NISTIR 5980

Programmer's Guide to the 1996 Demo Executor

D. Flater

Evan Wallace

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899-0001

NIST

NISTIR 5980

Programmer's Guide to the 1996 Demo Executor

D. Flater

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899-0001

February 1997

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

Programmer's Guide to the 1996 Demo Executor

Contents

List of Figures

List of Tables

This work was funded through the National Advanced Manufacturing Testbed (NAMT) project and the Systems Integration for Manufacturing Applications (SIMA) program.

Programmer's Guide to the 1996 Demo Executor

David Flater
Evan Wallace

March 6, 1997

Abstract

This report documents the interfaces and functions of the 1996 demo version of the Executor. The Executor is the shop controller for the National Advanced Manufacturing Testbed (NAMT) Framework demo, a prototype distributed manufacturing system that is used to validate pre-normative specifications for manufacturing-related protocols and interfaces.

1 Introduction

The National Advanced Manufacturing Testbed (NAMT) Framework[1] demo is a prototype distributed manufacturing system that serves as a testbed and trial implementation for emerging industry-developed specifications such as SEMATECH's Computer Integrated Manufacturing (CIM) Framework.[2] The core of the NAMT Framework demo is the Executor, which acts as the shop controller in the NAMT Framework architecture (see Figure 1).

The Executor is a multi-threaded Common Object Request Broker Architecture (CORBA) server. It dispatches discrete parts manufacturing jobs that are requested by a human operator via the Guardian. The Guardian is the user interface that accepts input from a terminal and translates it into CORBA commands understood by the Executor. Other components in the architecture are the Production Information Base (PIB), which uses a commercial object base to maintain lot information; the Product Data Manager (PDM), which uses another commercial product to maintain STEP (Standard for the Exchange of Product model data) Part 21 documents containing the routings for the lots; and the workcell, which is the next level of control down from the Executor, closer to the actual machines.

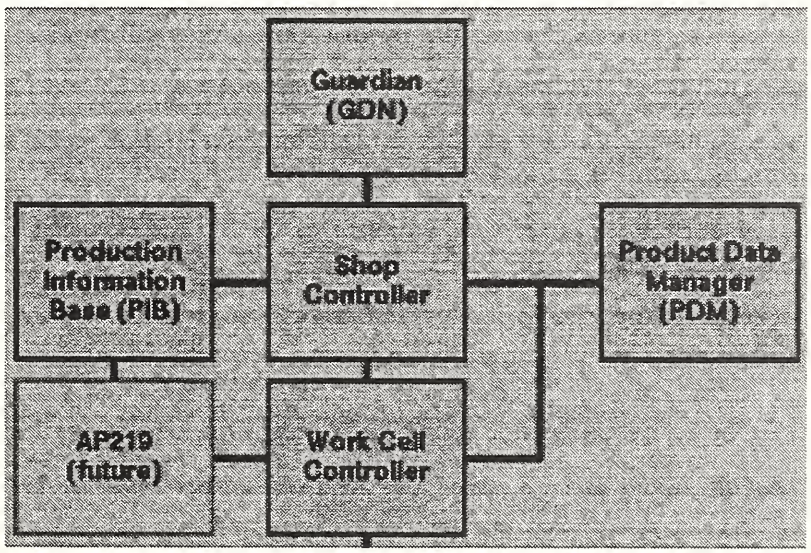

Figure 1: Architecture of the NAMT Framework Demo

This guide to the Executor is written at a level to be understood by a programmer, rather than a casual user, because the Executor itself does not interface directly with the user. It is the Guardian's job to provide user-friendly access to the Executor's functions, and to mediate between the keyboard and mouse on one hand and the Executor's CORBA API on the other. The purpose of *this* report is to document that API as it is seen by the Guardian or by any other clients of the Executor that may be built in the future. Documentation at an even lower level that discusses the internals of the Executor and various implementation issues is available on the NIST intranet at <URL:http://www-i.cme.nist.gov/framework/doc/papers96/ex96docs/>.

2 The Executor

The primary functions of the Executor are the following:

1. Respond to commands and status requests from the Guardian, which receives control input directly from the operator.
2. Retrieve lot information from the PIB to determine the routings.
3. Retrieve routings from the PDM and parse them.
4. Activate jobs.
5. Dispatch tasks to workcells.
6. Receive feedback from the workcells.
7. Abort jobs that are botched.
8. Write logging information.

The job control and status monitoring interfaces that the Executor offers to the Guardian are documented in following sections. The interface provided by the workcell to the Executor is largely the same as that provided by the Executor to the Guardian; both inherit the primary aspects of their interface from the same class. The other interfaces supported by the Executor, namely those to the PIB and the PDM, are out of the scope of this report, since they are defined unilaterally by the PIB and PDM; however, the interactions with the PIB and PDM are important to understanding how the Executor works.

In order to show the Executor in context, the following scenario describes its interactions with the Guardian, the PIB, the PDM, and the workcell by following the path of a job through the system. Figure 2 illustrates the associations between the various entities that will be discussed below.

- The Guardian asks the Executor to create a new shop-level job.
- The Guardian supplies values for the empty fields in the shop job (job name, lot, and priority), then informs the Executor that the job is ready to go.
- The Executor gets the lot ID from the job, retrieves the lot info from the PIB, and extracts the ID of the routing from the lot.
- The Executor retrieves the routing info (a STEP Part 21 document) from the PDM and parses it.
- The Executor constructs a list of ShopProcessJobs corresponding to the tasks in the routing and makes note of the workcells that are eligible to perform each one.
- When the shop has resources for another job, the Executor activates the job.
- When a workcell becomes available for the first task in the job, the Executor starts the task in the same way that the Guardian started the shop-level job: by asking the workcell to create a new job, filling in the fields, and then asking the workcell to start it.

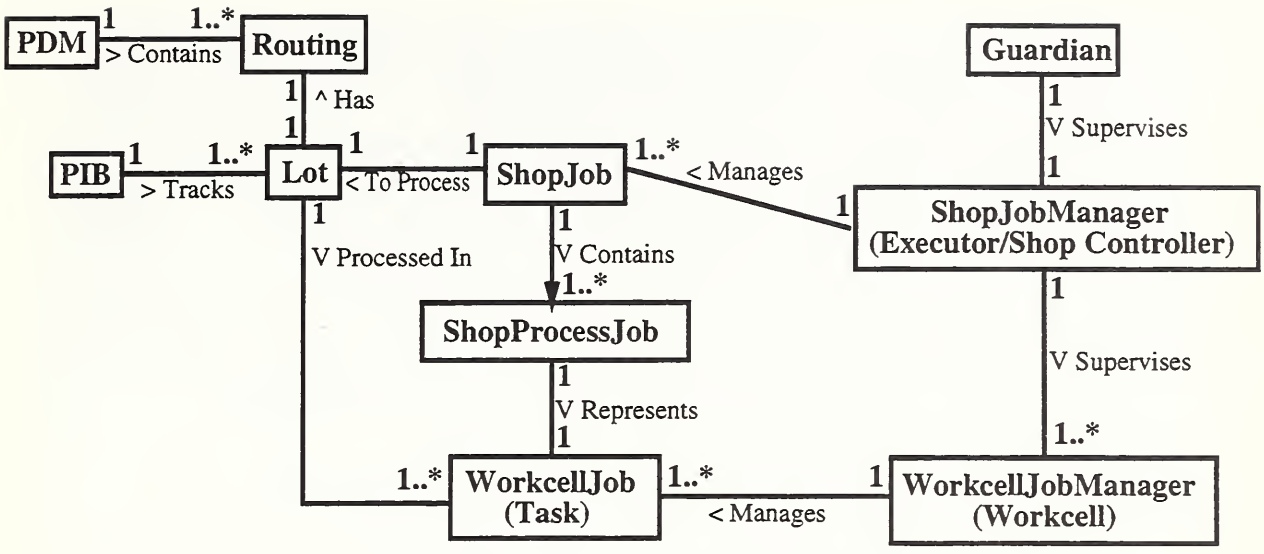

Figure 2: Association diagram

- The workcell performs the task, then informs the Executor when it is done. If the task is botched, the workcell informs the Executor of this too, so that the Executor can take appropriate action.
- The Executor recalculates the state of the shop-level job (e.g. Executing, Completed, Stopped, or Aborted) from the states of its constituent tasks. If the job is still Executing, the Executor proceeds to dispatch its next task.

In addition to the activity described above, the Executor is continuously polled for status information, and logs are maintained.

The Executor's job control and status monitoring interfaces are defined in IDL, the Interface Definition Language that is used for CORBA servers. Those IDL files are provided in the appendices and are explained in detail in following sections. The class hierarchy is very simple:

- JobManager: generic superclass for Executor and workcell controllers.
  - ShopJobManager: the class of the Executor.
  - WorkcellJobManager: the class of the workcell controller.
- ManagedJob: generic superclass for ShopJob, ShopProcessJob, and WorkcellJob.
  - ShopJob: the class of a shop-level job.
  - ShopProcessJob: the class that the Executor uses to keep record of workcell tasks.
  - WorkcellJob: the class of a workcell task, owned by the workcell.

2.1 Logging

The Executor maintains three of the four logs that are displayed by the Guardian (see Figure 3); two directly, and one indirectly via the PIB client library. Logging is used to track the nature of the message traffic between the Executor and the other components. The lines of text are color-coded to indicate which specification was the source for each message.

The format for a line in a log file is "`KEYWORD` text..." with `KEYWORD` being one of the following choices to indicate the source specification:

- `AP203`: STEP Part 203, Configuration Controlled Design

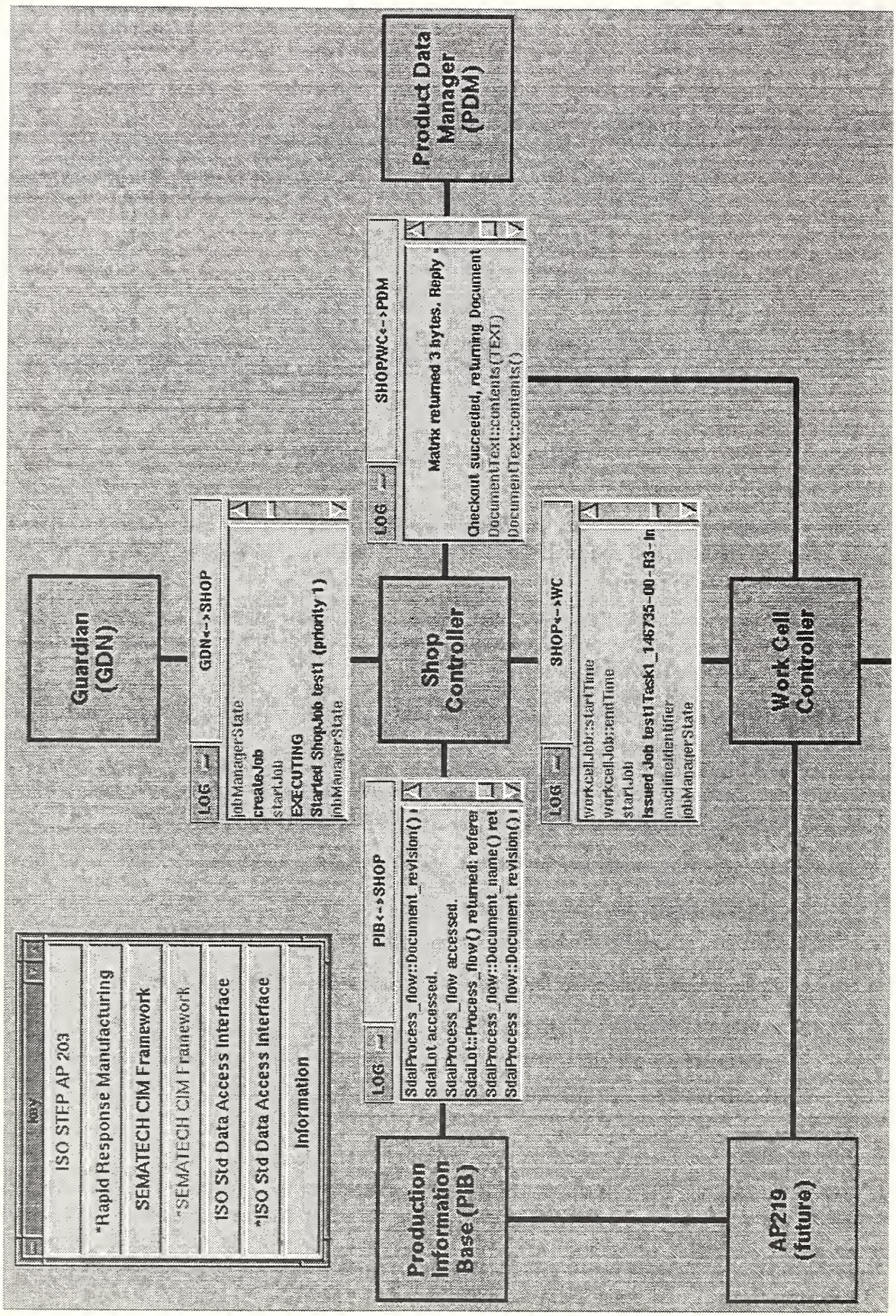

Figure 3: Guardian screen showing logs

- `RRM*`: Modified Rapid Response Manufacturing (RRM) for PDM/IDL
- `CIMF`: SEMATECH CIM Application Framework (CIMF)
- `CIMF*`: Modified CIMF
- `SDAI`: Standard Data Access Interface (SDAI)
- `SDAI*`: Modified SDAI
- `INFO`: Not a message, just useful information

2.2 Checking status

The Executor provides a set of top-level operations by which a client can check the Executor's status and get handles to shop jobs and workcells (Table 1). The client can get more information on jobs and workcells by calling operations on them directly once a handle has been retrieved. Table 2 shows the status operations provided by all ManagedJobs, while Tables 3 and 4 show the additional specialized status operations provided by ShopJobs and ShopProcessJobs respectively.

Table 1: Status operations available on the Executor

Operation	Description
boolean isPaused();	Returns true if the Executor is in the Paused state.
jm_state jobManagerState();	Returns the state of the Executor as an enum.
ManagedJob findJobNamed(in string jobName) raises (ManagedJobRetrievalFailedSignal);	Returns a handle to the requested job; raises ManagedJobRetrievalFailedSignal if not found.
ManagedJob findQueuedJobNamed(in string jobName) raises (ManagedJobRetrievalFailedSignal);	Like findJobNamed, but restrict search to jobs in the Queued or Queued-Held states.
ManagedJob findActiveJobNamed(in string jobName) raises (ManagedJobRetrievalFailedSignal);	Like findJobNamed, but restrict search to jobs in one of the following states: Executing; Pausing; Paused; Stopping; Aborting; Stopping-Aborting; Pausing-Stopping; Pausing-Aborting; Pausing-Stopping-Aborting.
ManagedJob findCompletedJobNamed(in string jobName) raises (ManagedJobRetrievalFailedSignal);	Like findJobNamed, but restrict search to jobs in the Completed state.
ManagedJobSequence allQueuedManagedJobs();	Returns a sequence of all jobs in the Queued or Queued-Held states.
ManagedJobSequence allActiveManagedJobs();	Returns a sequence of all jobs in one of the following states: Executing; Pausing; Paused; Stopping; Aborting; Stopping-Aborting; Pausing-Stopping; Pausing-Aborting; Pausing-Stopping-Aborting.
ManagedJobSequence allFinishedManagedJobs();	Returns a sequence of all jobs in one of the following states: Completed; Stopped; Aborted.
ManagedJobSequence allCompletedManagedJobs();	Returns a sequence of all jobs in the Completed state.

ManagedJobSequence allStoppedManagedJobs();	Returns a sequence of all jobs in the Stopped state.
ManagedJobSequence allAbortedManagedJobs();	Returns a sequence of all jobs in the Aborted state.
ManagedJobSequence allCancelledManagedJobs();	Returns a sequence of all jobs in the Cancelled state.
ManagedJobSequence allManagedJobs();	Returns a sequence of all jobs, excluding only those that have not yet been started (i.e. in the Created or Created-Held states).
WorkcellJobManagerSequence allWorkcells();	Returns a sequence of all the workcells' job manager objects.

Table 2: Status operations available on all ManagedJobs

Operation	Description
boolean isAborting();	True if job is Aborting, Pausing-Aborting, Stopping-Aborting, or Pausing-Stopping-Aborting
boolean isAborted();	True if job is Aborted
boolean isActive();	True if job is Executing, Pausing, Paused, Stopping, Aborting, Stopping-Aborting, Pausing-Stopping, Pausing-Aborting, or Pausing-Stopping-Aborting.
boolean isCancelled();	True if job is Cancelled.
boolean isCompleted();	True if job is Completed.
boolean isCreated();	True if job is Created or Created-Held.
boolean isFinished();	True if job is Completed, Stopped, or Aborted.
boolean isPausing();	True if job is Pausing, Pausing-Stopping, Pausing-Aborting, or Pausing-Stopping-Aborting.
boolean isPaused();	True if job is Paused, Created-Held, or Queued-Held.
boolean isQueued();	True if job is Queued or Queued-Held.
boolean isStopping();	True if job is Stopping, Pausing-Stopping, Stopping-Aborting, or Pausing-Stopping-Aborting.
boolean isStopped();	True if job is Stopped.
attribute mj_state jobState;	Returns the state of the job as an enum.
attribute string jobName;	The name of the job.

Table 3: Additional ShopJob status operations

Operation	Description
attribute Lot theLot;	The Lot that was specified for the job.
attribute short priority;	The priority that was specified for the job.
attribute ManagedJobSequence Tasks;	The sequence of ShopProcessJobs associated with this job.

Table 4: Additional ShopProcessJob status operations

Operation	Description
attribute WorkcellJob the_task;	The WorkcellJob corresponding to this ShopProcessJob.

attribute WorkcellJobManager workcell;	The workcell to which this task is currently assigned (null if not assigned).
attribute ShopJob parentjob;	The ManagedJob that contains this ShopProcessJob.
attribute DocumentSpecification ProcessOperationDocumentRef;	The document ref from the routing for this task.
attribute IdentifierSequence EligibleWorkcells;	Sequence of IDs of eligible workcells copied from the routing.
attribute Duration routingexpectedtimetocomplete;	Expected time to complete copied from the routing.

2.3 Commands

The Executor also provides operations that a client (i.e. the Guardian) can use to control both the state of shop-level jobs and the state of the Executor itself. The control functions are divided into three categories: the functions to control the Executor (Table 5), the functions to control single jobs (Table 6), and the functions to control all jobs (Table 7). The "all" functions simply broadcast one of the single-job functions to every job known to the Executor.

All of the operations described in this section are provided by the JobManager class, not by the ManagedJob class.

Table 5: Commands affecting the state of the Executor

Operation	Description
oneway void abortManagerOperations();	Causes the Executor to emergency-stop and cease to exist.
void pauseManagerOperations();	Causes the Executor to enter the Pausing state and stop dispatching new shop-level jobs onto the shop floor. When the shop floor is idle, the Executor enters the Paused state. New jobs will still be accepted and placed in the queue.
void stopManagerOperations();	Causes the Executor to enter the Stopping state and stop dispatching new shop-level jobs onto the shop floor. When the shop floor is idle, the Executor enters the Stopped state. New jobs will NOT be accepted.
void resumeManagerOperations();	Causes the Executor to return to the Executing or Waiting state from Pausing, Stopping, Pausing-Stopping, Paused, or Stopped.

Table 6: Commands affecting one shop-level job

Operation	Description
ManagedJob createJob();	Returns a handle to a newly created, uninitialized job. The caller should initialize the job and then pass it to startJob.

void startJob (in ManagedJob aManagedJob) raises (ManagedJobNotAcceptedSignal);	Place a new job into the queue so that the Executor will dispatch it. Raises ManagedJobNotAcceptedSignal if the Executor is Stopping or Stopped, or if the job has a problem like duplicate name or invalid lot.
void pauseJob (in ManagedJob aManagedJob) raises (InvalidStateTransitionSignal);	Causes the job to enter the Pausing state and stop dispatching new tasks to workcells. When all running tasks have completed, the job enters the Paused state and remains in the active list.
void stopJob (in ManagedJob aManagedJob) raises (InvalidStateTransitionSignal);	Analogous to pauseJob, except that when all running tasks have completed, the job enters the Stopped state and is considered finished. A stopJob directed at a job that is merely Queued will cause it to be Cancelled.
void resumeJob (in ManagedJob aManagedJob) raises (InvalidStateTransitionSignal);	Causes a paused job to return to the Executing state.
void abortJob (in ManagedJob aManagedJob) raises (InvalidStateTransitionSignal);	Causes a job to enter the Aborting state and cascade aborts to any currently executing tasks. When all tasks have aborted, the job enters the Aborted state and is considered finished. An abortJob directed at a job that is merely Queued will cause it to be Cancelled.
void removeFinishedJob(in ManagedJob aManagedJob);	Causes the Executor to free memory and forget about the specified job, which must be Completed, Stopped, Aborted, or Cancelled. (NOTE: "Finished" does not normally include "Cancelled;" this discrepancy results from extending the CIMF.)

Table 7: Commands affecting all shop-level jobs

Operation	Description
void makeAllManagedJobsPausing();	Effectively issues a pauseJob for each job. Exceptions are not propagated.
void makeAllManagedJobsNotPaused();	Effectively issues a resumeJob for each job. Exceptions are not propagated.
void makeAllManagedJobsAborting();	Effectively issues an abortJob for each job. Exceptions are not propagated.
void makeAllManagedJobsStopping();	Effectively issues a stopJob for each job. Exceptions are not propagated.

2.4 Initialized ShopJob fields

When the Guardian wants to start a new shop-level job, it must call createJob, initialize three fields in the job object that is returned, and then call startJob. The three fields are:

1. attribute string jobName;

   The name of the job. Must be unique.

2. attribute Lot theLot;

   Must be a valid key in the PIB to retrieve the lot from which the routing spec is retrieved.

3. attribute short priority;

   Optional. Controls the order in which queued shop jobs are dispatched onto the shop floor. 0 is the default; lower numbers cause jobs to be dispatched more quickly. (This is analogous to Unix process priorities.)

2.5 State diagrams

The state sets used by JobManager and ManagedJob are flattened implementations of concurrent state sets[1] derived from the CIMF (CIM Framework). Figure 4 shows the states of the JobManager class which the Executor implements. The Executor starts up in the Initializing state, then enters the Waiting state when it is ready to accept jobs. When it is given work to do, it enters the Executing state and only returns to the Waiting state when it is again idle with no jobs in its queue or active list.

Once it has received an abortManagerOperations message, the Executor irrevocably heads for the "aborted" state. ("Aborted" is not *really* a state; it represents the Executor having terminated.)

Figure 5 shows the states of the ManagedJob class which are used by ShopJob and ShopProcessJob. While a JobManager has only one terminal state, ManagedJobs can end up Completed, Stopped, Aborted, or Cancelled. A Cancelled job is one that was stopped or aborted before work on it even began. Unlike the analogous stopManagerOperations, a stopJob command is just as irrevocable as an abortJob. It is possible to throw a Stopping job into the Aborted state with an abortJob, but it is not possible to resume it.

As can be seen from reading the descriptions of the various state-changing commands in Tables 5 through 7, pausing or stopping involves reaching a convenient pause-point or stop-point. The JobManager does not pause or stop while there are active shop jobs; neither does a shop job pause or stop while there are active tasks. Instead, they refrain from starting any new work until the running jobs or tasks have completed, and then enter the Paused or Stopped state. Aborts, on the other hand, have no preconditions. When it is told to abort, the Executor will terminate immediately, without waiting for workcell activity to subside. A shop job that is aborted will cascade aborts to any active workcell tasks so that activity on the shop job will cease immediately. It is then left for the operator to clean up any problems that result.

3 Issues for Future Work

The preceding represents the interface of the Executor as it was used for the 1996 NAMT Framework demo. This was only the first year of a five-year project; the Executor is expected to continue to evolve in response to lessons learned, better understanding of the specifications, and improvements to the infrastructure. In support of that evolution, commentary on the limitations and possible problems with the 1996 interface is presented here.

[1] In a "concurrent" state model, the state of a machine is modeled using two or more independent state variables. For example, the state of a two-engine airplane may be modeled using independent state variables for each engine.

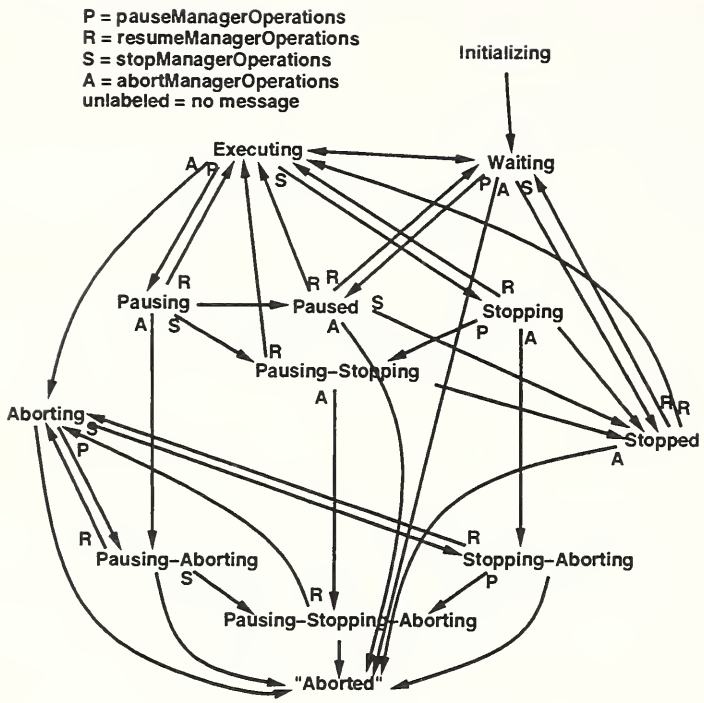

Figure 4: JobManager states

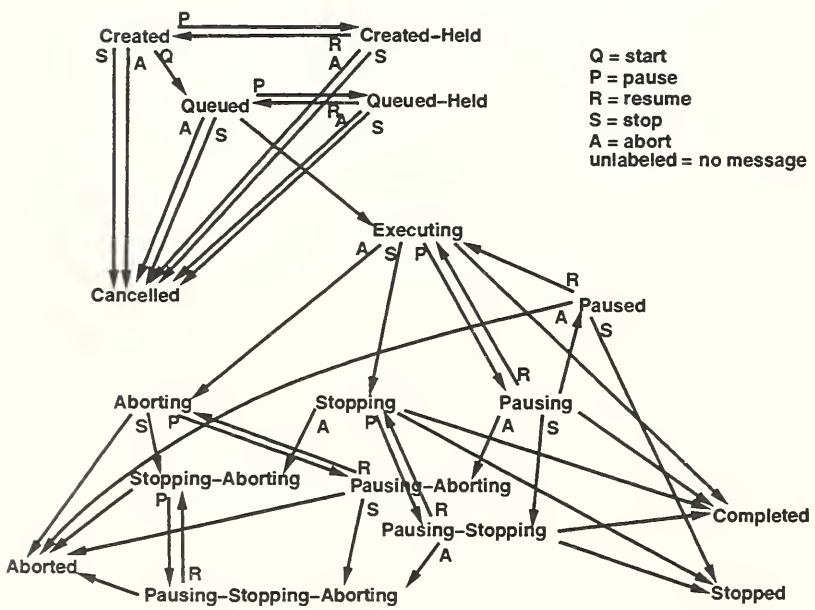

Figure 5: ManagedJob states

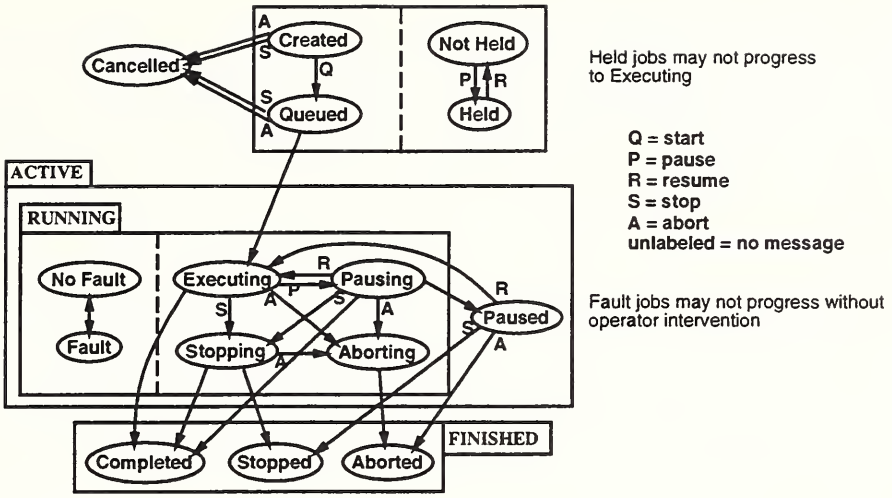

Figure 6: Proposed ManagedJob states (concurrent model)

3.1 States

The concurrent states model of the CIMF has been the subject of controversy and misunderstanding. Having discussed the issue with Ed Barkmeyer, another member of the Framework team, we now believe that the concurrency as implemented is not needed, and that another form of concurrency is needed but lacking.

The combinations of Pausing-Stopping, Pausing-Aborting, Stopping-Aborting, and Pausing-Stopping-Aborting are not meaningful to us. Aborts take precedence over stops, and stops take precedence over pauses. The notion that a job can pause in the middle of a stop or abort – to "freeze" immediately, then resume and continue on its way to the stop point or abort point – is not valid. If a machine is in the process of cutting metal, any kind of "freeze" is equivalent to an emergency stop, with ramifications even more serious than those of an abort.

Instead of this invalid concurrency, what is needed is a way to represent fault states. For example, if a workcell has been told to stop, but before it can reach a stop-point a machine gets stuck and cannot proceed, the state of the task is modeled by Stopping-Fault. The Executing, Pausing, Paused, Stopping, and Aborting states should all be mutually exclusive, but they may all be concurrent with the Fault state. That way, it is clear when human intervention is needed, and a task that is in a fault state can be recovered and continued after an operator has intervened. Figure 6 illustrates this proposed model.

3.2 Job control commands

In the interface described by this report, job control commands that affect a particular job are part of the JobManager interface and contain an argument specifying which job to affect. Since the state of the job can only be queried through the ManagedJob interface, it would make more sense for the state of the job to be controlled via the ManagedJob interface. That is the way it was organized in CIMF 1.3.

In the model underlying such an organization, the interface to a JobManager and the interfaces to related ManagedJobs are different aspects of a single agent. The JobManager provides access to services for controlling and monitoring the operation of the combined agent as well as some job control functions that can be applied to the aggregation of its related ManagedJobs. The ManagedJob interface itself provides for job control and monitoring of individual jobs. These interfaces are very similar in function to the Administrative and Task components of the Job Controller defined in the Manufacturing System Integration (MSI) project's Control Entity Interface (CEI).[3] What could not easily be done in the CEI, since an object oriented communications infrastructure was not available, was to make the tasks (jobs) directly addressable. But with CORBA, operations can be

defined directly on the job (task) interface.

The existing job control interface, a mixture of CIMF 1.3 and CIMF 1.2, was used because we could not find a working interpretation of CIMF 1.3. SEMATECH has since explained how they intended CIMF 1.3 to work, so the NAMT Framework testbed could and should be migrated in that direction in future development.

4 Further Reading

This is one of a number of documents that are to be written about the 1996 NAMT Framework project. Most significantly, a Framework Compendium is planned to provide an overview of the entire project, and a report will be produced that summarizes the results for the validation of the CIM Framework and SEMATECH's response. Currently the process of exchanging results with SEMATECH is still ongoing, and the other documents are not yet finalized.

Related reports will also be available on NIST's external and internal web servers from <URL:http://www.nist.gov/framework/> and <URL:http://www-i.cme.nist.gov/framework/doc/papers96/> respectively.

References

[1] Howard M. Bloom and Neil Christopher. A framework for distributed and virtual discrete part manufacturing. In *Proceedings of the CALS EXPO '96*, Long Beach, CA, October 1996.

[2] Lawrence Eng, Ken Freed, Jim Hollister, Carla Jobe, Paul McGuire, Alan Moser, Vinayak Parikh, Margaret Pratt, Fred Waskiewicz, and Frank Yeager. *Computer Integrated Manufacturing (CIM) Application Framework Specification 1.3.* SEMATECH, 2706 Montopolis Drive, Austin, TX 78741 U.S.A., 1996.

[3] Sarah Wallace, M. K. Senehi, Ed Barkmeyer, Steven Ray, and Evan K. Wallace. *Manufacturing Systems Integration: Control Entity Interface Specification.* National Institute of Standards and Technology, Interagency Report 5272, 1993. Available from the National Technical Information Service, Springfield, VA 22161 U.S.A.

A job_mgr.idl

```
// Interface definition for generic job and job manager components of the NAMT
// Framework

// Started:  1996-07-30 by Dave Flater
// Status:  Fairly stable

// The following definitions were originally from:
// Part 1:  IDL from CIM Framework, specifically:
// Modified IDL from CIM Framework Spec 1.3
// Updated by Paul McGuire, SEMATECH, March 29, 1996

// Many things have been modified for any or all of the following
// reasons:
//
//  1.  Working around inconsistencies in the CIMF;
//  2.  Correcting obvious errors in the CIMF;
//  3.  Reducing complexity for the trial implementation;
//  4.  Making NAMT components play together.

#define _JOBMGRIDL

#ifndef _RESOURCEIDL
#include "resource.idl"
#endif

interface ManagedJob;
#ifndef _MJS
typedef sequence<ManagedJob> ManagedJobSequence;
#define _MJS
#endif

// ManagedJob states.

// Flattened implementation of the concurrent state combinations from
// p. 41 of the CIMF.  Technically, any active state is "EXECUTING,"
// but here MJ_EXECUTING means very specifically the state in which
// we are active, not pausing, not stopping, and not aborting.
//
// Created-held and queued-held have been added
// because a makeAllManagedJobsPausing probably wants to "hold"
// any queued jobs even if a workcell becomes available.  Still
// lacking is a fault state, as described by these comments quoted
// from executor_i.cc:

  // If one of my tasks was stopped or aborted, but the ShopJob was
  // running normally, then I have a problem.  In the MSI model, the
  // ShopJob enters a fault state and a red light flashes on the
  // console to get a human to do something.  The situation is
  // resolved when the human sends something like Abort, Retry, Call
  // Complete, or Defer.  Maybe as future work we will add this to
  // the CIMF, but currently all we can do is cascade the stop or
  // abort upward.
```

```
// Also possible is future addition of MJ_PAUSED_STOPPING,
// MJ_PAUSED_ABORTING, and MJ_PAUSED_STOPPINGABORTING to model various
// strange fault states.

enum mj_state {MJ_CREATED, MJ_CREATED_HELD,
               MJ_QUEUED, MJ_QUEUED_HELD,
               MJ_EXECUTING, MJ_PAUSING_EXECUTION, MJ_PAUSED_EXECUTION,
               MJ_STOPPING, MJ_ABORTING, MJ_STOPPINGABORTING,
               MJ_PAUSINGSTOPPING, MJ_PAUSINGABORTING,
                 MJ_PAUSINGSTOPPINGABORTING,
               MJ_STOPPED, MJ_CANCELLED, MJ_ABORTED, MJ_COMPLETED};

// JobManager states.  These are based loosely on the
// MachineResource class of the CIMF (p. 223).  JobManager and
// ProcessJobManager don't seem to have state diagrams.

// Note that CIMF MachineResource doesn't admit to a STOPPED state.

enum jm_state {JM_INITIALIZING, JM_WAITING, JM_EXECUTING,
               JM_PAUSING_EXECUTION, JM_PAUSED_EXECUTION,
               JM_STOPPING, JM_ABORTING, JM_STOPPINGABORTING,
               JM_PAUSINGSTOPPING, JM_PAUSINGABORTING,
               JM_PAUSINGSTOPPINGABORTING, JM_STOPPED};

// Generic JobManager to be inherited by ShopJobManager and WorkcellJobManager.

interface JobManager
{
  // EXCEPTIONS

  /* The search operation for a job failed. */
  exception ManagedJobRetrievalFailedSignal {};

  /* Attempt to create or start a new job rejected.  This will happen
     if the JobManager is in the JM_STOPPED state. */
  exception ManagedJobNotAcceptedSignal {};

  /*
    OPERATIONS FOR TOP-DOWN CONTROL AND MONITORING

    These are taken from JobManager.  MachineResource comes with
    operations like shutdownNormal and shutdownImmediate that would be
    redundant here.

    Helpful hints:

    1.  abortManagerOperations means E-Stop.  The JobManager goes away.

    2.  pauseManagerOperations and stopManagerOperations both mean
        that you should let any running jobs finish but don't start
        any new ones.  The only difference between STOPPED and PAUSED
        is that you can queue up new jobs while PAUSED but not while
        STOPPED.
```

```
  3.  You can either stop or abort a job that is merely created or
      queued to send it to the MJ_CANCELLED state.
*/

// COMMANDS AFFECTING THE STATE OF THE JOB MANAGER
oneway void abortManagerOperations();
void pauseManagerOperations();
void resumeManagerOperations();
void stopManagerOperations();

// COMMANDS AFFECTING THE STATE OF ALL MANAGED JOBS
void makeAllManagedJobsPausing();
void makeAllManagedJobsNotPaused();
void makeAllManagedJobsAborting();
void makeAllManagedJobsStopping();

// COMMANDS AFFECTING THE STATE OF ONE MANAGED JOB (CIMF 1.2)
// queueJob has been removed; just use startJob.
ManagedJob createJob();
void startJob (in ManagedJob aManagedJob);
void pauseJob (in ManagedJob aManagedJob);
void resumeJob (in ManagedJob aManagedJob);
void abortJob (in ManagedJob aManagedJob);
void stopJob (in ManagedJob aManagedJob);

// OTHER ADMINISTRATIVE OPERATIONS
void removeFinishedJob(in ManagedJob aManagedJob);

// OPERATIONS TO QUERY JOB MANAGER STATE
boolean isPaused();
jm_state jobManagerState();

// OPERATIONS TO ACCESS MANAGED JOBS
ManagedJob findJobNamed(in string jobName)
       raises (ManagedJobRetrievalFailedSignal);
ManagedJob findQueuedJobNamed(in string jobName)
       raises (ManagedJobRetrievalFailedSignal);
ManagedJob findActiveJobNamed(in string jobName)
       raises (ManagedJobRetrievalFailedSignal);
ManagedJob findCompletedJobNamed(in string jobName)
       raises (ManagedJobRetrievalFailedSignal);
ManagedJobSequence allQueuedManagedJobs();
ManagedJobSequence allActiveManagedJobs();
ManagedJobSequence allFinishedManagedJobs();
ManagedJobSequence allCompletedManagedJobs();
ManagedJobSequence allStoppedManagedJobs();
ManagedJobSequence allAbortedManagedJobs();
ManagedJobSequence allCancelledManagedJobs();
ManagedJobSequence allManagedJobs();


// OPERATIONS FOR BOTTOM-UP REPORTING
// Task name instead of a ManagedJob handle is used here due to
```

```
  // implementation difficulties with the other approach.
  void informCompletedJob(in string aJobName);
  void informJobNotCompleted(in string aJobName);

  // Not implemented
  // void informJobWillBeLate (in ManagedJob aManagedJob,
  //                           in TimeStamp newFinishTime);

  // Also not implemented is a mechanism for a subordinate job manager
  // to report to its superior that it is going off-line.
};


// Generic ManagedJob to be inherited by ShopJob and WorkcellJob.

interface ManagedJob
{
  // The make* operations are not used in the current configuration.
  // The state of the job must be directly modified by its owner.
  // To make this easier, jobState is declared as an attribute.

  // OPERATIONS TO QUERY JOB STATE
  boolean isAborting();
  boolean isAborted();
  boolean isActive();
  boolean isCancelled();
  boolean isCompleted();
  boolean isCreated();
  boolean isFinished();
  boolean isPausing();
  boolean isPaused();   // Returns True for "held" jobs
  boolean isQueued();
  boolean isStopping();
  boolean isStopped();


  // ATTRIBUTES

  attribute mj_state jobState;

  // This attribute is called jobName instead of just name because there
  // was a conflict with some internal Orb function also called name.
  attribute string jobName;

  // Any job except the lowest-level job is comprised of a sequence of
  // "tasks" or "operations."  This attribute provides a place to
  // store and access that sequence.
  attribute ManagedJobSequence Tasks;

  // Priority queues may or may not be supported.
  attribute short priority;  // Like Unix, 0 = default
};
```

B executor.idl

```
// Interface definition for the Executor component of the NAMT
// Framework

// Started:  1996-06-03 by Dave Flater
// Major revision:  1996-07-30
// Status:  Fairly stable

#define _EXECUTORIDL

#ifndef _RESOURCEIDL
#include "resource.idl"
#endif

#ifndef _JOBMGRIDL
#include "job_mgr.idl"
#endif

#ifndef _PIBIDL
#include "pib.idl"
#endif

interface ManagedJob;
#ifndef _MJS
typedef sequence<ManagedJob> ManagedJobSequence;
#define _MJS
#endif
interface ShopJob;
#ifndef _SJS
typedef sequence<ShopJob> ShopJobSequence;
#define _SJS
#endif
interface ShopProcessJob;
#ifndef _SPJS
typedef sequence<ShopProcessJob> ShopProcessJobSequence;
#define _SPJS
#endif

interface WorkcellJob;

interface WorkcellJobManager;
#ifndef _WJMS
typedef sequence<WorkcellJobManager> WorkcellJobManagerSequence;
#define _WJMS
#endif

interface ShopJobManager: JobManager
{
  // This function is provided for the benefit of the Guardian.
  // c.f. EquipmentManager::allMachines, CIMF p. 208
  WorkcellJobManagerSequence allWorkcells();

  // Future work:  operation to return currently executing plan?
```

```
  // Heartbeat is a method that is invoked at regular intervals by
  // polld.
  //
  // heartbeat returns the pid of the polld associated with this
  // Executor.  polld is expected to die if this does not match.
  long heartbeat ();

  // Functions for testing and debugging.  DO NOT USE!
  long counter ();
  void makeTestJob ();

  // These are what you might call "internal" operations for the
  // Executor.  In fact I need to declare these here in order for
  // them to be easily accessible from the ManagedJob.
  void informCompletedShopJob (in ShopJob aShopJob);
  // void informShopJobWillBeLate (in ShopJob aShopJob,
  //                               in TimeStamp newFinishTime);
  void informShopJobNotCompleted (in ShopJob aShopJob);
};

interface ShopJob: ManagedJob
{
  attribute Lot theLot;
};

// This job type is for internal use by the ShopJobManager.
interface ShopProcessJob: ManagedJob
{
  // This is the WorkcellJob managed by the workcell.
  attribute WorkcellJob the_task;

  // The workcell to which this task is currently assigned (null if not
  // assigned).
  attribute WorkcellJobManager workcell;

  // The ManagedJob that this task belongs to.
  attribute ShopJob parentjob;

  // DocumentRef to give to the workcell.
  attribute DocumentSpecification ProcessOperationDocumentRef;

  // List of eligible workcells.
  attribute IdentifierSequence EligibleWorkcells;

  // This is the expected time to complete value from the routing.
  // It probably isn't useful to the real workcell, but may be used
  // by a workcell simulation.
  attribute Duration routingexpectedtimetocomplete;
};
```

C workcell.idl

```
// Interface definition for Workcell
// Started:  1996-06-03 by Dave Flater
// Major revision:  1996-07-08
// Status:  Semi-stable

#define _WORKCELLIDL

#ifndef _RESOURCEIDL
#include "resource.idl"
#endif

#ifndef _JOBMGRIDL
#include "job_mgr.idl"
#endif

#ifndef _EXECUTOR
#include "executor.idl"
#endif

interface WorkcellJobManager;
#ifndef _WJMS
typedef sequence<WorkcellJobManager> WorkcellJobManagerSequence;
#define _WJMS
#endif

interface WorkcellJobManager: JobManager
{
  /* Get a description for the machine. */
  readonly attribute string description;

  /* Get the unique identifier for the Machine. */
  readonly attribute string machineIdentifier;

  // Heartbeats to the workcell are issued by the Executor.
  void heartbeat ();
};

interface WorkcellJob: ManagedJob
{
  attribute string ShopProcessJobName;
  // DocumentSpecification is a 4-member struct defined in resource.idl
  attribute DocumentSpecification ProcessOperationDocumentRef;
  attribute Lot theLot;
  attribute AVPSequence settings;

  attribute TimeStamp startTime;
  attribute TimeStamp endTime;
  readonly attribute Duration estimated_time_to_completion;

  // Note that priority is inherited from ManagedJob.
};
```