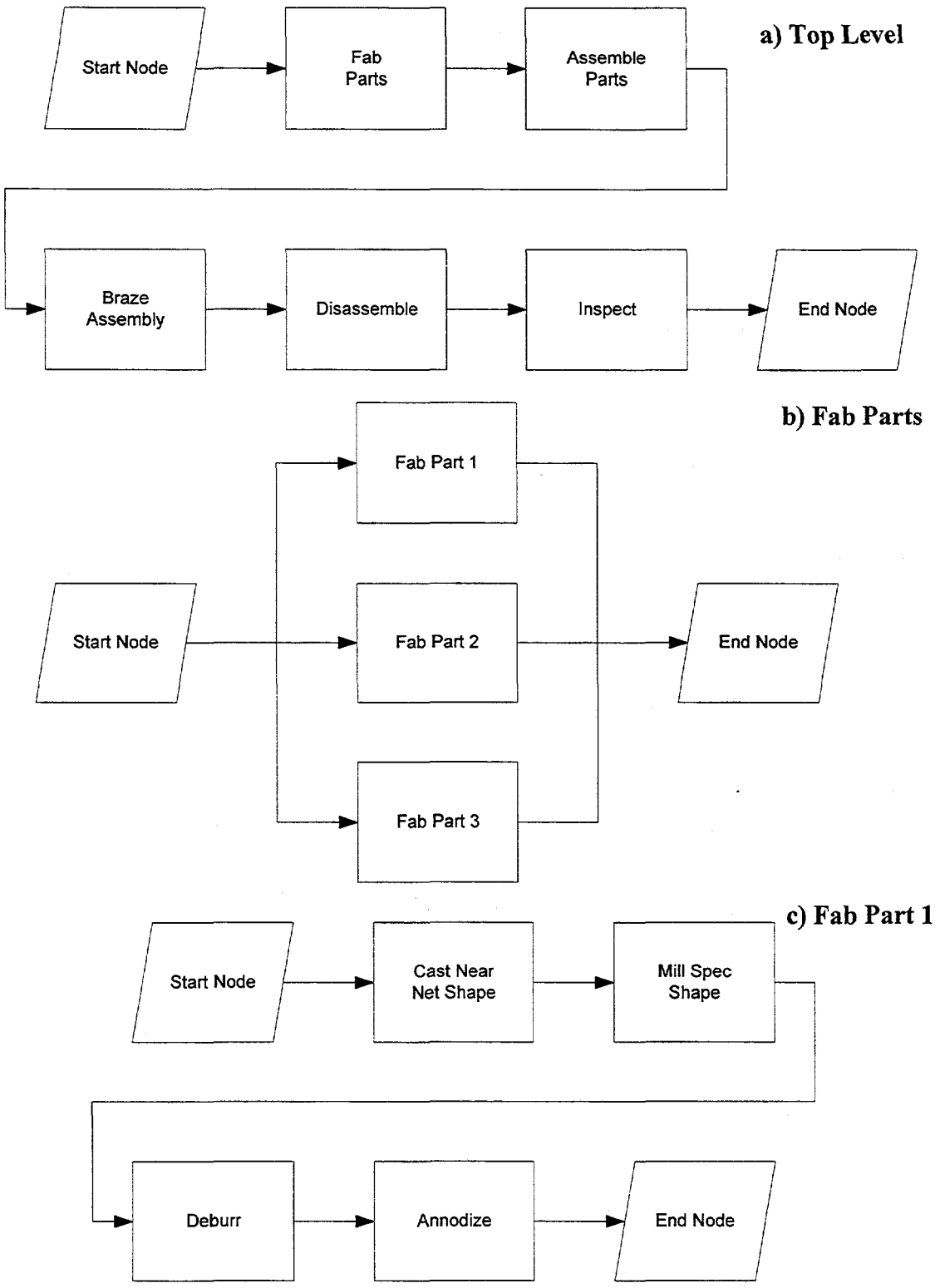


FIGURE 3-1 Hierarchical script decomposition: a) Top Level, b) Decomposition of Fab Parts, c) Decomposition of Fab Part 1

The application of the production script to the AMPS facility is described in Section 6.

4 Standard Manufacturing Modules

Key to the production script approach to information driven manufacturing is modular subsystems that have standardized interfaces. Basic features of subsystems that allow them to be easily integrated into automated systems are given in [7]. This section summarizes the work to develop the interface and the description grammar for a Standard Manufacturing Modules (SMM). The intent for the use of the SMM interface and description is illustrated in Figure 4-1. The manufacturing cell can rapidly be configured from available

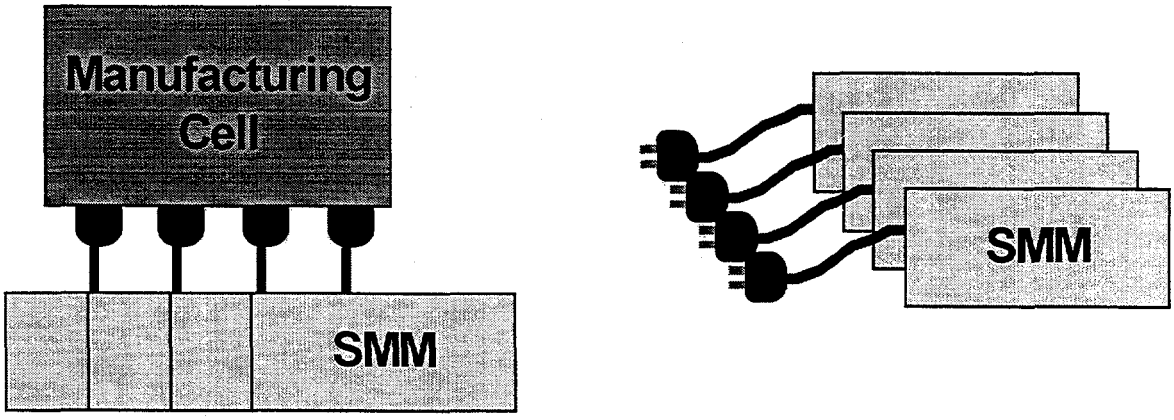


FIGURE 4-1 Use of plug and play SMMs

SMMs because of the plug and play interface. This section also describes the control architecture of the SMM that facilitates the development of manufacturing simulations.

4.1 Standard Interfaces

A general equipment interface was developed to allow remote control of production equipment. It utilizes much of the work by the semiconductor industry to develop their generic equipment model[8]. The complete definition is given in Appendix A. The basic features and motivation are given below.

The interface definition specifies remote control interface functionality that allows plug and play of manufacturing equipment. The intent is to define the interface in terms of standard equipment description information, and the standard message exchanges between the equipment and a supervisory control system during the interactions required to get the equipment to perform its tasks. The message content is specified independent of any particular communication link or specific equipment. The supervisory control system selects the equipment (or agent) to perform a task based upon capabilities and uses the standard interface interactions to get the task done on that equipment. The supervisor only needs to know that the equipment can perform the task within the desired specifications. Furthermore, the information provided in the equipment description together with the generic

interface is intended to allow simulation of equipment operation at the level detail required by a supervisory controller.

The interface definition covers the standard interactions required to set up and operate production equipment to perform the process steps of which it is capable. The interactions are described in terms of the function to be accomplished and the high level message protocol (sequence of messages) between the equipment and the supervisory control system for the cell of which the equipment is a part. The interaction protocol are intended to be independent of any particular communication link.

Process specific and equipment specific details are expressed in the characteristics description of the equipment or agent. Equipment descriptions and the data exchanged during the interactions are intended to provide a information driven interface to the equipment. When both the supervisor and the equipment adhere to the interface, a driver for the specific communication link (such as RS-232, TCP/IP, or GPIB) is all that will be required to do the software part of integrating the equipment into the whole system. The supervisory cell controller uses the equipment descriptions to determine whether the equipment can perform the desired process step, and to provide parameters that qualify and quantify the interactions required to get the step performed on the equipment.

The emphasis here is upon discrete manufacturing and material handoff between components in the system. The models and approach used here could most likely be generalized to accommodate continuous manufacturing equipment but such equipment has not been directly addressed.

4.1.1 Behavior State Models

State models are used to describe the expected behavior of both the equipment and the supervisory controller during operation of the equipment. The detailed internal states of the equipment are not addressed here if they do not effect the interactions between the equipment and the supervisor. State transitions occur due to events detected internally by the equipment or due to commands from the supervisor. Multiple state models deal with different aspects of the equipment behavior as seen by the supervisor and indicate which interactions are permitted at a particular time. In addition, for interactions with extended message exchange protocols, a separate state model of the interaction is defined. The state models ensure that both the supervisor and the equipment remain synchronized during interactions. Following the SEMI Generic Equipment Model[8], the Harel notation for state diagrams is used[9]. Those features of the notation used in Appendix A are briefly described below.

A state model consists of a state chart, a description of the states, and a transition table. Figure 4-2 illustrates the Harel state chart symbols. The Harel notation allows for hierarchical states and default initial substates when entering an encompassing parent state. Selectors can indicate which substate of a parent is to be entered. The history selector indicates that the system is to return to the substate that was active at the last transition out of the parent state. Transitions themselves are unidirectional but separate transitions can be used to toggle between states. Concurrent states are independent and do not directly cause

transitions in each other, but they do share common context. They can be thought of as weakly interacting subsystems. The use of concurrent states allows modularity of the model and greatly simplifies the individual state models.

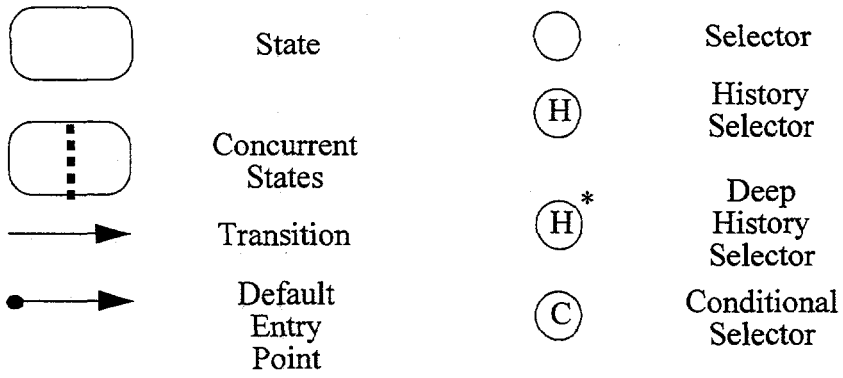


FIGURE 4-2 Harel State Chart

The main behavior of the equipment follows generic state models that address communications, local versus remote control, process states and alarm states. These state models provide the context for interactions that define the interface between the equipment and the supervisory controller.

The main processing state model in Appendix A for the generic equipment is shown in Figure 4-3. The complete set of behavioral state charts for the SMMs are:

- Processing State Chart
- Communications State Chart
- Remote Control State Chart
- Event Spooling State Chart
- Generic Alarm State Chart

The state charts are fully discussed in Appendix A.

4.1.2 Interaction State Models

Standardized state models are used to model the interactions of the supervisor with the process equipment. The main models cover communication states, remote versus local control, processing state and alarms. Interactions with the equipment assume that these main models are followed. Furthermore, each interaction follows an associated state model from initiation to termination. The interactions are defined by their function and the sequence of messages required to accomplish the function. The functions of the interactions fall into the following categories:

- Communications and control management
- Operations management

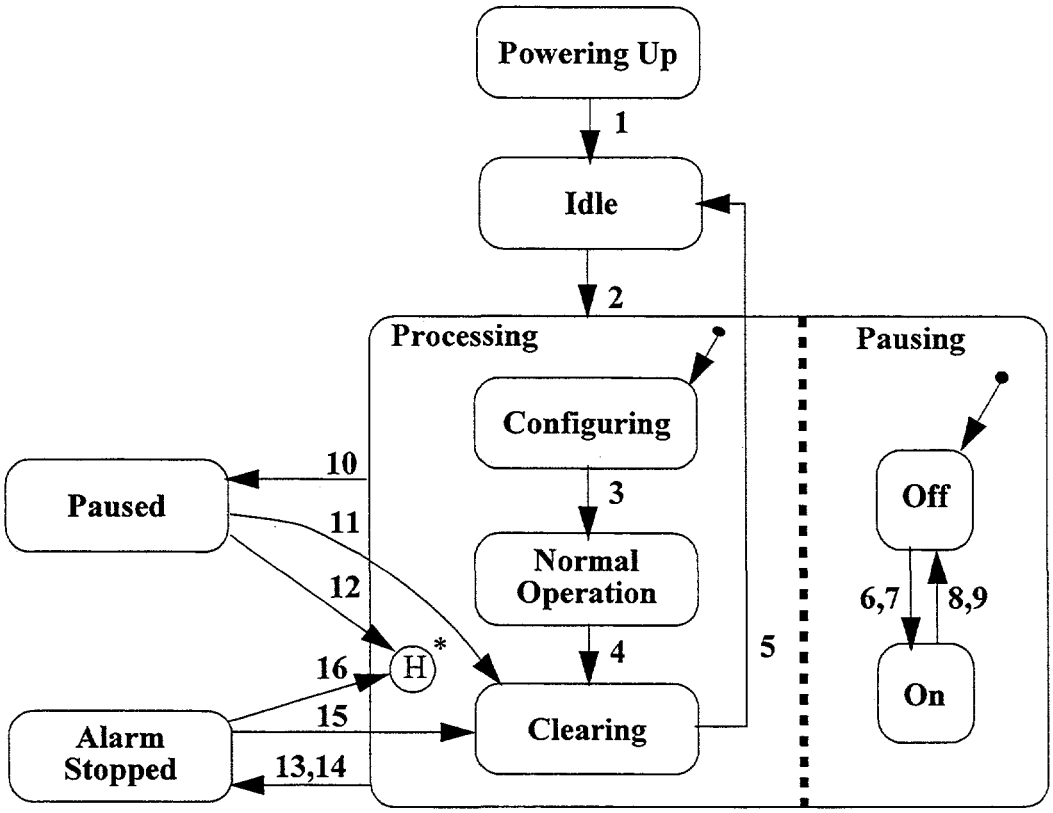


FIGURE 4-3 Processing State Chart

- Alarm management
- Material management
- Settings, queries and monitoring
- Logging management

Appendix A describes the interactions interface and the content of the characteristics information required to describe the equipment to a supervisory controller.

One of the more complex interaction state diagrams from Appendix A is shown in Figure 4-4. It illustrates the need for state models to provide for recovery from off normal conditions back to a state that is known by both partners in the interaction.

Characteristics description information is defined that allows the cell controller or supervisor to determine whether the equipment can perform the process steps. The description of the equipment also provides the information required during the interactions associated with getting the equipment to perform desired tasks. A grammar was developed to describe the equipment. It is also based upon lex and yacc[6].

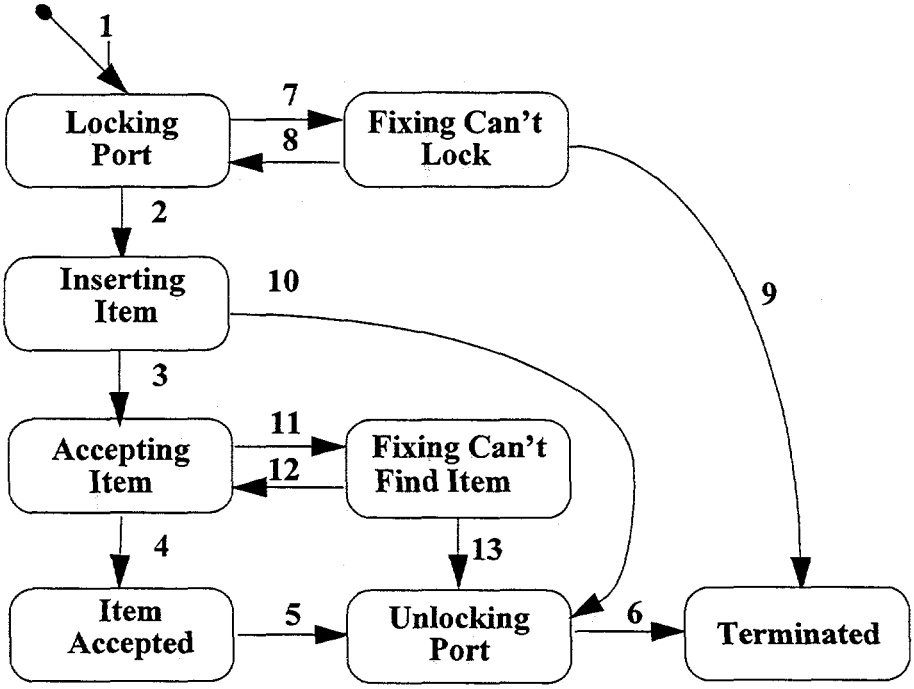


FIGURE 4-4 State Chart for the Item Receipt Interaction

4.2 SMM Control Architecture to Support Virtual Manufacturing

The control architecture for the SMMs is illustrated in Figure 4-5. Using the interface described above, the SMMs can be easily integrated into the supervisory control system for a cell. The display and record drivers are all that have to be adjusted to make the SMM run in a new manufacturing environment. The key to allowing cell simulation for virtual manufacturing is the use of both real and virtual drivers for the SMMs.The real mode of SMM execution uses the real driver while the simulation mode uses the virtual driver. The generic interface to the supervisor and the record and display drivers need not know whether the virtual or real drivers are being used. Thus, all interface and application code is executed in exactly the same manner for both the real and virtual modes. This control architecture is described in [11]. The use of identical software to run both real and virtual modes increases the confidence in the resulting simulations and allows complete validation of the code before hardware is run.

5 Task Sequence Control

A Task Sequence Controller (TSC) was developed to supervise the execution of the production scripts. It must decompose the scripts into production steps that can be performed by the SMMs under its control, dispatch the step to a selected SMM, route the required material to the SMM, ensure that results are recorded, and determine the next step to be executed.

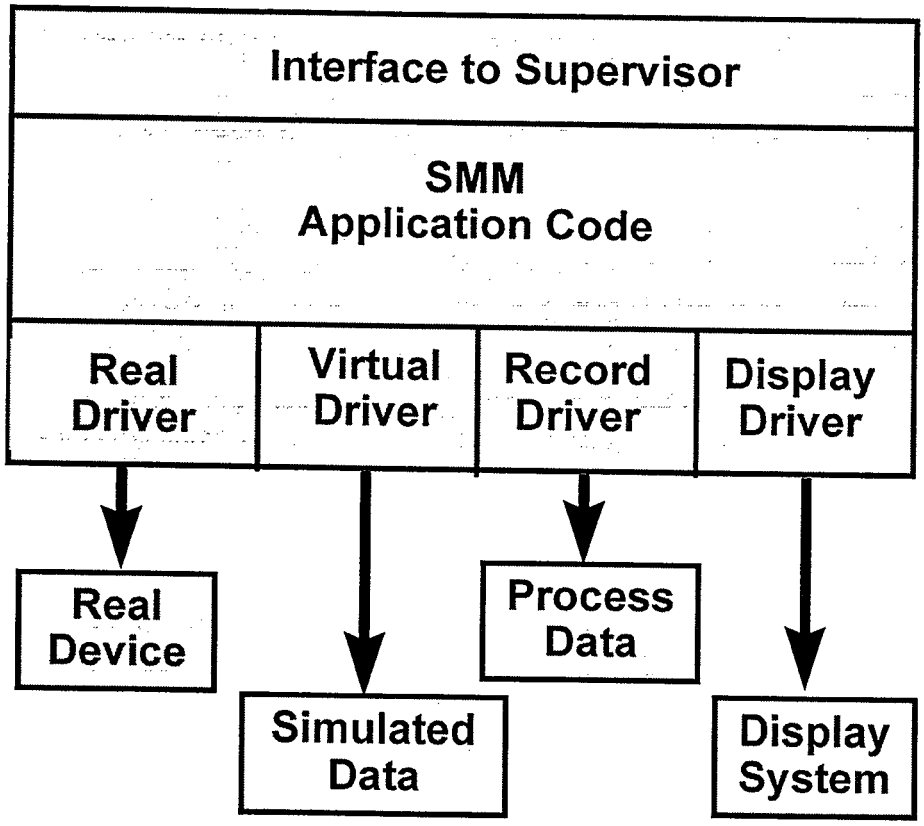


FIGURE 4-5 SMM architecture to allow virtual manufacturing

The main types of software modules that make up the TSC are:

- Process State Machines
- Process Managers
- Resource Manager
- User Interface
- Executive

The executive is a simple loop that gives each of the managers and the user interface a chance to do their work during each cycle. The processes that have to be managed are:

Job

A job is a work order that requires one or more production scripts to be executed.

Script

A production script is a sequence of manufacturing operation steps. A top level script is associated with a job. Other scripts are decompositions of the step in a higher level script.

Operation

An operation is an individual manufacturing step that is part of the decomposition of a script. The operation may be further decomposed into a lower level script.

Transport

A transport is a required movement of material from one SMM to another. Transports are required whenever a SMM needs material for an operation that is not already in its possession.

Interaction

An interaction is one of the interaction between the TSC and one of its SMM as described above.

Each process type follows a state machine through its life. At anytime, the TSC can support multiple instances of the each of the above processes. The manager for the process is responsible for ensuring that each instance of its managed type moves through its state machine in an orderly fashion. The managers negotiate among themselves to request services from each other and resolve resource contentions.

Each process instance is represented by a token that contains all of the information for the instance. To record desired information, the TSC can make a persistent copy of any data associated with any token.

A preliminary TSC was developed to test the basic features of the production script approach. This was applied to the AMPS cell as described in Section 6. Lessons learned in that application where used to refine the requirements and specifications for the TSC. At the end of this project, the refined TSC, as described above, was just coming on-line and going through initial testing.

6 Application to Agile Manufacturing Prototyping System

The Agile Manufacturing Prototyping System (AMPS), as described in [1], is a set of four modular robotic workcells that can be used to rapidly prototype automation concepts for manufacturing processes with an initial emphasis on assembly. AMPS was used as testbed for the concepts developed in this LDRD project. The application was to assemble a small weapon component that consisted of a stack of small parts into a fixture for brazing. Only one of the AMPS cells was required for this application. The layout of the call is shown in Figure 6-1.

6.1 AMPS Assembly SMM Software

The AMPS Assembly Cell was controlled using the software architecture described above and shown in Figure 4-5. The application code needed to track the items in the cell and be able to perform the assembly operations. The drivers and interface software also had to be provided.

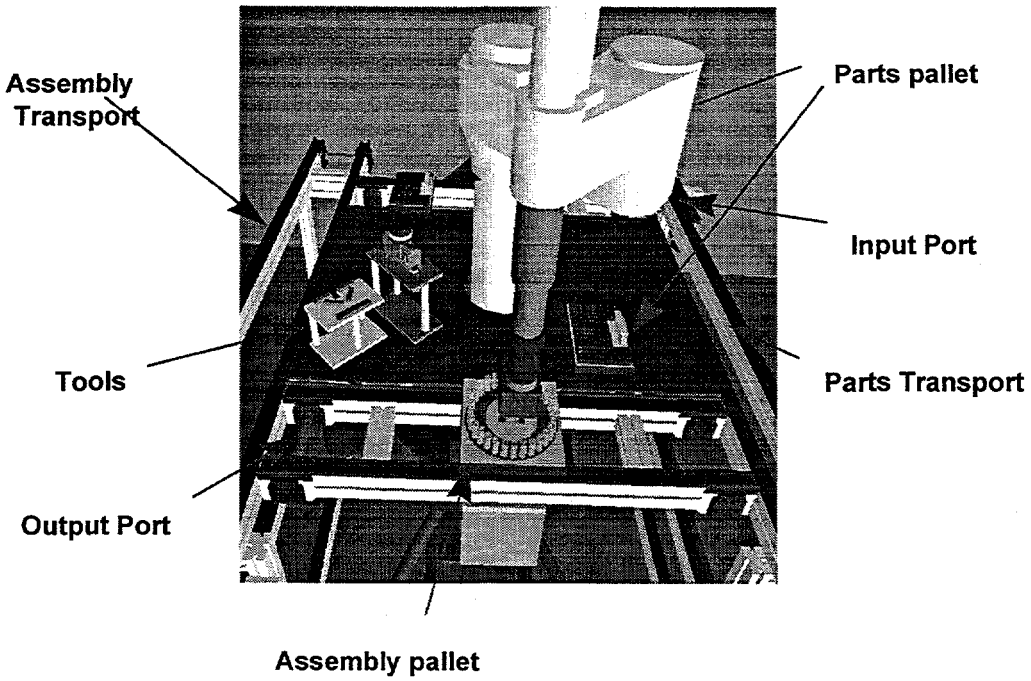


FIGURE 6-1 AMPS Assembly Cell

6.1.1 Assembly SMM Items and Workpoints

The items in the cells consisted of the tools, the storage stations for the tools, the pallets that held the parts, and the parts themselves.

Workpoints are points expressed with respect to the origin of the item. They are the points at which other items in the cell interact with the item. For example, the point where a tool is applied is a workpoint, as is the point where another item touches it. Items can be held by one other item but may themselves hold multiple other items. A configuration file is used to initialize the workpoint and attachment information for the cell. Tracking of item location is accomplished by updating attachment information after each operation.

6.1.2 Assembly SMM Operations

Each operation that an SMM can perform is parameterized so that it can be invoked by the TSC with the proper context supplied at runtime for the script information. The parametrization is done with respect to reference workpoints such as a tool grab point for the item allowing some of the parameters to be defaulted as standard offsets from the reference point. The parameterized algorithm for the pick operation using the anvil gripper in the AMPS Assembly Cell is shown in Figure 6-2.

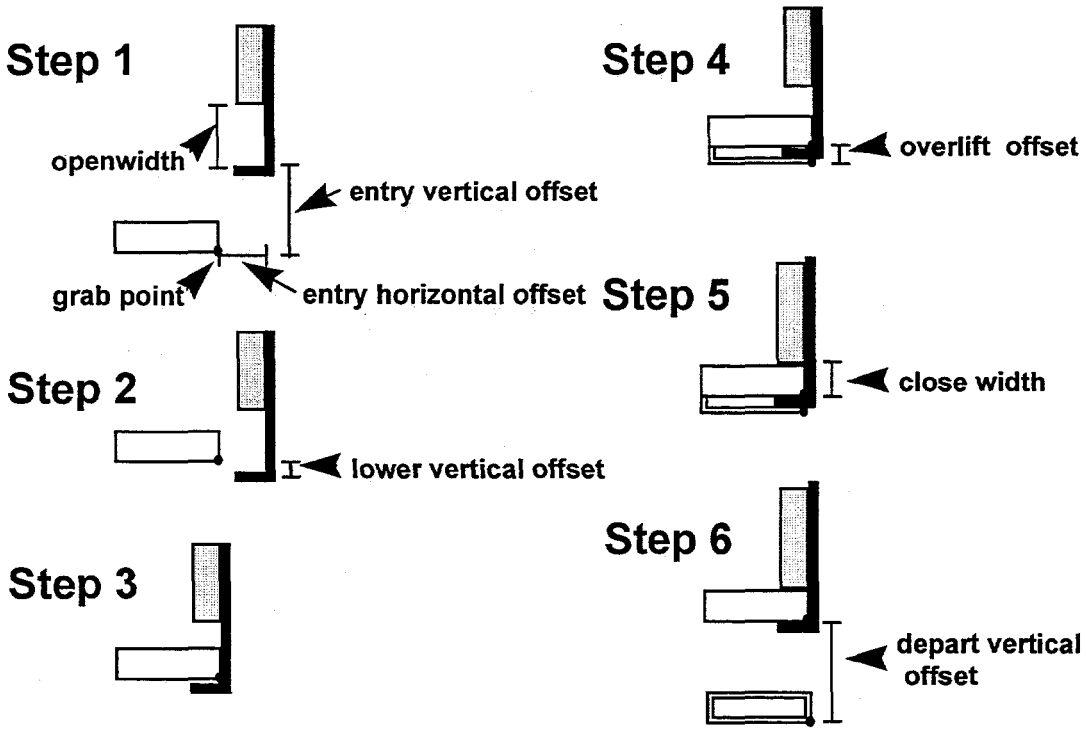


FIGURE 6-2 Parameterized algorithm for pick with anvil gripper

6.2 AMPS Application Results

The information for the configuration and scripts were generated by hand from the engineering drawing of the parts to be assembled, the input pallets, and the assembly pallets. Vision was used to locate the input palettes given their nominal location in the input ports of the cell. However, the robot was not taught the workpoints with the teach pendant, since all workpoints were based upon configuration information updated through the vision system and the item tracking system.

The system was used to demonstrate the concurrent execution of multiple scripts, serial execution of scripts, dispatch of operation steps according to SMM capability and availability, item tracking, and execution optimization through movement of the input pallets closer to the assembly pallet without changing of the script. Although the there was only one assembly SMM, simulation was used to investigate the allocation of the assembly execution across multiple cells. Thus, the feasibility of the production script approach has been verified.

6.2.1 Workpoint and Tool Algorithm Specification

The next part of the AMPS application was to use computer based geometry information to generate the configuration files used by the assembly SMM. The main sources of information are:

- item geometry and assembly information from CAD files
- presentation fixture information for items that are input to the cell for an assembly
- assembly fixturing information
- tool selection and workpoint information
- assembly sequence information
- approach direction information

A package was generated that took the above information and generated the configuration and script files for the AMPS assembly system. Generation of the inputs to the script and configuration file generator was more complicated than expected. For example, Archimedes[12] generates assembly sequence and approach information with respect to the origin of the assembly, but the input pallet and attached item workpoint information used different coordinate systems that those used by Archimedes. The symmetry of the parts made it impossible without experimentation, to determine a consistent set of work-point information. Similarly, tool grab point information was difficult to develop because of the multiple coordinate systems used by the various CAD tools used to generate the initial information.

Script execution at the TSC level was inefficient for the test assembly that was investigated. Since all of the assembly steps were done in the same SMM, there was no need to decompose the script at the TSC level. Rather the whole assembly script could be passed to the SMM which could perform the script autonomously. The SMM application code in Figure 4-5 was extended to process complete scripts that could be performed completely by one SMM.

7 Follow on Work

In order for the production script approach to be more useful, SMMs for various manufacturing processes have to be available. Developing the specifications for the SMMs requires process specific knowledge and information about the intended product application. Once the required process SMMs are identified, they have to be laid out and their operations orchestrated to perform the manufacturing steps.

The System Composer Project is a follow on effort to develop system integration tools based upon the key ideas and results of this LDRD research. The System Composer provides for the incremental development of the requirements and specification of manufacturing systems. System Composer will be used to manage information about:

- parts and assemblies that are to be processed by the system
- manufacturing operations that the system must be able to perform
- production scripts that the system must execute
- batching information
- system components and SMMs

- system layout

System composer will assist in the management of the requirements and specifications development and will marshal the use of advisors to help in the development of the next level of information given current information. For example, a tool and workpoint editor has been developed to facilitate the development of workpoint and tool algorithm information that was so problematic in the AMPS application as discussed in Section 6.2.1.

System Composer uses a less formal but graphical script than described above for the preliminary development of sequence information. As details are refined, the sequence information eventually contains all of the information specified in Section 3.

System Composer will provide graphic layout of the plug and play SMMs. A conceptual illustrated of the layout tool is shown in Figure 7-1.

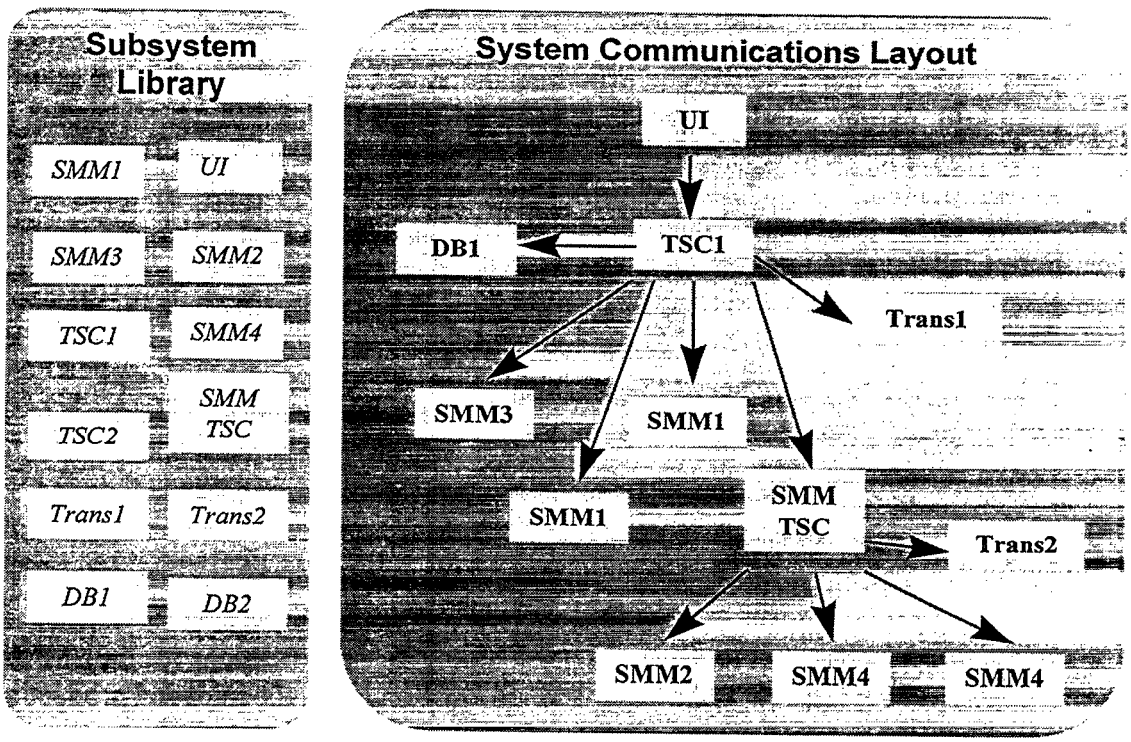


FIGURE 7-1 Graphical communications layout with new System Composer

8 References

1 Garcia, P., Woods, R.O., Rebeil, J.P., Jones,.J.F., Griesmeyer, J.M., Agile Manufacturing Prototyping System (AMPS).

2 Griesmeyer, J.M., McDonald, M.J., Harrigan, R.W., Butler, P.L., Rigdon, B, *Generic Intelligent System Control (GISC)*,SAND92-2159,September, 1992.

3 Griesmeyer, J.M., Urenda, T.D., Pacetti, R.M, Ferguson, J.J., "A Standard Control System for Modular Automation of Chemistry", *Laboratory Robotics and*

Automation, 6 (1994) 79-84.

4 FASTCAST

5 SMARTWELD

6 Mason, T., Brown, D., *lex & yacc*, O'Reilly & Associates, Inc., Sesastopol, CA, 1990.

7 Salit, M.L., Griesmeyer, J.M., "System Ready Behaviors for Integration", *Laboratory Robotics and Automation*, 9 (1997) to appear.

8 Semiconductor and Equipment International, SEMI *International Standards 1994: Equipment Automation/Software Vol. 2*, Mountain View, CA, 1994.

9 Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8 (1987) 231-274.

10 Salit, M., Guenther, F.R., Kramer, G.W., Griesmeyer, J.M., "Integrating Automated Systems With Modular Architecture", *Analytical Chemistry*, 66 (1994) 361-367.

11 Griesmeyer, J.M., Oppel, F.J., "Process Subsystem Architecture for Virtual Manufacturing", *1996 IEEE International Conference on Robotics and Automation*, (1996) 2371-2376.

12 Kaufman, S.G.,Wilson, R.H., Jones, R.E., Caltom, T.L., Ames, A.L., "ARCHIMEDES 2 Mechanical Assembly Planning System",*1996 IEEE International Conference on Robotics and Automation*, (1996) 3361-3368.

This page intentionally left blank.

General Equipment Interface Definition

J. Michael Griesmeyer
Sandia National Laboratories[1]

1. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Matrin Company, for the United States Department of Energy under Contract DC-ACO4-94AL85000.

Contents

List of Tables

List of Figures

General Equipment Interface Definition

J. Michael Griesmeyer[1]

1 Introduction

1.1 Purpose

The document specifies software remote control interface functionality that allows plug and play of manufacturing equipment. The intent is to define the interface in terms of standard equipment description information and the standard message exchanges between the equipment and a supervisory control system during the interactions required to get the equipment to perform its tasks. The message content is specified independently of any particular communication link or specific equipment. This allows the dispatching of tasks to the particular equipment based upon capabilities. The goal is to facilitate modular agent based control of manufacturing processes, and thereby significantly reduce system integration costs and the time required to introduce new products. The supervisory control system selects equipment (or agent) to perform a task based upon capabilities and uses the standard interface interactions to get the task done on that equipment. The supervisor need not know how the equipment performs the task. It only needs to know that the equipment can perform the task within the desired specifications. Furthermore, the information provided in the equipment description together with the generic interface is intended to allow simulation of equipment operation at the level detail required by a supervisory controller.

1.2 Scope

This document covers the standard interactions required to set up and operate production equipment to perform the process steps of which it is capable. The interactions are described in terms of the function to be accomplished and the high level message protocol (sequence of messages) between the equipment and the supervisory control system for the cell of which the equipment is a part. While examples may be in terms of particular communication links, the interaction protocol are intended to be independent of any particular communication link. Similarly, examples of particular process equipment may be used to illustrate some points of the interface.

The description of the plug and play interface for the process equipment is intended to be independent of both specific equipment to perform a process and even the process itself. Process specific and equipment specific details are expressed in the characteristics description of the equipment or agent. Equipment descriptions and the data exchanged

1.Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Matrin Company, for the United States Department of Energy under Contract DC-ACO4-94AL85000.

during the interactions are intended to provide a completely information driven interface to the equipment. When both the supervisor and the equipment adhere to the interface, a driver for the specific communication link (such as RS-232, TCP/IP, or GPIB) is all that will be required to do the software part of integrating the equipment into the whole system. The supervisory cell controller uses the equipment descriptions to determine whether the equipment can perform the desired process step, and to provide parameters that qualify and quantify the interactions required to get the step performed on the equipment.

The emphasis here is upon discrete manufacturing and material handoff between components in the system. The models and approach used here could most likely can be generalized to accommodate continuous manufacturing equipment but such equipment has not been directly addressed.

1.3 Overview

Standardized state models are used to model the interactions of the supervisor with the process equipment. The main models cover communication states, remote versus local control, processing state and alarms. Interactions with the equipment assume that these main models are followed. Furthermore, each interaction follows an associated state model from initiation to termination. The interactions are defined by their function and the sequence of messages required to accomplish the function. The functions of the interactions fall into the following categories:

- Communications and control management
- Operations management
- Alarm management
- Material management
- Settings, queries and monitoring
- Logging management

Characteristics description information is defined that allows the cell controller or supervisor to determine whether the equipment can perform the process steps. The description of the equipment also provides the information required during the interactions associated with getting the equipment to perform desired tasks.

This document describes the interactions interface and the content of the characteristics information required to describe the equipment to a supervisory controller.

2 Definitions

Agent Based Control

An approach to control in which agents are responsible for performing contracted tasks without detailed supervision. While agents may have varying degrees of autonomy in carrying out their tasks, the supervisor need not know the details of how the tasks are executed as long as the tasks are performed according to contract.

Supervisory Control

The control regime that addresses orchestration of tasks by assignment to agents and arbitration of material movement to and from the agents selected for particular tasks.

Equipment Control

The control regime that addresses the internal control of the equipment to carry out contracted tasks.

Interface Interaction

One of the interactions that define the interface between two systems.

Equipment State Model

A model of the equipment in which the behavior of the equipment depends upon a set of state variables. Transitions between states are events. The Equipment State Model provides the context for the interactions between the supervisor and the equipment.

State Variable

A variable that can take only discrete values which indicates some characteristic behavior regime of the equipment.

Data Variable

A variable that indicates some aspect of the equipment environment, or progress of operations.

Setting Variable

A variable that controls the manner in which the equipment operates or is controlled.

Characteristics Description

A description of the equipment that can be used by the supervisor to determine the equipment capabilities and the data required to execute the interface interactions.

Equipment Capabilities

The set of process operations that can be performed by the equipment together with a specification of the achievable quality and performance characteristics.

Event

An occurrence detected by a system component that requires a response by the system, or that is of interest to other system components

Alarm

An event that requires priority response to protect personnel, equipment or product.

Command Transaction

A synchronized exchange of messages between the supervisor and the equipment that is initiated by a command message from the supervisor. During the transaction data can be exchanged between the supervisor and equipment.

Event Report

A single message from the equipment to the supervisor reporting the occurrence and details of an event that has been detected by the equipment.

Interface Interaction Protocol

The sequence of message exchanges that are required between two systems to execute an interaction. The message exchanges in the protocol are either command transactions or event reports.

Interface Interaction State Model

A state model that tracks interface interactions through the message exchanges cause transitions between states of the interaction.

Process Description Language

A process specific language used to describe the manufacturing steps to be performed. Each class of process may have a unique process description language that can be interpreted by any equipment that performs production steps for that class of process.

SEMI

Semiconductor Equipment and Materials International.

Generic Equipment Model

The SEMI generic model of processing equipment for effective factory automation.[4]

Standard Laboratory Module

A device that performs chemistry analysis steps in the approach to the modular automation of chemical analyses being developed under the U.S. Department of Energy Robotics Technology Development Program.

3 General Description

The main behavior of the equipment follows generic state models that address communications, local versus remote control, process states and alarm states. These state models provide the context for interactions that define the interface between the equipment and the supervisory controller.

Standard interactions based upon the generic state models are used to define the interface to generic equipment. The function of each interaction is defined together with a message exchange protocol that keeps both sides of the interaction synchronized. The interactions can be initiated by commands from the supervisor to the equipment or by event reports from the equipment to the supervisor. The rest of the message exchange protocol for the interaction serves to pass information and keep both sides aware of the state of the other.

3.1 Some Communication Assumptions

The interface described here is independent of the communication link. However, some basic assumptions are made regarding message passing over the communication link. Every message must be acknowledged by the recipient before a second message can be sent to it. This prevents queueing of messages and the associated synchronization problems. Both the supervisor and the equipment must be able to detect when the communication link is lost and take appropriate action to restore communications. The communication link cannot be blocked while the equipment performs a task. For the purposes of the document message exchanges are modeled as instantaneous. If the equipment or the supervisor would normally require extended time to respond to a message, the response is considered as a separate message exchange.

Communications initiated by the supervisor are synchronized command transactions in which data messages can flow in either direction depending on the transaction type. Since the communications initiated by event reports from the equipment are asynchronous to the main thread of the supervisor, message exchanges initiated by the equipment consist of a single message. The supervisor response must be a separately initiated transaction. Furthermore, each event report from the equipment to the supervisor must contain a timestamp. However, timestamp associated with event reports will not be indicated in the description of the message exchanges described herein.

3.2 State Models

State models are used to describe the expected behavior of both the equipment and the supervisory controller during operation of the equipment. The detailed internal states of the equipment are not addressed here if they do not effect the interactions between the equipment and the supervisor. State transitions occur due to events detected internally by the equipment or due to commands from the supervisor. Multiple state models deal with different aspects of the equipment behavior as seen by the supervisor and indicate which interactions are permitted at a particular time. In addition, for interactions with extended message exchange protocols, a separate state model of the interaction is defined. The state models ensure that both the supervisor and the equipment remain synchronized during interactions. Following the SEMI Generic Equipment Model[4], the Harel notation for state diagrams is used[5]. Those features of the notation used in this document are briefly described below.

A state model consists of a state chart, a description of the states, and a transition table. Figure 3-1 illustrates the Harel state chart symbols. The Harel notation allows for hierarchical states and default initial substates when entering an encompassing parent state. Selectors can indicate which substate of a parent is to be entered. The history selector indicates that the system is to return to the substate that was active at the last transition out of the parent state. Transitions themselves are unidirectional but separate transitions can be used to toggle between states. Concurrent states are independent and do not directly cause transitions in each other, but they do share common context. They can be thought of as weakly interacting subsystems. The use of concurrent states allows modularity of the model and greatly simplifies the individual state models.

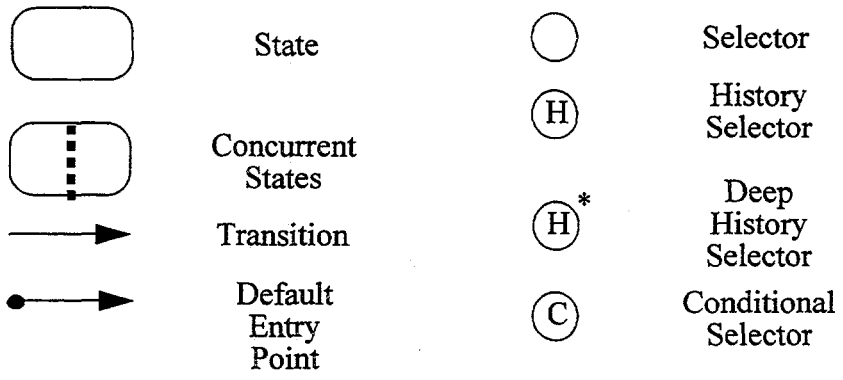


FIGURE 3-1 Harel State Chart

3.3 Interaction State Models

Each interaction that makes up the interface between the equipment and the supervisor follows a state model. The transitions in the interaction state models result from exchanges of messages. Many of the interactions involve a single exchange of messages and, thus, the initiating message exchange is also the terminating exchange. The state chart is shown in Figure 3-2 for interactions where the initial message exchange terminates the interaction. In the following sections that describe the interface interactions, separate interaction state models will not be given for interactions involving a single message exchange. However, the message content of all defined exchanges for an interaction will always be given.

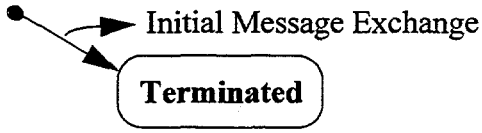


FIGURE 3-2 State Chart for Interactions with Single Message Exchange

In the tables describing interaction message exchanges, the => symbol indicates that the supervisor initiates the message exchange with the named command, while <= indicates that the equipment initiates the message exchange with the named event report. For both command transactions and event reports, arguments are indicated by (*arg1, arg2,...*) following the name of the command or event. In command transactions, download of data from the supervisor to the equipment is indicated by ->, and upload of data from the equipment to the supervisor is indicated by <-.

Any command transaction that cannot be fully executed by the equipment must be terminated gracefully by the equipment with a message to the supervisor containing an error code and a text explanation of the problem. When this occurs, the message exchange of the command transaction is terminated and the interaction of which it was a part is left in the state that it was before the aborted command transaction was initiated. This type of message exchange problem therefore will leave the equipment and the associated interac-

tion in a known state. Abnormal termination of interactions themselves is addressed in the state models of the interactions.

3.4 Communications Maintenance and Control Interactions

A simple state model is used to describe the communication state of the equipment. The details of communication initiation will be dependent on communication link used. However, the function of the interactions will be described independent of communication link. When communication is lost, the equipment must ensure that important event messages are spooled for later upload once communication is restored. The interactions and requirements associated with spooling are specified in this document. Finally, integration of the equipment into an automated facility requires that the equipment be remotely controlled by a supervisory control system. Interactions for negotiated handoffs between local and remote control are specified.

3.5 Processing Interactions

The mission of the equipment is to process product at the bidding of the supervisor. In order to perform the desired operations on product, the equipment must be configured or setup properly. While some equipment may require a setup step before processing each batch of product, other equipment may be able to perform operations on several batches without additional setup or reconfiguration. A general processing state model that accommodates both types of equipment is used to define the interactions required to get the equipment configured properly and to perform operations on product. The model also addresses interactions required for the handling of process interruptions caused by commands from the supervisor or alarms.

3.6 Alarm Management Interactions

The equipment must be able to detect various anomalous conditions and inform the supervisor so that they can be addressed. Interactions based upon a state model are defined to manage alarms. The actual handling of particular alarm conditions will be specific to the equipment and the alarm. Only the communication of alarm status is addressed here.

3.7 Material Movement Interactions

Material movement interactions are required to manage the following:

- product movement into and out of the processing equipment
- supply replenishment and waste removal
- access to common material resources

The interactions and their message exchange protocols are described using state models of the interactions where appropriate.

3.8 Settings, Queries, Symbol Access and Logging

The equipment must allow the supervisor to have access to all information that effects the interactions between them. Query capabilities are defined for standard equipment state variables. Symbol table capabilities are defined to provide a generic way for the supervisor to have access to equipment specific data and variables. Symbol monitoring features allow the supervisor set the thresholds at which it is informed when a symbol value changes. Generic logging features for record keeping and trouble shooting are discussed.

3.9 Characteristics Description

The Equipment Characteristics Description provides all of the equipment specific information required for the supervisor to interact with the equipment. This includes tables of operations, configurations, material input and output ports, support services, initial state variable values, and equipment specific symbols. Only the information content is discussed here. The actual media by which the supervisor has access to the Equipment Characteristics Description can be through a data file or some upload from the equipment. The generic requirement is that the information be made available to the supervisory control system. The characteristic description consists of the following tables:

Operation Table. A table of the process operations that can be performed by the equipment. For each operation, all information required to describe process step capability quality, duration, material flows and configuration needs is provided.

Configuration Table. A table of the allowable configurations of the equipment. For each configuration, parameters, and equipment settings are specified.

Supply Table. A table of the supply types that are required by the equipment to perform its operations. For each supply type, the storage capacity, units and material port through which the supply inters and exits the equipment are listed.

Waste Table. A table of the waste types that are generated by the equipment. For each waste type, the waste capacity, units and material port through which the waste exits the equipment are listed.

Tool Table. A table of the tools that are used by the equipment to perform its operations. For each tool, the tool class and entry/exit ports for the tool are listed

Material Port Table. A table of the material ports through which product, supplies and waste enter and leave the equipment. For each material port, the classes of material that enter and exit through the port are listed along with the physical characteristics of the port such as dimensions and location.

Shared Reservoir Table. A table of the material reservoirs (usually for fluids) that are required by the equipment as an intermittent source or sink of material. For each reservoir, the class of material and method of transport are listed.

Continuous Fluid Requirements Table. A table of the continuously delivered fluids that are required to service the equipment. For each fluid, the class of material and method of transport are listed along with normal flow requirements.

Alarm Table. A table of the alarms that can occur on the equipment. For each alarm, a category and severity must be specified together with a description of the condition.

Standard Variable Initialization. A table of the initialization values for standard equipment state and setting variables.

Equipment Specific Data Variable Table. A table of the equipment specific process variables. For each variable, a description, units and range are given, together with monitoring specification and parameters.

Equipment Specific Setting Variable Table. A table of the equipment specific setting variables. For each variable, a description, initial values and valid range are specified.

Equipment Specific Event Table. A table of the equipment specific events that can arise during the course of its operation. For each event, a description, category, and argument list are given.

4 Communication Maintenance and Locus of Control Interactions

4.1 Overview

The interactions for the establishment and maintenance of communications and supervisory control are described here. Although the exact set of messages required for establishment of communications will be dependent on the communications link employed, the required function of the interaction will be described. In order for the supervisor to use the equipment, the handoff from local to remote control must be accomplished. The interactions for that handoff are described. Upon loss of communications the equipment must spool event messages until communication has been restored. At that time, the supervisor can have them uploaded or purged. The interactions for spooling management are described.

4.2 Establishing and Maintaining Communications

It is the responsibility of the equipment to allow communications with the remote supervisor at all times after power up initializations are complete. The details of preparing for and initializing communications are dependent upon the physical communication link and low level message passing protocol. Here only an outline of the required behavior is given. Figure 4-1 shows the Communications State Chart. Table 4-1 describes the transitions in the Communications State Model. Communication Loss Pause is a flag state indicating whether the equipment is to pause upon loss of communication with the supervisor.

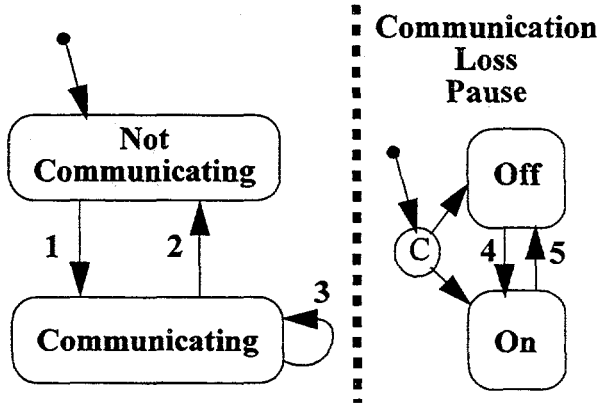


FIGURE 4-1 Communications State Chart

4.2.1 Description of Communication States

Not Communicating

The equipment is not communicating with the supervisory controller either because communications have not yet been established or they have been lost. Not Communicating is the entry state upon powering up.

Communicating

Communications are established and proceeding normally.

Communication Loss Pause On

The equipment will pause upon loss of communication.

Communication Loss Pause Off

The equipment will not pause upon loss of communication.

The initial value of the Communication Loss Pause flag must be specified in the equipment description.

TABLE 4-1. Communications State Transitions

#	Old State	Transition Event	New State	Comments
1	Not Communicating	Successful establishment of communication	Communicating	At start-up the equipment performs the steps required to establish communication
2	Communicating	Termination of communication by fault	Not Communicating	After initial connection, communication is not normally terminated except at shutdown. The equipment should perform the steps required to re-establish communications

TABLE 4-1. Communications State Transitions

#	Old State	Transition Event	New State	Comments
3	Communicating	Any message	Communicating	
4	Communication Loss Pause Off	Command to turn Communication Loss Pause On Received	Communication Loss Pause On	
5	Communication Loss Pause On	Command to turn Communication Loss Pause Off Received	Communication Loss Pause Off	

4.2.2 Requirements

- The equipment must allow communication with the supervisor at all times after power up initialization.
- Upon loss of communication with the supervisor, the equipment must perform the required steps to allow communication to be re-established.
- Even when under local control, the equipment must keep the supervisor informed of its state through the reporting of relevant events and response to queries.
- The equipment must have clearly defined behavior at loss of communication with the supervisor, and that behavior must be addressed in the equipment description.

4.2.3 Establishing Communication

The interaction for establishing communication is communication link specific and not specified here.

4.2.4 Setting Communication Loss Pause On

The interaction to turn communication loss pause on consists of a single command: CommLossPauseOn as shown in Table 4-2.

TABLE 4-2. Interaction for Setting Communication Loss Pause On

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>CommLossPauseOn	None	Terminated	

4.2.5 Setting Communication Loss Pause Off

The interaction to turn communication loss pause off consists of a single command: CommLossPauseOff as shown in Table 4-3.

TABLE 4-3. Interaction for Setting Communication Loss Pause Off

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>CommLossPauseOff	None	Terminated	

4.3 Local and Remote Control

The equipment must be in remote control for the supervisory control system to use it in an automated fashion. The control state at power up can be either local or remote, but the supervisor must be informed of the initial state. The value of this parameter must be provided in the equipment description data as described in Section 9.11, "Standard State Variable Initialization Table," on page 53. For safety considerations, the local operator and the supervisory control system must negotiate for control. Neither can take control from the other. However, both may at anytime cause an alarm stop. Figure 4-2 shows the Remote Control State Chart. The Remote Control State transitions are described in Table 4-4.

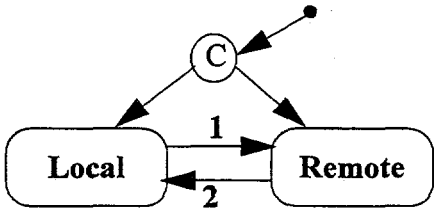


FIGURE 4-2 Remote Control State Chart

4.3.1 Description of Control States

Local

The local console operator has control of equipment operation. The supervisory control system can always query for the value of particular state variables and receives event reports that keep it aware of the equipment state. However, under local control the supervisory control system can not set equipment parameters or issue commands that change the state of the equipment.

Remote

The supervisory control system has access to all standard equipment control functions.

The initial value of the Remote Control State must be specified in the equipment description.

4.3.2 Requirements

- The equipment must allow all control functions for processing of product to be performed through remote control.
- The handoffs between local and remote control must be based on give rather than take.

TABLE 4-4. Remote Control State Transitions

#	Old State	Transition Event	New State	Comments
1	Local	Request for remote control received from supervisor and accepted	Remote	Handoff between local and remote control is negotiated. Neither side may *take* control from the other
2	Remote	Command to return to local control received	Local	The local operator can request local control and the supervisor can grant control through the command to return to local control. The supervisor can give control to the local operator event if it is not requested.

4.3.3 Transferring to Remote Control

Transferring to remote control consists of a single message exchange in which the supervisor requests remote control and the local operator grants or rejects the request. The interaction is shown in Table 4-5

TABLE 4-5. Interaction for Transferring to Remote Control

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>RequestRemoteControl <-RemoteControlGranted or <-RemoteControlDenied	None	Terminated	

4.3.4 Transferring to Local Control

Transfer from local to remote control can be initiated by either the supervisor or the local operator. The message exchange when initiated by the supervisor consists of a single message as show in Table 4-6.

TABLE 4-6. Interaction for Supervisor Initiated Transfer to Local Control

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>LocalControlGranted	None	Terminated	

When initiated by request from the local operator, the supervisor can reject or accept the request. The corresponding message exchange is shown in Table 4-7

TABLE 4-7. Interaction for Local Operator Initiated Transfer to Local Control

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=RequestLocalControl	None	LocalControlRequested	
2	=>LocalControlGranted or =>LocalControlDenied	LocalControlRequested	Terminated	

4.4 Event Spool Management

The equipment should ensure that essential information can be communicated to the supervisory control system even when communication link is lost for a period of time. The equipment must therefore spool event messages upon loss of communication so that they can be transmitted to the supervisor once communication has been restored. The size of the required spool area will be equipment dependent. Normally, the spool buffer should be large enough to allow the equipment to spool all event messages that would occur during a normal operation cycle, or at least until the next point in the operation from which the operation can be safely paused for restart at resumption of communications with the supervisor.

The event message spooling state model addresses these issues. It also covers the overflow of the buffer with communications still not restored. In that case, the first messages can be overwritten or the equipment can be paused before buffer overflow to preserve all messages relevant to normal operation. The Event Spooling State Chart is shown in Figure 4-3 and its transitions are described in Table 4-8.

4.4.1 Description of Spooling States

Spool Inactive

The Communication state is Communicating and the spool buffer is empty. This is the initial spooling state.

Spool Active

The Communication state is Not Communicating or the spool buffer is not empty.

Spool Not Full

The spool is not full. This is the initial spool loading substate.

Spool Full

The spool is full and additional event messages overwrite the oldest event massages.

Spool Not Unloading

The spool is not unloading. This is the initial spool unloading substate because communication has been lost.

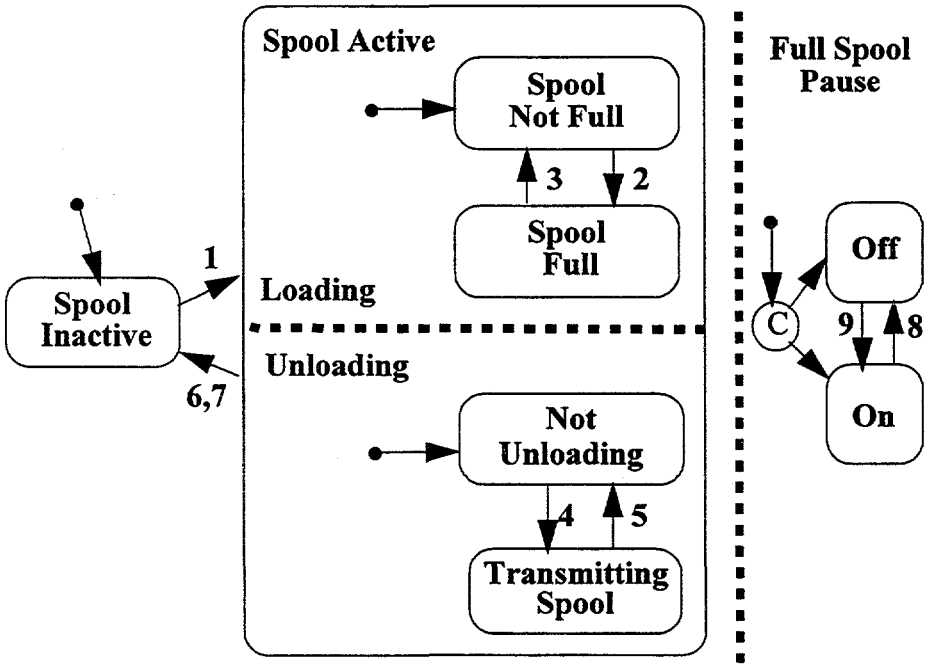


FIGURE 4-3 Event Spooling State Chart

Transmitting Spool. The spool is being transmitted to the supervisor because the command to transmit the spool has been received by the equipment.

Full Spool Pause Off

The equipment will not pause before the event spool is full.

Full Spool Pause On

The equipment will pause before the event spool is full.

TABLE 4-8. Event Spooling State Transitions

#	Old State	Transition Event	New State	Comments
1	Spool Inactive	Loss of communication	Spool Active	Entry at Spool Not Full and Not Unloading
2	Spool Not Full	Event spool buffer full	Spool Full	If Full Spool Pause is On, the equipment should be paused by the time the spool buffer is full. Otherwise, additional events are written over the oldest events in the spool
3	Spool Full	Some but not all of the event reports are uploaded to the supervisor	Spool Not Full	This occurs only after communications have been restored
4	Not Unloading	Command to transmit spool received	Transmitting Spool	Communication have been restored

TABLE 4-8. Event Spooling State Transitions

#	Old State	Transition Event	New State	Comments
5	Transmitting Spool	Communications lost during spool transmission	Not Unloading	
6	Spool Active/ Not Unloading	Command to purge spool received	Spool Inactive	It is assumed that purge of spool is instantaneous
7	Spool Active/ Transmitting Spool	Transmission of spool complete with communications still established	Spool Inactive	
8	Full Spool Pause On	Command to turn Full Spool Pause Off received	Full Spool Pause Off	
9	Full Spool Pause Off	Command to turn Full Spool Pause On received	Full Spool Pause On	The equipment must ensure that it has paused before the event spool has overflowed. Normally this will require setting Pausing to On (see Table 5-1 on page 21) at some time before the spool is full.

4.4.2 Requirements

- The equipment must ensure that short losses of communications with the supervisor do not cause loss of product or important processing information.
- The equipment must provide enough event spool buffer capacity to hold all the event messages expected during an operation cycle (which may be considered the event messages between resumable points in its operation cycle).
- The default setting of the Full Spool Pause flag must be specified in the equipment description.
- Spooled event reports must be uploaded by order of their timestamps.

4.4.3 Informing the Supervisor of the Event Spool State

The equipment must keep the supervisor informed of the spool state even though most of the state changes will only be communicated when the spool is transmitted to the supervisor after communication is restored. Timestamps on the spooled event reports will assist the supervisor during the transmission of the spooled reports. The spool state transitions that must be reported to the supervisor are transitions 1, 2, and 7 from Table 4-8 above.

The interaction to inform the supervisor of an Event Spool state change is shown in Table 4-9.

TABLE 4-9. Interaction for Event Spool State Change

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=SpoolStateChange (*Old Spool State, New Spool State*)	None	Terminated	

4.4.4 Transmitting Event Spool

The state chart of the interaction for transmitting the event spool is shown in Figure 4-4 and the message exchanges for the interaction are shown in Table 4-10.

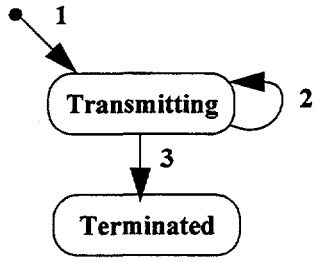


FIGURE 4-4 State Chart for Event Spool Transmission Interaction

TABLE 4-10. Interaction for Transmitting the Event Spool

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>TransmitSpool <-Current Length	None	Transmitting	During the transmission of the spool additional event reports may have to be spooled.
2	<=*Next event report*	Transmitting	Transmitting	The original event reports are transmitted in the order in which they where placed on the spool.
3	<=SpoolTransmitted	Transmitting	Terminated	The spool also becomes inactive and the supervisor must be informed according to the spool state change interaction of Table 4-9.

4.4.5 Purging Event Spool

The interaction to purge the spool consists of a single command: PurgeSpool as shown in Table 4-11.

TABLE 4-11. Interaction for Purging the Spool

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>PurgeSpool	None	Terminated	

4.4.6 Setting Full Spool Pause On

The interaction to turn full spool pause on consists of a single command: FullSpoolPauseOn as shown in Table 4-12.

TABLE 4-12. Interaction for Setting Full Spool Pause On

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>FullSpoolPauseOn	None	Terminated	

4.4.7 Setting Full Spool Pause Off

The interaction to turn full spool pause off consists of a single command: FullSpoolPauseOff as shown in Table 4-13.

TABLE 4-13. Interaction for Setting Full Spool Pause Off

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>FullSpoolPauseOff	None	Terminated	

5 Operation Interactions

5.1 Overview

This section describes the interactions required to

- prepare the equipment to process product
- process product
- deal with off normal conditions

The goal is to setup and configure the equipment for the desired operations, have the equipment perform operations on product, and deal with off-normal conditions that arise. Other than movement of material into and out of the equipment, it is assumed that the equipment is autonomous in the performance of its contracted operations. Interactions

required for movement of material are discussed in Section 6. The supervisor must be able to set equipment process variables and query the equipment about its state. Interactions for setting process variables, querying and monitoring equipment states are described in Section 7. Here, the standard interactions required to orchestrate the equipment through its processing states are presented. Internal states of the equipment are not of interest if they do not effect the interactions with the supervisor that are required to process product.

The standard processing interactions must accommodate wide variety of equipment. Some equipment processes one item at a time while some equipment handles batches of product. Some equipment performs single operations at a time while other equipment may perform operations in a streaming mode with each batch at a different stage in the process. Furthermore, some equipment can upload and download programs while other equipment can only perform preprogrammed operations.

A processing state model is used to provide context for the interactions with the supervisor that are required to process product.

5.2 Processing State Model

The Processing State Model is shown in Figure 5-1. The labeled transitions in the figure are described in Table 5-1. The equipment must perform the steps required to initiate communications during powering up. Thereafter, transitions from communicating to not communicating and back are concurrent with the processing state model.

The equipment remains in Idle until it receives a command to configure. After configuring is complete the equipment is ready to process product. In the Normal Operation state the equipment accepts input to operations, performs operations, and generates product. Under normal conditions, it cycles through these operations until it is commanded by the supervisor to clear itself and go back to the Idle state.

While processing, the equipment can transition to the Pausing state by a command from the supervisor, or detection of a condition by the equipment that requires the equipment to pause. While in the pausing state, the equipment will continue operate normally until it comes to an internal state from which it can pause and resume without aborting the current operation or loosing product. At that time, it will pause until it receives a command from the supervisor to resume or abort.

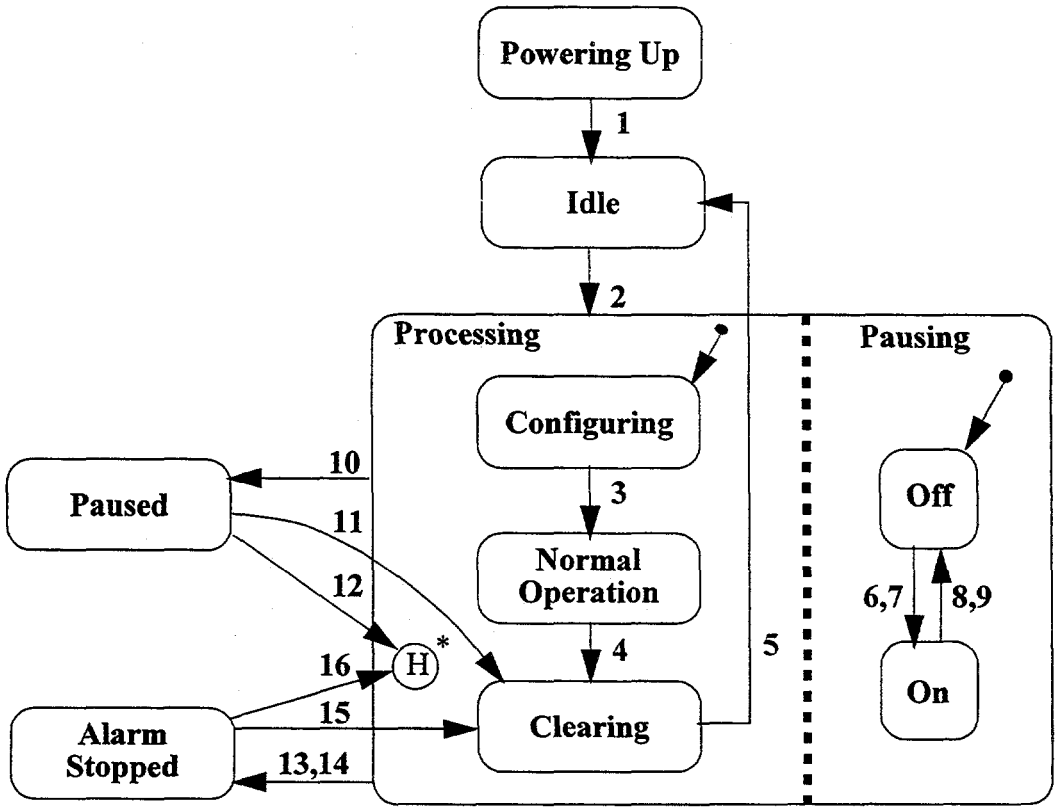


FIGURE 5-1 Processing State Chart

The equipment can also detect conditions that require it to alarm stop, or it can receive an alarm stop command from the supervisor. When either occurs, the equipment will immediately stop processing. Since there is no attempt to pause gracefully, the capability to resume from the Alarm Stopped state cannot be assured. The equipment remains in the Alarm Stopped state until it receives a command to resume or abort from the supervisor.

5.2.1 Description of Processing States

Powering Up

The equipment is performing power up initializations.

Idle

The equipment has performed power up initializations and is waiting for a command to configure for operations.

Configuring

The equipment is configuring for operation based on the configure command form the supervisor. This processing state may require material movement interactions.

Normal Operation

The equipment is configured and in normal operation state. In this state, the equipment can accept product for processing, perform operations on it, and provide output products. The equipment cycles through operations on product until it is command to clear

itself, receives an alarm stop, or reaches a resumable intermediate state with Pausing On. All material movement interactions (See "Material Movement Interactions" on page 28.) can occur in this state.

Clearing

The equipment is performing any activities required to clear itself and return to the idle state. This state may require material movement interaction especially if it was entered from the Paused or Alarm Stopped states upon receipt of an abort command.

Pausing Off

The equipment is not scheduled to pause.

Pausing On

The equipment will enter the Paused state at the next point in its processing from which it can resume.

Paused

The equipment is in a paused state in which processing has stopped but can be resumed from where it left off.

Alarm Stopped

The equipment has performed an alarm stop from which there is no assurance that normal processing can be resumed.

TABLE 5-1. Processing State Transitions

#	Old State	Transition Event	New State	Comments
1	Powering Up	Initialization complete	Idle	
2	Idle	Command to configure received	Processing	The Processing substate is Configuring and Pausing is Off
3	Configuring	Configuration complete	Normal Operation	
4	Normal Operating	Command to clear received	Clearing	
5	Clearing	Clearing completed	Idle	
6	Pausing Off	Command to pause received	Pausing On	
7	Pausing Off	Internal condition detected that requires the equipment to pause for handling	Pausing On	
8	Pausing On	Command to resume processing received	Pausing Off	
9	Pausing On	Command to abort processing received	Pausing Off	
10	Processing	A resumable internal condition reached with Pausing On	Paused	
11	Paused	Command to abort processing received	Clearing	

TABLE 5-1. Processing State Transitions

#	Old State	Transition Event	New State	Comments
12	Paused	Command to resume processing received	Previous internal processing state from which the Pause state was entered	
13	Processing	Command to Alarm Stop received	Alarm Stopped	
14	Processing	Internal condition detected that requires the equipment to alarm stop for handling	Alarm Stopped	
15	Alarm Stopped	Command to abort processing received	Clearing	
16	Alarm Stopped	Command to resume processing received	Previous internal processing state from which the Alarm Stopped state was entered	

5.3 Informing the Supervisor of the Processing State

The equipment must keep the supervisor informed of its processing state. Transitions 1, 3, 5, 7, 10, 14 shown in Table 5-1 are internal equipment events that should be reported to the supervisor using the ProcessingStateChange event report as shown in Table 5-2. All other transitions in the table are due to commands from the supervisor and acceptance of the command implies the occurrence of the transition. Reporting of those events using the ProcessingStateChange event report is optional.

TABLE 5-2. Interaction for Processing State Change

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=ProcessingStateChange (*Old Processing State, New Processing State*)	None	Terminated	

5.4 Configuration and Normal Processing Interactions

The normal path through the Processing State Chart takes the equipment all the way from start-up to its normal operating state where it can process product. It also provides a path for leaving the normal operating state to allow reconfiguration if needed for particular operations. The transition to Idle at the completion of powering up procedures is reported to the supervisor using the ProcessingStateChange event report.

5.4.1 Configuring the Equipment

The equipment stays in Idle until it receives the Configure command shown in Table 5-3. During configuration, the equipment may require material that it obtains through the material movement interaction described in Section 6. At the completion of configuring, the equipment transitions to the Normal Operation state.

TABLE 5-3. Interaction for Configuring Equipment

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>Configure(*configurationId, (Parameter list)*)	None	Terminated	The parameters of the configuration will be equipment specific and must be described in the equipment configuration table. See Section 9.3.

5.4.2 Clearing the Equipment

Under normal conditions the equipment stays in the Normal Operation state and processes product until it is commanded to clear by the supervisor. This occurs when shutting down the equipment or when the equipment needs to be reconfigured. The interaction for clearing the equipment is described in Table 5-4.

TABLE 5-4. Interaction for Clearing Equipment

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>Clear	None	Terminated	

5.5 Operation Initiation

In the Normal Operation state, the equipment receives product items to process and produces product. Once the equipment has received all of the input items for an operation, it is commanded to start the operation with the StartItemOperation command. After any required initialization, the equipment informs the supervisor that the operation has started. The interaction for Operation Initiation shown in Section 5-5.

TABLE 5-5. Interaction for Operation Initiation

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>StartItemOperation(*Operation Name, (Item ID list), (Parameter List), Operation Token*)	None	StartingOperation	The assigned operation token allows unambiguous reference to the particular execution of the operation.
2	<=ItemOperationStart(*Operation Token*)	StartingOperation	Terminated	

5.6 Operation Completion

When an operation is complete, the equipment must inform the supervisor. The event report must indicate the operation that was completed, and its completion code. Negative completion codes indicate that there was a problem. In particular, when the equipment is commanded to abort, all current operations of the equipment must complete with a negative completion code. The possible completion codes for an operation are included in definition of the operation in the operation table for the equipment as discussed in Section 9.2. The equipment informs the supervisor of operation completion through the ItemOperationComplete event report as shown in Table 5-6.

TABLE 5-6. Interaction for Operation Completion

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=ItemOperationComplete (*operation token, completion code*)	None	Terminated	The operation token is that given by the supervisor in the StartItemOperation command for the operation.

5.7 Off-Normal Processing Interactions

During the Processing state various off normal conditions can occur either at the equipment itself or elsewhere in the system that require processing to be interrupted. Those conditions detected by the equipment move it into the Pausing state or to Alarm Stopped depending upon the severity of the condition. These state transitions (7 and 14 in Table 5-1, "Processing State Transitions," on page 21) must be communicated to the supervisor through the ProcessingStateChange event report. When the off-normal condition is detected elsewhere in the system the supervisor can command the equipment to Pause or Alarm Stop. These commands are shown in Table 5-7 and Table 5-8 respectively.

TABLE 5-7. Interaction for Command Pausing the Equipment

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>Pause	None	Terminated	

TABLE 5-8. Interaction for Command Alarm Stopping the Equipment

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>AlarmStop	None	Terminated	

After the supervisor deals with the pause or alarm stop condition, it can command the equipment to either Resume or Abort processing. The Resume command causes the equip-

ment to continue from where it left off. The Abort command causes the equipment to go into the Clearing state. These commands are shown in Table 5-9 and Table 5-10.

TABLE 5-9. Interaction for Resuming Processing

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>Resume	None	Terminated	

TABLE 5-10. Interaction for Aborting Processing

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>Abort	None	Terminated	

5.8 Alarm States

The alarm state of the equipment is a composite of the states of all of the alarm conditions that can be detected by the equipment. Each detectable alarm condition has a unique identification, *AlarmId*, by which it is referenced. The standard state chart for each alarm condition is shown in Figure 5-2. The initial state of all alarms conditions is off. It is the responsibility of the equipment to detect all alarm conditions that can have an adverse impact on the safety of personnel or the quality of product, or that can damage equipment. The definition of the alarm state transitions are given in Table 5-11.

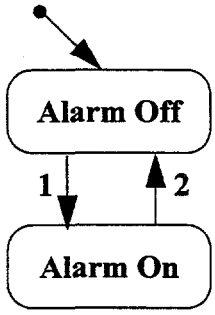


FIGURE 5-2 Generic Alarm State Chart

TABLE 5-11. Alarm State Transitions

#	Old State	Transition Event	New State	Comments
1	Alarm Off	Alarm condition is detected by the equipment	Alarm On	The equipment has a set of alarms that it can detect.
2	Alarm On	Alarm condition is no longer detected by the equipment	Alarm Off	The equipment may periodically check to determine if the alarm condition still exits. It also must check for the condition when asked by the supervisor to check for the alarm condition.

The interaction to inform the supervisor when an alarm condition is detected is shown in Table 5-12, while the interaction to inform the supervisor that an alarm condition has been cleared is shown in Table 5-13.

TABLE 5-12. Interaction for Notification of Alarm Condition On

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=AlarmOn (*AlarmId*)	None	Terminated	

TABLE 5-13. Interaction for Notification of Alarm Condition Off

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=AlarmOff (*AlarmId*)	None	Terminated	

When an alarm condition is on, the equipment must allow the supervisor to perform the necessary actions to handle the alarm. This may require that it set its Pausing State to On and inform the supervisor according to the interaction shown in Table 5-2. The supervisor will inform the equipment when it is through handling the alarm by commanding it to check the alarm condition according to the interaction shown in Table 5-14

TABLE 5-14. Interaction for Checking an Alarm Condition

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>CheckAlarm (*AlarmId*)	None	Terminated	This interaction ensures that the equipment will check for the condition when the supervisor thinks that the alarm has been cleared. Some equipment may not be checking unless told to check, while other equipment may periodically check the condition on its own. If upon checking the alarm state has changed, the equipment will issue the AlarmOn or AlarmOff notification.

5.9 Process Program Management

Manufacturing equipment can have many degrees of complexity, programmability, and autonomy. Some equipment will not be programmable except through the parameters that are passed to it with the StartItemOperation command shown in Table 5-5. Other equipment will be fully programmable in terms of the Process Description Language of the associated manufacturing process. In the latter case, there must be standard interactions for uploading and downloading equipment programs, and for selecting programs to be executed for desired products. Parameters of the StartItemOperation command may select the program to be run, or configuration specifications associated with the product production specifications may be used by the supervisor to setup the proper program. These interactions will be covered in latter versions of this document.

6 Material Movement Interactions

6.1 Overview

This section describes the material movement interactions for getting product and material into and out of the equipment, replenishing supplies, and removing waste. These interactions are intended to be process independent and thus are the same for a machining, welding or a chemical processing cell. For the complex interactions several off-normal states are modeled. Although the supervisor may not handle the off-normal interaction states in a automated fashion, the message exchanges are standardized to keep both the supervisor and the equipment informed of the state of the interaction.

6.2 Permit and Deny Input for Operation

The equipment must inform the supervisor when it can or cannot accept input for particular operations. For example, certain configurations may only be appropriate for selected operations or the equipment may only be able to process specified amounts of product at a time. When the equipment is configured, it must inform the supervisor of which operations it can perform. Similarly, when the input capacity for a particular operation is full, the equipment must inform the supervisor. The interactions for permitting and denying input for specified operations are shown in Table 6-1 and Table 6-2.

TABLE 6-1. Interaction for Permitting Input for Specified Operations

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= PermitInputForOperation (*OperationIdList*)	None	Terminated	

TABLE 6-2. Interaction for Denying Input for Specified Operations

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= DenyInputForOperation(*OperationIdList*)	None	Terminated	

6.3 Input Item Receipt

The interaction for entering an item into the equipment for processing starts when the supervisor tells the equipment to take an item for input to an operation. The equipment responds by indicating the port and location within the port in which to place the item. The port must be locked before the supervisor can interact with it. After the port is locked, the supervisor inserts the item and asks the equipment to confirm receipt. The port can then be unlocked and the interaction terminated. The state chart for the item receipt interaction is shown in Figure 6-1. The off-normal states of the interaction, Fixing Can't Lock and Fixing Can't Find Item, are shown as well as the normal states of the interaction. The message exchanges for the interaction are shown in Table 6-3.

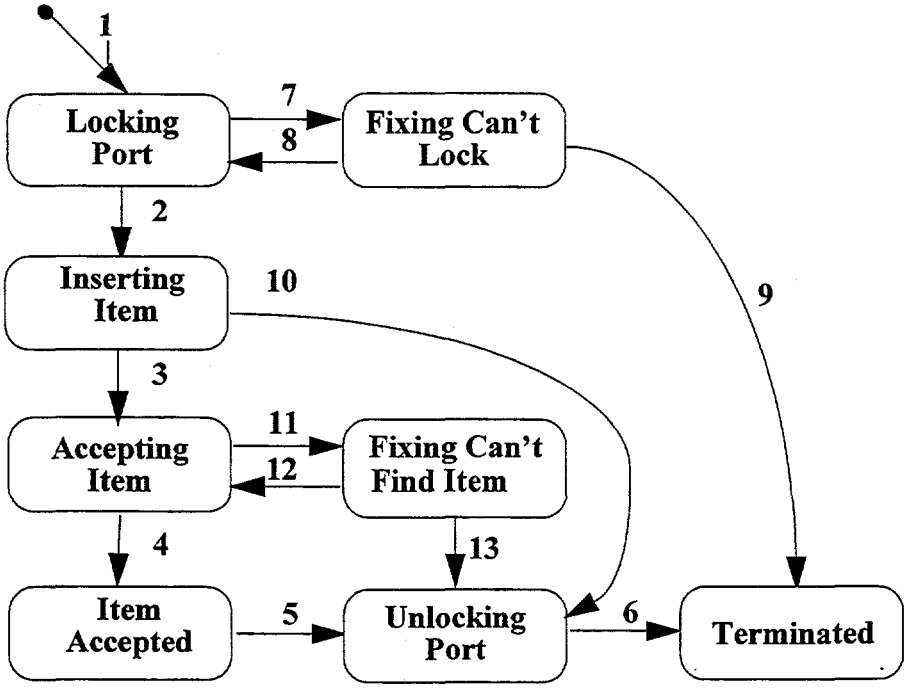


FIGURE 6-1 State Chart for the Item Receipt Interaction

TABLE 6-3. Interaction for Item Receipt

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=> TakeItem (*itemClass, elementalOperation, inputNumber*) <-(*portId, portIndex*)	None	Locking Port	
2	<= PortLocked (*portId*)	Locking Port	Inserting Item	
3	=>AcceptItem (*itemId, portId, portIndex*)	Inserting Item	Accepting Item	
4	<=ItemAcepted (*itemId*)	Accepting Item	Item Accepted	
5	=>UnlockPort (*portId*)	Item Accepted	Unlocking Port	
6	<=PortUnlocked (*portId*)	Unlocking Port	Terminated	
7	<=PortCantLock-Fault(*portId*)	Locking Port	Fixing Can't Lock	The supervisor will attempt recovery
8	=>ResumePortLock (*portId*)	Fixing Can't Lock	Locking Port	
9	=>AbortTakeItem (*portId*)	Fixing Can't Lock	Terminated	The supervisor gives up
10	=>UnlockAndAbort-TakeItem(*portId*)	Inserting Item	Unlock Port	

TABLE 6-3. Interaction for Item Receipt

#	Message Exchange	Old Inter. State	New Inter. State	Comment
11	<=CantFindItem (*itemId, portId, portIndex*)	Accepting Item	Fixing Can't Find Item	The supervisor will attempt recovery
12	=>AcceptItem (*itemId, portId, portIndex*)	Fixing Can't Find Item	Accepting Item	
13	=>UnlockAndAbort-TakeItem(*portId*)	Fixing Can't Find Item	Unlock Port	The supervisor gives up

6.4 Product Available Notification

When the operation is completed normally, all outputs are assumed to be available. For some operations, various outputs of the operation may become available before the operation completes. If an operation completes with an error completion code, not all of the intended products may be produced. A separate interaction to inform the supervisor of product availability is specified for these situations. The interaction for indicating product availability is shown in Table 6-4.

TABLE 6-4. Interaction for Product Available Notification

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= ProductAvailable (*opToken, outputNumber*)	None	Terminated	

6.5 Product and Item Removal

Normally, only product is removed from the equipment. However, when the supervisor aborts an operation, some of the input items for the operation may have to be removed. A single interaction is defined for product or item removal. Product removal is initiated by the *GiveProduct* command and item removal is initiated by the *GiveItem* command. The state chart for the product or item removal interaction is shown in Figure 6-2. The message exchanges for the interaction are shown in Table 6-5. The off-normal variations as well as the normal path through the interaction are shown.

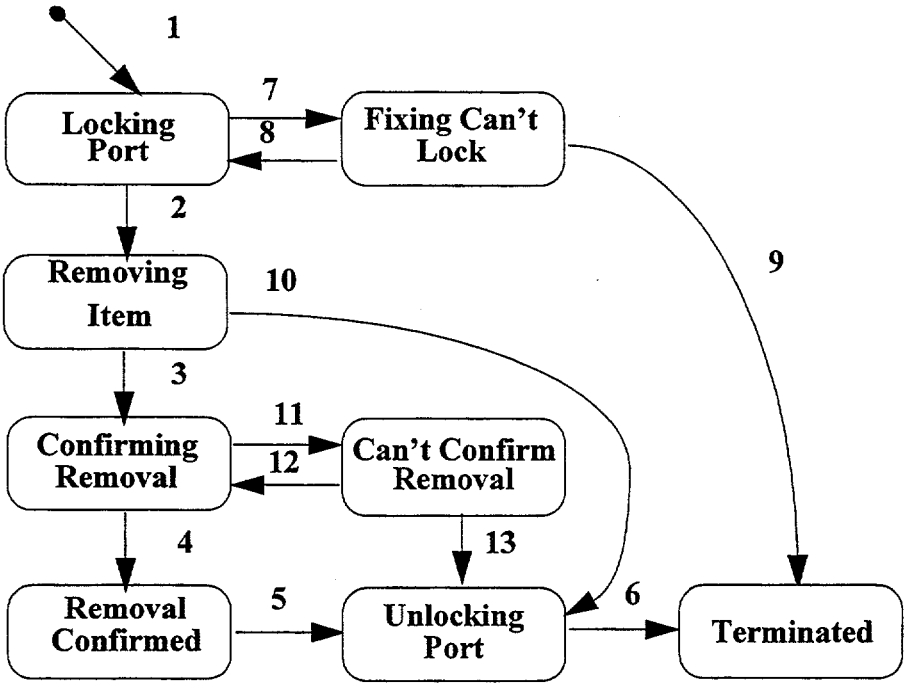


FIGURE 6-2 State Chart for Product or Item Removal Interaction

TABLE 6-5. Interaction for Product or Item Removal

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>GiveProduct(*opToken,outputNumber*) <-(*portId,portIndex,itemClass*) or =>GiveItem (*itemId*) <-(*portId,portIndex,itemClass*)	None	Locking Port	
2	<=PortLocked(*portId*)	Locking Port	Removing Item	
3	=>ConfirmRemoval (*portId, PortIndex*)	Removing Item	Confirming Removal	
4	<=RemovalConfirmed (*portId, PortIndex*)	Confirming Removal	Removal Confirmed	
5	=>UnlockPort(*portId*)	Removal Confirmed	Unlocking Port	
6	<=PortUnlocked(*portId*)	Unlocking Port	Terminated	
7	<=PortCantLockFault(*portId*)	Locking Port	Fixing Can't Lock	The supervisor will attempt recovery

TABLE 6-5. Interaction for Product or Item Removal

#	Message Exchange	Old Inter. State	New Inter. State	Comment
8	=>ResumePort-Lock(*portId*)	Fixing Can't Lock	Locking Port	
9	=>AbortGiveProduct (*portId*)	Fixing Can't Lock	Terminated	The supervisor gives up
10	=>UnlockAndAbortGive-Product (*portId*)	Removing Item	Unlock Port	
11	<=CantConfrimRemoval (*portId, portIndex*)	Confirming Removal	Can't Confirm Removal	The supervisor will attempt recovery
12	=>ConfirmRemoval (*portId, portIndex*)	Can't Confirm Removal	Confirming Removal	
13	=>UnlockAndAbortGive-Product (*portId*)	Can' t Confirm Removal	Unlock Port	The supervisor gives up

6.6 Supply Replenishment

The equipment can require consumable supplies of various types. The replenishment of supplies is an interaction in which the equipment requests that the supervisor load a particular port with the specified supply type and amount. The supply replenishment interaction is similar to the Item Receipt interaction. It is shown in Figure 6-3. The message exchanges for the interaction are shown in Table 6-6.

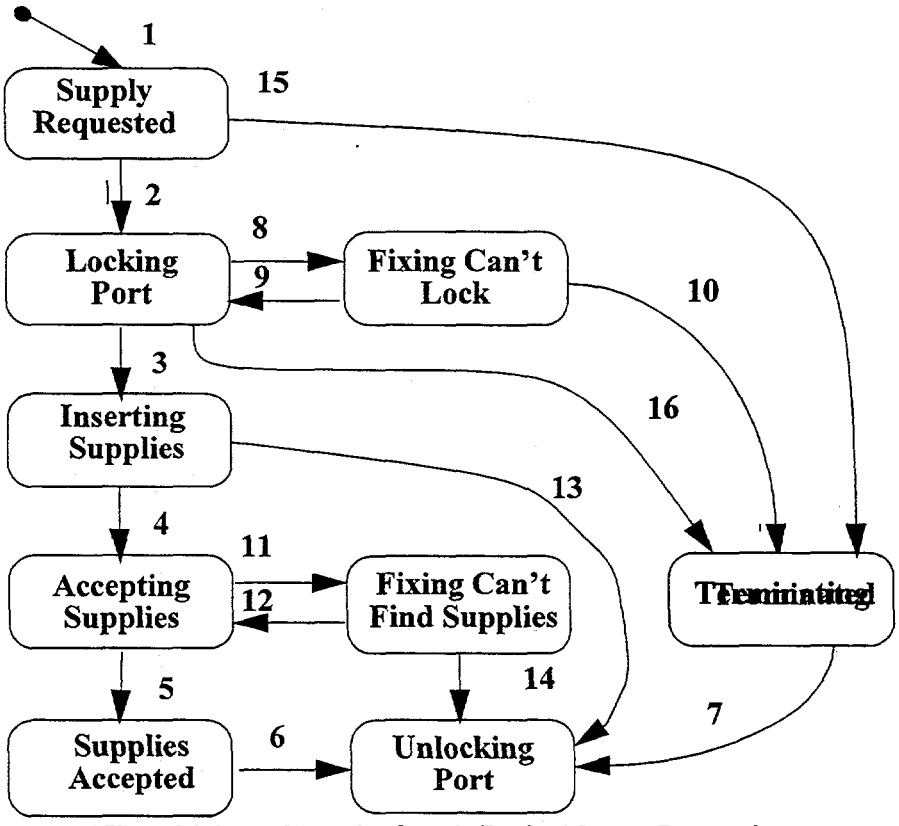


FIGURE 6-3 State Chart for Supply Replenishment Interaction

TABLE 6-6. Interaction for Supply Replenishment

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=SupplyReplenishmentRequest (*supplyId, portId, portIndex, quantity*)	None	Supply Requested	
2	=>LockPort(*portId*)	Supply Requested	Locking Port	
3	<= PortLocked (*portId*)	LockingPort	Inserting Supplies	
4	=>AcceptSupplies (*itemId, portId, portIndex*)	Inserting Supplies	Accepting Supplies	
5	<=SuppliesAccepted (*itemId*)	Accepting Supplies	Supplies Accepted	
6	=>UnlockPort (*portId*)	Supplies Accepted	Unlocking Port	
7	<=PortUnlocked (*portId*)	Unlocking Port	Terminated	
8	<=PortCantLockFault(*portId*)	Locking Port	Fixing Can't Lock	The supervisor will attempt recovery
9	=>ResumePortLock (*portId*)	Fixing Can't Lock	Locking Port	
10	=>AbortReplenishment (*portId*)	Fixing Can't Lock	Terminated	The supervisor gives up
11	<=CantFindSupplies (*itemId, portId, portIndex*)	Accepting Supplies	Fixing Can't Find Supplies	The supervisor will attempt recovery
12	=>AcceptSupplies(*itemId, portId, portIndex*)	Fixing Can't Find Supplies	Accepting Supplies	
13	=>UnlockAndAbortReplenishment (*portId*)	Inserting Supplies	Unlocking Port	
14	=>UnlockAndAbortReplenishment (*portId*)	Fixing Can't Find Supplies	Unlocking Port	The supervisor gives up
15	<=AbortReplenishmentRequest (*portId, portIndex*)	Supply Requested	Terminated	The equipment rescinds the replenishment request
16	<=AbortReplenishmentRequest (*portId, portIndex*)	Locking Port	Terminated	The equipment rescinds the replenishment request

6.7 Waste Removal

The equipment can generate waste of various types. The waste removal interaction is initiated by the equipment through a request to unload a specified amount of a type of waste from a particular port. The state chart for the waste removal interaction is shown in Figure 6-4 and the message exchanges are shown in Table 6-7. This interaction is similar to the Product and Item removal interaction.

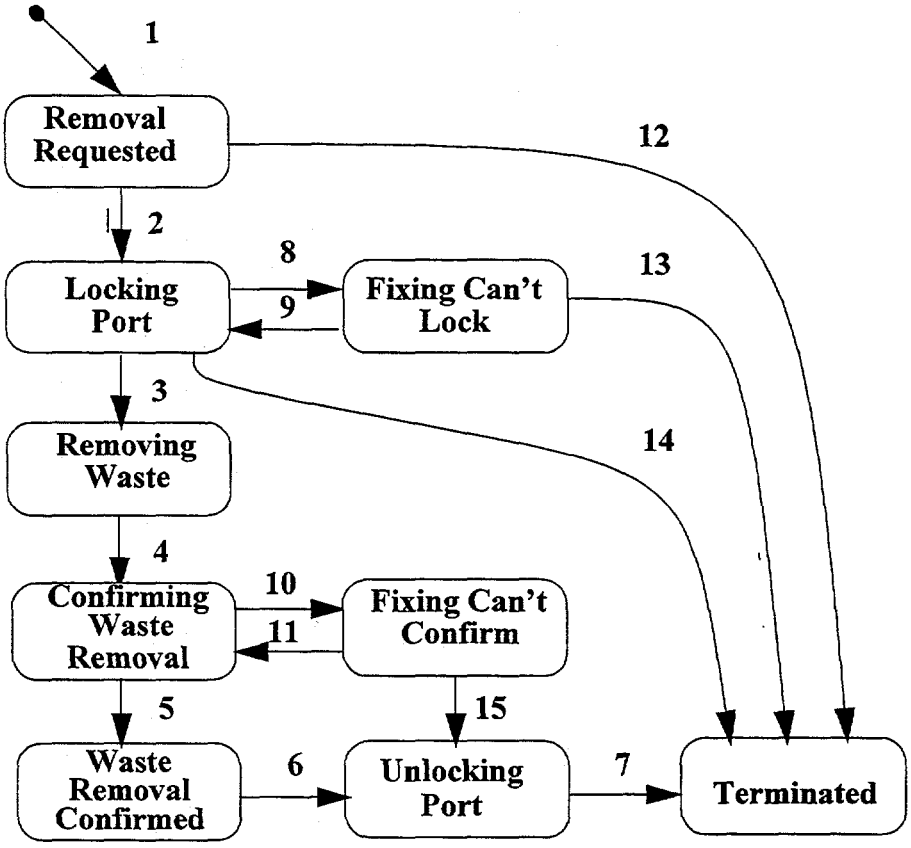


FIGURE 6-4 State Chart For Waste Removal Interaction

TABLE 6-7. Interaction for Waste Removal

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=WasteRemovalRequest(*portId,portIndex,wasteId*)	None	Removal Requested	
2	=>LockPort(*portId*)	Removal Requested	Locking Port	
3	<= PortLocked (*portId*)	Locking Port	Removing Waste	
4	<=ConfirmWasteRemoval (*portId, portIndex*)	Removing Waste	Confirming Waste Removal	

TABLE 6-7. Interaction for Waste Removal

#	Message Exchange	Old Inter. State	New Inter. State	Comment
5	<=ConfirmWasteRemoval (*portId, portIndex*)	Confirming Waste Removal	Waste Removal Confirmed	
6	=>UnlockPort (*portId*)	Waste Removal Confirmed	Unlocking Port	
7	<=PortUnlocked (*portId*)	Unlocking Port	Terminated	
8	<=PortCantLockFault(*portId*)	Unlocking Port	Fixing Can't Lock	The supervisor will attempt recovery
9	=>ResumePortLock (*portId*)	Fixing Can't Lock	Locking Port	
10	=>CantConfirmWasteRemovalFault(*portId,portIndex*)	Confirming Waste Removal	Can't Confirm	The supervisor will attempt recovery
11	=>ConfirmWasteRemoval (*portId, portIndex*)	Can't Confirm	Confirming Waste Removal	
12	<=AbortRemovalRequest (*portId, portIndex*)	Removal Requested	Terminated	The equipment rescinds the replenishment request
13	=>AbortWasteRemoval (*portId, portIndex*)	Fixing Can't Lock	Terminated	The supervisor gives up
14	<=AbortRemovalRequest (*portId, portIndex*)	Locking Port	Terminated	The equipment rescinds the replenishment request
15	=>UnlockAndAbortRemoval (*portId*)	Can't Confirm	Unlocking Port	The supervisor gives up

6.8 Input From Reservoir

Equipment can require input of material from reservoirs that are shared by other equipment in the cell. In this case, the supervisor must arbitrate the use of the shared reservoirs. The equipment requests to input from the reservoir. Then the supervisor grants or denies the request. If the request is granted, the equipment takes from the reservoir the requested amount and informs the supervisor when it is done.The state chart for the input form reservoir is shown in Figure 6-5 and the messages exchanges are shown in Table 6-8

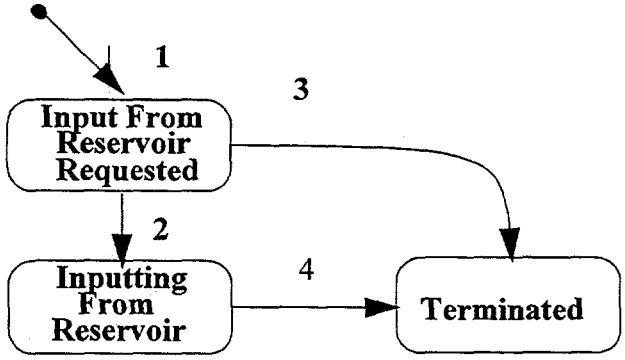


FIGURE 6-5 State Chart for Input From Reservoir Interaction

TABLE 6-8. Interaction for Inputting From Reservoir

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= InputFromReservoir-Request(*reservoirId,amount*)	None	Input From Reservoir Requested	
2	=> GrantInputFromReservoirRequest(*reservoirId*)	Input From reservoir Requested	Inputting From Reservoir	
3	=> InputFromReservoirDenied(*reservoirId*)	Input From reservoir Requested	Terminated	
4	<= InputFromReservoir-RequestClear(*reservoirId*)	Inputting From Reservoir	Terminated	

6.9 Output To Reservoir

Equipment may need to output material to reservoirs that are shared by other equipment in the cell. In this case, the supervisor must arbitrate the use of the shared reservoirs. The equipment requests to output the reservoir. Then the supervisor grants or denies the request. If the request is granted, the equipment outputs to the reservoir the requested amount and informs the supervisor when it is done.The state chart for the output to reservoir is shown in Figure 6-6 and the messages exchanges are shown in Table 6-9

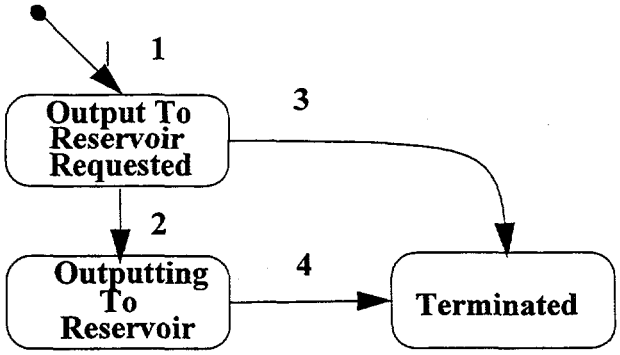


FIGURE 6-6 State Chart for Output to Reservoir Interaction

TABLE 6-9. Interaction for Outputting to Reservoir

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= OutputToReservoirRequest(*reservoirId,amount*)	None	Output To reservoir Requested	
2	=> GrantOutputToReservoirRequest(*reservoirId*)	Output To reservoir Requested	Outputting To Reservoir	
3	=> OutputToReservoirDenied(*reservoirId*)	Output To reservoir Requested	Terminated	
4	<= OutputToReservoirRequeastClear (*reservoirId*)	Outputting To Reservoir	Terminated	

6.10 Continuous Material Service Initiation and Termination

Many types of equipment require continuous supply of bulk material. For example, some equipment continuous supply of hydraulic fluid pressure, or a continuous source of cleaning fluid or solvent. In this case, the supervisor must initiate the service when requested by the equipment (typically at power up), and terminate the service when requested (typically at shut down). The bulk material service state chart is shown in Figure 6-7 and transitions are described in Table 6-10.

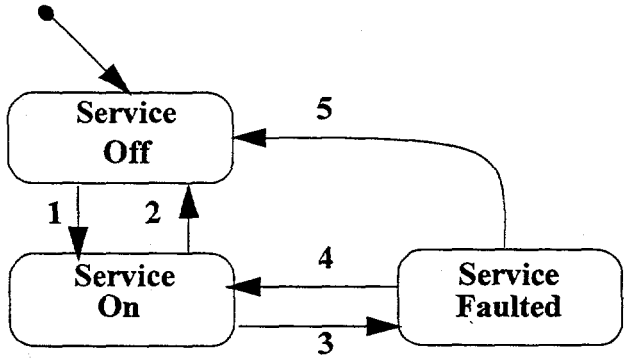


FIGURE 6-7 Bulk Material Service State Chart

TABLE 6-10. Bulk Material Service Transitions

#	Old State	Transition Event	New State	Comments
1	Service Off	Service is turned on by the supervisor at the request of the equipment	Service On	
2	Service On	Service is turned off by the supervisor at the request of the equipment	Service Off	
3	Service On	Service problems detected by the equipment	Service Faulted	The supervisor will try to recover service
4	Service Faulted	The supervisor is able to restore service	Service On	
5	Service Faulted	The supervisor is unable to restore service	Service Off	

Three interactions between the supervisor and the equipment deal with bulk material service: Material Service Initiation, Material Service Termination and Material Service Fault Handling. The state chart for the material service initiation interaction is shown in Figure 6-8 and the message exchanges is shown in Table 6-11. The state chart for material service termination is shown in Figure 6-9 and the message exchanges are shown in Table 6-12.

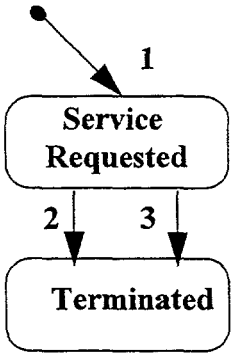


FIGURE 6-8 State Chart For Material Service Initiation Interaction

TABLE 6-11. Interaction for Bulk Material Service Initiation

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= MaterialServiceRequest(*materialId*)	None	Service Requested	
2	=> MaterialServiceOn (*materialId*)	Service Requested	Terminated	
3	=> MaterialServiceDenied (*materialId*)	Service Requested	Terminated	

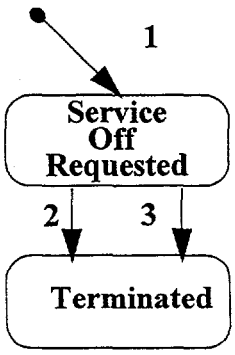


FIGURE 6-9 State Chart For Material Service Termination Interaction

TABLE 6-12. Interaction for Termination of Material Service

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= MaterialServiceOffRequest (*materialId*)	None	Service Off Requested	
2	=>MaterialServiceOff (*materialId*)	Service Off Requested	Terminated	
3	=>MaterialServiceOffDenied (*materialId*)	Service Off Requested	Terminated	

During processing, the equipment may detect problems with one of the material services that it requires. The state chart for the interaction to handle material service problems is shown in Table 6-10 and the message exchanges are shown in Table 6-13.

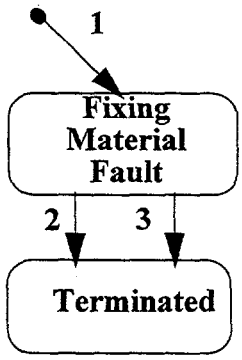


FIGURE 6-10 State Chart for Material Service Fault Interaction

TABLE 6-13. Interaction for Handling A Material Service Fault

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<= MaterialServiceFault(*materialId*)	None	Fixing fault	The supervisor will attempt recovery
2	=> MaterialServiceOn (materialId)	Fixing fault	Terminated	Supervisor thinks service problem has been fixed
3	=> HaltMaterialService (*materialId*)	Fixing fault	Terminated	Supervisor gives up

7 Settings, Queries, and Monitoring

7.1 Overview

The equipment must give the supervisor access to all relevant information about the equipment itself and the work in progress. This should be done though event reports as variables change, and through response to queries. This interface specification document has defined a set of standard state and equipment variables. Special queries are specified below for these variables. The description of the equipment will include any additional non-standard equipment setting and data variables to which the supervisor should have access through the equipment symbol tables. A set of generic symbol commands that access the symbols by name are defined here. Generic equipment capability to monitor its data variables is specified. The monitors can detect crossing of threshold values such as high temperature, low cooling fluid pressure, or other alarm and warning conditions. The supervisor can set some of the monitoring conditions for the data variables.

7.2 Standard Variable Access

The equipment must support queries for the current value of all the standard equipment variables. The queries take the form of single commands from the supervisor in which the equipment returns the value of the variable. The standard state variables are: Remote Control State, Communication Loss Pause, Event Spooling State, Full Spool Pause, Process State, Pausing State, Alarm States, and Port States.

The equipment must also respond to queries for current working variables including: Work in Progress, Status of Operation, Available Products, Port Contents, Supply Inventory, Waste Inventory, Pending Load Requests, Pending Unload Requests, Pending Alarms, Active Input From Reservoir Requests, and Active Output To Reservoir Requests. The standard queries with required arguments, if any, are listed in Table 7-1 along with a description of the expected response. As before, => indicates a command from the supervisor to the equipment and <- indicates the response returned by the equipment as part of the query command transaction.

TABLE 7-1. Standard Query Interactions

Query	Response	Comment
=>ReturnRemoteControlState	<-*RemoteControlState*	Local or Remote
=>ReturnCommunicationLoss-Pause	<-*CommunicationLoss-Pause*	On or Off
=>ReturnEventSpoolingState	<-*EventSpoolingState*	Active or Inactive
=>ReturnFullSpoolPause	<-*FullSpoolPause*	On or Off
=>ReturnSpoolLoadingState	<-*SpoolLoadingState*	SpoolFull or SpoolNotFull
=>ReturnSpoolUnloadingState	<-*SpoolUnloadingState*	NotUnloading, TransmittingSpool or PurgingSpool
=>ReturnProcessState	<-*ProcessState*	One of states described in Section 5.2.1

TABLE 7-1. Standard Query Interactions

Query	Response	Comment
=>ReturnPausingState	<-*PausingState*	On or Off
=>ReturnAlarmState(*AlarmId)*	<-*AlarmState*	On or Off
=>ReturnPortState(*PortId)*	<-*PortState*	Locked or Unlocked
=>ReturnWorkInProgress	<-(*Operation Token List*)	List of operation tokens which either have not yet completed or have output that has not yet been removed from the equipment. The operation token is assigned to the operation in the StartItemOperation Command given in Table 5-5.
=>ReturnStatusOfOperation (*OperationToken)*	<-(*Status, Completion-Code, CompletionTime)*	The status is one of Starting, Executing or Completed. If the operation is not yet complete, the completion code is 0 and the completion time is estimated.
=>ReturnAvailableProducts	<-(List of (*OperationToken, Output#*) pairs)	
=>ReturnPortContents (*PortId*)	<-(List of (*PortIndex, Item*) pairs)	The item is one of *ItemId* or (*OperationToken, Output#*). The *ItemId* refers to the item accepted in a Take Item interaction described in Section 6.3 above.
=>ReturnSupplyInventory (*SupplyType*)	<-*SupplyQuantity*	The supply types are listed in the supply table for the equipment described in Section 9.4 and along with the units for the quantity.
=>ReturnWasteInventory (*WasteType*)	<-*WasteQuantity*	The waste types are listed in the waste table for the equipment described in Section 9.5 and along with the units for the quantity.
=>ReturnPendingPortLoadRequests	<-(List of (*PortId, PortIndex, ItemClass, Quantity*))	The item class is a supply type.
=>ReturnPendingPortUnload Requests	<-(List of (*PortId, PortIndex, ItemClass, Quantity*))	The item class is a supply type, waste type or product type.
=>ReturnPendingAlarms	<-(*AlarmId List*)	The Alarm Ids are listed in the Alarm Table for the equipment described in Section 9.10.
=>ReturnActiveInputFromReservoirRequests	<-(List of (*ReservoirId, Quantity*) pairs)	The Reservoir Ids and the units for quantity are listed in the Shared Reservoir Table for the equipment described in Section 9.8.
=>ReturnActiveOutputToReservoirRequests	<-(List of (*ReservoirId, Quantity*) pairs)	The Reservoir Ids and the units for quantity are listed in the Shared Reservoir Table for the equipment described in Section 9.8.

7.3 Equipment Specific Variables

Access to the equipment data variables consists of query commands that specify the names of the variables whose values are to be returned. Access to equipment setting variables consists of the same query commands together with commands that set the values of variables specified by name. The definitions of the equipment data and setting variables are part of the equipment description tables as discussed in Section 9.12 and Section 9.13.

7.3.1 Getting Symbol Values

There are two query interactions by which the supervisor can obtain values of equipment variables. The first query is for a single variable value and the other is for a group of values. Each query involves a single message exchange. These are shown in Table 7-2 and Table 7-3.

TABLE 7-2. Interaction for Getting a Symbol Value

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>GetSymbolValue (*Name*) <-*Value*	None	Terminated	

TABLE 7-3. Interaction for Getting a List of Symbol Values

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>GetSymbolValueList (*Name List*) <-(List of (*Name, Value*) pairs)	None	Terminated	

7.3.2 Setting Symbol Values

There are two interactions by which the supervisor can set equipment setting variables. The first is for setting a single value and the other is for setting a list of values. Each of these consists of a single message exchange. These are shown in Table 7-4 and Table 7-5.

TABLE 7-4. Interaction for Setting a Symbol Value

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>SetSymbolValue (*Name, Value*)	None	Terminated	

TABLE 7-5. Interaction for Setting a List of Symbol Values

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>SetSymbolValueList (List of (*Name, Value*) pairs)	None	Terminated	

7.4 Variable Monitoring

The supervisor must be able to monitor all of the state, setting and data variables of the equipment. The supervisor is informed of changes in state variable values according to interactions described in Section 4 and Section 5. The setting variables can be changed by the local operator when the equipment is under local control and by the supervisor when the equipment is under remote control. In order to have a consistent mechanism for the supervisor to record and handle equipment setting changes, the equipment must inform the supervisor of setting changes through the interaction shown in Table 7-6.

Monitoring capabilities must be provided for the supervisor to be able to ensure quality of product and service of the equipment. Generic threshold monitoring functionality is specified together with a means to provide the supervisor with control over custom monitoring functionality that the equipment might have.

TABLE 7-6. Interaction for Setting Variable Change

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=SettingChange (*Name, Value*)	None	Terminated	

7.4.1 Threshold Monitoring

The monitoring of many data variables can be mapped on to a variation of the threshold monitor functionality described in the SEMI GEM Model. The value of a data variable can be in one of three zones: High, Normal or Low as shown in Figure 7-1. The boundaries between zones are thresholds. To prevent chatter in the reporting of transitions from one zone to another, deadzones are defined by the overlap of boundaries between zones. Transitions occur only when the variable value crosses the far boundary of the deadzone from its current zone. Thus, to transition from Normal to High the variable value must become greater than the High DeadZone Upper Limit, and to transition from High to Normal the

value must become lower than the High DeadZone Lower Limit. If chatter is not a concern, the upper and lower limits of the deadzones can be the same.

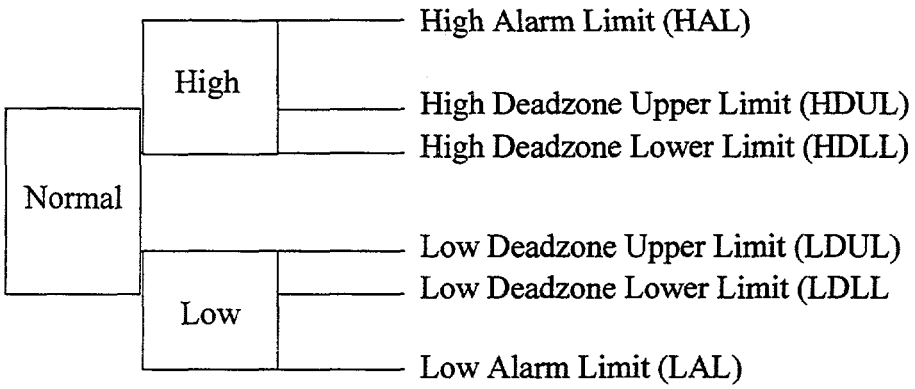


FIGURE 7-1 Zones and Threshold Monitoring Limits

Three types of data variables are identified based upon the required response of the supervisor to a zone transition. For consumable supply inventory variables, the usual concern is the replenishment of supplies that are consumed. For waste inventory variables, the usual concern is waste removal. Finally, for process variables that are important to the quality of the product, the concern is to keep the variable value within the normal range. Typically, the supervisor will need to set the boundaries of the normal range for all three types of data variables. However, the alarm limits are values that are characteristic of the equipment and serve to protect the equipment and surrounding objects (other equipment and personnel). These limits are not setable by the supervisor.

The equipment must maintain a zone state model for each of the data variables with threshold monitors. The general state chart for threshold monitored data variables is shown in Figure 7-2. Upon start-up, the equipment must determine whether the data variable is within the Normal zone and generate a zone transition event report if it is not. This is equivalent to having the initial zone state be Normal and treating the value at start-up as a new value.

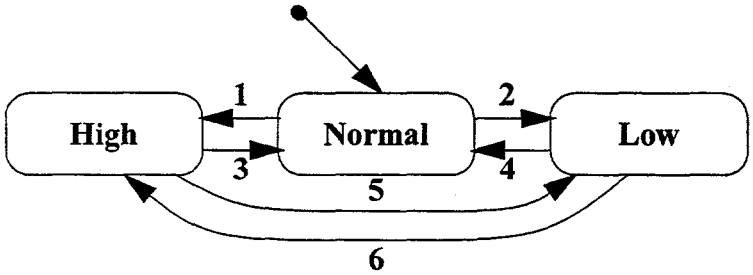


FIGURE 7-2 General Variable Value Zone State Chart

The monitor must be able to detect transitions shown in the state chart and described in Table 7-7. The interaction to inform the supervisor of a zone change for a data variable is shown in Table 7-8.

TABLE 7-7. Transitions for Data Variable Value Zone State Chart

#	Old State	Transition Event	New State	Comments
1	Normal	New value > High Deadzone Upper Limit	High	
2	Normal	New value < Low Deadzone Lower Limit	Low	
3	High	New value < High Deadzone Lower Limit and New value > Low Deadzone Lower Limit	Normal	
4	Low	New value > Low Deadzone Upper Limit and New value < High Deadzone Upper Limit	Normal	
5	High	New value < Low Deadzone Lower Limit	Low	
6	Low	New value > High Deadzone Upper Limit	High	

TABLE 7-8. Interaction for Variable Zone Change

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=DataZoneChange (*Variable-Name, Old Zone, New Zone*)	None	Terminated	

All defined alarm limits for data variables should have unique *AlarmId*'s. The monitor must detect when the variable value goes beyond the high or low alarm limits. When detected, these alarms should be communicated to the supervisor with the proper *AlarmId* using the interactions described in Section 5.8, "Alarm States," on page 25.

Two interactions are defined to allow the supervisor to query for the limit parameter values for data variables. The first query is by limit name. The other asks for the whole set of zone limits. These interactions are shown in Table 7-9 and Table 7-10.

TABLE 7-9. Interaction for Query of Variable Zone Limit

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>ReturnVariableZoneLimit (*VariableName, LimitName*) <-*LimitValue*	None	Terminated	*LimitName* is one of HAL, UDUL, UDLL, LDUL, LDLL, or LAL

TABLE 7-10. Interaction for Query of Variable Zone Limit Settings

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	<=ReturnVariableZoneLimitSettings (*VariableName*) <-(*HAL, UDUL, UDLL, LDUL, LDLL, LAL*)	None	Terminated	

The supervisor can set the limit values for the deadzones. This is accomplished with the interaction shown in Table 7-11.

TABLE 7-11. Interaction for Setting Variable Deadzone Limits

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>SetVariableDeadzoneLimit (*VariableName, LimitName, LimitValue*)	None	Terminated	*LimitName* is one of UDUL, UDLL, LDUL, or LDLL. The supervisor does not have write access to the alarm limits, HAL and LAL.

7.4.2 Custom Variable Change and Read Monitoring

The equipment may have custom monitoring capabilities that employ parameters that can be modified by the supervisor. Although the function of the custom monitor is equipment specific, the interactions for querying and setting the monitor parameters can be generic. Two types of custom variable monitors are accommodated: monitors invoked whenever the value changes and monitors invoked whenever the value is read. The supervisor should be able to query values of monitor parameters, and set the values of monitor parameters for which it has write permission. The data variable description table for the equipment as discussed in Section 9.12 provides a list of the custom monitor parameters.

The interactions for querying the current values of the custom monitor parameters are shown in Table 7-12 and Table 7-13. The interactions for setting custom monitor parameters are shown in Table 7-14 and Table 7-15.

TABLE 7-12. Interaction for Getting a Change Monitor Parameter

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>ReturnChangeMonitorParm (*VariableName, ParmName*) <-*ParmValue*	None	Terminated	

TABLE 7-13. Interaction for Getting a Read Monitor Parameter

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>ReturnReadMonitorParm (*VariableName, ParmName*) <-*ParmValue*	None	Terminated	

TABLE 7-14. Interaction for Setting a Change Monitor Parameter

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>SetChangeMonitorParm (*VariableName, ParmName, ParmValue*)	None	Terminated	

TABLE 7-15. Interaction for Setting a Read Monitor Parameter

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>SetReadMonitorParm (*VariableName, ParmName, ParmValue*)	None	Terminated	

7.5 Control of Event Notification and Customized Reports

Some equipment specific events such as intermediate steps in an operation or events intended for logging during trouble shooting may not normally be of interest to the supervisor. The interactions shown in Table 7-16 and Table 7-17 provide a means for enabling and disabling the reporting of particular events to the supervisor.

TABLE 7-16. Interaction for Enabling Event Report

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>EnableEventReport (*EventId*)	None	Terminated	

TABLE 7-17. Interaction for Disabling Event Report

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>DisableEventReport (*EventId*)	None	Terminated	

The supervisor may require information to be transmitted with the event report beyond that contained in the event arguments. For example, record of assembly and QA/QC may require the values of equipment settings and data variables at the start and completion of certain operations. The following interactions provide a means for managing custom reports to be transmitted with the standard event reports. The custom report is defined by a report name with list of equipment variables whose values are to be included in custom report. The interactions for defining a report and deleting a report are shown in Table 7-18 and Table 7-19

TABLE 7-18. Interaction for Defining Custom Report

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>DefineCustomReport (*ReportId, VariableName List*)	None	Terminated	

TABLE 7-19. Interaction for Deleting Custom Report

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>DeleteCustomReport (*ReportId*)	None	Terminated	

The interactions for linking and unlinking a custom report to an event are shown in Table 7-20 and Table 7-21. Finally, the interactions for enabling and disabling are shown in Table 7-22 and Table 7-23.

TABLE 7-20. Interaction for Linking Custom Report to Event

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>LinkCustomReportToEvent (*ReportId, EventId*)	None	Terminated	

TABLE 7-21. Interaction for Unlinking Custom Report to Event

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>UnlinkCustomReportToEvent (*ReportId, EventId*)	None	Terminated	

TABLE 7-22. Interaction for Enabling Custom Report

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>EnableCustomReport (*ReportId*)	None	Terminated	

TABLE 7-23. Interaction for Disabling Custom Report

#	Message Exchange	Old Inter. State	New Inter. State	Comment
1	=>DisableCustomReport (*ReportId*)	None	Terminated	

7.6 Equipment Information Requests

Machine Contents at Start-Up.

8 Logging Interactions

8.1 Overview

The equipment must support logging capabilities to ensure keeping of proper records of execution and provide diagnostic records for trouble shooting. These have not yet been defined.

9 Characteristics Description

9.1 Overview

The characteristics description of the equipment must describe the equipment to the supervisory control system with enough detail for the determination of whether the equipment can perform a desired process step with acceptable quality and schedule. Furthermore, the description must provide the information required for the interactions to configure the

equipment, to move material into the equipment prior to the performing the operation step, and to move material out of the equipment once the process step is completed. The broad areas of information covered are operation capabilities, configurations, material movement (See Section 6, "Material Movement Interactions," on page 28.), equipment variables and settings. The characteristics description consists of a set of tables.

9.2 Operation Table

The operation table lists the process operations that can be performed by the equipment. For each operation, all information required to describe process step capability quality, duration, material flows and configuration needs is provided. Some equipment may process products individually while other equipment may process product in streams. These characteristics must be described in the operation table.

operation table entry. operation Id, class, verb, parameter list, input list, output list, supply list, waste list, setup duration information, execution duration information, required configuration Id, configuration parameter specification, maximum batch size, maximum work in progress.

input. item class, quantity, entry port

output. item class, quantity, exit port

parameter . name, description, type, range information

supply. supply class, quantity

waste. waste class, quantity

9.3 Configuration Table

Some classes of equipment must be properly configured to perform specific operations on particular types of objects. The configuration table for the equipment allows the supervisor to get the equipment into the proper configuration for desired operations. A table of the allowable configurations of the equipment. For each configuration, parameters, and equipment settings are specified.

configuration table entry. configuration Id, description, parameter list, tool requirements

parameter. name, description, type, range information

tool. tool Id, entry/exit port

9.4 Supply Table

The supply table lists the supply types that are required by the equipment to perform its operations. For each supply type, the storage capacity, units and material port through which the supply inters and exits the equipment are listed.

Supply table entry. supply Id (data variable name), supply class, units, entry/exit port, capacity

9.5 Waste Table

The waste table lists the waste types that are generated by the equipment. For each waste type, the waste capacity, units and material port through which the waste exits the equipment are listed.

Waste table entry. waste Id (data variable name), waste class, units, entry/exit port, capacity

9.6 Tool Table

Tools cycle though the equipment. Some are generic and are used for many products. While others are specific to particular products. The specification of tool may be through class of operation to be performed or may be called out through the Process Description Language Program to be executed for an operation.

tool table entry. tool Id (data variable name), tool class, entry/exit port

9.7 Material Port Table

The material port table lists the material ports through which product, supplies and waste enter and leave the equipment. For each material port, the classes of material that enter and exit through the port are listed along with the physical characteristics of the port such as dimensions and location.

material port table entry. port Id, port location and geometry, port index list

port location and geometry. Describes the location of the port relative to some fiducial on the equipment; gives geometry information required to plan paths into and out of the port with the classes of material that can be handled by the port.

port index entry. index Id, location and geometry relative to port, material classes

9.8 Shared Reservoir Requirements Table

The shared reservoir table lists the material reservoirs (usually for fluids) that are required by the equipment as an intermittent source or sink of material. For each reservoir, the class of material and method of transport are listed.

shared reservoir table entry. reservoir Id, class of material, units, type (input from, output to)

9.9 Continuous Fluid Requirements Table

The continuous fluid table lists the continuously delivered fluids that are required to service the equipment. For each fluid, the class of material and method of transport are listed along with normal flow requirements.

continuous fluid table entry. fluid Id, class of material, method of transport, flow units, normal flow requirements

9.10 Alarm Table

The alarm table lists the alarms that can occur on the equipment. For each alarm, a category and severity must be specified together with a description of the condition.

alarm table entry. alarm Id, severity, category, description of alarm condition, list of handling requirements

9.11 Standard State Variable Initialization Table

The standard variable initialization table lists the initialization values for standard equipment state variables. The standard state variables and their possible initial values are:

Remote Control State . Local or Remote.

Communication Loss Pause . On or Off

Full Spool Pause. On or Off

9.12 Equipment Specific Data Variables

The equipment specific data variable table lists the equipment specific data variables. For each variable, a description, units and range are given, together with monitoring specification and parameters.

data variable table entry. data Id, description, type, units, range, zone monitoring flag, initial zone limit values (HAL, UDUL, UDLL, LDUL, LDLL, LAL), custom change

monitor flag, change monitor parameter list, custom read monitor, read monitor parameter list.

monitor parameter. parameter Id, description, type, range, initial value, external write flag

9.13 Equipment Specific Setting Variables

The equipment specific setting variable table lists the equipment specific setting variables. For each variable, a description, initial values and valid range are specified

setting variable table entry. setting Id, description, type, units, range, initial value, custom change monitor flag, change monitor parameter list, custom read monitor, read monitor parameter list.

monitor parameter. Parameter Id, description, type, range, initial value, external write flag

9.14 Equipment Specific Event Table

The equipment may generate events in addition to the events associated with the standard interaction. For example, events may be defined for indicating completion of intermediate process steps or for tracing operations during troubleshooting. The Equipment Specific Event Table describes these events.

equipment specific event table entry. eventId, category, argument list, description, initial report enable flag.

argument. argName, type, range

10 References

1 Griesmeyer, J. M., Urenda, T. D., Pacetti, R. M, Ferguson, J. J., "A Standard Control System for Modular Automation of Chemistry", *Laboratory Robotics and Automation*, 6 (1994) 79-84.

2 Salit, M., Guenther, F. R., Kramer, G. W., Griesmeyer, J. M., "Integrating Automated Systems With Modular Architecture", *Analytical Chemistry*, 66 (1994) 361-367.

3 Urenda, T. D., Griesmeyer, J. M., Contaminant Analysis Automation SLM INterface Requirements Specification, SAND-XXX, Draft 1994.

4 Semiconductor and Equipment International, *SEMI International Standards 1994: Equipment Automation/Software Vol. 2*, Mountain View, CA, 1994.

5 Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8 (1987) 231-274.

Equipment Description Grammar

J. Michael Griesmeyer[1]

1. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Matrin Company, for the United States Department of Energy under Contract DC-ACO4-94AL85000.

Contents

Equipment Description Grammar

J. Michael Griesmeyer[1]

1 Characteristics Preliminaries

This is intended for input into lex and yacc to parse out the grammar. We start with some production rules lifted primarily from C to allow declaration and possible initialization of various quantities that are needed by the grammar.

1.1 Declarations

There are three types of declaration lists. The plain declaration list can declare global or scoped entities that are used by other objects and it can be used to forward declare complex objects. The key value declaration list is used to declare variables that are used as setting or data variables in the headers for complex objects. The values can by accessed by naming the associated keyword. Finally, there is the parameter list which is used to declare the order dependent list of arguments for an operation. The identifiers in the declaration of key values and parameters cannot be scoped because they are always used to define the symbol table for the object being defined. When referencing the parameter or keyword to extract the value, scoping is allowed to indicate the intended object from which the key value is to be obtained. Similarly, scoping can be used to specify the key for which a value is to be set.

1.1.1 declaration-list

For simplicity, lets only declare one thing for each declaration specifier.

declaration:

declaration-specifier qualified-declarator-list ;

declaration-specifier init-declarator (declaration-qualifier-list)

declaration-list:

declaration

declaration-list declaration

declaration-qualifier-list:

declaration-qualifier

declaration-list, declaration-qualifier

---

1. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Matrin Company, for the United States Department of Energy under Contract DC-ACO4-94AL85000.

1.1.2 key-value-declaration-list

Key value declarations in object headers build tables of context keyword for the objects. Access to the keyword values is through qualified identifiers described below. The key value declaration allows for sub-keywords to be defined in a recursive manner.

key-value-declaration:

superKey(???? in grammar) identifier (key-value-declaration-list)

declaration-specifier indexed-identifier (declaration-qualifier-list)

declaration-specifier indexed-identifier = initializer (declaration-qualifier-list)

key-value-declaration-list:

key-value-declaration

key-value-declaration-list, key-value-declaration

The qualified identifier described in Section 1.3.2 can be used to reference any keyword or sub-keyword of an object.

1.1.3 parameter-declaration-list

The declarator for a parameter must be an unscoped identifier. If it is indexed, the index term indicates the size of the array.

parameter-declaration-list:

parameter-declaration

parameter-declaration-list, parameter-declaration

parameter-declaration:

declaration-specifier indexed-identifier declaration-qualifier-list

1.1.4 Declaration details

For now we will assume that storage class need not be specified. Automatic storage is used for all invocation items. Static storage is used for all definitions and other declarations.

declaration-specifier:

type-specifier

type-specifier: one of

integer real string workpieceClass processClass agent agentOperation agentMaterialPort agentConfiguration agentConsumable agentAlarm agentEvent typedef-name

Maybe later we will allow pointers.

declarator:

scoped-identifier

scoped-identifier:

scope-indicator indexed-identifier

scope-indicator scope-specifier indexed-identifier

The identifier in a scope-specifier should reference a script, a process or the problem scope. The indexed identifier must be declared before it can be referenced and may refer to any quantity associated with the scope of the qualified identifier. For example, the identifier could be one of the declared parameters or setting variables of the operation invoked by a script step.

scope-indicator

:: /* Problem scope */

up-scope-indicator

up-scope-indicator

\\

The term for the indexed identifier must return an integer.

indexed-identifier:

identifier

indexed-identifier [term]

initializer:

term

{ initializer-list }

initialize-list:

initializer

initializer-list, initializer

1.2 Declaration Qualifiers

The declaration qualifier is used to provide range and access information for header and parameter declarations. Calling arguments are always writable by the caller and therefore the write access need not be specified for parameters unless the execution can change the value for the parameter as supplied by the calling argument. The qualified declarations in a header must specify one of callerWrite, executionWrite or callerAndExecutionWrite.

declaration-qualifier-list:

declaration-qualifier

declaration-qualifier-list declaration-qualifier

declaration-qualifier:

range-specification

zone-model-specification

write-access

unit-specification

range-specification-list:

range-specification

range-specification-list, range-specification

range-specification:

number-range

number-range-choices

string-choices

{ range-specification-list }

number-range:

range-begin term range-end

range-begin term term range-end

range-begin:

(

[

range-end:

)

]

number-range-list:

number-range

number-range-list, number-range

number-range-choices:

(number-range-list)

string-choices:

(term-list)

unit-specification:

string-constant

A zone model specification can only be supplied for qualified declarations that can be written during execution, that is those with write access of either executionWrite or callerAndExecutionWrite. The limit declarations in the zone model specification give default values and write access information for each model limit in the following order:

HighAlarmLimit

HighDeadzoneUpperLimit

HighDeadzoneLowerLimit

LowDeadzoneUpperLimit

LowDeadzoneLowerLimit

LowAlarmLimit

zone-model-specification:

zoneModel limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration

limit-declaration:

term write-access

write-access:

callerWrite

callerWriteRequired

executionWrite

callerAndExecutionWrite

callerRequiredAndExecutionWrite

1.3 Evaluations

The production script provides means to specify variable whose values that must be determined based upon available information. Some bindings can be done at the time the script is read. Other bindings occur during script execution based upon context. Scripts are described as invocations of operations. The script grammar needs a means of referencing the information specified to control the operations and the data generated by the execution of the operations. The production script grammar is not intended to provide means to evaluate arbitrary expressions. Rather, terms are evaluated in context to access available information. Complex calculations, if required, are performed by the operations themselves or user supplied functions. Expression evaluations in the script are restricted to boolean expressions and comparison of term values required for decision making within the script. Terms and qualified identifiers are described here. The expressions using them are described later where they are needed.

1.3.1 Terms

The term is used to access information associated with objects. Implementations of script processors may evaluate as much of the term as possible at the time the script is read in. Generally, however, the complete evaluation of the term depends upon the context at the time it is used during execution of the script.

term-list:

term

term-list, term

term:

qualified-identifier

constant

constant:

integer-constant

string-constant

real-constant

literal-constant *****???? not in grammar

A literal constant is anything between parentheses. Its evaluation does not occur at the time the script is read in, but rather at the time of operation invocation. It thus, provides context dependence. More will be worked out on the use of literals later.

literal-constant:

‘ (...)

1.3.2 Qualified Identifiers

The qualified identifier allows access to fields in the objects that are declared and defined such as operations, scripts, agents and process classes. The key value declarations together with the qualified identifiers permit runtime definition of objects and access to their symbol tables (i.e. object data by reference to keywords). Identifiers without scope imply use of current scope. The qualified identifier is evaluated left to right. Standard fields are defined for some types and the user may define others for selected types. The terms for the indexed qualified identifier and the indexed identifier must be of integer type.

qualified-identifier:

scoped-idicator qualified-identifier-chain

query ****???? not in grammar

context-keyword ****???? not in grammar

qualified-identifier-chain:

qualified-identifier-link

qualified-identifier-link.qualified-identifier-link

qualified-identifier-link:

key-identifier

{ additive-expression }

contextFunction-invocation

contextFunction-invocation:

scoped-identifier (initializer-list)

initializer-list:

initializer

initializer-list, initializer

initializer:

additive-expression

{ initializer-list }

Querys have not yet been defined but probably will look like SQL in terms of terms. To be used in a qualified identifier chain, the query must return an identifier. The terms which are used to build a query can be qualified identifiers themselves.

Context keywords retrieve information about the object referenced by the qualified identifier to which they are appended. For example, if the qualified identifier refers to an script step, the agent keyword refers to the agent selected to perform the invoked operation. Similarly, the operation keyword would refer to the operation invoked by the script step. Obviously, each context keyword is valid only in selected contexts. Eventually, for each type specifier there will be a set of context keywords. Those context keywords that return an object identifier can be followed by additional context keywords or indexed identifiers to extend the qualified identifier. Keywords that refer to object variables are defined in the headers for the object using a key-value-declaration-list. The parameter, input workpiece, and output workpiece lists are also added to the keyword table of operations.

The following are examples of extended qualified identifiers.

machining::makePocket.width refers to the "width" argument for an invocation of the makePocket operation in the machining process class.

makePocket.width.valueRange refers to the acceptable range for the width argument in the makePocket operation of the current process.

script.script refers to the superscript of the script to which the current operation belongs.

script.agent refers to the agent of the script to which the current operation belongs.

agent.makePocket.width.valueRange refers to the acceptable range for the width argument in the makePocket operation of the agent selected for the current operation.

Mill532J.makePocket.width.valueRange refers to the acceptable range for the width argument in the makePocket operation of the agent, Mill532J.

The context keywords have not all been defined. A partial list is given in the following rule.

context-keyword:

operation
valueRange
defaultValue
HighAlarmLimit
HighDeadzoneUpperLimit
HighDeadzoneLowerLimit
LowDeadzoneUpperLimit

LowDeadzoneLowerLimit
LowAlarmLimit

2 Domain Declarations

The domain declaration list is the start of the parser.

domain-declaration:

processClass-declaration
agentClass-definition
agentOperation-definition
agentPDL-definition
agentMaterialPort-definition
agentConfiguration-definition
agentConsumable-definition
agentWaste-definition
agentAlarm-definition
agentEvent-definition
workpieceClass-definition
declaration

domain-declaration-list:

domain-declaration
domain-declaration-list, domain-declaration

2.1 The Process Class

The process classes for which the agent can preform operations must be declared.

processClass-definition:

processClass identifier (process-header) { processClass-initializer-list }

process-header:

/* A null process header is acceptable */
superProcess identifier
superProcess identifier, key-value-declaration-list
key-value-declaration-list

processClass-initializer-list:

/* A null processClass-initializer-list is acceptable */
domain-declaration-list

2.2 The Agent Class

The agent class definition need not be complete because the definitions in the agent intializers can be incrementally added using scoped identified

agentClass-definition:

agentClass identifier (agent-header)

agentClass identifier (agent-header), { agentClass-initializer-list }

agent-header:

/* A null process header is acceptable */

agentSuperclass identifier

agentSuperclass identifier key-value-declaration-list

key-value-declaration-list

agentClass-initializer-list:

agentClass-initializer

agentClass-initializer-list, agentClass-initializer

agentClass-initializer:

agentPDL-definition

agentOperation-definition

agentMaterialPort-definition

agentConfiguration-definition

agentConsumable-definition

agentWaste-definition

agentAlarm-definition

agentEvent-definition

The identifier for an agent declaration must not be scoped.

agent-declaration:

agentClass-definition indexed-identifier

scoped-identifier indexed-identifier /* Agent class previously defined */

2.3 The Workpiece Class

Only workpiece class names must be known by the script interpreter. The actual characteristics of the workpieces must be available at script execution through the workpiece class identifier and the workpiece identifier. We do not yet have a real definition for a workpiece class, but reserve the right to develop one later. In the meantime, the workpiece class definition below is just a declaration. The workpiece class identifier can be scoped to a particular process class or script.

workpieceClass-definition:

workpieceClass scoped-identifier (workpieceClass-header)

workpieceClass-header:

superWorkpieceClass scoped-identifier, key-value-declaration-list

superWorkpieceClass scoped-identifier

key-value-declaration-list

workpiece-declaration-list:

workpiece-declaration

workpiece-declaration-list, workpiece-declaration

The identifier for a workpiece declaration must not be scoped. The scope will always be that of the calling operation invocation and the workpiece identifier must be unique within the workpiece declaration list. However, for an operation invocation, an identifier in the workpiece input declaration can be the same as an identifier in the output declaration if the associatedworkpiece is not transformed by the operation. For example, a special fixture may be used for an operation but not be modified by the operation. In this case the workpiece is part of the facility and its identifier in the invocation can be scoped accordingly.

workpiece-declaration:

workpieceClass-definition indexed-identifier

scoped-identifier indexed-identifier /* Workpiece class previously defined */

2.4 The Agent Operation Prototype

The operation header declares the variables available to the execution of the operation. The parameter list declares the calling arguments of the operation. The declarators for the header variables, parameters, inputs and outputs are in the same scope and must be unique. That is, a header variable and a parameter cannot have the same name. Furthermore, except for the special case of unmodified workpieces as mentioned above, scope must not be supplied because the scope to be used is that of the invocation. In addition, the declarators for parameters, inputs and outputs must be simple identifiers without initializers because the invocation requires their specification. Defaults are not allowed. The first workpiece class list declares the classes of the expected input workpieces for the operation. The second workpiece class list declares the classes of the expected outputs of the operation. The calling setting variables, arguments, inputs and outputs of the operation invocation must be related to the declared header variables, parameters, inputs and outputs in the agent operation prototype. The agent workpiece declaration list must contain all of the workpiece input and output declarations that are declared for the process operation which it implements. It may have additional inputs and outputs that are peculiar to the manner in which it implements the operation.

operation-prototype:

(operation-header) (parameter-declaration-list) (workpieceClass-spec-list) (workpieceClass-spec-list)

operation-header:

/* A null operation header is acceptable */

key-value-declaration-list

2.5 The Agent Operation

The scoped identifier for the elemental operation indicates the agent and name of the operation. This identifier must be unique for the agent. The next scoped identifier indicates the process class and operation which the agent operation corresponds. The agent may have more than one operation that implements a particular process operation but with different acceptable ranges for the header and call argument variables. The configuration specification indicates how the agent must be configured to perform the operation.

agentOperation-definition:

agentOperation scoped-identifier scoped-identifier operation-prototype (operation-qualification-list)

operation-qualification-list:

operation-qualification

operation-qualification-list, operation-qualification

operation-qualification:

configuration-spec

duration-spec

consumption-spec

waste-generation-spec

configuration-spec:

configuration parameter-specification-list

2.6 The Agent PDL

The definition of a PDL supported by the agent lists the features and the acceptable ranges for variables associated with each feature.

PDL-definition:

PDL identifier, (PDL-header) ;

PDL-header:

processClass identifier (PDL-feature-declaration-list)

PDL-feature-declaration:

identifier (key-value-declaration-list)

PDL-feature-declaration-list:

PDL-feature-declaration

PDL-feature-declaration-list, PDL-feature-declaration

2.7 The Agent Material Port

The material port defines a physical port on the equipment that material (part, waste, supply, etc.) can enter or leave the equipment by. Defined are the classes of material along with the physical characteristics of the port such as dimensions and location. The location of the port if relative to some fiducial on the equipment and the geometry is that which will be required to plan paths into and out of the port. Geometry will be defined at a later date and is currently a place holder.

agentMaterial-definition:

agentMaterial port-identifier[1], (port-location), (port-geometry), (port-index-list);

port-location:

x, y, x, r, p, w[2]

port-geometry:

TBA

port-index-list:

port-index-declaration

port-index-list, port-index-declaration

port-index-declaration:

(index-identifier[3], (port-location), (port-geometry), (material-classes))

material-classes:

material-class[1]

material-classes, material-class

2.8 The Agent Configuration

The agent configuration defines a particular configuration that the equipment can be configured to. Agent operations refer to these configurations to identify the configuration or configurations that the equipment can be in and perform the operation.

agentConfiguration-definition:

agentConfiguration configuration-identifier[1], description[1], (parameter-declaration-list), (tool-requirements-list);

tool-requirements-list:

tool-identifier[1]

tool-requirements-list, tool-identifier

1. aString
2. Real
3. Integer

2.9 The Agent Consumable

The ...

agentConsumable-definition:

agentConsumable consumable-identifier[1], supply-class[1], capacity[2], entry-port-identifier, exit-port-identifier;

2.10 The Agent Waste

The ...

agentWaste-definition:

agentWaste port-identifier waste-identifier[1], waste-class[1], capacity[2], entry-port-identifier, exit-port-identifier;

2.11 The Agent Alarm

The ...

agentAlarm-definition:

agentAlarm alarm-identifier[1], severity[1], category[1], description, (handling-requirements-list);

handling-requirements-list:

handling-requirement

handling-requirements-list, handling-requirement

handling-requirement:

description[1]

2.12 The Agent Event

The ...

agentEvent-definition:

agentEvent event-identifier[1], category[1], description[1], initial-report-enable-flag[1], (parameter-declaration-list);

2.13 The Agent Tool

The ...

---

1. aString
2. real

agentTooldefinition:

agentTool tool-identifier[1], entry-port-identifier, exit-port-identifier;

2.14 The Agent Shared Reservoir

The ...

agentSharedReservoir-definition:

agentSharedReservoir shared-identifier[1], material-class[1], units[1], type;

type:

input | output

2.15 The Agent Continuous Fluid

The ...

agentContinuousFluid-definition:

agentContinuousFluid waste-identifier[1], material-class[1], method-of-transport ????, flow-units[1], flow-requirements;

Appendix 1 List of Declarations

declaration: 1
declaration-specifier qualified-declarator-list ; 1
declaration-specifier init-declarator (declaration-qualifier-list) 1
declaration-list: 1
declaration 1
declaration-list declaration 1
declaration-qualifier-list: 1
declaration-qualifier 1
declaration-list, declaration-qualifier 1
key-value-declaration: 2
superKey(???? in grammar) identifier (key-value-declaration-list) 2
declaration-specifier indexed-identifier (declaration-qualifier-list) 2
declaration-specifier indexed-identifier = initializer (declaration-qualifier-list) 2
key-value-declaration-list: 2
key-value-declaration 2
key-value-declaration-list, key-value-declaration 2
parameter-declaration-list: 2
parameter-declaration 2
parameter-declaration-list, parameter-declaration 2
parameter-declaration: 2

1. aString

declaration-specifier indexed-identifier declaration-qualifier-list 2

declaration-specifier: 2

type-specifier 2

type-specifier: one of 2

integer real string workpieceClass processClass agent agentOperation agentMaterialPort agentConfiguration agentConsumable agentAlarm agentEvent typedef-name 2

declarator: 2

scoped-identifier 2

scoped-identifier: 3

scope-indicator indexed-identifier 3

scope-indicator scope-specifier indexed-identifier 3

scope-indicator 3

:: /* Problem scope */ 3

up-scope-indicator 3

up-scope-indicator 3

\\ 3

indexed-identifier: 3

identifier 3

indexed-identifier [term] 3

initializer: 3

term 3

{ initializer-list } 3

initialize-list: 3

initializer 3

initializer-list, initializer 3

declaration-qualifier-list: 3

declaration-qualifier 3

declaration-qualifier-list declaration-qualifier 3

declaration-qualifier: 3

range-specification 3

zone-model-specification 3

write-access 3

unit-specification 4

range-specification-list: 4

range-specification 4

range-specification-list, range-specification 4

range-specification: 4

number-range 4

number-range-choices 4

string-choices 4

{ range-specification-list } 4

number-range: 4

range-begin *term range-end 4*

range-begin term term range-end 4

range-begin: 4

(4

[4

range-end: 4
) 4
] 4
number-range-list: 4
number-range 4
number-range-list, number-range 4
number-range-choices: 4
(number-range-list) 4
string-choices: 4
(term-list) 4
unit-specification: 4
string-constant 4
HighAlarmLimit 4
HighDeadzoneUpperLimit 4
HighDeadzoneLowerLimit 4
LowDeadzoneUpperLimit 4
LowDeadzoneLowerLimit 4
LowAlarmLimit 5
zone-model-specification: 5
zoneModel limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration 5
limit-declaration: 5
term write-access 5
write-access: 5
callerWrite 5
callerWriteRequired 5
executionWrite 5
callerAndExecutionWrite 5
callerRequiredAndExecutionWrite 5
term-list: 5
term 5
term-list, term 5
term: 5
qualified-identifier 5
constant 5
constant: 6
integer-constant 6
string-constant 6
real-constant 6
literal-constant *****????? not in grammar 6
literal-constant: 6
‘ (...) 6
qualified-identifier: 6
scoped-idicator qualified-identifier-chain 6
query ****???? not in grammar 6
context-keyword ****???? not in grammar 6
qualified-identifier-chain: 6

qualified-identifier-link 6
qualified-identifier-link.qualified-identifier-link 6
qualified-identifier-link: 6
key-identifier 6
{ additive-expression } 6
contextFunction-invocation 6
contextFunction-invocation: 6
scoped-identifier (initializer-list) 6
initializer-list: 6
initializer 6
initializer-list, initializer 6
initializer: 6
additive-expression 6
{ initializer-list } 7
context-keyword: 7
operation 7
valueRange 7
defaultValue 7
HighAlarmLimit 7
HighDeadzoneUpperLimit 7
HighDeadzoneLowerLimit 7
LowDeadzoneUpperLimit 7
LowDeadzoneLowerLimit 8
LowAlarmLimit 8
domain-declaration: 8
processClass-declaration 8
agentClass-definition 8
agentOperation-definition 8
agentPDL-definition 8
agentMaterialPort-definition 8
agentConfiguration-definition 8
agentConsumable-definition 8
agentWaste-definition 8
agentAlarm-definition 8
agentEvent-definition 8
workpieceClass-definition 8
declaration 8
domain-declaration-list: 8
domain-declaration 8
domain-declaration-list, domain-declaration 8
processClass-definition: 8
processClass identifier (process-header) { processClass-initializer-list } 8
process-header: 8
/* A null process header is acceptable */ 8
superProcess identifier 8
superProcess identifier, key-value-declaration-list 8
key-value-declaration-list 8

processClass-initializer-list: 8
/* A null processClass-initializer-list is acceptable */ 8
domain-declaration-list 8
agentClass-definition: 9
agentClass identifier (agent-header) 9
agentClass identifier (agent-header), { agentClass-initializer-list } 9
agent-header: 9
/* A null process header is acceptable */ 9
agentSuperclass identifier 9
agentSuperclass identifier key-value-declaration-list 9
key-value-declaration-list 9
agentClass-initializer-list: 9
agentClass-initializer 9
agentClass-initializer-list, agentClass-initializer 9
agentClass-initializer: 9
agentPDL-definition 9
agentOperation-definition 9
agentMaterialPort-definition 9
agentConfiguration-definition 9
agentConsumable-definition 9
agentWaste-definition 9
agentAlarm-definition 9
agentEvent-definition 9
9
agent-declaration: 9
agentClass-definition indexed-identifier 9
scoped-identifier indexed-identifier /* Agent class previously defined */ 9
workpieceClass-definition: 10
workpieceClass scoped-identifier (workpieceClass-header) 10
workpieceClass-header: 10
superWorkpieceClass scoped-identifier, key-value-declaration-list 10
superWorkpieceClass scoped-identifier 10
key-value-declaration-list 10
workpiece-declaration-list: 10
workpiece-declaration 10
workpiece-declaration-list, workpiece-declaration 10
workpiece-declaration: 10
workpieceClass-definition indexed-identifier 10
scoped-identifier indexed-identifier /* Workpiece class previously defined */ 10
operation-prototype: 10
(operation-header) (parameter-declaration-list) (workpieceClass-spec-list) (
workpieceClass-spec-list) 10
operation-header: 11
/* A null operation header is acceptable */ 11
key-value-declaration-list 11
agentOperation-definition: 11
agentOperation scoped-identifier scoped-identifier operation-prototype (operation-

qualification-list) 11
operation-qualification-list: 11
operation-qualification 11
operation-qualification-list, operation-qualification 11
operation-qualification: 11
configuration-spec 11
duration-spec 11
consumption-spec 11
waste-generation-spec 11
configuration-spec: 11
configuration parameter-specification-list 11
PDL-definition: 11
PDL identifier, (PDL-header) ; 11
PDL-header: 11
processClass identifier (PDL-feature-declaration-list) 11
PDL-feature-declaration: 11
identifier (key-value-declaration-list) 11
PDL-feature-declaration-list: 11
PDL-feature-declaration 11
PDL-feature-declaration-list, PDL-feature-declaration 11
agentMaterial-definition: 12
agentMaterial port-identifier, (port-location), (port-geometry), (port-index-list); 12
port-location: 12
x, y, x, r, p, w 12
port-geometry: 12
TBA 12
port-index-list: 12
port-index-declaration 12
port-index-list, port-index-declaration 12
port-index-declaration: 12
(index-identifier, (port-location), (port-geometry), (material-classes)) 12
material-classes: 12
material-class1 12
material-classes, material-class 12
agentConfiguration-definition: 12
agentConfiguration configuration-identifier1, description1, (parameter-declaration-list), (tool-requirements-list); 12
tool-requirements-list: 12
tool-identifier1 12
tool-requirements-list, tool-identifier 12
agentConsumable-definition: 13
agentConsumable consumable-identifier, supply-class1, capacity, entry-port-identifier, exit-port-identifier; 13
agentWaste-definition: 13
agentWaste port-identifier waste-identifier1, waste-class1, capacity2, entry-port-identifier, exit-port-identifier; 13
agentAlarm-definition: 13

agentAlarm alarm-identifier1, severity1, category1, description, (handling-requirements-list); 13

handling-requirements-list: 13

handling-requirement 13

handling-requirements-list, handling-requirement 13

handling-requirement: 13

description1 13

agentEvent-definition: 13

agentEvent event-identifier1, category1, description1, initial-report-enable-flag1, (parameter-declaration-list); 13

agentTooldefinition: 14

agentTool tool-identifier, entry-port-identifier, exit-port-identifier; 14

agentSharedReservoir-definition: 14

agentSharedReservoir shared-identifier1, material-class1, units1, type; 14

type: 14

input | output 14

agentContinuousFluid-definition: 14

agentContinuousFluid waste-identifier1, material-class1, method-of-transport ????, flow-units1, flow-requirements; 14

Production Script Grammar

J. Michael Griesmeyer[1]

1. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Matrin Company, for the United States Department of Energy under Contract DC-ACO4-94AL85000.

Contents

Production Script Grammar

J. Michael Griesmeyer[1]

1 Production Script Preliminaries

This is intended for input into lex and yacc to parse out the grammar. We start with some production rules lifted primarily from C to allow declaration and possible initialization of various quantities that are needed by the grammar.

1.1 Declarations

There are three types of declaration lists. The plain declaration can declare global or scoped entities that are used by other objects, and it can be used to forward declare complex objects. The key value declaration list is used to declare variables that are used as setting or data variables in the headers for complex objects. The values can by accessed by naming the associated keyword. Finally, there is the parameter list which is used to declare the order dependent list of arguments for an operation. The identifiers in the declaration of key values and parameters cannot be scoped because they are always used to define the namespace for the object being defined. When referencing the parameter or keyword to extract the value, scoping is allowed to indicate the intended object from which the key value is to be obtained. Similarly, scoping can be used to specify the key for which a value is to be set.

1.1.1 declaration

declaration:

declaration-specifier quailfied-declarator-list ;

qualified-declarator:

init-declarator

init-declarator (declaration-qualifier-list)

quailfied-declarator-list:

qualified-declarator

qualified-declarator-list , qualified-declarator

init-declarator:

declarator

---

1. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Matrin Company, for the United States Department of Energy under Contract DC-ACO4-94AL85000.

declarator = initializer

The following are examples of declarations. More complex examples are given when some of the details of declarations are discussed.

```
// Declare some integers and a real. The allowable range for 'midval' is specified.
int bigval, midval (30 100), lowval;
real power;
// Declare a script in the scope of the welding manufacturing process.
script welding::simpleWeldScript;
```

1.1.2 key-value-declaration-list

Key value declarations in object headers build tables of context keyword for the objects. Access to the keyword values is through qualified identifiers described below. The key value declaration allows for sub-keywords to be defined in a recursive manner by using superkey as the declaration specifier.

key-value-declaration:

superkey identifier (key-value-declaration-list)

declaration-specifier identifier (declaration-qualifier-list)

declaration-specifier identifier = initializer (declaration-qualifier-list)

key-value-declaration-list:

/* Null is OK */

key-value-declaration

key-value-declaration-list , key-value-declaration

The qualifed identifier described in Section 1.3.3 can be used to reference any keyword or sub-keyword of an object. The following are examples of key value declaration lists.

```
// Key value declarations can must specify write access and can specify range
// information through the declaration qualifier list. The key value declaration can
// also supply initial/default values.
real realVal (executionWrite), real upperThreshold = 50.67 (callerWrite, [15.9, 200.0))

// The key can be a super keyword with nesting of sub-keywords
superkey qualitySettings (real surfaceFinish (callerWrite, [15.9, 200.0)), real normalTolerance = .005 (callerWrite, [.001, .20)))
```

1.1.3 parameter-declaration-list

The declarator for a parameter must be an unscoped identifier. If it is indexed, the index term indicates the size of the array.

parameter-declaration:

declaration-specifier indexed-identifier

declaration-specifier indexed-identifier (declaration-qualifier-list)

parameter-declaration-list:

/* Null is OK */

parameter-declaration

parameter-declaration-list , parameter-declaration

The following is an example of a parameter declaration list.

```
// Defaults are not allowed for parameters which must be supplied by the caller in order
// Write access need not be specified if only the caller can write to the parameter.
real bigVal ((300.0 400.0], (550.0 700.0)), real smallVal ((3.0 4.0], (5.5 7.0)),
string yesNo ("y", "Y", "N" ,"n")
```

1.1.4 Declaration details

For now we will assume that storage class need not be specified. Automatic storage is used for all invocation items. Static storage is used for all definitions and other declarations.

declaration-specifier:

type-specifier

type-specifier: one of

integer real string script scriptStep scriptNode scriptGenerator elementalOperation processProgram invocationGenerator workpieceClass processClass PDL contextFunction typedef-name

Maybe later we will allow pointers.

declarator:

scoped-identifier

scoped-identifier:

indexed-identifier

scope-specifier indexed-identifier

scoped-identifier-list

scoped-identifier

scoped-identifier-list, scoped-identifier

The identifier in a scope specifier should reference a script, a process or the problem scope. The indexed identifier must be declared before it can be referenced and may refer to any quantity associated with the scope of the qualified identifier. For example, the identifier could be one of the declared parameters or setting variables of the operation invoked by a script step.

scope-specifier:

:: /* Problem scope */

indexed-identifier ::

The term for the indexed identifier must return an integer.

indexed-identifier:

identifier

indexed-identifier [term]

initializer:

term

{ initializer-list }

initialize-list:

/* Null is OK */

initializer

initializer-list , initializer

1.2 Declaration Qualifiers

The declaration qualifier is used to provide range and access information for header and parameter declarations. Calling arguments are always writable by the caller and therefore the write access need not be specified for parameters unless the execution can change the value for the parameter as supplied by the calling argument. The qualifed declarations in a header must specify the write access for the variable.

declaration-qualifier:

unit-specification

write-access

range-specification

zone-model-specification

declaration-qualifier-list:

declaration-qualifier

declaration-qualifier-list , declaration-qualifier

1.2.1 Write Access

Each variable in a keyword or or parameter list must indicate who can write to it. The calling parameters for an operation are defaulted to callerWrite access; meaning that the invocation of an operation can (and must) supply values for the parameters. However, if the execution can change the value of a variable associated with a calling parameter, the parameter must be declared as allowing execution write. The following rule indicates the valid write access specifications that can be supplied in the declaration qualification list.

write-access:

callerWrite

callerWriteRequired

executionWrite

callerAndExecutionWrite

callerRequiredAndExecutionWrite

1.2.2 Range Specifications

Valid ranges can be specified in the declaration qualifiers for variables in headers and parameter lists.

range-specification:

number-range

number-range-choices

string-choices

{ range-specification-list }

range-specification-list:

range-specification

range-specification-list , range-specification

number-range:

[term]

[term term]

[term term)

(term term]

(term term)

number-range-list:

number-range

number-range-list , number-range

number-range-choices:

(number-range-list)

The following are examples of range specifications.

```
// Single value range
[50]

// Bounded ranges
(5 9]
[45.0 46.0]

// Bounded range choices
([45.0 70.0], [250 300], [1000 1200])
```

string-choices:

(term-list)

The terms in the term list must resolve to srings. The following is an example of string choices.

```
// Acceptable values of a yes or no string
("y","Y","yes","Yes","n","N","no","No")
```

1.2.3 Zone Model

A zone model can be used for the value of a data variable. A value can be in one of three zones: High, Normal or Low as shown in Figure 1-1. The boundaries between zones are thresholds. To prevent chatter in the reporting of transitions from one zone to another, deadzones are defined by the overlap of boundaries between zones. Transitions occur only when the variable value crosses the far boundary of the deadzone from its current zone. Thus, to transition from Normal to High the variable value must become greater than the High DeadZone Upper Limit, and to transition from High to Normal the value must become lower than the High DeadZone Lower Limit. If chatter is not a concern, the upper and lower limits of the deadzones can be the same.

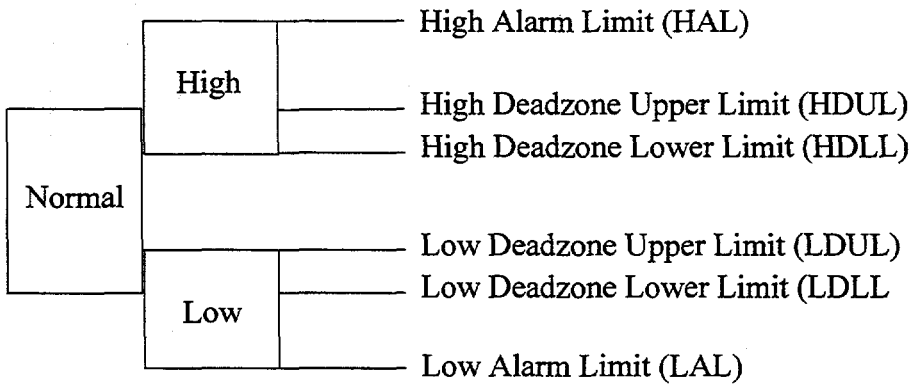


FIGURE 1-1 Zones and Threshold Monitoring Limits

Typically, an operation invocation will need to set the boundaries of the normal range for data variables. However, the alarm limits can be characteristic of equipment or operations and serve to protect the equipment and surrounding objects (other equipment and personnel). These limits would normally not be set by the operation invocation.

A zone model specification can only be supplied for qualified declarations that can be written during execution, that is those with write access of executionWrite, callerAndExecutionWrite, or callerRequiredAndExecutionWrite. The limit declarations in the zone model specification give default values and write access information for each model limit in the following order:

HighAlarmLimit
HighDeadzoneUpperLimit
HighDeadzoneLowerLimit
LowDeadzoneUpperLimit
LowDeadzoneLowerLimit

LowAlarmLimit

zone-model-specification:

zoneModel limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration

limit-declaration:

term write-access

The following is an example of a zone model specification in which the caller can set the deadzone limits but not the alarm limits for the variable.

zoneModel 45.0 executionWrite 40.0 callerWrite 39.0 callerWrite 10.0 callerWrite 9.0 callerWrite 2.0 executionWrite

1.3 Evaluations

The production script provides means to specify variables whose values must be determined based upon available information. Some bindings can be done at the time the script is read. Other bindings occur during script execution based upon context. Scripts are described as invocations of operations. The script grammar needs a means of referencing the information specified to control the operations and the data generated by the execution of the operations. The production script grammar is not intended to provide means to evaluate arbitrary expressions. Rather, terms are evaluated in context to access available information. Complex calcualtions, if required, are performed by the operations themselves or user supplied functions. Expression evaluations in the script are restricted to boolean expressions and comparison of term values required for decision making within the script. Terms and qualified identifiers are described here. The expressions using them are described later where they are needed.

1.3.1 Terms

The term is used to access information associated with objects. Implementations of script processors may evaluate as much of the term as possible at the time the script is read in. Generally, however, the complete evaluation of the term depends upon the context at the time it is used during execution of the script.

term:

qualified-identifier

constant

term-list

term

term-list, term

constant:

integer-constant

string-constant

real-constant

literal-constant

A literal constant is anything between parentheses. Its evaluation does not occur at the time the script is read in, but rather at the time of operation invocation. It thus, provides context dependence. More will be worked out on the use of literals later.

literal-constant:

‘ (...)

1.3.2 Context functions

Context functions retrieve information about an object. Context fucntions are declared by specifying the return type, the object type to which they are associated, and any required calling parameters.

contextFunction-declaration:

contextFunction scoped-identifier (type-specifier , type-specifier , parameter-declaration-list) ;

The first type-specifier is the return type of the function and the second type-specifier is the object type for the function.

The invocation of the context function only suppiles the required parameters because the object for the function is determined from the context of the invocation which is always within a qualified identifier.

context-function-invocation:

scoped-identifier (initializer-list)

1.3.3 Qualified Identifiers

The qualified identifier allows access to fields in the objects that are declared and defined such as operations, scripts, agents and process classes. The key value declarations together with the qualified identifiers permit runtime definition of objects and access to their symbol tables (i.e. object data by refererence to keywords). Identifiers without scope imply use of current scope. The qualified identifer is evaluated left to right. Standard namespace keywords are defined for some types and the user may define others for selected types. The terms for the index of an indexed identifier must be of integer type.

qualified-identifier:

scoped-identifier

query

context-function-invocation

qualified-identifier . indexed-identifier

qualified-identifier . context-function-invocation

qualified-identifier . (term-expression)

Querys have not yet been defined but probably will look like SQL in terms of terms. To be used in a qualified identifier chain, the query must return an identifier. The terms which are used to bulid a query can be qualified identifiers themselves.

The indexed identifiers in a qualified identifier are keywords in the namespace of the object refered to by the qualified identifier to which it is appended. For example, if the qualified identifier refers to a script step, the agent keyowrd refers to the agent selected to perform the invoked operation. Similarly, the operation keyword would refer to the operation invoked by the script step. Context functions retrieve information about the object referenced by the qualified identifer to which they are appended. Obviously, each context function or keyword is valid only in selected contexts. Eventually, for each type specifier there will be a set of context functions and standard namespace keywords. Those context functions or keywords that return an object identifier can be followed by additional context functions or indexed identifiers to extend the qualified identifier. Keywords refer to object variables that are defined in the headers for the object using a key-value-declaration-list. The parameter, input workpiece, and output workpiece lists are also added to the keyword table for operations with parm, input and output as the respective superkeys.

The following are examples of extended qualified identifers.

machining::makePocket.width

refers to the "width" argument for an invocation of the makePocket operation in the machining process class.

makePocket.width.valueRange

refers to the acceptable range for the width argument in the makePocket operation of the current process.

script.script

refers to the superscript of the script to which the current operation belongs.

script.agent

refers to the agent of the script to which the current operation belongs.

agent.makePocket.width.valueRange

refers to the acceptable range for the width argument in the makePocket operation of the agent selected for the current operation.

Mill532J.makePocket.width.valueRange

refers to the acceptable range for the width argument in the makePocket operation of the agent, Mill532J.

The standard namespace keywords have not all been defined. A partial list follows.

operation

script
agent
input
output
parm
valueRange
HighAlarmLimit
HighDeadzoneUpperLimit
HighDeadzoneLowerLimit
LowDeadzoneUpperLimit
LowDeadzoneLowerLimit
LowAlarmLimit

2 Domain Declarations

The domain declaration is the starting rule for the parser. Process Classes must be declared. Elemental Operations must be declared and defined. Scripts must be declared and defined. Process Description Language programs must be declared and defined. Forward declarations are allowed for each *type-specifier* that requires definition. Thus, just like in C, the declaration may include a definition, but a definition is not required. Only one definition, whether complete or not, may be supplied for a scoped identifier. Multiple declarations for a scoped identifier (a definiton followed by a declaration, or more than a single forward declaration and a single definition) will result in a warning. The *declarator* for a forward declaration should normally be a *scoped-identifier.* Many of the definitions can be incremental in that, once a partial definition has been provided, additional components can be added through scoping of the associated identifiers. For example, elementalOperations can be added to a processClass after it has been initally defined by scoping the elementalOperation identifier to indicate the process class to which it belongs. While forward declarations are allowed, the definition must be supplied before any information other than the identifier of the object is used.

Each domain declaration, whether a definition or a simple declaration, is terminated by a semi-colon.

domain-declaration:

processClass-definition
elementalOperation-definition
processProgram-definition
script-definition
scriptStep-definition
scriptNode-definition

workpieceClass-definition

PDL-definition

contextFunction-declaration:

declaration

domain-declaration-list:

domain-declaration

domain-declaration-list domain-declaration

2.1 The Process Class

Process Class definitions provide manufacturing process domains for scoping of process information associated with a particular process class. For now, the process class identifier must not be scoped and therefore must be unique. The superprocess keyword in the process header is used to indicate that the process class is a subprocess of the named process. The declarations in the process header can be used to hold setting varaiables that control the environment of the process. These must give the caller write access. The subprocess inherits the declarations in the header of the its superclass. They can, however, be overridden by declarations in the subprocess header with the same identifiers as those for the superprocess. The same inheritance approach is used for scripts and elemental operations scoped to a particular process class. The process class initializer list need not be specified with the definition of the process class because the scripts, elemental operations, and process programs are scoped when they are defined. That scoping has the effect of adding them to the appropriate process script, elemental operation, or process program lists.

processClass-definition:

processClass identifier (process-header) { processClass-initializer-list } ;

process-header:

/* A null process header is acceptable */

superProcess identifier

superProcess identifier , key-value-declaration-list

key-value-declaration-list

processClass-initializer-list:

/* Null is OK */

domain-declaration-list

The following are examples of process class definitions.

```
processClass manufacturing (){};
processClass machining  ( superProcess manufacturing ) {};
processClass fixturing ( superProcess machining ) {};
processClass turning (superProcess machining) {};
processClass milling (superProcess machining) {};
processClass weldning  ( superProcess manufacturing ) { processClass GTAweldning  (
superProcess welding )};
```

2.2 The Workpiece Class

Workpiece class names must be known by the script interpreter. The actual charateristics of the workpieces must be available at script execution through the workpiece class identifier and the workpiece identifier. We do not yet have a real definition for a workpiece class, but reserve the right to develop one later. In the meantime, the workpiece class definition below just indicates that header information might be supplied to help define the workpiece. The workpiece class identifier can be scoped to a particular process class or script. The workpiece description should point to the data that describes geometry, materials, composition, etc. This could be an PDES/STEP file, a ProEngineer file, or some other product description.

workpieceClass-definition:

workpieceClass scoped-identifier (workpieceClass-header) ;

workpieceClass-header:

/* Null header is OK */

superWorkpieceClass identifier

superWorkpieceClass identifier , key-value-declaration-list

key-value-declaration-list

The following are examples of workpiece class definitions.

workpieceClass artifact (string geometryFile (callerWrite), real[6][] workpointList (callerWrite),string productionScript (callerWrite));
workpieceClass assembly (superWorkpieceClass artifact, string compositionFile (callerWrite));
workpieceClass fixture (superWorkpieceClass artifact, string fixtureMethod (callerWrite), string machineMountMethod (callerWrite));
workpieceClass rawStock (string geometryFile (callerWrite), string materialSpec (callerWrite));
workpieceClass casting (superWorkpieceClass artifact, string materialSpec (callerWrite));

A workpiece that is introduced to a facility rather than being an output of an operation must be declared before it can be used in an operation invocation. The declaration gives the class, workpiece name, and possibly values for some of the keyword variables

workpiece-declaration:

scoped-identifier indexed-identifier (setting-list);

The identifier for a workpiece declaration must not be scoped. The scope will always be that of the calling operation invocation and the workpeice identifier must be unique within the workpiece declaration list. However, for an operation invocation, an identifier in the workpiece input specification can be the same as an identifier in the output declaration if the associatedworkpiece is not transformed by the operation. For example, a special fixture may be used for an operation but not be modified by the operation. In this case the workpiece is part of the facility and its identifier in the invocation can be scoped accordingly.

The following are examples of workpiece declarations.

```
assembly assembly1 (compositionFile = "assembly1_composition.cmp")
casting casting1 (geometryFile = "casting1.geo", materialSpec = "316ss")
```

The input and output workpiece declarations in operation prototypes must indicate the workpiece class and its name.

io-workpiece-declaration-list:

/* Null is OK */

scoped-identifier identifier

io-workpiece-declaration-list, scoped-identifier identifier

2.3 The Process Description Language

A Process Description Language (PDL) can be defined for a manufacturing process. The definition of the PDL lists the features that can be invoked in a program written in the PDL. The feature declaration lists feature variables. These features are named in the description of a PDL program so that they can be compared against the features supported by agents that might be selected to perform the program.

PDL-definition:

PDL identifier , (PDL-header) ;

PDL-header:

processClass identifier PDL-feature-declaration-list

PDL-feature-declaration:

identifier (key-value-declaration-list)

PDL-feature-declaration-list:

/* Null is OK */

PDL-feature-declaration

PDL-feature-declaration-list , PDL-feature-declaration

The following is an example of a PDL definition.

```
PDL StdMachiningLanguageA, (processClass machining);
```

2.4 The Operation Prototype

The operation header declares the variables avaiable to the execution of the operation. Some are setting variables that can be set by name in the operation invocation. These must give the caller access. Furthermore, those with caller required write access must be supplied in an operation invocation either directly or through default. Other varables hold data for the execution of the operation and can be specified in the record list of an operation invocation. These must have execution write access. The parameter list declares the calling arguments of the operation. The declarators for the header variables, parameters,

inputs and outputs are in the same scope and must be unique. That is, a header variable and a parameter cannot have the same name. Furthermore, scope must not be supplied because the scope to be used is that of the invocation. In addition, the declarators for parameters, inputs and outputs must be simple identifiers without initializers because the invocation requires their specification. Defaults are not allowed. The first workpiece class list declares the classes of the expected input workpieces for the operation. The second workpiece class list declares the classes of the expected outputs of the operation. Any pre-processing required for an operation is considered just another operation and therefore is called out in a preceeding script step. The calling setting variables, arguments, inputs and outputs of the operation invocation must be related to the declared header variables, parameters, inputs and outputs in the operation prototype.

operation-prototype:

(operation-header) , (parameter-declaration-list), (io-workpiece-declaration-list) , (io-workpiece-declaration-list)

operation-header:

key-value-declaration-list

The workpieceClass specifications in the prototype of elemental operations normally will not include classes with complete descriptions. For example, the makePocket operation for machining can be executed on many different workpiece geometries and materials. The invocation of the makePocket operation supplies parameters that locates the feature relative to the origin of the workpiece and provides details of the workpiece geometry.

2.5 The Elemental Operation

The scoped identifier for the elemental operation indicates the process class to which the elemental operation belongs. The identifier must be unique within the process class.

elementalOperation-definition:

elementalOperation scoped-identifier , operation-prototype ;

The following are examples of elemental operation definitions.

```
// scoped operation name
elementalOperation welding:spotWeld,
//Operation prototype
// operation header
    (real[6] pointTolerance (callerWrite) , energyTolerance (callerWrite),
    powerTolerance (callerWrite), real energyDeposition (executionWrite),
    real peakPower (executionWrite), real[6] theWeldedPoint (executionWrite)),
// operation parameter list
    (real[6] weldPoint, real weldArea, real flangeThickness, real energy, real power),
// input list
    (assembly assemblyToWeld),
// output list
    (assembly weldedAssembly);

// Mount Fixture
```

```
elementalOperation fixturing::mountFixture,
//Operation prototype
// operation header
     (real[6] pointTolerance (callerWrite), real[6] locationError (executionWrite)),
// operation parameter list
     (real[6] machineMountPoint, real[6] fixtureMountPoint,
     real[3] matingAxis, real[3] approachOffset),
// input list
     (fixture fixtureToMount),
// output list
     (fixture mountedFixture);
```

2.6 The Process Program

Process programs written in the PDL can be invoked by an operation invocation. The scoped identifer provides the unique name of the process program. The PDL specification names the PDL in which the program is written and provides a description of the program. The description is a list of the PDL features that are invoked. The features are listed by name and the values for the feature variables are given. If a feature is invoked by the program more than once with different values for the variables, the feature should be listed more than once in the program description. Any tools required by the program should be included in the input and output specifications of the operation prototype. If the process program requires a special setup, the setup is specified by the invocation of an operation to perform the setup.

processProgram-definition:

processProgram scoped-identifier , operation-prototype , PDLSpec ;

PDLSpec:

identifier { program-description-list }

identifier { program-description-list } , setup operation-invocation

PDL-feature-spec:

identifier (setting-list)

program-description-list:

/* Null is OK */

PDL-feature-spec

program-description-list , PDL-feature-spec

The folowing are examples of process program definitions.

```
// Define a process program
processProgram makeBulkShapeA,
//Operation prototype
// operation header
     (real surfaceFinish (callerWrite)),
// operation parameter list
     (real[3] scale),
```

```
// input list
    (rawStock inputStock),
// output list
    (artifact bulkShape),
// PDLSpec
    StdMachiningLanguageA {};
```

2.7 The Script

script-definition:

script scoped-identifier , operation-prototype , script-ends , { script-definer-list } ;

script scoped-identifier , operation-prototype , scriptGenerator scoped-identifier ;

The scoped identifier for the script indicates the process class to which the script belongs. If the script involves more than one process, problem scope should be specified. This is done by using the scope specifier without an identifier (i.e. the first production rule for *scope-specifier*). The identifier must be unique within the indicated scope.

The script header is intended to declare and intitialize variables that are local to the script. These can be used for settings or results in the scope of the script. The scoped-identifiers in script settings should refer to variables declared in the header for the script or an associated process.

The identifiers in the *script-ends* refer to the Script Starting Decision Node and the Script Terminating Decision Node. The identifiers must refer to script nodes that are defined in the script definer list.

scriptEnds:

(identifier identifier) /* Starting Node and Terminating Node */

Because of the cross linking between steps and nodes, the short forms for many of the productions can be used. Other productions generate the required cross-linking. Care should be taken to only give a definition once for each scoped-identifier.

script-definer-list:

/* Null is OK */

domain-declaration-list

The script definer list must include the starting and ending script nodes as well as at least one script step.

The scripts that are defined with a script generator are filled in at the time the script is invoked. The script generator identifier is the name of a function that is supplied to the script processor. It takes as arguments the resolved information supplied in an operation invocation of the script and returns the name of a file that contains the script step and node definitions to be used for this invocation of the script. This allows for online generation of scripts decompositions by whatever means the user chooses to employ.

Scope of the identifers for domain declarations in the script definer list can be omitted if script scope is desired because the scope for the script being define is implied. Scopes can be nested since scripts can be defined in the definer list of other scripts. Identifiers not found in the scope of the current script are looked for in the scope of the encompassing script. Scoped identifiers must of course return the proper type. All steps and nodes of a script are scoped to the script. Errors will be returned if the specified scopes and names are not consistent.

2.7.1 Script Steps

scriptStep-definition:

scriptStep scoped-identifier , operationInvocation , previous-node-spec , next-node-spec ;

previous-node-spec:

node-spec

next-node-spec:

node-spec

node-spec:

scoped-identifier /* previous/next script step */

scoped-identifier integer-constant /* previous/next node and exit/entry set number */

2.7.2 Script Nodes

scriptNode-definition:

scriptNode scoped-identifier { entrance-set-list } { exit-set-list } ;

entrance-set-list:

/* Null is OK */

(scoped-identifier-list)

entrance-set-list, (scoped-identifier-list)

exit-set-list:

/* Null is OK */

selection-criterion (scoped-identifier-list)

exit-set-list , selection-criterion (scoped-identifier-list)

The selection criterion for an exit step set of a script node is a boolean term, a boolean result of comparing two terms, or the boolean result of logical combination of two criteria. The comparison operators are defined for strings, integers, and doubles. The types to be compared must be comparable. Thus, strings cannot be compared to numbers. If either term is a double, the comparison is evaluated for doubles after any required conversion of the other term from integer to double. Terms that return names are compared as strings.

selection-criterion:

or-criterion

or-criterion:

and-criterion

or-criterion || and-criterion

and-criterion:

compare-criterion

and-criterion && compare-criterion

compare-criterion:

primary-criterion

term compare-operator term

primary-criterion:

term

(selection-criterion)

compare-operator:

==

!=

>

<

<=

>=

Finally, a special script node class needs to be defined that has the effect of providing the script executor options on how to perform a step. In this case, the decision criteria must be structured so that the script executor can select only one of the options (i.e. only one exit step set).

2.8 The Operation Invocation

operationInvocation:

scoped-identifier , (setting-list) , (initializer-list) , (input-spec-list) , (output-spec-list) , (record-spec-list) , (agent-spec)

scoped-identifier , invocationGenerator scoped-identifier

For operations invoked with an invocationGenerator, the named invocation generator is a function that takes as input arguments the parent script name, the resolved invocation information for the parent script, the script step name and operation name. It returns the setting, argument, input specification, output specification and record specification lists, and the agent specification for the invocation of the operation. This allows the user to provide a function to map between the calling information of the parent script and the calling information for the steps of which it is composed.

The scoped identifier in the *operationInvocation* must refer to an elemental operation, a process program, or a script. The settings must refer to identifiers declared in the header

list for the operation or in the header lists for the associated processes. Identifiers without scope refer to variables declared in the operation header. The calling argument values are supplied as an initializer list. All of the values for settings, arguments, inputs and outputs are in the scope of the invocation. The values go out of scope (i.e. are no longer available) when the script invocation to which the operation invocation belongs is complete. Thus, they are available to other operation invocations in the same script level.

The compiler or interpreter of the script must check type and number of arguments, input specification and output specification lists. It must also check the types of these values.

The following is an example of an operation invocation. The forms for settings, arguments, input and output specifications, record specifications and agent specifications are discussed in subsequent sections.

```
// Operation name
    welding::spotWeld ,
// Setting list
    ( pointTolerance = {.2,.2,.1,.1,.1,.1} , energyTolerance = .20) ,
// Argument list
    ( assembly1.wrkpt1, .35, .65, 1500., 1500. ) ,
// Input specification list
    ( assemblyStep1 output.theAssembly) ,
// Output deposition list
    ( assemblyStep2 input.part1 ) ,
// Record specification list
    ( energyDeposition (end), peakPower (end), theWeldedPoint (end)) ,
// Agent specification
    ( agentClass welding::spotwelder && certificationCriterion ANSI12Ba
    && maintenance maintenance_methodA (byCycle 10000)
    && calibration calibr_methodB (byOperationTime 5000))
```

2.8.1 Setting Values

The settings must refer to a key value declared in the header for the object associated with the qualified identifier. Normally, the setting is for one of the key values of the current operation. In which case, the qualified indentifier is just the identifer for the key value as declared in the header of the operation prototype.

setting-list:

/* Null is OK */

setting

setting-list , setting

setting:

qualified-identifier = initailizer

2.8.2 Argument Specifications

The arguments for the formal parameters for the invoked operation are supplied as an intializer list.

2.8.3 Input and Output Specification

The inputs must refer to a new workpiece, an output of a previous operation invocation, or an input to the script of which the invocation is a part.

input-spec-list:

/* Null is OK */

input-spec

input-spec-list , input-spec

input-spec:

scoped-identifier /* Previously declared workpiece */

(workpiece-declaration) /* New workpiece */

identifier . output . identifier /* Direct output of the script step refered to by first identifier */

input . identifier /* Input to the parent script */

The output depositions must refer to a termination stream, the input to a subsequent operation invocation in the script, or an output of the script. The primary parent of an operation output is used to reflect a persistent object that is transformed by an operation. For example, when machinning a feature into an exiting workpiece, it is convenient to refer to the resulting workpiece by the same name. This is done by indicating the primary parent of the operation output. The primary parent of an operation output is one of the inputs to the operation or the primary parent of one of the inputs. The latter case uses the qualified identifier composed of the identifer for the input and the parent context keyword. For example, inputOne refers to "inputOne" of the current operation invocation, and inputOne.parent refers to the parent of "inputOne" of the current operation invocation..

output-spec-list:

/* Null is OK */

output-spec

output-spec-list , output-spec

output-spec:

output-deposition

output-deposition output-parent

output-deposition:

workpiece-termination-spec /* Terminate the workpiece */

identifier . input . identifier /* Direct input of script step refered to by first identifier */

output . identifier /* Output of the parent script */

output-parent:

qualified-identifier /* Indication of the workpiece that was the direct parent */

The workpiece termination specification is probably not going to be fleshed out soon.

workpiece-termination-spec:

literal-constant

2.8.4 Records Specification

The records specification list indicates what must be recorded to verify that the operation was performed according to specifications, and to provide data for process improvement. The record specification consists of an identifier from the header declaration or parameter list and specifications of when to record its value. Note that the calling arguments may be modified during the performance of an operation.

The zone specification assumes three regions: High, Normal and Low with high and low alarms and boundaries with deadzones between the three zones. Six terms are required to specify the parameters of the zone model. The zone specification results in the recording of the initial zone for the identifier and any crossings of zone boundaries. See *General Equipment Interface Definition* for a discussion of the zone model.

This record specification is not complete but it handles lots of what we need.

record-spec-list:

/* Null is OK */

record-spec

record-spec-list , record-spec

record-spec:

key-identifier (sampling-spec-list)

sampling-spec-list:

sampling-spec

sampling-spec-list , sampling-spec

Obviously, the units of the terms below cannot be ignored but we will for now.

sampling-spec:

start

end

onChange

samplingRate term

zoneSpecification (zone-limit-spec-list)

The zone specification can only be given for those operation variables for which a zone model has been declared. Furthermore, limits may be changed from the defaults only for those limits to which the caller has been given write access.

zone-limit-spec

HighAlarmLimit term

HighDeadzoneUpperLimit term

HighDeadzoneLowerLimit term

LowDeadzoneUpperLimit term

LowDeadzoneLowerLimit term

LowAlarmLimit term

zone-limit-spec-list:

/* Null is OK */

zone-limit-spec

zone-limit-spec-list , zone-limit-spec

2.8.5 Agent Specification

Generally, the agent specification will give criteria for determining whether an agent is certified to perform the operation, or it will reference an agent that has previously been selected either for the parent script or a previous script step. The agent specification allows designation of the agent assigned to the current script by use of a qualified identifier composed of the context keyword combination, script.agent. The agent of a previous script step can be specified using a qualified identifier that ends with the context keyword, agent. For example, the the agent of the step named "firstPass" in the current script is specified by script.firstPass.agent

The actual agent constriants have yet to be defined. At minimum, the agent must be able to accept the operation with the calling information in the operation invocation. Calibration and maintenance invocation protocols indicate when the methods should be performed. The bySPC protocol assumes a zone model for the specified identifier which should refer to an operation variable for past executions of the operation on the candidate agent. The method should be invoked whenever the value leaves the normal zone. The values for the zone specification would presummably be based upon statistical quality control data. The agent certification criterion references the standards by which the agent and or the operators of the agent should be certified. For example, a welding operation should be performed by a certified weld engineer and with equipment certified to perform the operation.

Other standard constriants will be dreamed up later. The rules provided here allow for the logical combination of constriants.

agent-spec:

/* Null is OK */

qualified-identifier

agent-constraint

agent-constaint:

or-constraint

or-constraint:

and-constraint

or-constraint || and-constraint

and-constraint:

primary-constraint

and-constraint && primary-constraint

primary-constraint:

agentClass identifier

maintenance identifier (invocation-protocol-list)

calibration identifier (invocation-protocol-list)

certificationCriterion term

literal-constant

(agent-constraint)

invocation-protocol-list:

invocation-protocol

invocation-protocol-list , invocation-protocol

invocation-protocol:

byCycle term

byOperationTime term

byTotalTime term

bySPC qualified-identifier (zone-limit-spec-list)

Appendix 1 List of Declarations

declaration:

declaration-specifier quailfied-declarator-list ;

qualified-declarator:

init-declarator

init-declarator (declaration-qualifier-list)

quailfied-declarator-list:

qualified-declarator

qualified-declarator-list , qualified-declarator

init-declarator:

declarator

declarator = initializer

key-value-declaration:

superkey identifier (key-value-declaration-list)

declaration-specifier identifier (declaration-qualifier-list)

declaration-specifier identifier = initializer (declaration-qualifier-list)

key-value-declaration-list:

/* Null is OK */
key-value-declaration
key-value-declaration-list , key-value-declaration

parameter-declaration:
declaration-specifier indexed-identifier
declaration-specifier indexed-identifier (declaration-qualifier-list)

parameter-declaration-list:
/* Null is OK */
parameter-declaration
parameter-declaration-list , parameter-declaration

declaration-specifier:
type-specifier

type-specifier: one of
integer real string script scriptStep scriptNode scriptGenerator elementalOperation processProgram invocationGenerator workpieceClass processClass PDL context-Function typedef-name

declarator:
scoped-identifier

scoped-identifier:
indexed-identifier
scope-specifier indexed-identifier

scoped-identifier-list
scoped-identifier
scoped-identifier-list, scoped-identifier

scope-specifier:
:: /* Problem scope */
indexed-identifier ::

indexed-identifier:
identifier
indexed-identifier [term]

initializer:
term
{ initializer-list }

initialize-list:
/* Null is OK */
initializer
initializer-list , initializer

declaration-qualifier:
unit-specification
write-access
range-specification
zone-model-specification

declaration-qualifier-list:
declaration-qualifier
declaration-qualifier-list , declaration-qualifier

write-access:
callerWrite

callerWriteRequired
executionWrite
callerAndExecutionWrite
callerRequiredAndExecutionWrite

range-specification:
number-range
number-range-choices
string-choices
{ range-specification-list }

range-specification-list:
range-specification
range-specification-list , range-specification

number-range:
[term]
[term term]
[term term)
(term term]
(term term)

number-range-list:
number-range
number-range-list , number-range

number-range-choices:
(number-range-list)

string-choices:
(term-list)
HighAlarmLimit
HighDeadzoneUpperLimit
HighDeadzoneLowerLimit
LowDeadzoneUpperLimit
LowDeadzoneLowerLimit
LowAlarmLimit

zone-model-specification:
zoneModel limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration limit-declaration

limit-declaration:
term write-access

term:
qualified-identifier
constant

term-list
term
term-list, term

constant:
integer-constant
string-constant
real-constant
literal-constant

literal-constant:
' (...)
contextFunction-declaration:
contextFunction scoped-identifier (type-specifier , type-specifier , parameter-declaration-list) ;
context-function-invocation:
scoped-identifier (initializer-list)
qualified-identifier:
scoped-identifier
query
context-function-invocation
qualified-identifier . indexed-identifier
qualified-identifier . context-function-invocation
qualified-identifier . (term-expression)
operation
script
agent
input
output
parm
valueRange
HighAlarmLimit
HighDeadzoneUpperLimit
HighDeadzoneLowerLimit
LowDeadzoneUpperLimit
LowDeadzoneLowerLimit
LowAlarmLimit
domain-declaration:
processClass-definition
elementalOperation-definition
processProgram-definition
script-definition
scriptStep-definition
scriptNode-definition
workpieceClass-definition
PDL-definition
contextFunction-declaration:
declaration
domain-declaration-list:
domain-declaration
domain-declaration-list domain-declaration
processClass-definition:
processClass identifier (process-header) { processClass-initializer-list } ;
process-header:
/* A null process header is acceptable */
superProcess identifier
superProcess identifier , key-value-declaration-list

key-value-declaration-list

processClass-initializer-list:

/* Null is OK */

domain-declaration-list

workpieceClass-definition:

workpieceClass scoped-identifier (workpieceClass-header) ;

workpieceClass-header:

/* Null header is OK */

superWorkpieceClass identifier

superWorkpieceClass identifier , key-value-declaration-list

key-value-declaration-list

workpiece-declaration:

scoped-identifier indexed-identifier (setting-list);

io-workpiece-declaration-list:

/* Null is OK */

scoped-identifier identifier

io-workpiece-declaration-list, scoped-identifier identifier

PDL-definition:

PDL identifier , (PDL-header) ;

PDL-header:

processClass identifier PDL-feature-declaration-list

PDL-feature-declaration:

identifier (key-value-declaration-list)

PDL-feature-declaration-list:

/* Null is OK */

PDL-feature-declaration

PDL-feature-declaration-list , PDL-feature-declaration

operation-prototype:

(operation-header) , (parameter-declaration-list), (io-workpiece-declaration-list)

, (io-workpiece-declaration-list)

operation-header:

key-value-declaration-list

elementalOperation-definition:

elementalOperation scoped-identifier , operation-prototype ;

processProgram-definition:

processProgram scoped-identifier , operation-prototype , PDLSpec ;

PDLSpec:

identifier { program-description-list }

identifier { program-description-list } , setup operation-invocation

PDL-feature-spec:

identifier (setting-list)

program-description-list:

/* Null is OK */

PDL-feature-spec

program-description-list , PDL-feature-spec

script-definition:

script scoped-identifier , operation-prototype , script-ends , { script-definer-list } ;

```
        script scoped-identifier , operation-prototype , scriptGenerator  scoped-identifier ;
scriptEnds:
        ( identifier  identifier )  /* Starting Node and Terminating Node */
script-definer-list:
        /* Null is OK */
        domain-declaration-list
scriptStep-definition:
        scriptStep scoped-identifier , operationInvocation , previous-node-spec , next-node-
        spec ;
previous-node-spec:
        node-spec
next-node-spec:
        node-spec
node-spec:
        scoped-identifier                    /* previous/next script step  */
        scoped-identifier integer-constant  /* previous/next node and exit/entry set number */
scriptNode-definition:
        scriptNode scoped-identifier { entrance-set-list } { exit-set-list } ;
entrance-set-list:
        /* Null is OK */
        ( scoped-identifier-list )
        entrance-set-list,  ( scoped-identifier-list )
exit-set-list:
        /* Null is OK */
        selection-criterion ( scoped-identifier-list )
        exit-set-list , selection-criterion ( scoped-identifier-list )
selection-criterion:
        or-criterion
or-criterion:
        and-criterion
        or-criterion || and-criterion
and-criterion:
        compare-criterion
        and-criterion && compare-criterion
compare-criterion:
        primary-criterion
        term compare-operator term
primary-criterion:
        term
        ( selection-criterion )
compare-operator:
        ==
        !=
        >
        <
        <=
        >=
```

operationInvocation:
scoped-identifier , (setting-list) , (initializer-list) , (input-spec-list) , (output-spec-list) , (record-spec-list) , (agent-spec)
scoped-identifier , invocationGenerator scoped-identifier
setting-list:
/* Null is OK */
setting
setting-list , setting
setting:
qualified-identifier = initailizer
input-spec-list:
/* Null is OK */
input-spec
input-spec-list , input-spec
input-spec:
scoped-identifier /* Previously declared workpiece */
(workpiece-declaration) /* New workpiece */
identifier . output . identifier /* Direct output of the script step refered to by first identifier */
input . identifier /* Input to the parent script */
output-spec-list:
/* Null is OK */
output-spec
output-spec-list , output-spec
output-spec:
output-deposition
output-deposition output-parent
output-deposition:
workpiece-termination-spec /* Terminate the workpiece */
identifier . input . identifier /* Direct input of script step refered to by first identifier */
output . identifier /* Output of the parent script */
output-parent:
qualified-identifier /* Indication of the workpiece that was the direct parent */
workpiece-termination-spec:
literal-constant
record-spec-list:
/* Null is OK */
record-spec
record-spec-list , record-spec
record-spec:
key-identifier (sampling-spec-list)
sampling-spec-list:
sampling-spec
sampling-spec-list , sampling-spec
sampling-spec:
start
end

onChange
samplingRate term
zoneSpecification (zone-limit-spec-list)

zone-limit-spec
HighAlarmLimit term
HighDeadzoneUpperLimit term
HighDeadzoneLowerLimit term
LowDeadzoneUpperLimit term
LowDeadzoneLowerLimit term
LowAlarmLimit term

zone-limit-spec-list:
/* Null is OK */
zone-limit-spec
zone-limit-spec-list , zone-limit-spec

agent-spec:
/* Null is OK */
qualified-identifier
agent-constraint

agent-constaint:
or-constraint

or-constraint:
and-constraint
or-constraint || and-constraint

and-constraint:
primary-constraint
and-constraint && primary-constraint

primary-constraint:
agentClass identifier
maintenance identifier (invocation-protocol-list)
calibration identifier (invocation-protocol-list)
certificationCriterion term
literal-constant
(agent-constraint)

invocation-protocol-list:
invocation-protocol
invocation-protocol-list , invocation-protocol

invocation-protocol:
byCycle term
byOperationTime term
byTotalTime term
bySPC qualified-identifier (zone-limit-spec-list)

Distribution

Internal Distribution:

1	MS 1002	P. J. Eicker, 9600
1	MS 1010	M. E. Olson, 9622
10	MS 1010	J. M. Griesmeyer, 9622
1	MS 0188	LDRD Office, 4523
1	MS 9018	Central Technical File, 8940-2
5	MS 0899	Technical Library, 4961
2	MS 0619	Review & Approval Desk, 12690 for DOE/OSTI
1	MS 0161	Patentand Licensing Office, 11500
6		RMSEL Library