```
const string JobStateChangedSubject =
        "/JobSupervision/Job/StateChanged";
```

/* This enumerated type identifies the states of Jobs. It is used in event notifications of state changes. */

```
enum JobState {
        JobUndefined,
        JobCreated,
        JobQueued,
        JobActive,
        JobExecuting,
        JobNotPaused,
        JobPausing,
        JobPaused,
        JobNotStopping,
        JobStopping,
        JobNotAborting,
        JobAborting,
        JobFinished,
        JobCanceled,
        JobCompleted,
        JobStopped,
        JobAborted };

struct JobStateChangedFilters {
        Global::Property name;
        Global::Property previousState;
        Global::Property newState;
};
```

JobStateChangedFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "Name" | string | The name of the Job. |
| "PreviousState" | string | Previous state preceding the most recent change. |
| "NewState" | RegistrationState | New state following the most recent change. |

```
struct JobStateChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        JobStateChangedFilters eventFilterData;
        Global::Properties eventNews;
        Job aJob;
};
```

/* If the Job cannot be completed by the specified deadline, a JobDeadlineCannotBeMetEvent should be sent as early as possible, not necessarily after the deadline has passed. */

```
const string JobDeadlineCannotBeMetSubject =
        "/JobSupervision/Job/DeadlineCannotBeMet";

struct JobDeadlineCannotBeMetFilters {
        Global::Property name;
        Global::Property deadline;
};
```

JobDeadlineCannotBeMetFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "Name" | string | The name of the Job. |
| "Deadline" | Global::TimeStamp | Value of the Deadline that cannot be met. |

```
struct JobDeadlineCannotBeMetEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        JobDeadlineCannotBeMetFilters eventFilterData;
        Global::Properties eventNews;
        Job aJob;
};
```

/* This event is posted when a Job's deadline date has changed. */

```
const string JobDeadlineChangedSubject =
        "/JobSupervision/Job/DeadlineChanged";

struct JobDeadlineChangedFilters {
        Global::Property name;
        Global::Property previousDeadline;
        Global::Property newDeadline;
};
```

JobDeadlineChangedFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "Name" | string | The name of the Job. |
| "PreviousDeadline" | Global::TimeStamp | Previous Deadline. |
| "NewDeadline" | Global::TimeStamp | New Deadline. |

```
struct JobDeadlineChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        JobDeadlineChangedFilters eventFilterData;
        Global::Properties eventNews;
        Job aJob;
};
```

Exceptions: None.

Provided Services:

/* Ask the Job for its JobRequestor. */

```
JobRequestor getJobRequestor ()
        raises (Global::FrameworkErrorSignal);
```

/* Get a named Job property from its Job Specification. */

```
Global::Property getJobProperty (
        in Global::PropertyName aPropertyName)
        raises (Global::FrameworkErrorSignal,
        Global::InvalidPropertyNameSignal,
        Global::PropertyNotFoundSignal);
```

/* Set a named Job property in its Job Specification. */

```
void setJobProperty (
        in Global::Property aProperty)
        raises (Global::FrameworkErrorSignal,
        Global::SetValueOutOfRangeSignal,
        Global::InvalidPropertyNameSignal,
        Global::UnsupportedPropertySignal,
        Global::ReadOnlyPropertySignal);
```

/* Indicates whether results are available for this Job. Each specialization may determine what constitutes results. */

```
boolean areJobResultsAvailable()
        raises( Global::FrameworkErrorSignal );
```

/* Retrieve the latest results. Each implementation determines what constitutes relevant results. This may be used for returning current results for complex Jobs. */

```
JobSupervisor::Results getJobResults()
        raises( Global::FrameworkErrorSignal );
```

/* Begin the process to pause the Job at the next safe opportunity. Results in the transition to Pausing state and eventually Paused state. */

```
void makePaused ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* Request Job resume activity from the previous Pause. Results in the transition to the executing state. */

```
void makeExecuting ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* Request to cancel the Job. This operation is only valid if the Job is in the Queued state (e.g. the Job cannot be canceled once it is Active). This operation results in the transition to the Canceled state. */

```
void makeCanceled ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* Begin the process to stop the Job. This is an orderly termination and should never cause irreparable problems (e.g. should not stop etching a wafer in mid-cycle). This operation results in the transition to the Stopping state and eventually the Stopped state. */

```
void makeStopped ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* Begin the process to abort the Job. Caution should be used with this operation. Aborting a Job requires immediate termination of the Job and could result in irrecoverable change to factory or material state. This operation results in the transition to the Aborting state and eventually the Aborted state. */

```
void makeAborted ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* Determine whether the Job is in state indicated. */

```
boolean isAborting ( )
        raises (Global::FrameworkErrorSignal);

boolean isAborted ( )
        raises (Global::FrameworkErrorSignal);
```

```
boolean isActive ( )
        raises (Global::FrameworkErrorSignal);

boolean isCanceled ( )
        raises (Global::FrameworkErrorSignal);

boolean isCompleted ( )
        raises (Global::FrameworkErrorSignal);

boolean isExecuting ( )
        raises (Global::FrameworkErrorSignal);

boolean isFinished ( )
        raises (Global::FrameworkErrorSignal);

boolean isPausing ( )
        raises (Global::FrameworkErrorSignal);

boolean isPaused ( )
        raises (Global::FrameworkErrorSignal);

boolean isQueued ( )
        raises (Global::FrameworkErrorSignal);

boolean isStopping ( )
        raises (Global::FrameworkErrorSignal);

boolean isStopped ( )
        raises (Global::FrameworkErrorSignal);
```

/* Return the estimated time remaining until Job completion. The quality of this estimate is dependent both on the specific Job derivative and on the implementation. If the Job is Finished or Canceled, a zero Duration will be returned. */

```
Duration timeRemaining ( )
        raises (Global::FrameworkErrorSignal);

}; // Job
```

Contracted Services:

| *Interface* | *Component* | *Service* |
|---|---|---|
| JobRequestor | JobSupervision | informJobStateChange |

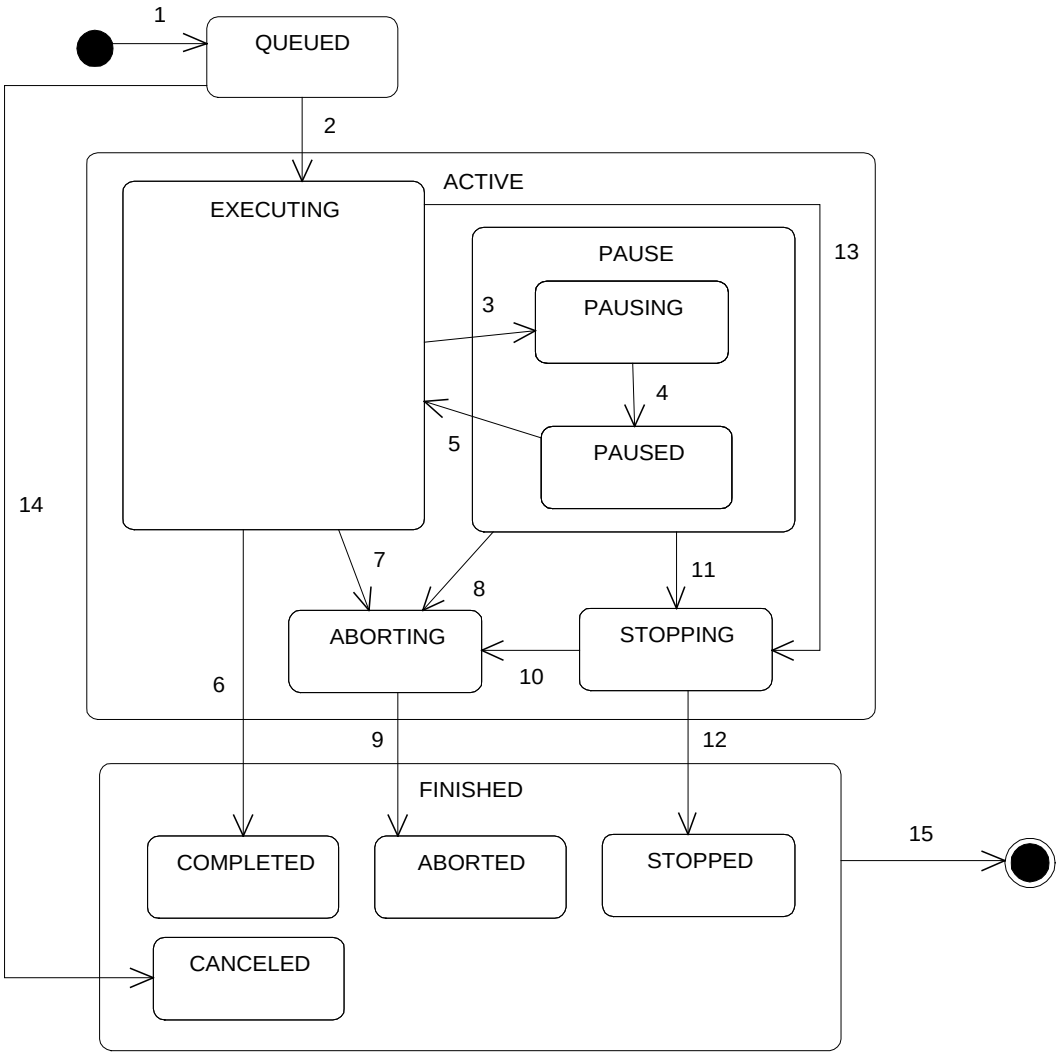


**Figure 5**
**Job Dynamic Model**

6.6.3.1 For implementations (e.g., Job derivatives), the Executing state is expected to be extended by partitioning it into at least two “orthogonal” states. One would hold the Pause states. The other would contain the implementation behavior of Executing.

Object State Tables:

**Table 3 Job State Definitions and Query Table**

| *State* | *Definition* | *Query for State via* |
|---|---|---|
| ABORTED | In this state the Job has aborted execution. This is a substate of Finished. | boolean isAborted(); sent to an instance of Job; returns TRUE<br>JobSequence allFinishedJobs (); sent to instance of JobSupervisor will provide some indication. |
| ABORTING | ABORTING represents an immediate termination of the Job and activities not completed before the aborting will be terminated. After ABORTING, execution is not intended to continue. | boolean isAborting(); sent to an instance of Job; returns TRUE |
| ACTIVE | This is a parent state representing that the Job is ACTIVE for the JobSupervisor; i.e., the current status of the Job is tracked by the JobSupervisor when the Job is in the ACTIVE state. | boolean isActive(); sent to an instance of Job; returns TRUE.<br>JobSequence allActiveJobs (); or Job findActiveJobNamed (in string jobName); sent to instance of JobSupervisor. |
| CANCELED | In this state the Job has been removed from the Queue and will never become Active. This is a substate of Finished. | boolean isCanceled(); sent to an instance of Job; returns TRUE |
| COMPLETED | This state represents that the Job has successfully completed execution. This is a substate of Finished. | boolean isCompleted(); sent to an instance of Job; returns TRUE.<br>Job findCompletedJobNamed (in string jobName); sent to instance of JobSupervisor<br>JobSequence allFinishedJobs (); sent to instance of JobSupervisor will provide some indication. |
| EXECUTING | This state represents that the Job is EXECUTING. Specializations of Job will normally develop substates representing the specialized Job execution behavior. | boolean isExecuting(); sent to an instance of Job; returns TRUE. |
| PAUSE | Parent state of PAUSING, PAUSED | None. |
| PAUSED | In this state the Job has paused execution. | boolean isPaused(); sent to an instance of Job; returns TRUE. |
| PAUSING | In this state the Job is being paused by the executor of the Job. Execution is intended to continue. | boolean isPausing(); sent to an instance of Job; returns TRUE. |
| QUEUED | In this state the Job is waiting to become active. | boolean isQueued(); sent to an instance of Job; returns TRUE. JobSequence allQueuedJobs (); or Job findQueuedJobNamed (in string jobName); sent to instance of JobSupervisor. |
| STOPPED | In this state the Job has stopped execution. This is a substate of Finished. | boolean isStopped(); sent to an instance of Job; returns TRUE.<br>JobSequence allFinishedManagedJobs (); sent to instance of JobSupervisor will provide some indication. |
| STOPPING | In this state the Job is being stopped. Execution is not intended to continue.<br>STOPPING represents an ordered termination of the Job Activities. Job Activities not completed before stopping may or may not be performed, depending on the implementation. | boolean isStopping(); sent to an instance of Job; returns TRUE. |
| FINISHED | This is a parent state representing that the Job has finished execution, through either successful execution, abort, or stop. This is the superstate of Completed, Aborted, Stopped, and Canceled. | boolean isFinished(); sent to an instance of Job; returns TRUE.<br>JobSequence allFinishedJobs (); sent to instance of JobSupervisor. |

**Table 4 Job State Transitions**

| # | *Current State* | *Trigger* | *New State* | *Action* | *Comment* |
|---|---|---|---|---|---|
| 1 | Non Existent | Job creation. | QUEUED | JobStateChanged Event published by the instance of Job | None. |
| 2 | QUEUED | Internal to component. | EXECUTING | JobStateChanged Event published by the instance of Job | Job has started. |
| 3 | EXECUTING | void makePaused ( ); sent to the Job; or void PauseAllJobs ( ); sent to an instance of JobSupervisor. | PAUSING | JobStateChanged Event published by the instance of Job | Job has been told to pause. |
| 4 | PAUSING | Internal to component | PAUSED | JobStateChanged Event published by the instance of Job | Job has completed PAUSING activities. Wait for resume. |
| 5 | PAUSED | void makeExecuting ( ); sent to the Job or void resumeAllJobs (); sent to an instance of JobSupervisor. | EXECUTING | JobStateChanged Event published by the instance of Job | Restart EXECUTING from the point PAUSED at. |
| 6 | EXECUTING | Internal to component | COMPLETED | Execution of the Job was successful or completed normally. | None. |
| 7 | EXECUTING | void makeAborted ( ); sent the Job or void abortAllJobs ( ); sent to the JobSupervisor. | ABORTING | Execution of the Job stops immediately. Product or Job will be unfinished. | None. |
| 8 | PAUSE | void makeAborted ( ); sent the Job or void abortAllJobs ( ); sent to the JobSupervisor. | ABORTING | Active Job is aborted immediately. | None. |
| 9 | ABORTING | Internal to component | ABORTED | JobStateChanged Event published by the instance of Job | Job has completed aborting activities. |
| 10 | STOPPING | void makeAborted ( ); sent the Job or void abortAllJobs ( ); sent to the JobSupervisor. | ABORTING | JobStateChanged Event published by the instance of Job | None. |
| 11 | PAUSE | void makeStopped( ); sent to the Job or void stopAllJobs( ); sent to the JobSupervisor. | STOPPING | JobStateChanged Event published by the instance of Job | None. |
| 12 | STOPPING | Internal to component | STOPPED | JobStateChanged Event published by the instance of Job | Job has completed stopping activities. |
| 13 | EXECUTING | void makeStopped( ); sent to the Job or void stopAllJobs( ); sent to the JobSupervisor. | STOPPING | JobStateChanged Event published by the instance of Job | None. |
| 14 | QUEUED | void makeCanceled( ); sent to the Job. | CANCELED | JobStateChanged Event published by the instance of Job | Job has been canceled before starting execution. It cannot be restarted. |
| 15 | FINISHED | void removeFinished-Job (in Job aJob); sent to an instance of JobSupervisor. | Non Existent | JobStateChanged Event published by the instance of Job | Job has been removed from the finished queue. |

6.6.4 *JobRequestor Interface*

Module: AbstractIF

Interface: JobRequestor

Inherited Interface: Implementation Dependent

Description: In order to request work of a JobSupervisor using the requestJob operation, a component must implement the JobRequestor interface. This is a companion interface to JobSupervisor. The JobRequestor may also subscribe to the state change events of the Job, if more detail is required.

```
interface JobRequestor {
```

Exceptions: None.

Published Events: None.

Provided Services:

/* The Job has transitioned to a new state. Required for transition to Executing from Queued and for any transition to a Finished sub-state. This operation is in addition to the required JobStateChangedEvent notifications. */

```
void informJobStateChange (
        in Job aJob,
        in Job::JobState previousState,
        in Job::JobState newState)
        raises (Global::FrameworkErrorSignal);

}; // JobRequestor
```

Contracted Services: None.

Dynamic Model: None.

Scenario:

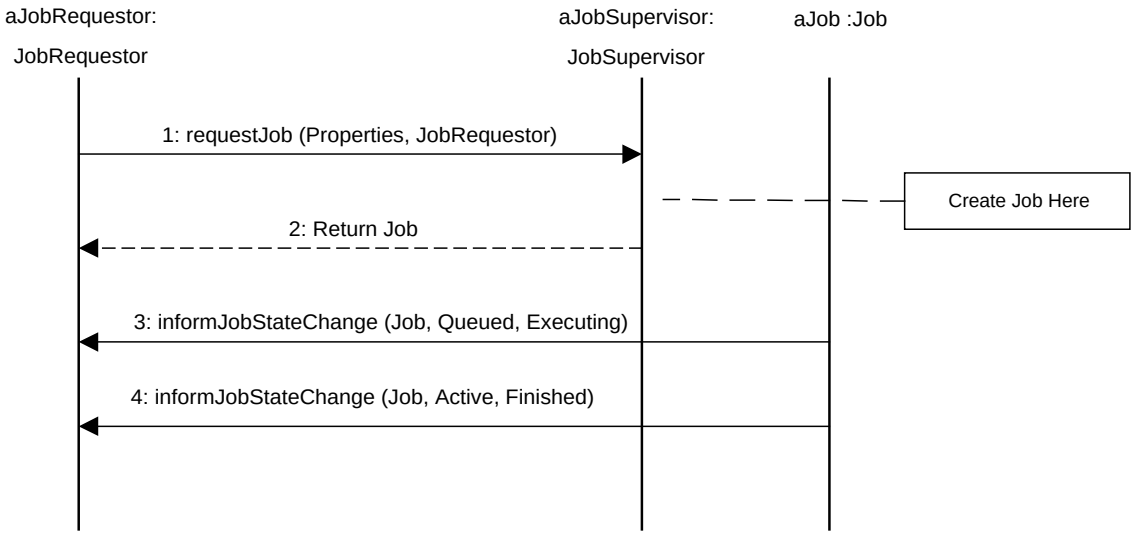


**Figure 6**
**Job Supervision Scenario**

6.6.4.1 Figure 6 shows the most basic of scenarios for Job Supervision interactions. It proceeds in this fashion:

6.6.4.1.1 The JobRequestor populates a JobSpecification then requests a Job according to that specification.

6.6.4.1.2 In response to the Job request, the JobSupervisor facilitates the creation of a Job to represent the task. A handle to the Job is returned to the JobRequestor (assuming the Job request is accepted).

6.6.4.1.3 The Job Supervision component (e.g. in the form of the Job) informs the JobRequestor when the Job begins.

6.6.4.1.4 The Job Supervision component (e.g. in the form of the Job) informs the JobRequestor when the Job has completed (assuming successful completion). It also issues events for each state change (not shown in the scenario diagram).

6.7 *Factory Component*

6.7.1 The Factory interfaces provide configuration services to specify the existence and connectivity of factory resources that constitute a factory. This includes area configuration and the registration of CIM system components, and the ability to dynamically configure a factory to enforce business policy.

6.7.1.1 These interfaces are included here to satisfy dependencies and meet needs of other CIM Framework Components. These interfaces will be moved to the Factory Operations Component in a subsequent revision ballot.

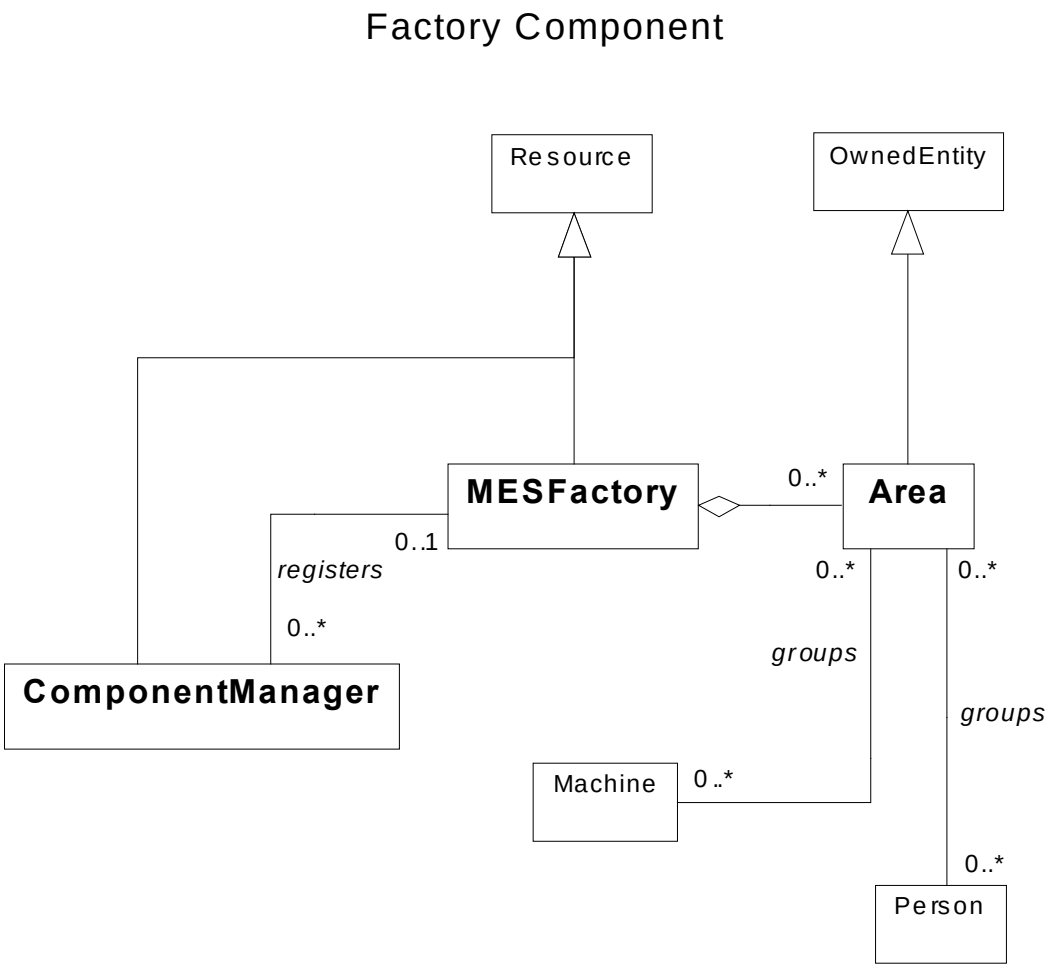


**Figure 7**
**Factory Component Information Model**

6.7.2 *Factory Declarations*

6.7.2.1 The following declarations are used by the interfaces of the Factory Component.

```
interface ComponentManager;

interface Area;

typedef sequence <ComponentManager> ComponentManagerSequence;

typedef sequence <Area> AreaSequence;

exception AreaNotFoundSignal {Area requestedArea;};

exception AreaDuplicateSignal { };

exception AreaNotAssignedSignal { };

exception AreaRemovalFailedSignal { };
```

6.7.3 *MESFactory Interface*

| | |
|---|---|
| Module: | FactoryOperations |
| Interface: | MESFactory |
| Inherited Interface: | AbstractIF::Resource |
| Description: | The MESFactory interface represents one particular factory. This instance is a composite object referring to the objects that represent factory resources, particularly CIM system components. The factory instance provides overall startup and shutdown capability. |

```
interface MESFactory : AbstractIF::Resource {
```

Exceptions: None.

Published Events:

/* MES Factory state change event definition. */

```
const string MESFactoryStateChangedSubject = "/Factory/MESFactory/StateChanged";
```

/* This enumerated type identifies the states of the MES Factory. It is used in event notifications of state changes. */

```
enum MESFactoryState {
        FactoryUndefined,
        FactoryStartingUp,
        FactoryOperating,
        FactoryGoingToStandby,
        FactoryStandby,
        FactoryShuttingDownImmediately,
        FactoryShuttingDownNormally,
        FactoryOff };

struct MESFactoryStateChangedFilters {
        Global::Property name;
        Global::Property previousState;
        Global::Property newState;
};
```

MESFactoryStateChangedFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "MESFactoryName" | string | The name of the MES Factory. |
| "PreviousState" | MESFactoryState | Previous state preceding the most recent change. |
| "NewState" | MESFactoryState | New state following the most recent change. |

```
struct MESFactoryStateChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        MESFactoryStateChangedFilters eventFilterData;
        Global::Properties eventNews;
        MESFactory aMESFactory;
};
```

/* Registration of Component Manager has changed */

```
const string ComponentManagerRegistrationChangedSubject =
"/Factory/MESFctory/ComponentManagerRegistrationChanged";
```

/* This enumerated type identifies the states of Component Manager registration. It is used in event notifications of state changes. */

```
enum RegistrationState {
        RegistrationUndefined,
        Registered,
        NotRegistered };

struct RegistrationChangedFilters {
        Global::Property MESFactoryName;
        Global::Property componentManagerName;
        Global::Property newState;
};
```

RegistrationChangedFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "MESFactoryName" | string | The name of the MES Factory. |
| "ComponentManagerName" | string | The name of the Component Manager. |
| "NewState" | RegistrationState | New state following the most recent change. |

```
struct ComponentManagerRegistrationChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        RegistrationChangedFilters eventFilterData;
        Global::Properties eventNews;
        MESFactory aMESFactory;
        ComponentManager aComponentManager;
};
```

**Provided Services:**

/* Add an area to the receiver. Returns the area. */

```
Area addArea (in Area anArea)
        raises (AreaDuplicateSignal,
        Global::FrameworkErrorSignal);
```

```
/* Remove an area from the receiver. Returns the area removed. */

Area removeArea (in Area anArea)
        raises (Global::FrameworkErrorSignal,
        AreaRemovalFailedSignal,
        AreaNotAssignedSignal);

/* Returns the factory areas */

AreaSequence allAreas ( )
        raises (Global::FrameworkErrorSignal);

/* Returns collections of various factory resources. */

EquipmentTracking::MachineSequence allMachines ( )
        raises (Global::FrameworkErrorSignal);

/* Returns a collection of the component managers for the factory. */

ComponentManagerSequence allComponentManagers ( )
        raises (Global::FrameworkErrorSignal);

/* A component informs the factory that it has completed startup. */

void informComponentManagerIsOperating (in ComponentManager aComponentManager)
        raises (Global::FrameworkErrorSignal);

/* A component informs the factory that it has completed shutdown. */

void informComponentManagerIsStopped (in ComponentManager aComponentManager)
        raises (Global::FrameworkErrorSignal);

/* Factory is requested to go to STARTING UP state. Note MESFactory inherits from the Resource interface and is
started up using the operations defined in that interface. During the startup and shutdown the factory delegates
appropriate requests to all registered components. */

void makeStartingUp ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

/* Factory is requested to specified state. */

void makeOperating ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

void makeStandby ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

void makeShuttingDownNormaly ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

void makeShuttingDownImmediately ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

void makeOff ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

/* Answer whether the factory is in the state indicated */
```

```
boolean isOutOfService ( )
        raises (Global::FrameworkErrorSignal);

boolean isOff ( )
        raises (Global::FrameworkErrorSignal);

boolean isStartingUp ( )
        raises (Global::FrameworkErrorSignal);

boolean isInService ( )
        raises (Global::FrameworkErrorSignal);

boolean isOperating ( )
        raises (Global::FrameworkErrorSignal);

boolean isShuttingDownNormaly ( )
        raises (Global::FrameworkErrorSignal);

boolean isShuttingDownImmediately ( )
        raises (Global::FrameworkErrorSignal);

boolean isGoingToStandby ( )
        raises (Global::FrameworkErrorSignal);

boolean isStandby ( )
        raises (Global::FrameworkErrorSignal);

}; // MESFactory
```

Contracted Services: None.

Dynamic Model:

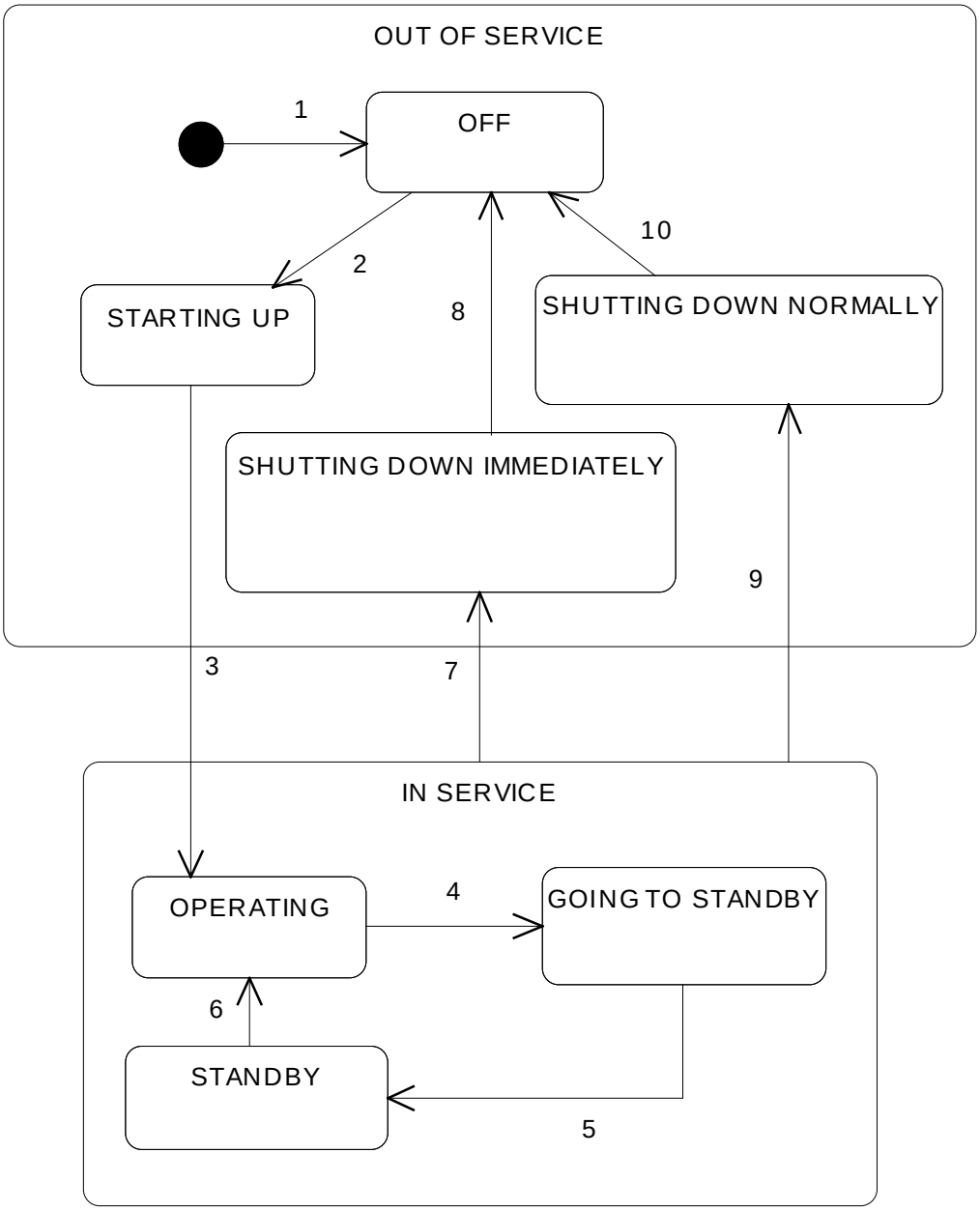


**Figure 8**
**MESFactory Dynamic Model**

**Table 5 MESFactory State Definitions and Query Table**

| *State* | *Definition* | *Query for State via* |
|---|---|---|
| OUT OF SERVICE | A superstate, inherited from Resource, encompassing the next four substate definitions. | boolean isOutOfService ( );<br>sent to the instance of MESFactory returns FALSE. |
| OFF | In this state the MESFactory has a building, machines and other resources. No CIM activities should be allowed. ComponentManagers should not be registered yet. | boolean isOff ( );<br>sent to the instance of MESFactory returns TRUE. |
| STARTING UP | In this state the MESFactory has requested startup sequences for all resources. | boolean isStartingUp ( );<br>sent to the instance of MESFactory returns TRUE. |
| SHUTTING DOWN NORMALY | In this state MESFactory resources and material are brought to a safe state in preparation for terminating in an orderly fashion. | boolean isShuttingDownNormaly ( );<br>sent to the instance of MESFactory returns TRUE. |
| SHUTTING DOWN IMMEDIATELY | In this state the MESFactory is shutting down without regard for the safe state of material or potential product and data loss. All processes are terminated immediately. | boolean isShuttingDownImmediately ( );<br>sent to the instance of MESFactory returns TRUE. |
| IN SERVICE | A superstate, inherited from Resource, encompassing the next three substate definitions. | boolean isInService ( );<br>sent to the instance of MESFactory returns TRUE. |
| OPERATING | In this state the MESFactory is able to process product. Applications are prepared to support factory operations to process product. | boolean isOperating ( );<br>sent to the instance of MESFactory returns TRUE. |
| GOING TO STANDBY | In this state the MESFactory is performing sequences to make the transition to STANDBY. It brings all product and equipment to a safe stopping place. | boolean isGoingToStandby ( );<br>sent to the instance of MESFactory returns TRUE. |
| STANDBY | STANDBY means nearly available for immediate use. In this state the MESFactory is idle and available; applications are able to respond to a subset of selected messages. | boolean isStandby ( );<br>sent to the instance of MESFactory returns TRUE. |

**Table 6 MESFactory State Transition Table**

| *#* | *Current State* | *Trigger* | *New State* | *Action* | *Comment* |
|---|---|---|---|---|---|
| 1 | non-existent | No CIM Framework trigger necessary | OFF | Building(s), machines and other resources are added after this transition. | The MESFactory object instance is unique in the Framework. It is the only object that must be created by the implementation. |
| 2 | OFF | void makeStartingUp ( ); sent to the instance of MESFactory. | STARTING UP | | Startup sequence performed, including delegated messages to component managers. Resource inherited interface also defines startup. |
| 3 | STARTING UP | void makeOperating ( ); sent to the instance of MESFactory. | OPERATING | MESFactory is operating event published by the instance of MESFactory. | Relevant registered components inform the factory when they have completed startup prior to startup being complete. |

| # | *Current State* | *Trigger* | *New State* | *Action* | *Comment* |
|---|---|---|---|---|---|
| 4 | OPERATING | void makeStandby ( ); sent to the instance of MESFactory. | GOING TO STANDBY | | MESFactory is requested to go to STANDBY state. |
| 5 | GOING TO STANDBY | completion of Standby transition by MESFactory. | STANDBY | MESFactory is in STANDBY state event published by the instance of MESFactory. | All component managers, machines and material are idle and in a safe state. |
| 6 | STANDBY | void makeOperating ( ); sent to MESFactory | OPERATING | void startup ( ); sent to an instance of Component-Manager. | Since standby means nearly available for immediate use, this startup transition should be minimal. |
| 7 | IN SERVICE | void makeShuttingDown Immediately ( ); sent to the instance of MESFactory. | SHUTTING DOWN IMMEDI-ATELY | void shutdownImmediate ( ); sent to all instances of registered Component Managers. | As an action, messages delegated to component managers. The Resource inherited interface also implements shutdownImmediate |
| 8 | SHUTTING DOWN IMMEDI-ATELY | void makeOff (); sent to the instance of MESFactory. | OFF | void informComponent ManagerIsStopped (in ComponentManager aComponentManager); sent to the instance of MESFactory by the Component Managers. MESFactory is off event published by the instance of MESFactory. | The MESFactory polls the ComponentManagers and Resources for completion of shutdown before the MESFactory state transitions to OFF. |
| 9 | IN SERVICE | void makeShuttingDown Normally ( ); sent to the instance of MESFactory. | SHUTTING DOWN | void shutdown Normal ( ); sent to all instances of registered Component-Managers. | As an action, messages delegated to component managers. |
| 10 | SHUTTING DOWN NORMALLY | void makeOff ( ); sent to the instance of MESFactory. | OFF | void informComponent ManagerIsStopped (in ComponentManager aComponentManager); sent to the instance of MESFactory by the Component Managers. MESFactory is off event published by the instance of MESFactory. | The MESFactory polls the ComponentManagers and Resources for completion of shutdown before the MESFactory state transitions to OFF. |

6.7.4 *Area Interface*

Module: FactoryOperations

Interface: Area

Inherited Interface: OwnedEntity

Description: Area is the interface corresponding to a physical or logical grouping of factory resources (the complement of machines and/or personnel assigned to it). Area may represent a singular entity or it may represent a collection of other Areas. For example, an Area may represent an entire facility for maintenance purposes, or an Area may represent a processing area such as a “bay” which is comprised of “zones.”

The association between an Area and its composite Areas may be hierarchical or there may simply be a collection of peer Areas without any explicit or implicit relationship.

Area may or may not be an optional construct, depending on such issues as security.

```
interface Area {
```

Exceptions: None.

Published Events:

/* Area configuration changed event definition. */

```
const string AreaConfigurationChangedSubject =
"/Factory/Area/AreaConfigurationChanged";
```

/* This enumerated type identifies the types of configuration changes for Areas. It is used in event notifications of state changes. */

```
enum AreaChangeType {
        MachinesChanged,
        PersonsChanged,
        SubAreaChanged };

struct AreaConfigurationChangedFilters {
        Global::Property name;
        Global::Property changeType;
};
```

AreaConfigurationChangedFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "Name" | string | The name of the Area. |
| "ChangeType" | AreaChangeType | The type of change to the Area. |

```
struct AreaConfigurationChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        AreaConfigurationChangedFilters eventFilterData;
        Global::Properties eventNews;
        Area anArea;
};
```

Provided Services:

/* Answer the Area to which this Area is associated. If no membership has been established, nil is returned. */

```
Area getSuperArea ( )
        raises (Global::FrameworkErrorSignal);
```

/* Get the unique identifier for the Area. */

```
string getAreaIdentifier ( )
        raises (Global::FrameworkErrorSignal);
```

/* Adds a machine to the receiver. Returns the machine added. */

```
EquipmentTracking::Machine addMachine (in EquipmentTracking::Machine aMachine)
        raises (Global::FrameworkErrorSignal,
        EquipmentTracking::MachineDuplicateSignal);
```

/* Create an association between an Area and the Area to which it belongs. The service will add the Area indicated by the argument to the receiver's set of subareas. The service will also update the superarea for the argument. The service returns the argument. */

```
Area addSubArea (in Area anArea)
        raises (Global::FrameworkErrorSignal,
        AreaDuplicateSignal);
```

/* Adds a person to the receiver. Returns the person added. */

```
Labor::Person addPerson (in Labor::Person aPerson)
        raises (Global::FrameworkErrorSignal,
        Labor::PersonDuplicateSignal);
```

/* Remove the association between an Area and the Area to which it belongs. The service will remove the Area indicated by the argument from the receiver's set of subareas. The service will also nullify membership (ownership) for the argument. */

```
void removeSubArea (in Area anArea)
        raises (Global::FrameworkErrorSignal,
        AreaNotAssignedSignal,
        AreaRemovalFailedSignal);
```

/* Removes a machine from the receiver. */

```
void removeMachine (in EquipmentTracking::Machine aMachine)
        raises (Global::FrameworkErrorSignal,
        EquipmentTracking::MachineNotAssignedSignal,
        EquipmentTracking::MachineRemovalFailedSignal);
```

/* Removes a person from the receiver. */

```
void removePerson (in Labor::Person aPerson)
        raises (Global::FrameworkErrorSignal,
        Labor::PersonNotAssignedSignal,
        Labor::PersonRemovalFailedSignal);
```

/* Set the unique identifier for the Area. */

```
void setAreaIdentifier (in string identifier)
        raises (Global::FrameworkErrorSignal,
        Global::DuplicateIdentifierSignal);
```

/* Returns the set of subareas associated with this Area, that is, the Areas "contained" within this higher-level Area. If no membership has been established, an empty set is returned. */

```
AreaSequence allSubAreas ( )
        raises (Global::FrameworkErrorSignal);
```

/* Returns the receiver's set of process machines. */

```
EquipmentTracking::MachineSequence allMachines ( )
        raises (Global::FrameworkErrorSignal);
```

/* Returns the receiver's set of persons */

```
Labor::PersonSequence allPersons ( )
        raises (Global::FrameworkErrorSignal);
```

```
}; // Area
```

Contracted Services: None.

Dynamic Model: None.

6.7.5 *Component Manager Interface*

Module: FactoryOperations

Interface: ComponentManager

Inherited Interface: Resource

Description: The ComponentManager is an abstract interface that supports the registration and control (enabling/disabling) of a component's interface and for managing the resources in its domain.

Exceptions: None.

Published Events:

```
interface ComponentManager : Resource {

/* Component Manager state has changed */

const string ComponentManagerStateChangedSubject =
"/Factory/ComponentManager/ComponentManagerStateChanged";

/* This enumerated type identifies the states of Component Managers. It is used in event notifications of state changes. */

enum ComponentManagerState {
        ComponentManagerUndefined,
        ComponentManagerStopped,
        ComponentManagerStartingUp,
        ComponentManagerShuttingDown };

struct ComponentManagerStateChangedFilters {
        Global::Property name;
        Global::Property previousState;
        Global::Property newState;
};
```

ComponentManagerStateChangedFilters Properties:

| *Name* | *Value Type* | *Description* |
|---|---|---|
| "Name" | string | The name of the ComponentManager. |
| "PreviousState" | ComponentManagerState | Previous state prior to the most recent transition. |
| "NewState" | ComponentManagerState | New state following the most recent transition. |

```
struct ComponentManagerStateChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        ComponentManagerStateChangedFilters eventFilterData;
        Global::Properties eventNews;
        ComponentManager aComponentManager;
};
```

Provided Services:

/* This operation causes the component to do its portion of the registration interchange with the factory indicated by the argument. */

```
void makeRegistered (in MESFactory aFactory)
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* This operation causes the component to remove its registration from the factory. */

```
void makeNotRegistered (in MESFactory aFactory)
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* This operation causes a registered component to perform its startup sequence. Each manager gets itself to the point where it is capable of interacting with other components. When it is ready to support all services defined in the interface, the component manager tells the factory that component startup is complete. */

```
void makeStartingUp ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* This operation causes the component to perform its shutdown sequence and then enter the state STOPPED. During shutting down activities, time is allotted to bringing the resources of a component to a safe stopping condition. The component manager tells the factory that component shutdown is complete. */

```
void makeShuttingDown ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* This operation causes the component to go into the state STOPPED without regard to data loss or the stopping condition of resources or material. There is no communication with the factory. */

```
void makeStopped ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* This operation causes a registered component to go into the state IN SERVICE from the state STOPPED. */

```
void makeInService ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);
```

/* Answer whether the status of the component is that indicated. */

```
boolean isStopped ( )
        raises (Global::FrameworkErrorSignal);

boolean isStartingUp ( )
        raises (Global::FrameworkErrorSignal);

boolean isShuttingDown ( )
        raises (Global::FrameworkErrorSignal);

boolean isNotRegistered ( )
        raises (Global::FrameworkErrorSignal);

boolean isRegistered ( )
        raises (Global::FrameworkErrorSignal);

}; // ComponentManager
```

Contracted Services: None.

Dynamic Model:

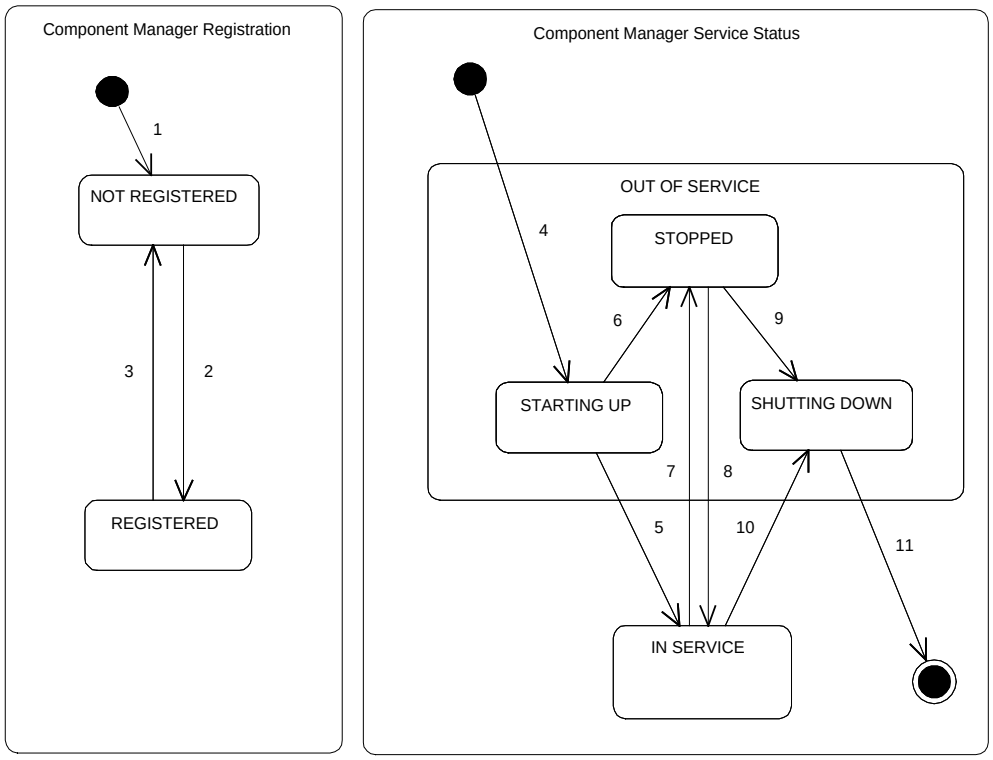


**Figure 9**
**Component Manager Dynamic Model**

**Table 7 ComponentManager State Definitions and Query Table**

| *State* | *Definition* | *Query for State via* |
|---|---|---|
| NOT REGISTERED | Component is not registered with a factory and is not connected to the system. | boolean isNotRegistered ( ); sent to the instance of ComponentManager returns TRUE. |
| REGISTERED | Component is registered with a factory and is connected to the system. | boolean isRegistered ( ); sent to the instance of ComponentManager returns TRUE. |
| STOPPED | Component is not able to provide services. | boolean isStopped ( ); sent to the instance of ComponentManager returns TRUE. |
| STARTING UP | Component is performing a startup sequence. | boolean isStartingUp ( ); sent to the instance of ComponentManager returns TRUE. |
| IN SERVICE | Component is capable of interacting with other components. | boolean isInService ( ); sent to the instance of ComponentManager returns TRUE. |
| SHUTTING DOWN | Component is performing shutdown sequence. | boolean isShuttingDown ( ); sent to the instance of ComponentManager returns TRUE. |
| OUT OF SERVICE | Component is not able to provide services. | None. |

**Table 8 ComponentManager State Transitions**

| # | *Current State* | *Trigger* | *New State* | *Action* | *Comment* |
|---|---|---|---|---|---|
| 1 | Non Existent | No CIM Framework trigger necessary. | NOT REGISTERED | Instance creation is done by the specialization. | ComponentManager is an abstract inherited interface designed to provide common behavior for specializations. |
| 2 | NOT REGISTERED | makeRegistered() sent to the instance of ComponentManager. | REGISTERED | Component manager registers with a Factory instance via Trader. | None. |
| 3 | REGISTERED | makeNotRegistered() sent to the instance of ComponentManager. | NOT REGISTERED | Component manager removes registration from the Factory via Trader. | None. |
| 4 | Unknown | makeStartingUp() sent to the instance of ComponentManager. | STARTING UP | Component manager is requested to startup. | Implementation can use startup() or makeStartingUp() services. |
| 5 | STARTING UP | Internal to component. | IN SERVICE | Report to Factory that Component is able to provide services, via events. | Component manager has finished its startup sequence. |
| 6 | STARTING UP | Internal to component. | STOPPED | None. | Component Manager could not complete startup procedure and is stopped for further corrective action. |
| 7 | IN SERVICE | makeStopped() sent to the instance of Component Manager. | STOPPED | Component will also go out of service. | ComponentManager is stopped and it may be resumed as needed. |
| 8 | STOPPED | makeInService() sent to the instance of Component Manager. | IN SERVICE | Component Manager has resumed operations and is in service for execution. | None. |
| 9 | STOPPED | makeShuttingDown() sent to the instance of ComponentManager. | SHUTTING DOWN | As an action it reports to the Factory that resources and material are brought to a safe state. | The component manager receives “shutdown normal” message. |
| 10 | IN SERVICE | makeShuttingDown() sent to the instance of ComponentManager. | SHUTTING DOWN | As an action it reports to the Factory that resources and material are brought to a safe state. | The component manager receives “shutdown normal” message. |
| 11 | SHUTTING DOWN | makeNotRegistered() sent to the instance of Component Manager. | Non Existent | The Component Manager is unregistered. | None. |

# APPENDIX 1
# FULL IDL SPECIFICATION

NOTE: The material in this appendix is an official part of SEMI E97 and was approved by full letter ballot procedures on October 21 and December 15, 1999 by the North American Regional Standards Committee.

```
module CIMFW {
#ifndef _CIMFW_GLOBAL_
#define _CIMFW_GLOBAL_

    module Global {

            typedef string Identifier;

            typedef unsigned long Flags;

            struct NamedValue{
                Identifier name;
                any argument;
                long len;
                Flags arg_modes;
            };

            typedef NamedValue NameValue;

            typedef sequence <NamedValue> NameValueSequence;

            typedef string PropertyName;

            struct Property{
                PropertyName Property_name;
                any Property_value;
            };

            typedef sequence <Property> Properties;

            typedef sequence <string> StringSequence;

            typedef string Unit;

            typedef string Units;

            typedef sequence <any> anySequence;

            typedef sequence <long> longSequence;

            enum PriorityOfEvent { Low,
                Medium,
                High,
                Alarm };

            enum LifecycleState { Undefined,
                Created,
                Deleted,
                Moved,
                Copied };

            enum ReservationState { UndefinedReservationState,
                Reserved,
                UnReserved };

            enum E10State { E10Productive,
                E10Standby,
                E10Engineering,
                E10ScheduledDowntime,
                E10UnscheduledDowntime,
                E10NonscheduledTime };
```

```
            struct ulonglong
                {unsigned long low ;
                 unsigned long high ;
            } ;

            typedef ulonglong TimeT ;

            typedef TimeT TimeStamp;

            typedef sequence <TimeStamp> TimeStampSequence;

            struct IntervalT {
                TimeT lower_bound;
                TimeT upper_bound;
            };

            typedef IntervalT TimeWindow;

            typedef TimeT Duration;

            struct ResourceSchedule {
                TimeStamp plannedStartTime;
                TimeStamp plannedEndTime;
                TimeStamp actualStartTime;
                TimeStamp actualEndTime;
            };

            typedef sequence <ResourceSchedule> ResourceScheduleSequence;

            exception NotFoundSignal { string errorMessage; };

            exception DuplicateIdentifierSignal {
                string errorMessage;
                string duplicateIdentifier; };

            exception InvalidStateTransitionSignal {
                string errorMessage; };

            exception SetValueOutOfRangeSignal {
                string errorMessage; };

            exception TimePeriodInvalidSignal {
                string errorMessage; };

            exception InvalidPropertyNameSignal {};

            exception PropertyNotFoundSignal {};

            exception UnsupportedPropertySignal {};

            exception ReadOnlyPropertySignal {};

            exception FrameworkErrorSignal {
                string errorMessage;
                unsigned long errorCode;
                any errorInformation;};

            const unsigned long NOT_IMPLEMENTED = 0;

            const unsigned long IMPLEMENTED_BY_SUBCLASS = 1;

            const unsigned long UNKNOWN_EXCEPTION = 2;

            const unsigned long COMPLETION_UNKNOWN = 3;

    }; // module Global
#endif // _CIMFW_GLOBAL_

    module EquipmentTracking {

        interface Machine {}; // Stub
```

```
            typedef sequence <Machine> MachineSequence;

            exception MachineDuplicateSignal { };

            exception MachineNotAssignedSignal { };

            exception MachineRemovalFailedSignal { };

    }; // module EquipmentTracking

    module Labor {

        interface Person {}; // Stub

            typedef sequence <Person> PersonSequence;

            exception PersonDuplicateSignal { };

            exception PersonNotAssignedSignal { };

            exception PersonRemovalFailedSignal { };

    }; // module Labor

#ifndef _CIMFW_ABSTRACT_IF_
#define _CIMFW_ABSTRACT_IF_

    module AbstractIF {

        interface Resource;

        interface Material;

        interface MaterialGroup;

        interface JobSupervisor;

        interface Job;

        interface JobRequestor;

            typedef sequence <Resource> ResourceSequence;

            typedef sequence <Material> MaterialSequence;

            typedef sequence <MaterialGroup> MaterialGroupSequence;

            typedef sequence <Job> JobSequence;

            typedef sequence <JobSupervisor> JobSupervisorSequence;

        interface NamedEntity {

            void setName (in string name)
                raises (Global::FrameworkErrorSignal);

            string getName ( )
                raises (Global::FrameworkErrorSignal);

            boolean isNamed (in string testName)
                raises (Global::FrameworkErrorSignal);

        }; // NamedEntity

        interface OwnedEntity : NamedEntity {

            void setOwner (in NamedEntity owner)
                raises (Global::FrameworkErrorSignal);

            NamedEntity getOwner ( )
```

```
            raises (Global::FrameworkErrorSignal);

    }; // OwnedEntity

    interface Resource : OwnedEntity {

        typedef sequence <Resource> ResourceSequence;

        void startUp ( )
            raises (Global::FrameworkErrorSignal);

        void shutdownNormal ( )
            raises (Global::FrameworkErrorSignal);

        void shutdownImmediate ( )
            raises (Global::FrameworkErrorSignal);

        string resourceLevel ( )

            raises (Global::FrameworkErrorSignal);

        string nameQualifiedTo (in string resourceLevel)
            raises (Global::FrameworkErrorSignal);

        ResourceSequence subresources ( )
            raises (Global::FrameworkErrorSignal);

        boolean isInService ( );

        boolean isOutOfService ( );

    }; // Resource

    interface Material : NamedEntity {

        string getIdentifier ( )
            raises (Global::FrameworkErrorSignal);

        void setIdentifier (in string identifier)
            raises (Global::FrameworkErrorSignal,
            Global::DuplicateIdentifierSignal);

        MaterialGroupSequence materialGroups ( )
            raises (Global::FrameworkErrorSignal);

        boolean isMemberOf (in MaterialGroup aMaterialGroup)
            raises (Global::FrameworkErrorSignal);

    }; // Material

    interface MaterialGroup : NamedEntity {

        exception DuplicateMaterialSignal {Material aMaterial;};

        exception DuplicateMaterialGroupSignal {Material
        aMaterialGroup;};

        exception MaterialRemovalFailedSignal {Material aMaterial;};

        exception MaterialGroupRemovalFailedSignal {
            MaterialGroup  aMaterialGroup;};

        string getIdentifier ( )
            raises (Global::FrameworkErrorSignal);

        void setIdentifier (in string identifier)
            raises (Global::FrameworkErrorSignal,
            Global::DuplicateIdentifierSignal);

        void addMaterials (in MaterialSequence aMaterialSequence)
            raises  (Global::FrameworkErrorSignal,
```

```
        DuplicateMaterialSignal);

    void addMaterial (in Material aMaterial)
        raises (Global::FrameworkErrorSignal,
        DuplicateMaterialSignal);

    void removeMaterial (in Material aMaterial)
        raises  (Global::FrameworkErrorSignal,
        MaterialRemovalFailedSignal,
        Global::NotFoundSignal);

    MaterialSequence removeAllMaterials ( )
        raises (Global::FrameworkErrorSignal);

    void addMaterialGroup (in MaterialGroup aMaterialGroup)
        raises (Global::FrameworkErrorSignal,
        DuplicateMaterialGroupSignal);

    void removeMaterialGroup (in MaterialGroup aMaterialGroup)
        raises (Global::FrameworkErrorSignal,
        MaterialGroupRemovalFailedSignal,
        Global::NotFoundSignal);

    MaterialSequence allMaterials ( )
        raises (Global::FrameworkErrorSignal);

    MaterialGroupSequence allMaterialGroups ( )
        raises (Global::FrameworkErrorSignal);

    long size ( )
        raises  (Global::FrameworkErrorSignal);

}; // MaterialGroup

interface JobSupervisor : Resource {

    typedef Global::NameValueSequence Results;

    exception JobRejectedSignal { string errorMessage; };

    exception JobNotFoundSignal {
        string errorMessage;
        string missingJobName; };

    const string JobLifecycleSubject
    =”/JobSupervision/JobSupervisor/JobLifecycle”;

    struct JobLifecycleFilters {
        Global::Property name;
        Global::Property lifecycle;
    };

    struct JobLifecycleEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        JobLifecycleFilters eventFilterData;
        Global::Properties eventNews;
        Job aJob; // on Delete, aJob is nil
    };

    Job requestJob (
        in Global::Properties aJobSpecification,
        in JobRequestor aJobRequestor)
        raises (Global::FrameworkErrorSignal,
        JobRejectedSignal);

    Results runJob (
        in Global::Properties aJobSpecification)
        raises (Global::FrameworkErrorSignal,
        JobRejectedSignal);
```

```
boolean canPerform (
    in Global::Properties aJobSpecification)
    raises (Global::FrameworkErrorSignal);

void pauseAllJobs ()
    raises (Global::FrameworkErrorSignal);

void resumeAllJobs ()
    raises (Global::FrameworkErrorSignal);

void abortAllJobs ()
    raises (Global::FrameworkErrorSignal);

void stopAllJobs ()
    raises (Global::FrameworkErrorSignal);

void removeFinishedJob (in Job aJob)
    raises (Global::FrameworkErrorSignal);

Job findJobNamed (in string jobName)
    raises (Global::FrameworkErrorSignal,
    JobNotFoundSignal);

Job findQueuedJobNamed (in string jobName)

    raises (Global::FrameworkErrorSignal,
    JobNotFoundSignal);

Job findActiveJobNamed (in string jobName)
    raises (Global::FrameworkErrorSignal,
    JobNotFoundSignal);

Job findCancelledJobNamed (in string jobName)
    raises (Global::FrameworkErrorSignal,
    JobNotFoundSignal);

Job findFinishedJobNamed (in string jobName)
    raises (Global::FrameworkErrorSignal,
    JobNotFoundSignal);

JobSequence allJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allQueuedJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allCanceledJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allActiveJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allExecutingJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allPausingJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allPausedJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allStoppingJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allAbortingJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allFinishedJobs ( )
    raises (Global::FrameworkErrorSignal);

JobSequence allStoppedJobs ( )
```

```
        raises (Global::FrameworkErrorSignal);

    JobSequence allAbortedJobs ( )
        raises (Global::FrameworkErrorSignal);

    JobSequence allCompletedJobs ( )
        raises (Global::FrameworkErrorSignal);

}; // JobSupervisor

interface Job : OwnedEntity {

    const string JobStateChangedSubject =
    "/JobSupervision/Job/StateChanged";

    enum JobState {
        JobUndefined,
        JobCreated,
        JobQueued,
        JobActive,
        JobExecuting,
        JobNotPaused,
        JobPausing,
        JobPaused,
        JobNotStopping,
        JobStopping,

        JobNotAborting,
        JobAborting,
        JobFinished,
        JobCanceled,
        JobCompleted,
        JobStopped,
        JobAborted };

    struct JobStateChangedFilters {
        Global::Property name;
        Global::Property previousState;
        Global::Property newState;
    };

    struct JobStateChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        JobStateChangedFilters eventFilterData;
        Global::Properties eventNews;
        Job aJob;
    };

    const string JobDeadlineCannotBeMetSubject =
        "/JobSupervision/Job/DeadlineCannotBeMet";

    struct JobDeadlineCannotBeMetFilters {
        Global::Property name;
        Global::Property deadline;
    };

    struct JobDeadlineCannotBeMetEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        JobDeadlineCannotBeMetFilters eventFilterData;
        Global::Properties eventNews;
        Job aJob;
    };

    const string JobDeadlineChangedSubject =
        "/JobSupervision/Job/DeadlineChanged";

    struct JobDeadlineChangedFilters {
        Global::Property name;
        Global::Property previousDeadline;
```

```
    Global::Property newDeadline;
};

struct JobDeadlineChangedEvent {
    string eventSubject;
    Global::TimeStamp eventTimeStamp;
    JobDeadlineChangedFilters eventFilterData;
    Global::Properties eventNews;
    Job aJob;
};

exception InvalidPropertyNameSignal {};

exception PropertyNotFoundSignal {};

exception UnsupportedPropertySignal {};

exception ReadOnlyPropertySignal {};

JobRequestor getJobRequestor ()
    raises (Global::FrameworkErrorSignal);

Global::Property getJobProperty (
    in Global::PropertyName aPropertyName)

    raises (Global::FrameworkErrorSignal,
    InvalidPropertyNameSignal,
    PropertyNotFoundSignal);

void setJobProperty (
    in Global::Property aProperty)
    raises (Global::FrameworkErrorSignal,
    Global::SetValueOutOfRangeSignal,
    InvalidPropertyNameSignal,
    UnsupportedPropertySignal,
    ReadOnlyPropertySignal);

boolean areJobResultsAvailable()
    raises( Global::FrameworkErrorSignal);

JobSupervisor::Results getJobResults()
    raises( Global::FrameworkErrorSignal);

void makePaused ( )
    raises (Global::FrameworkErrorSignal,
    Global::InvalidStateTransitionSignal);

void makeExecuting ( )
    raises (Global::FrameworkErrorSignal,
    Global::InvalidStateTransitionSignal);

void makeCanceled ( )
    raises (Global::FrameworkErrorSignal,
    Global::InvalidStateTransitionSignal);

void makeStopped ( )
    raises (Global::FrameworkErrorSignal,
    Global::InvalidStateTransitionSignal);

void makeInService ( )
    raises (Global::FrameworkErrorSignal,
    Global::InvalidStateTransitionSignal);

void makeAborted ( )
    raises (Global::FrameworkErrorSignal,
    Global::InvalidStateTransitionSignal);

boolean isAborting ( )
    raises (Global::FrameworkErrorSignal);

boolean isAborted ( )
```

```
                raises (Global::FrameworkErrorSignal);

            boolean isActive ( )
                raises (Global::FrameworkErrorSignal);

            boolean isCanceled ( )
                raises (Global::FrameworkErrorSignal);

            boolean isCompleted ( )
                raises (Global::FrameworkErrorSignal);

            boolean isExecuting ( )
                raises (Global::FrameworkErrorSignal);

            boolean isFinished ( )
                raises (Global::FrameworkErrorSignal);

            boolean isPausing ( )
                raises (Global::FrameworkErrorSignal);

            boolean isPaused ( )
                raises (Global::FrameworkErrorSignal);

            boolean isQueued ( )
                raises (Global::FrameworkErrorSignal);

            boolean isStopping ( )
                raises (Global::FrameworkErrorSignal);

            boolean isStopped ( )
                raises (Global::FrameworkErrorSignal);

            Global::Duration timeRemaining ( )

                raises (Global::FrameworkErrorSignal);

        }; // Job

        interface JobRequestor {

            void informJobStateChange (
                in Job aJob,
                in Job::JobState oldState,
                in Job::JobState newState)
                raises (Global::FrameworkErrorSignal);

        }; // JobRequestor

    }; // module AbstractIF

#endif // _CIMFW_ABSTRACT_IF_
#ifndef _CIMFW_FACTORY_OPERATIONS_
#define _CIMFW_FACTORY_OPERATIONS_

    module FactoryOperations {

        interface ComponentManager;

        interface Area;

            typedef sequence <ComponentManager> ComponentManagerSequence;

            typedef sequence <Area> AreaSequence;

            exception AreaNotFoundSignal {Area requestedArea;};

            exception AreaDuplicateSignal { };

            exception AreaNotAssignedSignal { };

            exception AreaRemovalFailedSignal { };
```

```
interface MESFactory : AbstractIF::Resource {

    const string MESFactoryStateChangedSubject =
    "/Factory/MESFactory/StateChanged";

    enum MESFactoryState { FactoryUndefined,
        FactoryStartingUp,
        FactoryOperating,
        FactoryGoingToStandby,
        FactoryStandby,
        FactoryShuttingDownImmediately,
        FactoryShuttingDownNormally,
        FactoryOff };

    struct MESFactoryStateChangedFilters {
        Global::Property name;
        Global::Property previousState;
        Global::Property newState;
    };

    struct MESFactoryStateChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        MESFactoryStateChangedFilters eventFilterData;
        Global::Properties eventNews;
        MESFactory aMESFactory;

    };

    struct RegistrationChangedFilters {
        Global::Property MESFactoryName;
        Global::Property componentManagerName;
        Global::Property newState;
    };

    struct ComponentManagerRegistrationChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        RegistrationChangedFilters eventFilterData;
        Global::Properties eventNews;
        MESFactory aMESFactory;
        ComponentManager aComponentManager;
    };

    Area addArea (in Area anArea)
        raises (AreaDuplicateSignal,
        Global::FrameworkErrorSignal);

    Area removeArea (in Area anArea)
        raises (Global::FrameworkErrorSignal,
        AreaRemovalFailedSignal,
        AreaNotAssignedSignal);

    AreaSequence allAreas ( )
        raises (Global::FrameworkErrorSignal);

    EquipmentTracking::MachineSequence allMachines ( )
        raises (Global::FrameworkErrorSignal);

    ComponentManagerSequence allComponentManagers ( )
        raises (Global::FrameworkErrorSignal);

    void informComponentManagerIsOperating (in ComponentManager
    aComponentManager)
        raises (Global::FrameworkErrorSignal);

    void informComponentManagerIsStopped (in ComponentManager
    aComponentManager)
        raises (Global::FrameworkErrorSignal);
```

```
    void makeStartingUp ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

    void makeOperating ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

    void makeStandby ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

    void makeShuttingDownNormaly ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

    void makeShuttingDownImmediately ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

    void makeOff ( )
        raises (Global::FrameworkErrorSignal,
        Global::InvalidStateTransitionSignal);

    boolean isOff ( )
        raises (Global::FrameworkErrorSignal);

    boolean isStartingUp ( )

        raises (Global::FrameworkErrorSignal);

    boolean isOperating ( )
        raises (Global::FrameworkErrorSignal);

    boolean isShuttingDownNormaly ( )
        raises (Global::FrameworkErrorSignal);

    boolean isShuttingDownImmediately ( )
        raises (Global::FrameworkErrorSignal);

    boolean isGoingToStandby ( )
        raises (Global::FrameworkErrorSignal);

    boolean isStandby ( )
        raises (Global::FrameworkErrorSignal);

}; // MESFactory

interface Area : AbstractIF::OwnedEntity {

    const string AreaConfigurationChangedSubject =
    "/Factory/Area/AreaConfigurationChanged";

    enum AreaChangeType {
        MachinesChanged,
        PersonsChanged,
        SubAreaChanged };

    struct AreaConfigurationChangedFilters {
        Global::Property name;
        Global::Property changeType;
    };

    struct AreaConfigurationChangedEvent {
        string eventSubject;
        Global::TimeStamp eventTimeStamp;
        AreaConfigurationChangedFilters eventFilterData;
        Global::Properties eventNews;
        Area anArea;
    };
```

```
        Area getSuperArea ( )
            raises (Global::FrameworkErrorSignal);

        string getAreaIdentifier ( )
            raises (Global::FrameworkErrorSignal);

        Area addSubArea (in Area anArea)
            raises (Global::FrameworkErrorSignal,
            AreaDuplicateSignal);

        void removeSubArea (in Area anArea)
            raises (Global::FrameworkErrorSignal,
            AreaNotAssignedSignal,
            AreaRemovalFailedSignal);

        void setAreaIdentifier (in string identifier)
            raises (Global::FrameworkErrorSignal,
            Global::DuplicateIdentifierSignal);

        AreaSequence subAreas ( )
            raises (Global::FrameworkErrorSignal);

        EquipmentTracking::Machine addMachine (in
        EquipmentTracking::Machine aMachine)

            raises (Global::FrameworkErrorSignal,
            EquipmentTracking::MachineDuplicateSignal);

        void removeMachine (
            in EquipmentTracking::Machine aMachine)
            raises (Global::FrameworkErrorSignal,
            EquipmentTracking::MachineNotAssignedSignal,
            EquipmentTracking::MachineRemovalFailedSignal);

        Labor::Person addPerson (
            in Labor::Person aPerson)
            raises (Global::FrameworkErrorSignal,
            Labor::PersonDuplicateSignal);

        void removePerson (
            in Labor::Person aPerson)
            raises (Global::FrameworkErrorSignal,
            Labor::PersonNotAssignedSignal,
            Labor::PersonRemovalFailedSignal);

        EquipmentTracking::MachineSequence machines ( )
            raises (Global::FrameworkErrorSignal);

        Labor::PersonSequence persons ( )
            raises (Global::FrameworkErrorSignal);

    }; // Area

    interface ComponentManager : AbstractIF::Resource {

        const string ComponentManagerStateChangedSubject =
        "/FactoryOperations/ComponentManager/ComponentManagerStateChang
        ed";

        enum ComponentManagerState { ComponentManagerUndefined,
            ComponentManagerStopped,
            ComponentManagerStartingUp,
            ComponentManagerShuttingDown};

        struct ComponentManagerStateChangedFilters {
            Global::Property name;
            Global::Property previousState;
            Global::Property newState;
        };

        struct ComponentManagerStateChangedEvent {
```

```
            string eventSubject;
            Global::TimeStamp eventTimeStamp;
            ComponentManagerStateChangedFilters eventFilterData;
            Global::Properties eventNews;
            ComponentManager aComponentManager;
        };

        const string ComponentManagerRegistrationChangedSubject =
        "/Factory/ComponentManager/ComponentManagerRegistrationChanged"
        ;

        enum RegistrationState {
            RegistrationUndefined,
            Registered,
            NotRegistered };

        void makeRegistered (in MESFactory aFactory)
            raises (Global::FrameworkErrorSignal,
            Global::InvalidStateTransitionSignal);

        void makeNotRegistered (in MESFactory aFactory)
            raises (Global::FrameworkErrorSignal,
            Global::InvalidStateTransitionSignal);

        void makeStartingUp ( )
            raises (Global::FrameworkErrorSignal,
            Global::InvalidStateTransitionSignal);

        void makeShuttingDown ( )
            raises (Global::FrameworkErrorSignal,
            Global::InvalidStateTransitionSignal);

        void makeStopped ( )
            raises (Global::FrameworkErrorSignal,
            Global::InvalidStateTransitionSignal);

        boolean isStopped ( )
            raises (Global::FrameworkErrorSignal);

        boolean isStartingUp ( )
            raises (Global::FrameworkErrorSignal);

        boolean isShuttingDown ( )
            raises (Global::FrameworkErrorSignal);

        boolean isNotRegistered ( )
            raises (Global::FrameworkErrorSignal);

        boolean isRegistered ( )
            raises (Global::FrameworkErrorSignal);

    }; // ComponentManager

}; // module FactoryOperations
#endif // _CIMFW_FACTORY_OPERATIONS_

}; // module CIMFW
```

**NOTICE:** SEMI makes no warranties or representations as to the suitability of the standard set forth herein for any particular application. The determination of the suitability of the standard is solely the responsibility of the user. Users are cautioned to refer to manufacturer's instructions, product labels, product data sheets, and other relevant literature respecting any materials mentioned herein. These standards are subject to change without notice.

The user's attention is called to the possibility that compliance with this standard may require use of copyrighted material or of an invention covered by patent rights. By publication of this standard, SEMI takes no position respecting the validity of any patent rights or copyrights asserted in connection with any item mentioned in this standard. Users of this standard are expressly advised that determination of any such patent rights or copyrights, and the risk of infringement of such rights, are entirely their own responsibility.

# SEMI E98-1102
# PROVISIONAL STANDARD FOR THE OBJECT- BASED EQUIPMENT MODEL (OBEM)

This provisional standard was technically approved by the Global Information and Control Committee and is the direct responsibility of the North American Information and Control Committee. Current edition approved by the North American Regional Standards Committee on November 27, 2001. Initially available at www.semi.org December 2001; to be published March 2002. Originally published February 2000; previously published March 2002.

NOTICE: The designation of SEMI E98 was updated during the 1102 publishing cycle to reflect revisions to SEMI E98.1.

## 1 Purpose

1.1 Purposes of the Object-Based Equipment Model include the following:

- Define a standard model for interfacing to multi-process equipment and other complex equipment.
- Define standard equipment components so that communications can "discuss" component-related issues.
- Provide an equipment model that can be easily integrated with SEMI E81 CIM Framework systems by connecting an OBEM-compliant equipment to a Machine object.

1.2 The purpose of the Object-Based Equipment Model (OBEM) standard is to provide definitions, services, and behavior, as seen through communications with the factory, for the common types of physical and logical objects of which equipment is typically composed, including the equipment itself. The definition of standardized objects allows the equipment to describe its makeup to the factory and provides the factory visibility into the equipment.

## 2 Scope

2.1 This is a provisional standard that defines concepts, behavior, and services to support the integration of production equipment within a semiconductor factory. The scope of this standard includes all semiconductor manufacturing equipment that provides an interface to the factory host systems. Some services may not be applicable to some material handling systems.

2.2 Sections that must be completed in order for the provisional status of OBEM to be removed include the following:

1. Section 11.2 — Access Management
2. Section 14 — OBEM Compliance

2.3 Detail standards will also be added in the future to specify OBEM mappings to different protocols such as SECS-II, CORBA IDL, and DCOM.

2.4 This standard does not purport to address safety issues, if any, associated with its use. It is the responsibility of the users of this standard to establish appropriate safety and health practices and determine the applicability of regulatory limitations prior to use.

## 3 Limitations

3.1 This standard is not intended to define the attributes, behavior, or services of systems that are aggregates of equipment, such as cells.

3.2 The decomposition of equipment into different objects is chosen by the equipment supplier to map the physical equipment to the characteristics of the objects defined by this standard.

3.3 Object-oriented technology is not required for implementations of OBEM. However, object-oriented implementations should be compatible with OBEM.

## 4 Referenced Standards

4.1 This section lists documents referenced by this standard.

4.2 *SEMI Standards*

SEMI E5 — SEMI Equipment Communications Standard 2 Message Content (SECS-II)

SEMI E10 — Standard for Definition and Measurement of Equipment Reliability, Availability, and Maintainability (RAM)

SEMI E15 — Specification for Tool Load Port

SEMI E15.1 — Provisional Specification for 300 mm Tool Load Port

SEMI E30 — Generic Model for Communications and Control of Manufacturing Equipment (GEM)

SEMI E39 — Object Services Standard: Concepts, Behavior, and Services

SEMI E40 — Standard for Processing Management

SEMI E41 — Exception Management (EM) Standard

SEMI E42 — Recipe Management Standard: Concepts, Behavior, and Message Services

SEMI E53 — Event Reporting

SEMI E54 — Sensor/Actuator Network Standard

SEMI E58 — Automated Reliability, Availability, and Maintainability Standard (ARAMS): Concepts, Behavior, and Services

SEMI E81 — Provisional Specification for CIM Framework Domain Architecture

SEMI E87 — Provisional Specification for Carrier Management (CMS)

SEMI E90 — Specification for Substrate Tracking

SEMI E94 — Provisional Specification for Control Job Management

NOTE 1: As listed or revised, all documents cited shall be the latest publications of adopted standards.

## 5 Terminology

5.1 *Abbreviations and Acronyms*

5.1.1 *AGV* — Automated Guided Vehicle

5.1.2 *AMHS* — Automated Material Handling System

5.1.3 *APC* — Advanced Process Control

5.1.4 *ARAMS* — Automated Reliability, Availability, and Maintainability Standard (SEMI E58)

5.1.5 *CIM* — Computer Integrated Manufacturing

5.1.6 *CJM* — Control Job Management

5.1.7 *CMS* — Carrier Management Standard

5.1.8 *FDC* — Fault Detection Control

5.1.9 *FIMS* — Front-Opening Interface Mechanical Standard (reference SEMI E62)

5.1.10 *FOUP* — Front-Opening Unified Pod

5.1.11 *FPD* — Flat Panel Display

5.1.12 *OBEM* — Object-Based Equipment Model

5.1.13 *OSS* — Object Services Standard (SEMI E39)

5.1.14 *PGV* — Personal Guided Vehicle

5.1.15 *R2R* — Run-to-Run Control

5.1.16 *RMS* — Recipe Management Standard (SEMI E42)

5.1.17 *SMIF* — Standard Mechanical Interface (SEMI E19)

5.1.18 *STS* — Specification for Substrate Tracking

5.2 *Definitions*

5.2.1 *abstract object type* — an object supertype that is not instantiated directly but only through one of its subtypes.

5.2.2 *actuator* — an analog or digital output device that is used to affect changes in the physical environment. Examples of actuators include mass flow controllers (MFCs) and open/closed valves.

5.2.3 *advanced process control (APC)* — techniques covering both feedforward and feedback control and automated fault detection, applied both by the equipment (in situ) and by the factory (ex situ).

5.2.4 *Automated Material Handling System (AMHS)* — a factory system used to transport and store carriers. AMHS has two major types of components: an automated transport system and one or more storage systems (stockers).

5.2.5 *automated transport system* — the component of AMHS used to transport carriers between stockers and/or production equipment.

5.2.6 *carrier* — a container with one or more fixed positions at which material may be held.

NOTE 2: Positions within a carrier may be considered as material locations owned by the carrier.

5.2.7 *clock* — a device that is used to provide real-time date and time information.

5.2.8 *container* — a durable that is used to hold other material, including other containers, for transport, storage, or shipping. Types of containers include carriers and boxes.

5.2.9 *dry run (mechanical dry run)* — a complete equipment cycle that allows the material handling and software capabilities of the equipment to be exercised without requiring full facilities hookups and without changing the physical state of the wafer. Environmental control subsystem (e.g., vacuum, nitrogen purge, particle detection) should not be affected by a dry run, and process consumables are not used.

5.2.10 *durable* — a type of material used to facilitate manufacturing but not normally consumed in the process that is removable, reusable, and trackable. Examples include containers, reticles, and pellicles.

5.2.11 *environmental subsystem* — a subsystem of equipment with the purpose of monitoring or maintaining one or more specific environmental conditions or used to handle product or durables. Environmental subsystems include vacuum systems, particle detection systems, and nitrogen purge systems.

5.2.12 *equipment* — equipment (manufacturing equipment) performs one or more of the following manufacturing functions in the factory: material process, material transport, or material storage. Equipment is made up of various parts: modules, subsystems and sensors/actuators. Equipment has at least one carrier port. Equipment communicates with the factory.

5.2.13 *equipment element* — a component of the equipment that behaves as a unit, performs work, and may or may not contain lower-level components.

5.2.14 *equipment module (module)* — a major component of equipment that contains at least one material location and performs some task on material. Equipment modules may be aggregates of equipment subsystems, i/o devices, and other modules.

5.2.15 *fault detection* — analysis of data for early detection of process faults before yield loss becomes significant.

5.2.16 *Front-Opening Unified Pod (FOUP)* — a front-opening pod with an integrated (non-removable) cassette.

5.2.17 *implementation* — the internal view of a type, class, or instance, including any non-public properties and behavior. The specific code and functionality that implements an interface. (See SEMI E81.)

5.2.18 *interface* — the external view of an object type, class, or object that defines its public properties and services without regard to the internal structure and internal behavior. (See also SEMI E81.)

5.2.19 *interface inheritance* — the construction of an interface by incremental modification of other interfaces (see implementation inheritance). (See SEMI E81.) OBEM specifies interface inheritance but not implementation inheritance.

5.2.20 *I/O device* — a general term for any type of sensor or actuator or aggregation of sensor and/or actuator.

5.2.21 *linked equipment* — two or more equipment that are physically and logically connected and function as a single installation of equipment. In this case, the individual component equipment are modeled as high-level modules of the linked equipment.

5.2.22 *load port* — The physical interface provided for the exchange of carriers with an agent of the factory (operator or automated material handling system). (Reference SEMI E15.)

5.2.23 *Manufacturing Execution System (MES)* — the factory system responsible for managing the manufacturing process, including logistics and process flow.

5.2.24 *material* — (1) any material used in, or required by, the manufacturing process. Material is classified as consumable, durable, or product. (2) an abstraction of the various types of things used during manufacturing, such as wafers, carriers, and chemicals, which require some management.

5.2.25 *material location* — a reference to a place within the equipment or an equipment component that can hold material, such as the top surface of an indexer or substrate chuck or the end effector of a substrate handler.

5.2.26 *measured value* — a value representing a measurement, with a numerical value, measurement units, and a valid range.

5.2.27 *measurement equipment* — equipment whose intended function is to measure or inspect the product and to report results. Measurement of the product is the factory's means of gaining feedback on the manufacturing process.

5.2.28 *Object-Based Equipment Model* — a model of equipment, its components, behaviors, attributes, and services, as defined by this document.

5.2.29 *object type* — a declaration (specification) that describes the common properties and behavior for a collection of objects. Types classify objects according to a common interface; classes classify objects according to a common implementation. (See also SEMI E39 and E81.)

5.2.30 *object specifier* — designates a logical path pointing to a specific instance of an object through a hierarchy of owners. See SEMI E39.

5.2.31 *Personal Guided Vehicle (PGV)* — a manually guided and operated vehicle capable of placing and removing carriers to and from a carrier port.

5.2.32 *pod* — as used in this document, a container providing environmental control, such as a SMIF or FIMS pod[1].

5.2.33 *production equipment* — process equipment and measurement equipment.

5.2.34 *process durable* — a specialized durable used by process equipment and specified by the user as part of the process, such as a reticle or burin-in board.

5.2.35 *process equipment* — equipment whose intended function is to process product, adding value to the product.

5.2.36 *product* — (1) from the equipment's perspective, product is a synonym for substrate, and includes non-product substrates such as test substrates and send-ahead substrates; (2) from the factory perspective, product is the material being processed and produced by the factory.

5.2.37 *run-to-run control* — techniques for varying settings in one run based on analysis of either incoming product (feed-forward) or product from an earlier run.

5.2.38 *sensor* — a component that responds to changes in the physical environment and provides an analog or digital input value.

5.2.39 *sensor/actuator device* — a device consisting of one or more sensors and/or actuators on the physical tool. See SEMI E54 for a precise definition of "sensor or actuator" and for a description of the internal structure of an sensor/actuator network Common Device Model definition.

5.2.40 *setup* — 1. (verb) the performance of one or more steps that puts the equipment into a known state in which it is ready to perform a specific process; 2. (noun) the state of the equipment once it has been setup.

5.2.41 *standardized object* — an object that is formally defined and compliant to SEMI E39, Object Services Standard (reference SEMI E42).

5.2.42 *storage equipment (stocker)* — equipment whose intended function is primarily to provide storage, either short-term or long-term, for carriers.

5.2.43 *subassembly* — a component of equipment that provides some limited functionality.

5.2.44 *substrate* — basic unit of material on which work is performed to create a product. Examples include wafers, die, plates used for masks, flat panels, circuit boards, and leadframes.

5.2.45 *subsystem* — a subsystem is an intelligent aggregate that behaves as a unit. A subsystem is made up of sensors and/or actuators and may contain mechanical assemblies. Subsystems may be shared by multiple modules.

5.2.46 *subtype* — an object type that is based on (derived from) another type and adds some specialization or overrides some properties or services. The type from which the subtype is derived is the supertype. For additional detail, see SEMI E39, Object Services Standard.

5.2.47 *supertype* — an object type which is used as a basis from which specializations are derived. The derived types are called subtypes. For additional detail, see SEMI E39.

5.2.48 *transport equipment* — equipment whose intended function is primarily to move material from

1 The term "pod" was originally defined as a bottom-opening pod with a SMIF interface.

one location in the factory to another location. Transport equipment may also provide short-term storage for material. (See also AMHS.)

5.2.49 *virtual sensor (synthetic sensor, derived sensor)* — one or more calculated measured values that are based on one or more sensor readings. This may include results based on neural nets, statistical analysis, etc. or may be based on a single sensor value.

5.2.50 *work* — a group of one or more substrates that undergo processing in a factory. Something that may be work in one kind of factory, such as reticles and leadframes, may have a different role in other types of factories. Work includes, but is not limited to, material intended as product. For example, it may include product substrates, test substrates, and filler substrates.

5.2.50.1 From the point of view of the equipment, work is either new (processing has not started), completed (all intended processing has been performed, terminated, or aborted, including rejected and resorted work, and no further processing is to be done) or incomplete (work in progress, on hold).

## 6 Conventions

6.1 This section defines the conventions followed by this document.

6.2 *Object Conventions* — This document conforms to the conventions for objects established by SEMI E39, including object diagrams, object terminology, and requirements for standardized objects. Accordingly, notation is based on Object Modeling Technique (OMT) as described in Object Oriented Modeling Design.[2]

6.2.1 *Formal Name of an Object* — The text capitalizes formal object name references. Similar to the way capitalization is normally used when discussing entities. When describing something in the general (like cities) lower case is used, but when a specific entity is of interest (New York City), then first letters are capitalized.

6.2.2 *Components of Complex Attributes* — The names of object attributes defined in tables are left-justified. The individual elements of complex attributes are right-justified in order of appearance below the complex attribute.

2 Rumbaugh, James, et al, Object Oriented Modeling Design, Prentice Hall, Englewood Cliffs, NJ, c1991.

6.2.3 *Names of OBEM Objects* — The names of abstract object types start with the word "Abstract" and are not intended to be directly implemented. All other objects defined in OBEM are concrete types that may be directly implemented.

6.3 *State Model Conventions*

6.3.1 This document uses the Harel state chart convention for describing dynamic operation of defined objects. The outline of this convention is described in an attachment of SEMI E30. The official definition of this convention is described in "State charts: A Visual Formalism for Complex Systems"[3].

6.3.2 The Harel convention has not the concept of state models of "creation" and "extinction" for expressing a temporary entity. The "job" described in this document is such an entity, and a copy of the same state model is used for an independent job newly created. In this document, a circle with a black circle inside is used for expressing extinction of an entity. A filled black circle denotes the entry to the state model (the entity creation).

6.3.3 Transition tables are provided in conjunction with the state diagrams to explicitly describe the nature of each state transition. A transition table contains columns for Transition number, Previous State, Trigger, New State, Actions, and Comments. The "trigger" (column 3) for the transition occurs while in the "previous" state. The "actions" (column 5) includes a combination of:

1. Actions taken upon exit of the previous state.
2. Actions taken upon entry of the new state.
3. Actions taken which are most closely associated with the transition.

6.3.3.1 No differentiation is made between these cases.

| *Num* | *Previous State* | *Trigger* | *New State* | *Actions* | *Comments* |
|---|---|---|---|---|---|
| | | | | | |

6.4 *Service Message Representation* — Services are functions or methods that may be provided by either the equipment or the host. A service message may be either a request message, which always requires a response, or a notification message, that does not require a response.

6.4.1 *Service Definition*

6.4.1.1 A service definition table defines the specific set of messages for a given service resource, as shown in the following table:

| *Message Service Name* | *Type* | *Description* |
|---|---|---|
| | | |

6.4.1.2 Type can be either "N" = Notification or "R" = Request & Response.

6.4.1.3 Notification type messages are initiated by the service provider (e.g., the equipment) and the provider does not expect to get a response from the service user. Request messages are initiated by a service user (e.g., the host). Request messages ask for data or an activity from the provider. Request messages expect a specific response message (no presumption on the message content).

6.4.2 *Service Parameter Dictionary*

6.4.2.1 A service parameter dictionary table defines the description, format and its possible value for parameters used by services, as shown in the following table:

| *Parameter Name* | *Description* | *Format: Possible Value* |
|---|---|---|
| | | |

6.4.2.2 A row is provided in the table for each parameter of a service.

3 D. Harel, "State charts: A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987.

6.4.3 *Service Message Definition*

6.4.3.1 A service message definition table defines the parameters used in a service, as shown in the following table:

| *Parameter* | *Req/Ind* | *Res/Cnf* | *Comment* |
|---|---|---|---|
| | | | |

6.4.3.2 The columns labeled REQ/IND and RSP/CNF link the parameters to the direction of the message. The message sent by the initiator is called the "Request". The receiver terms this message the "Indication" or the request. The receiver may then send a "Response" which the original sender terms the "Confirmation".

6.4.3.3 The following codes appear in the REQ/IND and RSP/CNF columns and are used in the definition of the parameters (eg., how each parameter is used in each direction):

| | |
|---|---|
| M | Mandatory Parameter — Must be given a valid value. |
| C | Conditional Parameter — May be defined in some circumstances and undefined in others. Whether a value is given may be completely optional or may depend on the value of the other parameter. |
| U | User-Defined Parameter. |
| - | The parameter is not used. |
| = | (For response only.) Indicates that the value of this parameter in the response must match that in the primary (if defined). |

6.5 *OBEM Standard Structure*

6.5.1 The remaining part of this document is organized as follows:

6.5.1.1 Section 7 contains background information to provide a context for the Object-Based Equipment Model.

6.5.1.2 Sections 8 provides an overview of two major views of the equipment: the functional view and the internal composition view.

6.5.1.3 Section 9 introduces the OBEM object model: the interface inheritance hierarchy and the rules of aggregation that together form the foundation of the OBEM model of equipment.

6.5.1.4 Section 11 defines the requirements for the component objects within the equipment interface hierarchy: and other related objects of significance not defined elsewhere.

6.5.1.5 Section 12 defines the message services used in OBEM that are not defined in other standards.

6.5.1.6 Section 13 defines the services that are required of the user (factory system, remote access, and operator).

6.5.1.7 Section 14 specifies the minimum requirements and optional capabilities for compliance to the OBEM standard.

6.5.1.8 Section 15 provides scenarios showing typical message flows during operation.

6.5.2 Additional sections are provided as related information: examples and additional material that are not part of the standard itself. These include models for linked litho, 300 mm equipment, the relationship of OBEM and the CIM Framework, and representations of date and time.

## 7 Background

7.1 Both modern manufacturing processes and modern manufacturing equipment are increasingly complex. A single installation of equipment may have hundreds or thousands of sensors and actuators. In order to manage this complexity, better methods of referencing the internal components of equipment are needed. Use of the object paradigm provides a means for the equipment to describe its internal composition to the factory in a natural way.

7.2 Definition of standardized objects allows the factory to be specific about its requirements and its need for information.

7.3 *Computer Manufacturing Integration Business Goals*

7.3.1 The intent of this section is to provide a context for, and insight into, those requirements of industries such as semiconductor and flat panel display (FPD) manufacturing businesses that affect the object-based equipment model.

7.3.2 The primary purpose of computer integrated manufacturing (CIM) technologies is to improve factory productivity.[4] Other inter-related secondary CIM business goals are listed below.

- Maximize product yields (line/mechanical yield).
- Maximize device yields (electrical/functional yield).
- Maximize total factory product substrate throughput.
- Increase individual equipment product substrate throughput.
- Reduce product variability.
- Reduce process variability.
- Optimize ability to center processes in a "sweet spot".
- Reduce the use of non-product substrates.
- Reduced time to utilization for equipment (i.e., the time to install, qualify, characterize and ramp production).
- Increase the usability, accuracy, and reliability of data used for metrics.
- These business goals can be met by addressing certain concrete objectives, which are listed below.

7.4 *OBEM Functional Objectives*

7.4.1 OBEM will standardize specific functional capabilities to be implemented on semiconductor/FPD and other manufacturing equipment, providing a hierarchical view of equipment for effective factory integration.

7.4.2 The OBEM functional objectives are as follows:

- Manage material into and through the equipment.
- Manage the association of the process instructions with the material.
- Report data associated with the equipment, the process, and the material.
- Facilitate equipment performance monitoring.

7.4.3 These OBEM functional objectives, individually and collectively, can be shown to directly address the overall business goals:

7.4.4 The Object Based Model objective directly affects the ability to implement most of the other objectives, especially in the case of highly modular equipment.

7.4.5 Equipment performance monitoring has the effect of improving product variability, device yield and can reduce the need for non-product test substrates. It can also provide a means of targeting a specific process window to improve device characteristics such as speed.

7.4.6 Management of the association of process instructions with the material can reduce scrap due to misprocessing, thus improving product yield. The material management objectives impact on the throughput of individual equipment and the total factory throughput.

7.5 *Relevant Factory Environment* — Equipment must support a variety of different factory environments. This is necessary because factory business practices and factory configurations vary not only from company to company but also from one facility within a company to another. Items will be added to this section as their relevance becomes apparent.

7.5.1 *Material Handling Systems* — Material may be loaded and unloaded manually by a fab technician or it may be loaded and unloaded using semi-automated and automated transport systems. Types of systems include:

- Automated Guided Vehicles (AGV),
- Personal Guided Vehicles (PGV),
- Overhead Transport Systems (OTS), including Overhead Hoist Transport (OHT), and
- Fixed Arm Robots.

7.5.2 *Containers* — Containers may be open (e.g., cassettes) or closed (pods, including reticle pods). Pods may be bottom-opening (SMIF), with a removable cassette, or front-opening (FIMS), which may have either a removable cassette or an integrated (non-removable) cassette (FOUP).

7.5.3 *Factory Interface* — The equipment must be able to support different levels of automation, including:

4 For a more detailed discussion and list, see the Guidance and Guideline documents at http://www.sematech.org/public/docubase/abstract/tech-30.htm

- stand-alone (with no connection to the factory systems),
- fully on-line and operated locally (by the operator),
- fully on-line and operated remotely (by the factory systems), and
- fully on-line and able to support and coordinate interactions from multiple factory users and systems at the same time.

## 8 Equipment Overview

8.1 The Object-Based Equipment Model defines the objects that are generic components of equipment as well as the object representing the equipment itself. OBEM does not dictate the makeup of equipment. Through support of OBEM, the equipment is able to describe its own makeup to the factory. However, OBEM does require certain visibility and access to those parts of the equipment that control and/or monitor the environment or the location of the product.

8.2 Two view areas are of importance: the functional view of the equipment and the internal composition view of the equipment.

8.3 *Functional View of Equipment* — From a functional view, equipment is internally composed of logical subsystems with different areas of responsibility that are at different levels within a control hierarchy, as illustrated in Figure 1. There are three general levels. Equipment Control is at the highest level, both responsible for, and representing, the equipment as an integrated whole. The middle level provides management of specific areas, while the lowest level of functionality has specific time-critical responsibilities and handles all direct interaction with the equipment's I/O (sensors and actuators). The third level is below the factory level of visibility and is discussed here for completeness.

NOTE 3: This is not intended to represent the design of an actual implementation.

8.3.1 The functional areas are discussed below in alphabetical order.

8.3.2 *Access Management* — Access Management is responsible for communications with the factory, including factory computers, local and remote operators, third party systems, and alternate users (desktop access by process engineers, maintenance personnel, supplier remote diagnostics, etc.) Communications with the local operator include input devices (such as keyboards, wands, buttons, and optical character readers) and display devices (console, light pole, and LCD panel) as well as interpretation of operator requests.

8.3.3 *Communications Link* — Communications Link is responsible for low-level communications, including establishing a connection with a communications partner, sending messages, and receiving messages.

8.3.4 *Date/Time Management* — Date/Time Management is responsible for maintaining an accurate date and time-of-day, and for providing current date/time information to the rest of the system. This may include maintenance of regular time-based scheduling.

8.3.5 *Environment Control* — Environment Control is responsible for maintaining the internal environment according to the equipment's specifications. While Process Control is specific to a process and recipe, other monitoring activities may be required regardless of whether the equipment is processing or idle. Such activities include monitoring for particles, humidity, or temperature.

8.3.6 *Equipment Control* — Equipment Control is the supervisory level with overall high-level control. Equipment Control represents the entire equipment as an integrated whole to the factory and represents the decision-making authority within the equipment.

8.3.7 *Event Management* — Events continually occur in all equipment states. A variety of these events are of interest to the factory, including those events that generate a change of state in any standardized object. The factory requires notification when selected events occur, and in many cases, requires reports of the values of specified information at the time that the event occurred. Event Management is responsible for tracking those events and the reports associated with those events.

8.3.8 *Exception Management* — Exception Management is responsible for determining the proper response to an action or operation that the equipment was unable to perform which raised an exception condition. It prompts notification to all affected components, including internal components and currently connected users. In some cases, the proper action may have to be resolved by the user. Exception Management is a high level activity that is in addition to underlying hardware and software interlocks.

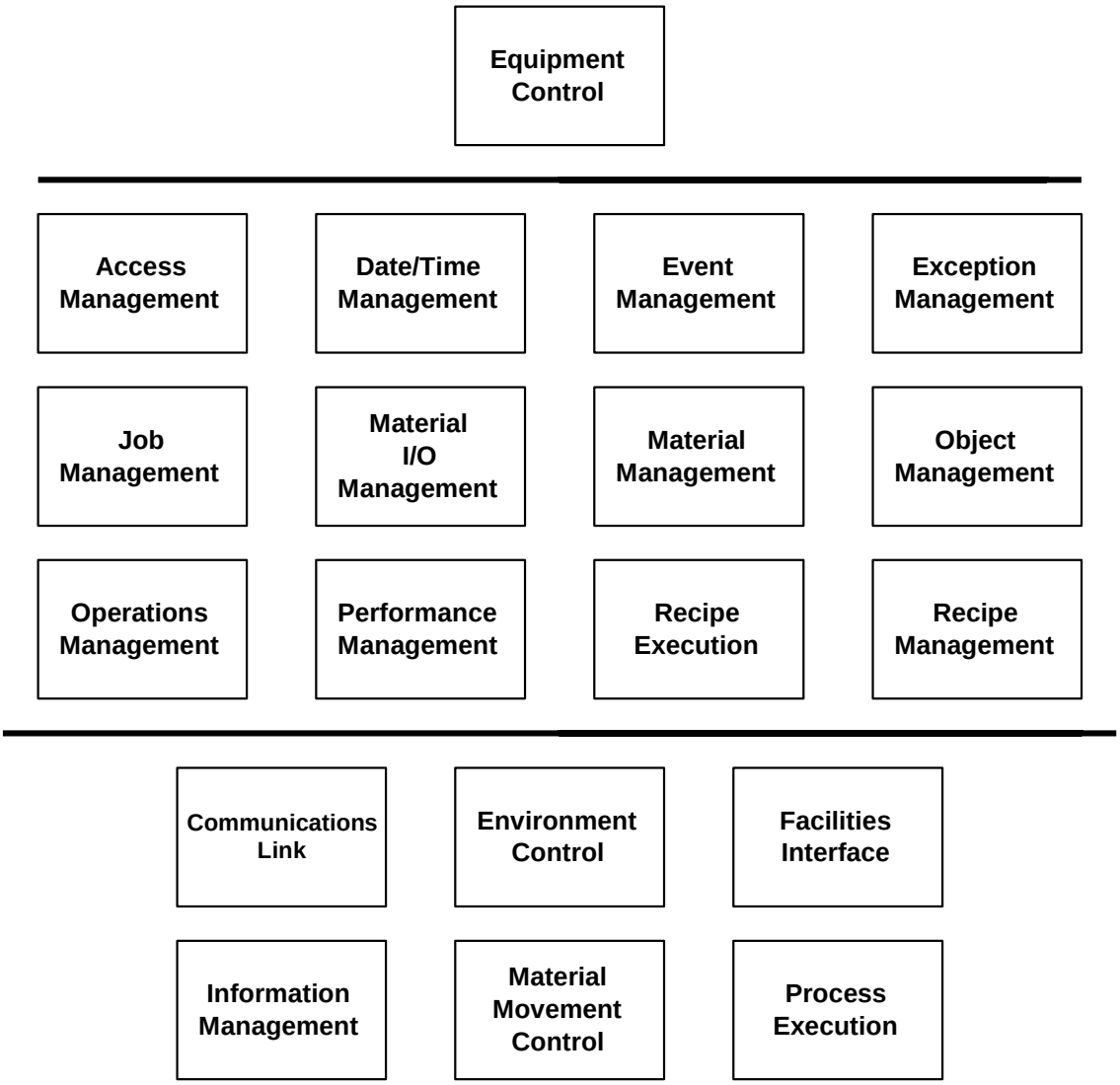


**Figure 1**
**Functional View of Equipment**

8.3.9 *Facilities Interface* — The Facilities Interface is responsible for managing the physical interfaces (hookups) to the factory. This includes bulk fill, continuous chemical services, factory vacuum, factory exhaust, and the electrical environment of the equipment.

8.3.10 *Information Management* — Information Management is responsible for the information and data stored by the equipment, including information required for the user as well as various internal event and data logs.

8.3.11 *Job Management* — Job Management is responsible for all jobs, including process jobs, job queues, and job execution.

8.3.12 *Material I/O Management* — The Material Input/Output (I/O) Management is responsible for loading and unloading material to and from the factory. This includes the AMHS interface (parallel I/O), pod interface, carrier management, and carrier-related services such as reading, writing, and slot mapping (identifying unoccupied, correctly occupied, and incorrectly occupied slots in a carrier).

8.3.13 *Material Management* — Material Management is responsible for tracking all material, including carriers, product, and consumables, within or used by the equipment. This includes providing historical information required for product history.

8.3.14 *Material Movement Control* — Material Movement Control consists of low-level control of

internal subsystems, subassemblies, and i/o devices used in moving material within the equipment, such as robots, location sensors, proximity sensors, motors, centering and alignment systems, and material identifier readers.

8.3.15 *Object Management*

8.3.15.1 Object Management consists of management of OBEM objects, their attributes, and internal communications. It includes all elements of configuration definition, both fixed and user-configurable, that pertain to the equipment. Configuration settings consist of those attributes that affect the global behavior of the equipment and are generally static and change only on request. They are in effect at all times regardless of the current recipe(s) and/or processing states. They control activities that maintain the environment when "not processing".

8.3.15.2 Configuration settings shall be retained in non-volatile storage. Some elements of configuration may be distributed. For example, individual process chambers may have their own configuration elements.

8.3.15.3 Elements of configuration management include:

- configuration of individual physical chambers, and
- configuration of individual logical objects.

8.3.16 *Operations Management* — Operations Management is responsible for the overall operation of the equipment in all operational modes: automatic, semi-automatic, and manual.

8.3.17 *Performance Management* — Performance Management is responsible for managing information and operations related to the performance of the equipment and equipment modules. This includes oversight for manual mode operations performed when the equipment and equipment modules are out of service. For implementations of ARAMS, this also includes ARAMS state changes and data as well as oversight for manual mode operations performed during downtime and non-scheduled time.

8.3.18 *Process Execution* — Process Execution covers those fixed algorithms and procedures that are not reachable or changeable by the user. This includes any embedded control and sequence algorithms not contained in recipes. It consists of low-level control of subsystems, sensors, and actuators not covered by Material Management Control, such as, chemical control (valves, exhaust), motion control (rotational, acceleration, positional) and the control of the environment during processing of the product (temperature, etc.). It also includes product environment control and any fixed embedded fault detection classification, and/or fixed low-level in-situ run-to-run control for advanced process control.

8.3.19 *Recipe Execution*

8.3.19.1 A recipe represents the pre-planned and reusable set of instructions, algorithms, and settings that are used by process execution to control process, including variable in situ process control algorithms. Recipes are created by the user, and in some cases by the equipment as well. Recipes may be of a variety of types, such as flow sequence, metrology, models, abort, and load maps, as well as etch, clean, etc.

8.3.19.2 Recipe Execution is responsible for the proper and safe execution of recipes, including loading the recipe into the execution area, verification of the recipe, validation of recipes (ensuring the recipe does not conflict with the current equipment configuration), and initiation of process execution based on recipe instructions (SEMI E42).

8.3.20 *Recipe Management* — Recipe Management consists of the management of stored recipes. This is differentiated from short-term storage of recipes and the selection and execution of recipes performed by Recipe Execution (SEMI E42). Recipes are classified (organized) according to their primary application function: process, environment, service (maintenance), etc.

8.4 *Relationships with Other Standards*

8.4.1 Only those functional areas in the middle in Figure 1 are of interest to the host. The top level of Equipment Control represents all of the functionality below it, while the functional areas at the bottom are considered to be low level and proprietary to the equipment supplier.

8.4.2 Table 1 shows those functional areas that are defined by OBEM and those that are defined by other SEMI standards. In some cases, OBEM may extend or limit the functionality defined elsewhere.

**Table 1 Functional Area Definition**

| *Functional Area* | *Where Defined* | *Comments* |
|---|---|---|
| Access Management | SEMI E98 (OBEM) | Defines different kinds of user control. |
| Date/Time Management | SEMI E98 (OBEM) | Addresses timestamp, date/time synchronization. |
| Event Management | SEMI E53 (ERS) | SEMI E53 may be required for SECS-II implementations. |
| Exception Management | SEMI E41 (EMS) | Required for reporting alarms and exceptions. |
| Material I/O Management | SEMI E87 (CMS) | Required for Carrier Management. |
| Material Management | SEMI E90 (STS) | Required for Substrate Tracking. |
| Object Management | SEMI E39 (OSS) | Required |
| Operations Management | SEMI E98 (OBEM) | Overall coordination. |
| Performance Management | SEMI E58 (ARAMS) | Optional for EquipmentModule and Equipment. Not used for lower level components. |
| Job Management | SEMI E40 (PM), SEMI E94 (CJM) | Process Managment and Control Job Management |
| Recipe Execution | SEMI E42 (RMS) | Required for processing by EquipmentModule. |
| Recipe Management | SEMI E42 (RMS) | Required for long-term storage by Equipment. |

8.5 *Internal Composition View of the Equipment* — The physical makeup of equipment is of interest to the factory, particularly for equipment that is complex, multi-module, and/or multi-process. Productivity and maintenance tracking, for example, requires that the factory be able to specify individual subsystems and/or modules for maintenance activities, where it is possible to do so without removing the entire equipment from manufacturing scheduling. For example, one or more baths in a wet bench may be down for maintenance even though the wet bench itself continues to process.

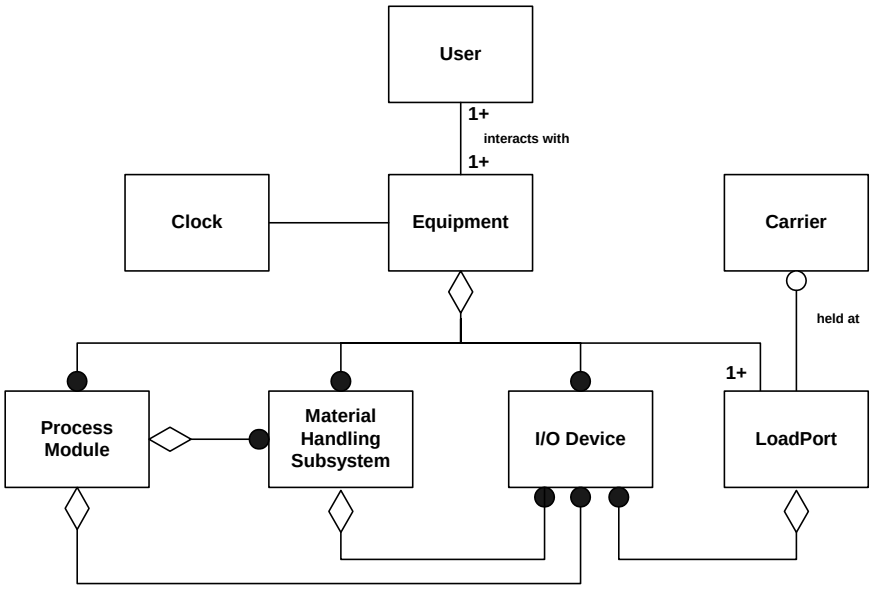


**Figure 2**
**An Example of Equipment Internal Composition**

## 9 OBEM Object Model

9.1 OBEM defines generic component objects of Equipment, and the Equipment object itself. Equipment is made up of elements (units or parts) of different levels of intelligence and complexity, such as modules, subsystems, and I/O devices. Each of these elements may itself be made up of several smaller elements, some of which may also be intelligent, and this allows the complexity of the equipment to be distributed to smaller functional units. Many of

these elements may be of interest to the factory. In particular, process modules, which are intelligent and may be independently operable, are very interesting to the factory, since these are the units where the product is actually processed. The factory requires processing modules to be highly visible and individually addressable and to support certain of the same remote commands that are required of the equipment. Other elements of interest include subsystems for material handling, alignment, and measurement.

9.2 The equipment is responsible for all communications at all times, including messages directed to a specific part of the equipment. Service requests directed to components of the equipment shall be managed by the equipment to ensure equipment integrity.

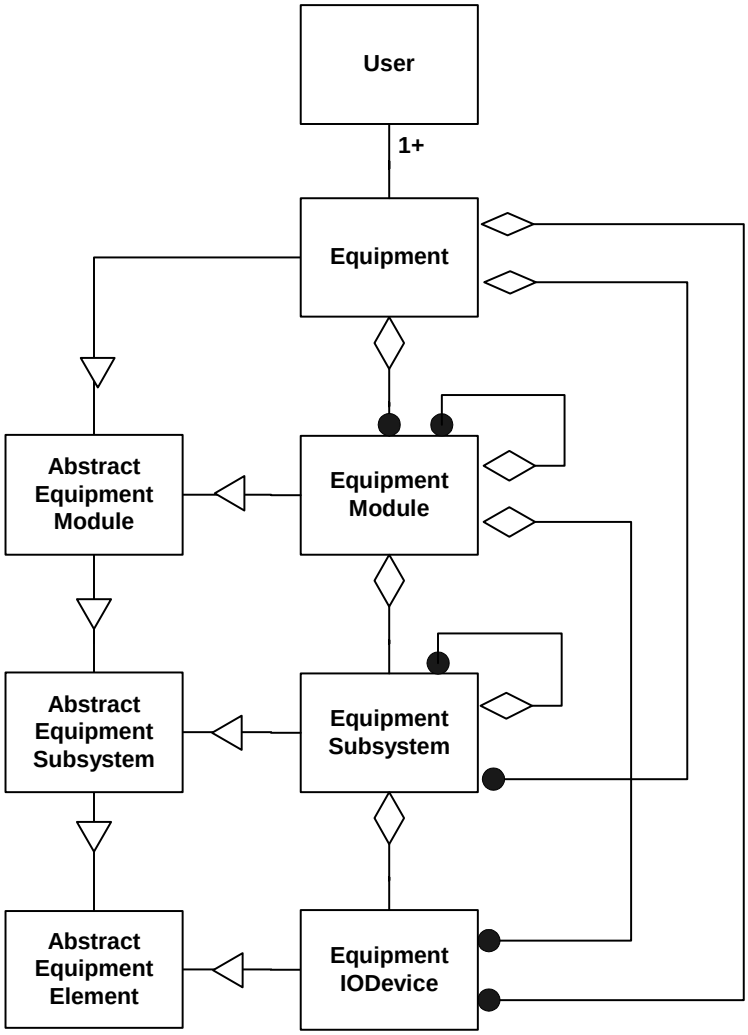


**Figure 3**
**Equipment Object Model**

9.3 In Figure 3, two hierarchies are shown. On the left is an inverted interface inheritance hierarchy, and on the right the concrete subtypes where rules of aggretation are shown. The interface inheritance shows the objects that define the attributes, state models, and services of the subtype objects as viewed externally. These are presented upside down from the usual presentation so that they may be directly related to the aggregation hierarchy on the right. In both cases, the simpler objects are below the more complex objects.

9.4 Those object types starting with the word “Abstract” are abstract objects not intended to be implemented directly. Their purpose is solely to define the inherited attributes, state models, and services for those objects used to build an OBEM model of equipment. The remaining objects shown in Figure 3 are concrete objects. All rules of aggregation are defined for concrete objects only.

9.5 *OBEM Object Requirements* — By definition, subtypes of objects inherit the properties (attributes, services, and relationships) of their supertypes. In some cases, properties of the subtype may be further specialized.

9.5.1 *Object Services Requirements* — All objects formally defined by this standard shall be compliant to the fundamental requirements of SEMI E39 (OSS). All OBEM-defined objects that are aggregates, containers, or managers of other objects shall comply with the additional OSS requirements for object owners. According to OSS, an owner is any aggregate, container, or manager of one or more other objects. An owner is required to respond to queries about the types of objects that it owns. Owners have additional responsibilities, as specified in OSS.

9.5.2 *Object Non-volatility*

9.5.2.1 All objects defined by OBEM shall be persistent. The individual object persists across powerdown and powerup conditions, and all current values of static attributes (attributes that do not change dynamically indicating the object' s status) shall be maintained and restored upon powerup. It may be important to maintain other critical values as well, depending upon the object and the implementation. When the equipment is powdered on or reset[5], all state models are restored. Following initialization, the object is considered to be operational. Figure 4 shows this convention for a generic OBEM object. However, since the state model can not be accessed by the user until the object is operational, the default entry state for a specific state model is considered to be within the Operational state.

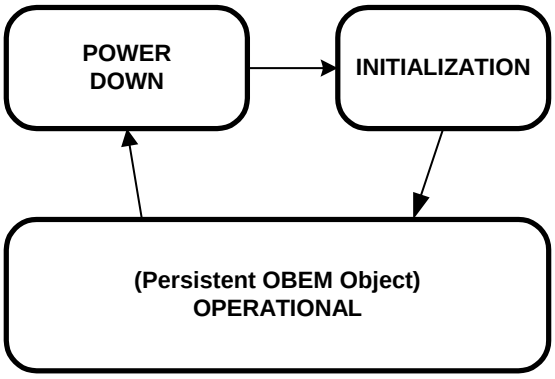


**Figure 4**
**Persistence of OBEM Objects**

9.5.2.2 POWERDOWN and INITIALIZATION are common to all OBEM objects. Therefore, they are not specific to any object and are not generally shown. When operational, the OBEM object is capable of maintaining state information. From the user view, an instantiation of an OBEM object shall follow the behavior or state model as shown in Figure 4. The equipment representation, which consists of an aggregation of OBEM objects, shall also reflect the state model shown here.

9.5.2.3 The equipment is responsible for managing the exchange of any of its component parts, including parts exchanged during powerdown. This may be accomplished through use of intelligent components that are able to identify themselves or through the user interface.

9.5.3 *Shared Resources* — When two (or more) objects cooperate in using the services provided by a third object, then the third object should not be modeled as a component of either of the first two objects. If the two cooperating objects have a common owner, either logical or physical, then the shared resource object should belong to the common owner.

9.5.4 *Object Factory Communications* — OBEM objects other than the Equipment object are neither required nor expected to communicate directly with the factory. Factory communications are handled by the Equipment instantiation.

9.5.5 *Object Event Reporting*

9.5.5.1 Event reporting allows a user to receive notification of events together with related data of interest. OBEM compliance requires that the equipment provide a standard mechanism for reporting events of interest to the user, together with the current values of user-selectable data.

9.5.5.2 All transitions in state models are of interest to the user and reportable unless otherwise stated in the state model definition.

9.6 *OBEM Interface Inheritance Hierarchy*

9.6.1 An interface inheritance hierarchy begins with a simple interface at the highest (most abstract) level, and lower levels within the hierarchy represent added functionality (specialization). A subtype of an object inherits the attributes, behavior, relationships, and services of the supertype and adds to and/or modifies (overrides) them.

5 For a more detailed discussion of powerdown, reset, and soft reset, see SEMI E58.

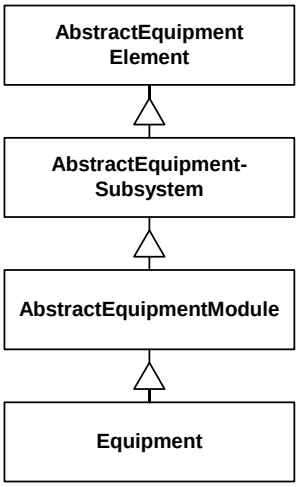


**Figure 5**
**Object Interface Hierarchy Concept**

9.6.1.1 The physical view shown in Figure 2 is concerned with the relationships between different objects. In that view, the equipment is at the highest level and owns (is responsible for) the lower level objects of which it is made up. A process chamber is considered to be at a higher level than subsystems such as substrate handlers.

9.6.1.2 From the view of the object interface hierarchy, this order is reversed, with AbstractEquipmentElement appearing at the top level as shown in Figure 5. From the view of an object interface — the interface to an object — the higher the level, the more simple the interface. This is because of inheritance, where the "child" object inherits all of the attributes, behavior, and services of the "parent" object and at the same time adds some degree of specialization that will be reflected in either additional attributes, behavior, or services, or in restrictions on the more general object.

9.6.1.3 All objects represent the view as seen by the factory, not the internal view of equipment control. From this view, the information and services required for an equipment part such as a pod door opener is relatively simple. The view of a module such as a process chamber is more complex but contains all the elements of the view provided for the simpler part (functional description, immutable id, etc.). The view of the equipment is the most complex and includes all of the attributes and services of the equipment element, equipment subsystem, equipment module, and the equipment itself.

9.6.1.4 The object model of equipment presented to the factory is based on SEMI E39 (OSS). OSS services allow the factory to "discover" the actual physical makeup and aggregation hierarchy of the physical view of equipment illustrated in Figures 2 and 3.

9.6.1.5 Each object is defined in terms of its requirements, attributes, behavior (state models), and the services that it is required to support. The equipment owns all of the objects that it is made of and is responsible for providing the required behavior.

9.6.1.6 All objects in OBEM inherit the attributes and services defined for the Top Object as specified in the Object Service Standard (OSS). This allows the factory to use object services to request the equipment to describe its physical view by reporting which objects that it owns.

9.6.1.7 Note that equipment support for an OBEM interface to the factory does not imply or require direct access from the factory to any equipment element.

## 10 OBEM Object Definitions

10.1 OBEM objects are defined in this section.

10.2 *AbstractEquipmentElement Object* — The supertype object of the interface hierarchy is AbstractEquipmentElement, which is an abstraction of any equipment component that can perform work. AbstractEquipmentElement is an abstract type that is not implemented directly. There are two subtypes of AbstractEquipmentElement: AbstractEquipmentSubsystem, and EquipmentIODevice. AbstractEquipmentElement is an abstract type, so that implementations are of one of the subtypes.

10.2.1 *AbstractEquipmentElement Requirements*

10.2.1.1 *Object Exception Management*

10.2.1.1.1 SEMI E41 defines a model for Exception Conditions. An Exception Condition may be either an Alarm Condition or an Error Condition. Error Conditions may, in some cases, have a set of associated Recovery Actions that can be performed by the AbstractEquipmentElement to attempt to recover from the abnormal situation.

10.2.1.1.2 An OBEM object shall comply with the fundamental requirements of SEMI E41, Section 10.4. Exception Condition objects shall be provided in conformance with SEMI E41 and shall be accessible through services defined in SEMI E39. The AbstractEquipmentElement owns all exceptions that it generates. Therefore, it shall report all of its Exception Condition objects through OSS services.

10.2.2 *AbstractEquipmentElement Subtypes* — The AbstractEquipmentElement has two subtypes, the EquipmentIODevice and the AbstractEquipment-Subsystem.