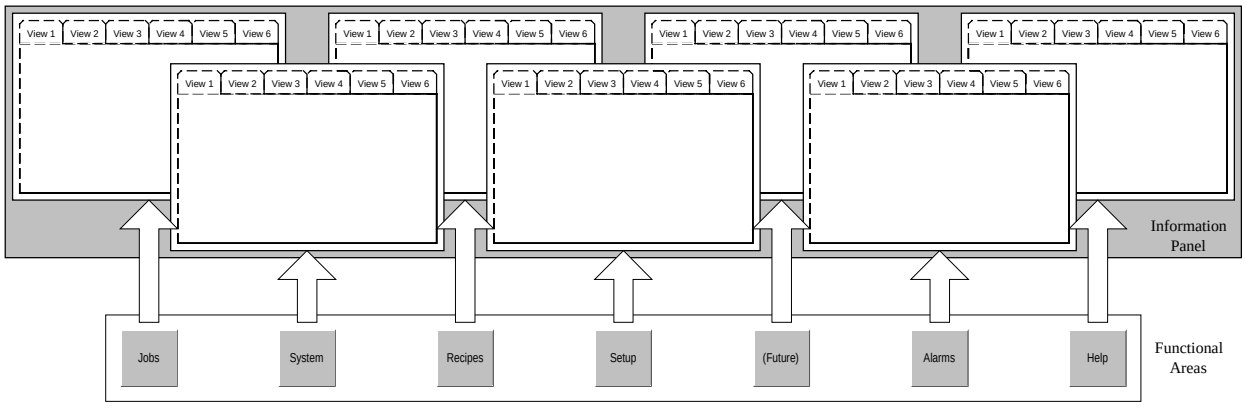


Figure 6
Network Navigation Model — Tab Sub-navigation

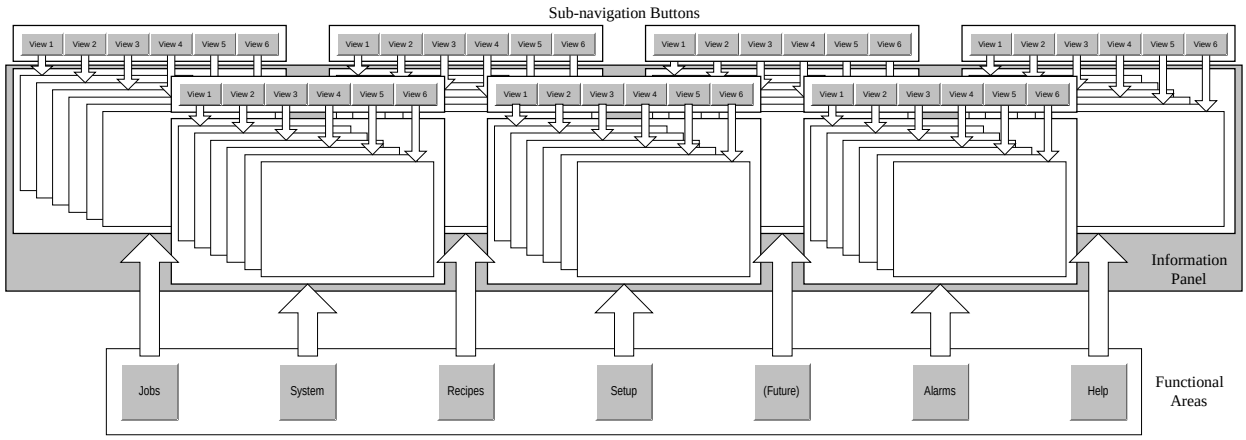


Figure 7
Network Navigation Model — Button Sub-navigation

5.3.5 *Display Layout* *Description*

5.3.5.1 The display layout is designed for ease of use with touchscreen input devices and does not require a keyboard or other pointing device. By dividing the screen into rectangular panels, provision is made to accommodate the display and input of information organized by the tasks users must accomplish in managing and monitoring processing, maintaining and repairing the equipment, and other relevant work.

5.3.6 *Basic Layout* *Mandatory*

5.3.6.1 The basic layout shall contain four panels as shown and oriented in Figure 8. At a minimum, the interface shall support the orientation of the command panel on the right-hand side, unless the enhanced layout (Section 5.3.7) is implemented.

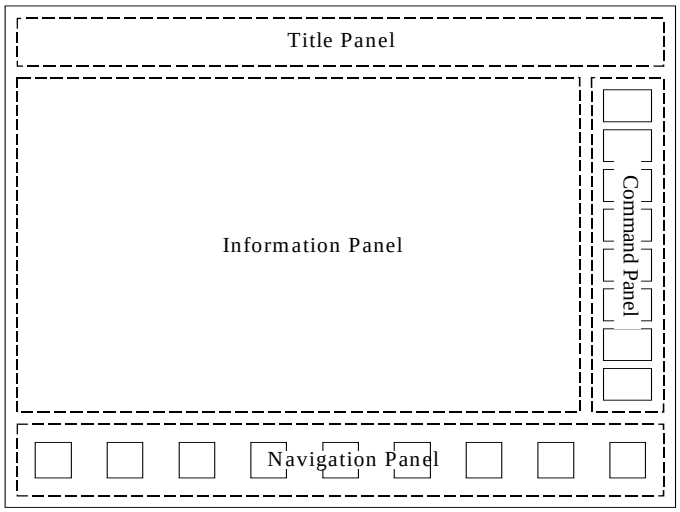
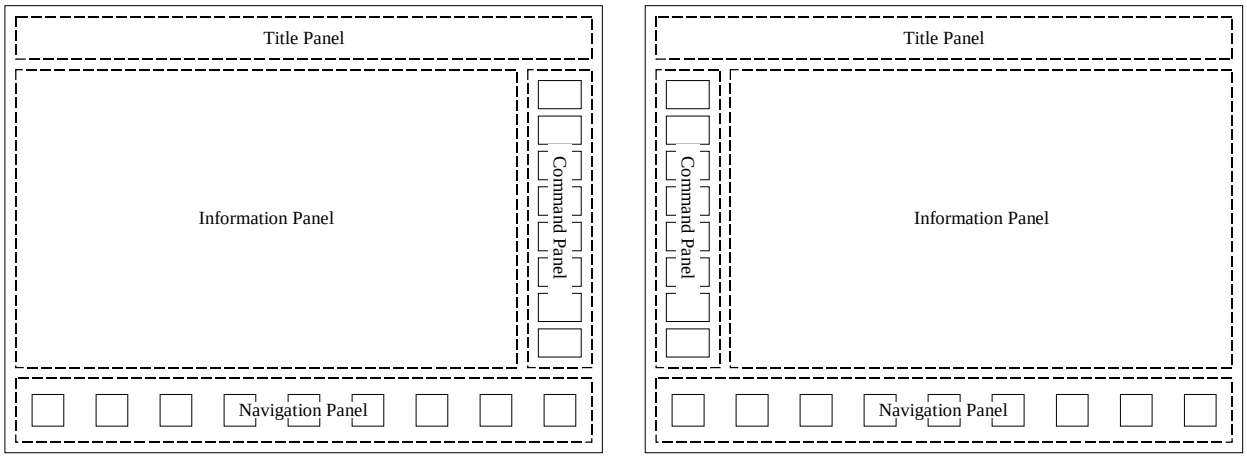


Figure 8
Basic Layout

5.3.6.2 All the panels are tiled edge to edge to create the display, and only the relative position of the panels is specified in this standard. Panels may or may not display a visible border. For an interface that is the primary display (typically, but not always at the front of equipment), an outer window frame allowing window resizing, closing, or positioning shall not be shown or enabled. This is to prevent the user from mistakenly "losing" the window, which may result in a dangerous condition. If desired, a logged-in user with sufficient privileges may be allowed to resize, but not minimize or close, the primary display window. Secondary instances of the interface (e.g., displayed at a maintenance node or displayed at a remote node) may show and enable the outer window frame.

5.3.7 *Enhanced Layout* *Recommended*

5.3.7.1 It is strongly recommended that left-handed users be allowed to change the location of the command panel to the left-hand side (see Figure 9(b)) to avoid obscuring the screen when reaching with their left hand to make selections on the command panel when it is located on the right-hand side of the screen.



(a) (b)

Figure 9
Enhanced Layout – Right and Left Command Panel Orientation

5.3.8 *Title Panel* *Description*

5.3.8.1 The title panel is a horizontal area above the information and command panels, at the top of the interface window. It is always displayed and contains the host communications status display (if host communications is supported), date/time display, Login/Logout button (if security is supported), message display area, and the name of the current view. It may optionally contain a corporate identifier or logo, a display of critical parameters, an audible alarm silencing button, orientation graphics, a light tower representation, and other items that should always be displayed to ensure effective operation.

5.3.9 *Title Panel Basic Information* *Mandatory*

5.3.9.1 Shown below is the title panel with the mandatory display objects. The relative positions shown, with the top portion of the title panel containing the date/time display at the left, the current view name to its right, and with the message area below the top portion, are mandatory.

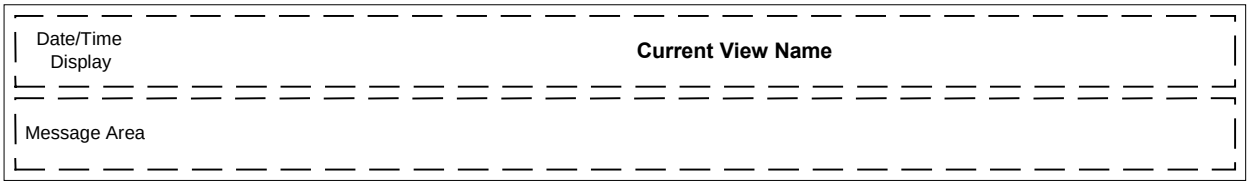


Figure 10
Title Panel

5.3.10 *Title Panel with Conditional Information* *Conditional*

5.3.10.1 Shown below is the title panel with the mandatory display objects, plus the conditional host communications status display and the conditional Login/Logout button. The relative positions shown, with the host communications status display left-most, and the Login/Logout button at the upper right, are mandatory.

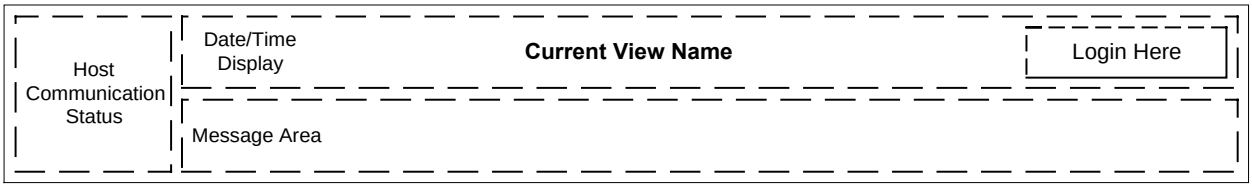


Figure 11
Title Panel

5.3.10.2 *Title Panel Host Communications Status* *Conditional*

5.3.10.2.1 If the equipment supports host communication then status information shall be included in the title panel. Information such as communications status (i.e., whether communications is active), communications state (i.e., connected, disconnected, etc.), and whether the equipment is in a local or remote mode may be displayed here. The display of specific information is dependent on the host communication protocol which may impose additional specific requirements on what is displayed.

5.3.10.3 *Title Panel Login/Logout Button* *Conditional*

5.3.10.3.1 The Login/Logout button label reads “Login Here” until a user is logged in, then displays a user identifier until the user logs out. User selection of the Login/Logout button invokes a dialog box where the user may enter a user identifier and password, or, if already logged in, may select a button to log out. If required by the implementation, when this dialog box is displayed, all other functions in the interface window may be disabled, including the navigation panel.

5.3.11 *Title Panel with Additional Information* *Recommended*

5.3.11.1 Shown below is an example of a layout for the title panel incorporating some recommended display objects and their relative positions.

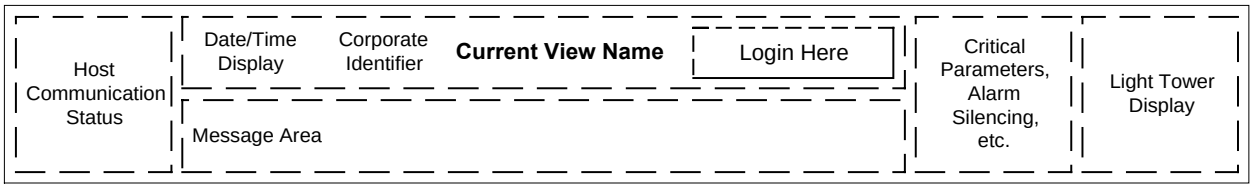


Figure 12
Title Panel with Some Additional Display Objects

5.3.11.2 *Title Panel Alarms Button* *Recommended*

5.3.11.2.1 Although not recommended for new designs, the title panel may also contain an Alarms button that allows the user to respond to cautions and severe alarms. In this case, the Alarms navigation button in the navigation panel shall be omitted, and any alarms accessed through a title panel Alarms button shall be displayed in a dialog box, not as an information panel and its command panel.

5.4 *Navigation Panel* *Mandatory*

5.4.1 Navigation buttons shall have a text label. In addition, they may also include an icon to graphically represent their function. When no icon is present, the button label shall be centered on the button. If an icon is present, the label shall be centered below the icon. Navigation buttons are arranged horizontally along the bottom of the display, in the navigation panel, which shall always be present.

5.4.2 *Required Navigation Functions* *Mandatory*

5.4.2.1 At a minimum, the user shall always be able to immediately access and respond to alarm and caution notifications, even when a dialog box is displayed on the current view. Dialog boxes shall not obscure the navigation panel. Additionally, the user shall always be able to immediately access other parts of the interface if such access is required to ensure the safe operation of the equipment. Only when prohibited by the operating system or other implementation limitations such that a displayed dialog box cannot be maintained during, or redisplayed after navigation, it is allowed that such access may be accomplished by displaying another dialog box that completely covers the originally displayed dialog box. When the overlaying dialog box is dismissed, the underlying dialog box is redisplayed, in the same state it was in prior to the invocation of the overlying dialog box (i.e., given the stated prohibition or limitations, it is not mandatory that access be provided through navigation using the navigation panel). Immediate access shall mean that the user shall not have to dismiss or otherwise interact with any displayed dialog box in order to perform the required access. When the user navigates back or otherwise returns from the required access, the last selected view shall be displayed, along with any dialog box that was displayed, in the same state it was in. If no dialog box was displayed, the last selected view shall be displayed.

5.4.2.2 An allowed exception is a login and/or logout dialog box or screen if an implementation requires modal operation while logging in or out.

5.4.3 *Conditional Navigation Functions* *Conditional*

5.4.3.1 Except when absolutely prevented by the operating system or implementation limitations, the navigation panel shall always be available for user selection, even when a dialog box is displayed on the current view. This makes it possible for the user to directly and immediately access any functional area from anywhere within the user interface. Immediate access shall mean that the user shall not have to dismiss or otherwise interact with any displayed dialog box in order to perform the required access. When the user navigates back to a functional area, the last selected view is displayed, along with any dialog box that was displayed, in the same state it was in.

5.4.3.2 An allowed exception is a login and/or logout dialog box or screen if an implementation requires modal operation while logging in or out.

5.4.4 *Navigation Panel Layout* *Mandatory*

5.4.4.1 The figure below shows the navigation panel, with three buttons labeled "(Future)" indicating the positions where buttons may be placed if required by the specific implementation of the interface, or as a result of modifications or enhancements in future releases of the software. It is recommended that the navigation panel contain no more than ten buttons.

5.4.4.2 The navigation buttons shall be sequenced from left to right in descending order of expected frequency of use. The most frequently selected navigation button shall be left-most within the navigation panel; and the least frequently selected button shall be right-most.

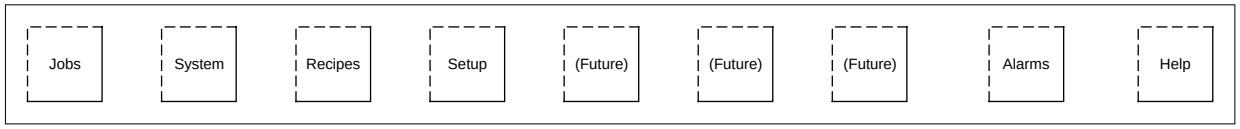


Figure 13
Navigation Panel

5.4.5 *Navigation Panel Alarm and Help Buttons* *Conditional*

5.4.5.1 The two exceptions to the above ordering are the Alarms and Help navigation buttons, which, when they are supported in an implementation, shall be the next to right-most and right-most buttons, respectively. This placement ensures that the position of these buttons shall remain unchanged, even if subsequent interface modifications or enhancements require additional buttons. The Alarm button shall be placed so that the spacing between it and adjacent buttons is larger than the spacing between other buttons, to allow its selection quickly, and without error.

5.4.6 *Navigation Button Labels* *Conditional*

5.4.6.1 For each functional area, there is a corresponding navigation button identified by a text label (mandatory) and icon (recommended) identifying the functionality and information provided. The table below shows text labels (conditional) for the navigation buttons, a description of each functional area, and some recommended alternative labels.

Table 1 Functional Areas

Navigation Button Label	*Description*	*Alternate Labels*
Jobs	Operations related to product processing, including any pre- and post-production equipment setup	Lot Operations, Operation, Operations, Processing, Main, Run
System	Equipment status, manual move, maintenance, service, calibration, & other engineering-level functions	Overview, Service, Status, System Status, Maintenance
Recipes	Recipe management, including creation, editing, storing, etc.	None
Datalog	Data histories, event logs, SPC functions (If supported)	History, Analysis, Logs, Data
Setup	User account administration, host communications control, user preferences, parameters, hardware configuration/options, light tower programming, etc.	Configuration, Options
Alarms	Alarm and caution summary to acknowledge and clear posted alarms, current event log	None (see Section 5.4.5)
Help	Help files on operations, procedures, and the interface	None (see Section 5.4.5)

5.4.6.2 The top to bottom ordering of the table reflects the left to right ordering of navigation buttons. Also allowed, but not recommended for new designs, is a left to right ordering of: System, Jobs, Equipment Setup, Recipes, History, Maintenance, and Configuration. The alternative labels specified in the table may be applied to this

ordering also. Additional buttons, if required for a particular implementation, shall be added between the Setup and Alarm button positions.

5.4.7 *Navigation Panel Saliences* *Conditional*

5.4.7.1 Only one navigation button at a time shall display a pressed appearance. Additionally, navigation buttons shall display colored salience coding for a number of purposes: 1, to indicate the user is viewing a functional area (medium blue salience); 2, to indicate an unfinished task (typically an open dialog box) in a functional area not currently displayed (medium blue salience); and 3, to inform the user that there are new or unacknowledged cautions or alarms (saturated yellow or saturated red salience, respectively). The caution and alarm saliences are displayed on the Alarms navigation button only. As an example, if the user has opened a dialog box in the Jobs functional area, and then selects the Recipes navigation button, the Recipes button shall display a pressed (down) appearance *and* a medium blue salience, and the Jobs button shall display an unpressed (up) appearance *and* a medium blue salience (Figure 14). This reminds the user that there is an open dialog box in the Jobs functional area. More than one navigation button may display the unfinished task salience.

5.4.7.2 The Jobs button may also display a medium green salience (not shown) to notify the user that the equipment is "Ready to Load," "Ready to Unload," "Ready to Run," or is in a similar state such that the user's attention is requested in the Jobs functional area. This is useful when the user has navigated to another functional area of the interface. If there is an unfinished task, its medium blue salience shall remain displayed, even if the user's attention is requested.

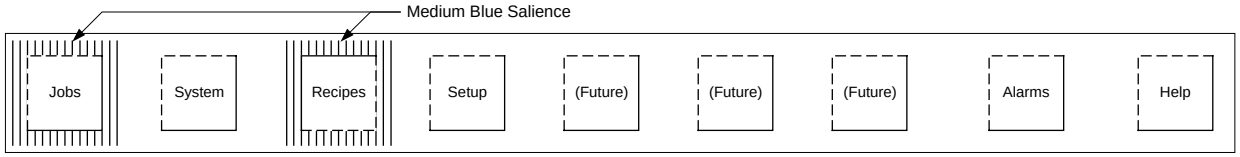


Figure 14
Navigation Button Saliences

5.4.7.3 The Alarm navigation button, in addition to the medium blue salience, displays a saturated (bright) yellow salience when there are new or unacknowledged cautions, or displays a saturated red salience when there are new or unacknowledged alarms. Only the severest level is displayed. That is, when there are both cautions and alarms, the red alarm salience shall be displayed. When there are no alarms and only cautions, the caution salience shall be displayed. The figures below show the same situation as Figure 14, with Figure 15 showing a caution salience, and Figure 16 showing an alarm salience. If there are no cautions or alarms, the Alarm button displays a medium blue salience if the user is viewing the Alarms functional area, or if there is an unfinished task and another functional area is being viewed. If a caution or alarm occurs, the medium blue salience is replaced with the appropriate salience, and is only re-displayed when all cautions and alarms have been acknowledged or cleared.

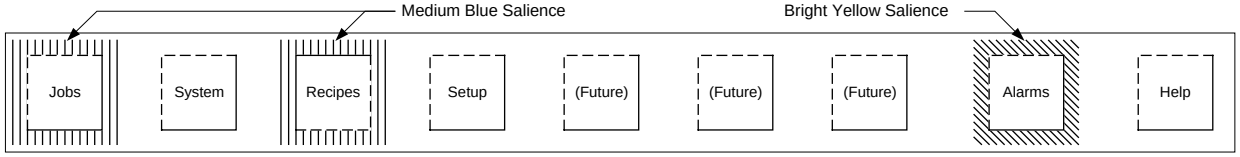


Figure 15
Warning Salience

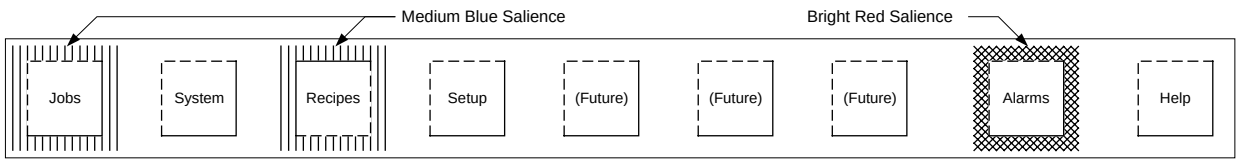


Figure 16
Alarm Salience

5.4.8 *Sub-navigation* *Conditional*

5.4.8.1 When sub-navigation is supported it shall be by a single row of tabs or buttons in a sub-navigation panel as shown below.

5.4.8.2 *Sub-navigation Layout A — Tabs*

5.4.8.2.1 Shown below are two orientations of the layout (right-hand and left-hand command panels), with sub-navigation using tabs. This is the preferred method for new designs where more than one view per functional area is needed. User selection of a tab brings the tab to the front, displays its information and command panel, and allows the user access to its display objects. Use of tabs in each functional area must be consistent throughout the interface, even if there is only one view in a functional area, and thus, one tab.

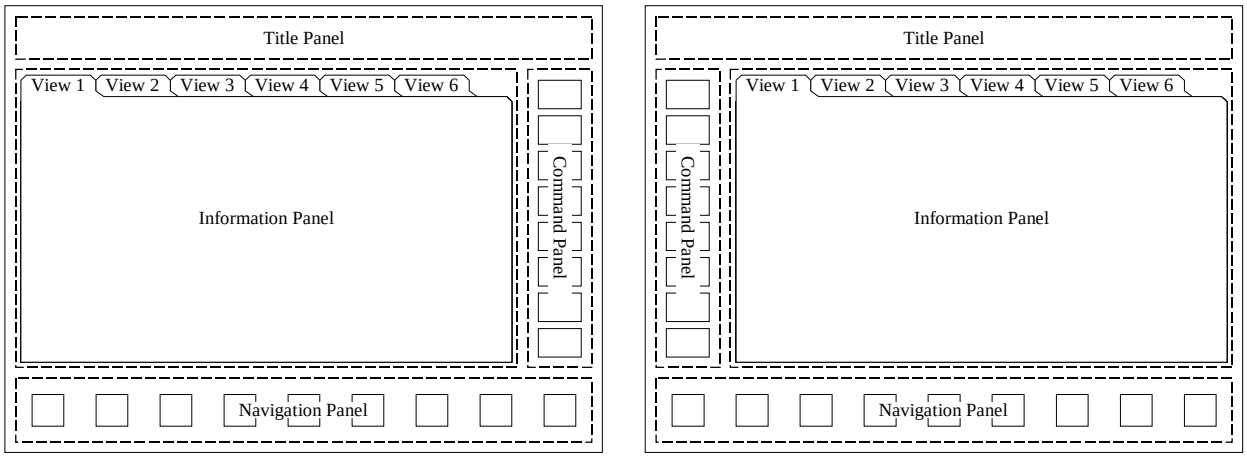


Figure 17
View Sub-navigation Using Tabs

5.4.8.3 *Sub-navigation Layout B — Sub-navigation Panel With Buttons*

5.4.8.3.1 Shown below are two orientations of the layout, with sub-navigation using view selection buttons in a sub-navigation panel. The figure shows one possible relative location for a sub-navigation panel, but is not intended to restrict implementation. Other arrangements are allowed. However, if a sub-navigation panel is used, its size and location in each functional area must be consistent throughout the interface, even if there is only one view in a functional area, and thus, no buttons in the panel.

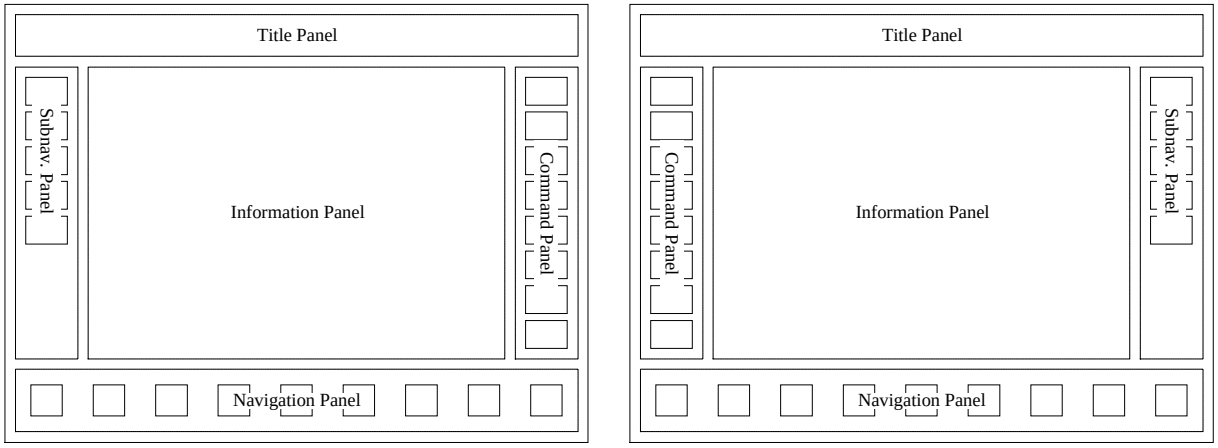


Figure 18
View Sub-navigation Using Buttons in Separate Panel

5.4.8.3.2 It is important in the layout to separate sub-navigation methods from the global commands in the command panel. This limits the number of buttons needed in the command panel; and reduces or eliminates the need for multiple columns of buttons, which would alter the information panel display aspect ratio. If an information panel has a different aspect ratio than the others, its contents may appear to "jump" sideways when navigating, distracting the user. The separation of sub-navigation from commands accomplishes two important objectives; a) users do not become confused trying to differentiate sub-navigation from commands, and b) the aspect ratio of the information panel display is consistent for all views across all functional areas.

5.5 *Information Panel* *Mandatory*

5.5.1 The information panel displays a view or views of the information and graphics for each functional area. Graphics and other display objects are placed in this panel to achieve the control and monitoring capabilities required. If necessary, multiple views of information may be displayed within a functional area, one at a time, in the information panel.

5.5.2 When any functional areas have more than one view, the user must be able to switch between those views while remaining within the context of the current functional area. The ways the user may select among multiple views presented in this standard are called sub-navigation methods to distinguish them from user navigation between functional areas using the navigation panel.

5.6 *Command Panel* *Mandatory*

5.6.1 The command panel is a vertical column of command buttons located to the right of the information panel (to the left if switched to accommodate left-handed users). Only buttons for common or global commands related to the current view displayed in the information panel shall be located in the command panel. If there are no common commands for an information panel, the command panel shall have no buttons. Each view in a functional area shall have its own command panel. To limit the number of command buttons needed in each command panel, user selection of a different view shall display that view and its associated command panel, with commands that apply only to the selected view. A command panel may be used for more than one view if it is suitable for that purpose. Command buttons or other display objects that have a more limited scope shall be located in the information panel. Restricting locally-acting commands and functions to the information panel makes clear to the user that only general, global commands are located in the command panel. Buttons for navigation (i.e., that invoke the display of another view in the information panel) shall not be located in the command panel. It is recommended that multiple columns of buttons in the command panel be avoided.

6 Compliance Statement

6.1 In order to be compliant with this specification, the documentation accompanying an equipment shall include a Human Computer Interface (HCI) Compliance Statement that accurately indicates compliance with the individual requirements defined in this document. Requirements and recommended capabilities are defined in Table 2.

6.2 In order to be compliant with HCI, equipment must meet all requirements in each of three categories, as follows:

6.2.1 Mandatory: In order to be compliant with this standard, all of the mandatory requirements shall be both implemented and compliant as defined in this specification.

6.2.2 Conditional: In order to be compliant with this standard, each conditional requirement shall either be implemented as defined in this specification or shall both not be implemented in the user interface and not be supported in some other way by the equipment. (i.e., no conditional capability which is present on the equipment shall be implemented in a manner other than as defined in this specification).

6.2.3 Recommended: Implementation of these features is at the discretion of the implementers. The only requirement for compliance with this specification for these capabilities is that they be accurately documented in the compliance statement for the equipment.

6.3 Each requirement/capability shall be marked "Yes" under "Implemented" if the equipment includes a feature which provides equivalent functionality as that defined in this specification even if that feature appears in a different form. Otherwise it shall be marked "No".

6.4 Each requirement/capability shall be marked "Yes" under "HCI Compliant" if the equipment includes a feature which conforms to all aspects of the requirement or recommended capability as defined in this specification. Otherwise it shall be marked "No".

Table 2 HCI Compliance Statement

HCI Compliance Statement			
Mandatory Requirements	*Reference*	*Implemented*	*HCI Compliant*
Button Size	5.2.2.1	Yes No	Yes No
Button Behavior	5.2.2.3	Yes No	Yes No
Button Text	5.2.2.4	Yes No	Yes No
Dialog Boxes	5.2.5	Yes No	Yes No
Basic Network Navigation Model	5.3	Yes No	Yes No
Basic Layout	5.3.6	Yes No	Yes No
Title Panel Basic Information	5.3.9	Yes No	Yes No
Navigation Panel	5.4	Yes No	Yes No
Required Navigation Panel Functions	5.4.2	Yes No	Yes No
Navigation Panel Layout	5.4.4	Yes No	Yes No
Information Panel	5.5	Yes No	Yes No
Command Panel	5.6	Yes No	Yes No
Conditional Requirements	*Reference*	*Implemented*	*HCI Compliant*
Saliences	5.2.3	Yes No	Yes No
Information Dialog Boxes	5.2.6	Yes No	Yes No
Input/Selection Dialog Box	5.2.7	Yes No	Yes No
Message Dialog Box	5.2.8	Yes No	Yes No
Network Navigation Model with Sub-navigation	5.3.4	Yes No	Yes No
Title Panel with Conditional Information	5.3.10	Yes No	Yes No
Title Panel Host Communications Status	5.3.10.2	Yes No	Yes No
Title Panel Login/Logout Button	5.3.10.3	Yes No	Yes No
Conditional Navigation Panel Functions	5.4.3	Yes No	Yes No
Navigation Panel Alarm & Help Buttons	5.4.5	Yes No	Yes No
Navigation Button Labels	5.4.6	Yes No	Yes No
Navigation Panel Saliences	5.4.7	Yes No	Yes No
Sub-navigation	5.4.8	Yes No	Yes No
Recommended Capabilities	*Reference*	*Implemented*	*HCI Compliant*
Button Dimensions	5.2.2.2	Yes No	Yes No
Enhanced Layout	5.3.7	Yes No	Yes No
Title Panel with Additional Information	5.3.11	Yes No	Yes No
Title Panel Alarms Button	5.3.11.2	Yes No	Yes No

7 Related Documents

7.1 *SEMATECH Documents*[1]

Computer Integrated Manufacturing (CIM) Application Framework Specification

SCC User-Interface Style Guide

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.

1 SEMATECH, 2706 Montopolis Drive, Austin, TX, 78741-6499. Website: www.sematech.org

SEMI E96-1101
GUIDE FOR CIM FRAMEWORK TECHNICAL ARCHITECTURE

This guide 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 European Regional Standards Committee on June 11, 2001, and by the North American Regional Standards Committee on July 19, 2001. Initially available at www.semi.org August 2001; to be published November 2001. Originally published February 2000. This document replaces PR5-0699 in its entirety.

1 Purpose

1.1 This guide describes technical architecture choices that enable application components to cooperate in a Computer Integrated Manufacturing (CIM) environment and reduce the effort required to integrate those components into a working solution. The CIM Framework technical architecture guide builds on publicly available specifications for distributed object computing. It defines manufacturing production systems requirements for the technical infrastructure needed for improved component interoperability, substitutability, and extensibility. It provides guidance for specifying components and addresses options for using an underlying distributed object communication infrastructure.

1.2 This guide provides guidance for the technical foundation of the SEMI Computer Integrated Manufacturing (CIM) Framework standards. It discusses a component-based architecture using object-oriented and framework technology that helps implementers achieve component interoperability and substitutability, application extensibility, and reuse. It establishes the role of distributed object communications infrastructure in providing necessary support for the framework technology. Specification methods for mapping a CIM Framework specification to alternative infrastructure technologies are also addressed by this technical architecture. However, these mappings are not intended to be prescriptive. Further work may be required to define additional mappings to emerging technologies. Many implementation issues that should be resolved for a particular software implementation are outside the scope of this guide.

1.3 Adhering to this guide for technical architecture alone does not provide interoperability between applications. While the technical architecture provides a foundation for interoperability, it is limited by the following factors:

- Multiple infrastructure implementation choices are possible, and interoperability across these environments is not guaranteed.
- The technical architecture intentionally limits its scope to only the most fundamental infrastructure requirements, leaving additional technical issues for future guide upgrades or for implementers' discretion.
- Conformance to a specification for CIM Framework Domain Architecture is also required for interoperability of domain components.
- More complete semantics (including behavioral constraints and collaboration patterns) for components are needed to ensure consistent interactions among components developed by separate suppliers.

1.4 A guide for technical architecture is a necessary, but not a sufficient, basis to achieve the goals of the CIM Framework specifications. It does not mandate specific solutions to address the identified technical requirements because there are multiple implementation choices that meet these requirements. Rather, the technical architecture identifies those crucial technical requirements that should be considered by both CIM software suppliers and consumers. The proposed standard identifies the technical capabilities implementations should provide, but leaves the implementation options open. It is the responsibility of suppliers to provide and explain an implementation of each capability, and the responsibility of consumers to assess particular implementations for use in their factories.

1.5 This guide provides guidance on the technical tradeoffs for services provided by the distributed computing infrastructure for the purpose of supporting and enabling the domain specifications of CIM Framework components. These areas are:

- *Distributed Object Communication* — Provides the basic services to enable implementations supporting the CIM Framework interfaces to transparently locate other, possibly distributed implementations and exchange messages requesting standard CIM Framework operations. Interface Definition Language provides a formal specification of the CIM Framework interfaces that can be automatically transformed into conformant implementations ready for integration and interoperation.
- *Exception Declarations* — Identify the form and structure of return messages that inform requestors that a requested operation resulted in an anticipated, but abnormal outcome.

- *Event Specification* — Establishes the delivery mechanism, identification conventions, and data structures for reporting the occurrence of anticipated state changes to CIM Framework objects.
- *Distributed Transactions* — Define mechanisms needed to coordinate the start, completion or rollback of units-of-work that cross CIM Framework component boundaries.
- *Component Manager Support* — Identifies the component-level operations needed to create, locate, or remove instances of objects (and manage collections of those objects) that support the CIM Framework specified interfaces.

2 Scope

2.1 *Intended Audience*

2.1.1 This document is intended for developers of components and applications, and integrators of MES systems that adhere to the CIM Framework specifications. It is also intended for system architects who contribute to the evolution of the CIM Framework architecture and guides based on implementation experience. A guide for technical architecture is focused on the software technologies that support the architectural goals for the CIM Framework rather than on the manufacturing domain concepts that the CIM Framework encompasses. The technical architecture perspective complements SEMI E81.

2.2 *Architectural Issues Not Covered*

2.2.1 A number of architectural issues are not covered within this document because they are beyond the scope of the CIM Framework standards and are not expected to come within the scope of the standards as they are revised. They are itemized here because a product architecture layered on the CIM Framework Technical Architecture should address these additional architecture issues. In these cases, other more general specifications emerging in the infrastructure technology areas are expected to provide these needed standards. The CIM Framework domain specifications do not require specific conformance in these areas to support component specifications.

2.2.2 *Persistence*

2.2.2.1 Persistence refers to the ability of an object to maintain a nonvolatile copy of its current state such that the object could recreate the state during a future initialization. There are various operations for object persistence, and problems can occur if objects with cross-references do not coordinate their persistence strategies and mechanisms. The CIM Framework excludes persistence as an implementation mechanism.

2.2.3 *System Performance*

2.2.3.1 System performance is highly dependent on the selection of hardware and software platforms for system execution. Tests should be performed to verify adequate system performance and scalability for the anticipated operating environment. Performance tuning mechanisms or measurement tools are excluded from the CIM Framework specifications as an implementation dependent mechanism.

2.2.4 *Data Replication*

2.2.4.1 Data replication is a technique used to provide additional fault tolerance or improve system performance in certain situations. The CIM Framework excludes specification of replication strategies as an implementation dependent mechanism.

2.2.5 *Change Management*

2.2.5.1 Change management is the ability to introduce and control changes to the system configuration. The CIM Framework encompasses change management in the domain context of document control, but the CIM Framework excludes the broader treatment of change management for the MES software configuration itself.

2.2.6 *Externalization*

2.2.6.1 Externalization can be used to provide a form of persistence or to transfer object state between disjoint implementations. The ability of an object to externalize its data and state supports recovery of data and state for objects that terminated from memory. The CIM Framework excludes externalization as an implementation dependent mechanism.

2.3 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 The CIM Framework should continue to evolve to meet the needs of a competitive and vital industry. The content of this framework represents a significant amount of real development experience from a number of commercial software suppliers and their customers. These specifications reflect the product architectures of those companies, as well as the requirements of their customers. This evolution process should continue as more products based on the CIM Framework are developed.

3.1.1 This guide acknowledges the following deficiencies that should be addressed in future revisions. These deficiencies are identified in the following sections.

3.2 *Mapping to Alternate Distributed Computing Infrastructures*

3.2.1 The CIM Framework provides a specification for MES software components, specified in terms of generalized manufacturing production systems requirements, that may be implemented using a variety of technical infrastructure foundations. While the intent of this guide is to provide both rigor in specification and flexibility to make infrastructure implementation choices, these goals often conflict. The use of mapping techniques complicates the task of integrating applications across technology boundaries.

3.2.2 While it is anticipated that a conformant implementation using either CORBA®[1] or the Microsoft®[2] Distributed Component Object Model (DCOM) is feasible by mapping the specifications to the implementation, it is recognized that cross-infrastructure integration is significantly more difficult (for example, merging transaction models). The mapping described here offers more diverse implementation choices, but it does not guarantee that all of those chosen technologies easily work together in a single heterogeneous implementation.

3.2.3 Although the DCOM mapping provides a straightforward transform from the OMG® Interface Definition Language (IDL)™[3] specifications for static invocation, the Microsoft OLE Automation interfaces may be required for dynamic invocation. There appears to be greater risk in being able to successfully map the CIM Framework to the OLE Automation interfaces. The requirement for dynamic invocation should be evaluated with this in mind.

3.2.4 Another issue with DCOM mapping concerns exceptions. DCOM returns exceptions using its return value HRESULT. Many CIM Framework operations already use return values and would not be able to return a HRESULT without restructuring the return mechanism for the operation results.

3.2.5 Finally, there has not yet been a detailed analysis of the CIM Framework interfaces to verify that they can be successfully mapped using the CORBA Interworking Architecture.[4] This is especially true of the OLE Automation mapping resolution.

1 CORBA is a registered trademark of Object Management Group, Inc. in the United States and other countries.

2 Microsoft is a registered trademark of Microsoft Corporation, Inc. in the United States and other countries.

3 OMG Interface Definition Language (IDL) is a trademark of Object Management Group, Inc. in the United States and other countries.

4 Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.2, Object Management Group, 492 Old Connecticut Path, Framingham, MA: Object Management Group, 1998.

3.3 *Business Rules*

3.3.1 The management of factory objects requires the use of a set of business rules; that is, procedures representing common business practices that should be applied under a given set of circumstances in response to some factory event. For instance, "Do not assign a process job to a machine which is scheduled for maintenance within 24 hours." Factory systems implementations typically specify business rules as event-driven ECA (event-condition-action) or ECAA (event-condition-action-alternative action) rules. For example:

- *event* — request to edit a process specification;
- *condition* — invalid user access privilege;
- *action* — deny access;
- (*alternative action* — deny access and report breach of security).

3.3.2 Business rules can also be embodied in the sequencing logic of sequential process definitions. In this case the business rules define the criteria for making sequencing decisions that effect the flow of work through the factory.

3.3.3 Business rules are intended to be addressed in future revisions of this guide.

3.4 *Security and Access Control*

3.4.1 The management of sensitive information regarding business processes and product specifications requires that MES implementations (especially distributed systems) provide some level of security and access control services. Typically, such services:

- identify and authenticate any factory object seeking sensitive information,
- permit access to information or operations based upon identity and privilege,
- provide security-related audit trails,
- provide secure communications (not susceptible to being intercepted nor malicious or inadvertent modification), and
- administer an enterprise's security policy.

3.4.2 Security and access control is intended to be addressed in future revisions of this guide.

3.5 *Internationalization*

3.5.1 Specifications that deal with issues related to internationalization are emerging from several sources, including the OMG. This guide should encompass the need and ability to incorporate internationalization features into the CIM Framework specifications.

3.5.2 Internationalization are intended to be addressed in future revisions of this guide.

3.6 *Object Properties*

3.6.1 Object properties refers to a technique that allows additional attributes (data) to be dynamically associated with an object without changing the interfaces of the objects to which the properties are attached. This can be used as a convenient dynamic extensibility mechanism that may be considered in future CIM Framework specifications.

3.6.2 Object properties are intended to be addressed in future revisions of this guide.

3.7 *Object Collections and Queries*

3.7.1 Object collections and queries allow flexible access to aggregate data for a group of objects. This capability may be a candidate to replace the limited operations for collections found in the component manager interface.

3.7.2 Object collections and queries are intended to be addressed in future revisions of this guide.

4 Referenced Standard

4.1 *SEMI Standard*

SEMI E81 — Provisional Specification for CIM Framework Domain Architecture

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 *ACID* — Atomicity Consistency Isolation Durability

5.1.2 *CIM* — Computer Integrated Manufacturing

5.1.3 *ECA* — Event-Condition-Action (rule)

5.1.4 *ECAA* — Event-Condition-Action-Alternative Action (rule)

5.1.5 *ENS* — Event Notification System

5.1.6 *ERP* — Enterprise Resource Planning

5.1.7 *GUI* — Graphical User Interface

5.1.8 *MES* — Manufacturing Execution System

5.1.9 *ODL* — Object Definition Language

5.1.10 *OMA* — Object Management Architecture

5.1.11 *OTS* — Object Transaction Service

5.2 *Definitions*

5.2.1 *application* — 1. one or more programs consisting of a collection of interoperating objects which provide domain specific functionality to an end user or other applications. 2. functionality provided by one or more programs consisting of a collection of interoperating objects.

5.2.2 *application interface* — the interface provided by an application or application program.

5.2.3 *application object* — an object implementing an application interface.

5.2.4 *architecture* — the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

5.2.5 *attribute* — an identifiable association between an object and a value. An attribute may have functions to set and retrieve its value.

5.2.6 *behavior* — the effects of performing a requested service, including its results (e.g., changes in the state of an object).

5.2.7 *binding* — a specific choice of platform technologies and other implementation-specific criteria.

5.2.8 *class* — the shared common structure and common behavior of a set of object implementations.

5.2.9 *client* — an object that uses the services of another object by sending messages to it or referencing its state.

5.2.10 *collection* — an object containing references to (collections of) other objects with services for managing them and providing access to them as a related group of objects.

5.2.11 *component* — a reusable package of encapsulated objects and/or other components with well-specified, published interfaces. The component is the element of standardization and substitutability for the CIM Framework.

5.2.12 *Computer Integrated Manufacturing* — an approach that leverages the information handling capability of computers to manage manufacturing information and support or automate the execution of manufacturing operations.

5.2.13 *conformance* — adherence to a standard or specification in the implementation of a product, process, or service.

5.2.14 *conformance requirement* — identification in the specification of behavior and/or capabilities required by an implementation for it to conform to that specification.

5.2.15 *conforming implementation* — an implementation that satisfies all relevant specified conformance requirements.

5.2.16 *event* — an asynchronous message denoting the occurrence of some incident of importance. For example, state change or new object created.

5.2.17 *event channel* — the intermediate object that forwards published events to interested subscribers.

5.2.18 *exception* — an infrastructure mechanism used to notify a calling client of an operation that an unusual condition occurred in carrying out the operation.

5.2.19 *extensibility* — the ability to extend or specialize existing components and add new object classes or components while preserving architectural integrity and component conformance to standards.

5.2.20 *framework* — a collection of classes or components that provide a set of interoperable services and functionality for a particular domain.

5.2.21 *implementation* — the internal view of a class, object or module, including any non-public behavior. The specific code and functionality that implements an interface.

5.2.22 *implementation conformance statement* — a statement made by the supplier of an implementation or system claiming to conform to one or more specifications and stating which capabilities have been implemented. It specifically includes the relevant optional capabilities and limits.

5.2.23 *infrastructure* — the services, facilities, and communications mechanisms that support the collaboration between and lifecycle of distributed objects.

5.2.24 *inheritance* — the ability to derive new classes, types or interfaces from existing classes, types or interfaces. For example, a derived class ("subclass") inherits the instance variables and methods of the base class ("superclass") and may add new instance variables and methods. In the CIM Framework, inheritance applies to interfaces and their specification of operations rather than implementations of classes.

5.2.25 *instance* — a software entity that has state, behavior and identity. The terms instance and object are interchangeable. An object is an instance of an interface if it provides the operations, signatures, and semantics specified by that interface. An object is an instance of an implementation if its behavior is provided by that implementation.

5.2.26 *interface* — the external view of a class, object, or module that emphasizes its abstraction while hiding its structure and internal behavior. An interface definition ideally includes the semantics of attributes and operations.

5.2.27 *interoperability* — the ability for two applications or the parts of an application to cooperate. In the CIM Framework, interoperability requires that application components be able to support specified relationships, share data, invoke each others' behavior (operations), return exceptions, and exchange events.

5.2.28 *lifecycle* — the life of an object, including creation, deletion, copy, and equivalence.

5.2.29 *message* — in object oriented systems a message is the means by which a client object invokes the behavior specified by an operation of a server object.

5.2.30 *message bus* — a software infrastructure that provides distributed communication between objects in component implementations. It can refer to an Object Request Broker, Microsoft DCOM, Java Remote Method Invocation or other infrastructure for conveying messages between objects.

5.2.31 *name-value pair* — a data structure that associates a name with an arbitrary value, typically used as an extensibility mechanism for conveying information by name-based retrieval.

5.2.32 *namespace* — a namespace is a bounded collection of names with a constraint to ensure that each name is unique within the collection.

5.2.33 *object* — a software entity that has state, behavior, and identity. The terms instance and object are interchangeable. An object is an instance of an interface if it provides the operations, signatures, and semantics specified by that interface. An object is an instance of an implementation if its behavior is provided by that implementation.

5.2.34 *object services* — interfaces for general services that are likely to be used in any program based on distributed objects.

5.2.35 *operation* — an operation is a specification entity, identified by an operation identifier, that denotes a service that can be requested. An operation has a signature that describes the legitimate values of request parameters and returned results, including any exceptions.

5.2.36 *persistent object* — an object that can survive the process or thread that created it.

5.2.37 *productive entity* — productive entity is an abstraction of a physical unit, which is involved in any way in a production process (e.g. production or supporting equipment). A productive entity has its own

internal logic and provides a software interface to access this logic

5.2.38 *query* — a message sent to a server (e.g. the productive entity) by a client interested in some information from the server (state of the productive entity). A query may or may not have arguments and it always has an answer. The semantics of a query is that some information from the server is returned, but the query cannot effect any change to the state of the server.

5.2.39 *service* — a function provided by a service provider that is performed through an operation specified by the provider.

5.2.40 *service provider, server* — an object providing services to other objects as specified by its published operations.

5.2.41 *signature* — a signature is the name, parameters, return values, and exceptions for a specific operation.

5.2.42 *substitutability* — the ability to replace a given component from one supplier with a functionally equivalent component from another supplier without impacting the other components or its clients in the system.

5.2.43 *trader service* — a collection of names with associated properties of features for each name and methods for manipulating and inspection that collection.

5.2.44 *type* — a declaration 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.

6 Technical Architecture Guidance

6.1 The computing infrastructure provides the distributed computing environment for CIM Framework applications. This infrastructure includes the operating system, networking and communications, data storage and access, user interface and presentation services, event distribution, systems management, and many other elements. Of these many infrastructure elements, a guide for technical architecture specifies a small subset of key services that need to be standardized in order to facilitate and streamline system integration between conformant CIM Framework implementations.

6.2 The CIM Framework relies on publicly available specifications to define the use of infrastructure services wherever such published specifications exist. The largest single source for openly defined specifications for distributed object services is the Object Management Architecture.[5] Reference to this guide does not imply that CIM Framework conformant implementations should implement the referenced services. The implementations that realize these infrastructure technologies are outside the scope of this guide. The technology choices made by implementers should be kept transparent to the CIM Framework to the greatest extent possible.

6.3 *Distributed Object Communication*

6.3.1 The CIM Framework documents assume the use of software infrastructure to provide distributed communication between objects in an implementation. The acronym ORB was originated by the OMG to describe its distributed object communication infrastructure, but is sometimes used in a more general sense. In this document ORB is used only to refer to the OMG specified technology and the more general term “message bus” is used for the diverse class of distributed communication mechanisms for communication between objects. The message bus is used to allow objects to make requests and receive responses from other objects. An object can communicate through the message bus with objects that are local or remote. Location transparency allows the object to remain ignorant of the actual location of the object with which it communicates.

6.3.2 A primary criterion for a message bus implementation is its ability to deliver all messages specified by the interfaces of the CIM Framework components. To accomplish this, the message bus should provide the ability to support or map a specified interface, including its inherited features, data types, operations, object references, and exceptions to and from runtime marshaled transport formats.

6.3.3 Alternate message bus implementation technologies are supported by the CIM Framework by mapping the OMG IDL for the CIM Framework interfaces into a specific message bus implementation. The Common Object Request Broker: Architecture and Specification[4] contains sections that define this mapping from CORBA to COM and from CORBA to OLE Automation. These sections, called the “Interworking Architecture,” cover detailed rules for mapping OMG IDL, types, and exceptions to compatible interfaces in COM and OLE Automation.

6.3.4 From a high level perspective, the DCOM and CORBA message buses are comparable. The DCOM capabilities are roughly equivalent to those of an ORB. However, with a lower level analysis, differences show up in data types, inheritance, object identity, and the handling of exceptions. The CORBA Interworking

5 Object Management Group. The Object Management Architecture Guide, Revision 3.0, John Wiley and Sons, New York NY, 1995.

Architecture[4] (Chapter 15, "Interworking Architecture," Chapter 16, "Mapping COM to CORBA," and Chapter 17, "Mapping OLE Automation to CORBA") defines mapping approaches covering:

Interface Mapping,

Interface Composition Mapping, and

Identity Mapping.

6.3.5 These areas should be addressed in order to provide a mapping between the OMG IDL used to specify CIM Framework interfaces and the message bus used for implementation. If the mapping is not specific (i.e., can occur in multiple ways) then two implementations may not necessarily be able to communicate even if they use the same message bus type. Although the CORBA Interworking Architecture is specific to Microsoft technologies, it could provide the foundation for future interworking mappings.

6.3.6 The current CIM Framework interfaces are specified in OMG IDL. The interfaces can be directly compiled and used with any of the available ORB implementations on the market.

6.3.7 The only way to provide such a direct solution using Microsoft DCOM would be to create additional CIM Framework interface specifications in DCOM MIDL and/or OLE Automation ODL. This would allow direct support for message bus functionality using DCOM. For example, the CIM Framework memory management requirement of "in" for parameter passing (see Section 6.3.15) would be directly supported by MIDL and DCOM but since the CORBA exception model is significantly richer than the DCOM exception model, mapping CORBA exceptions to COM would require an additional protocol to be defined for DCOM.

6.3.8 The CORBA Interworking Architecture supports mapping the current CIM Framework interfaces defined in OMG IDL to DCOM MIDL or OLE Automation ODL. This mapping is detailed enough that the mapping should always provide the same MIDL/ODL solution. Even though the current scope does not include interoperability between implementations on DCOM and CORBA (see Section 3.2), the issue of mapping the interface is still the same. This would also provide a step towards true interoperability using CORBA/DCOM bridge products that are beginning to become available.

6.3.9 Microsoft also provides an extension called OLE Automation. These interfaces are described in Object Definition Language (ODL). The OLE interfaces can be invoked dynamically by a client with no compile-time interface knowledge. The OLE data types are a subset of the types supported in DCOM, and there is no support for user-defined constructed types. The mapping solution differs for the DCOM and the OLE Automation. OLE Automation does not provide as clean a mapping from OMG IDL as DCOM does. This limitation may not allow some of the interfaces to translate completely to an OLE Automation implementation. Thus, component suppliers using an OLE Automation implementation should explain impact on interfaces that were not fully supported due to the Automation restrictions.

6.3.10 The CORBA Interworking Architecture covers mapping issues for the major areas of concern for the CIM Framework. The areas of primary importance are the Interface mapping, Interface composition mapping, Identity mapping, and Exception mapping. The CORBA Interworking Architecture gives detailed mappings for each of these areas and deals with the DCOM and OLE Automation mappings separately. The following five subsections summarize these mapping issues.

6.3.11 *Interface Mapping*

6.3.11.1 The OMG IDL primitives, constructed data types, and object references map closely to DCOM. The inherited CORBA interfaces may be represented as multiple DCOM interfaces. The CORBA attributes may be mapped to get and set operations in DCOM interfaces.

6.3.11.2 The OMG IDL primitives map to OLE primitives except for special cases. The OLE interfaces do not support constructed data types and should be mapped to specially constructed interfaces. CORBA object references map to OLE Automation interface pointers. There are difficulties in mapping CORBA multiple inheritance to OLE Automation interfaces documented in the CORBA specification.[4] CORBA attributes may be mapped to get and set operations in OLE Automation interfaces.

6.3.12 *Aspects* — The total behavior of a piece of a productive entity in a factory can be viewed as the union of distinct behaviors. Each such isolated behavior (or functional area) is called an aspect of the productive entity.

6.3.12.1 There is great variety in productive entity behavior. There are generic aspects that are shared by all or most productive entities (such as recipe management or process state model or material tracking) and there are aspects that are specific to one productive entity type or to a particular productive entity model. The behavior of each productive entity is the union of the particular aspects of that productive entity.

6.3.12.2 Saying that two pieces of productive entities have a certain aspect does not necessarily mean that they behave absolutely the same way. There are two

ways by which behavioral variation within an aspect can be modeled: Parameters and Variants.

6.3.13 *Parameterized Aspects* — An aspect can have parameters. Differences in productive entities behavior are modeled by assigning different values to the parameters. For example if a physical structure aspect of the productive entity specifies that the productive entity has a material buffer, the number of material units (buffer size) that can be placed on the buffer is a possible parameter. The number of buffers the productive entity has can be another parameter.

6.3.14 *Variants* — While parameterization is a very powerful tool, there are variations in behavior that cannot be simply modeled as different parameter values. In this case one can use variants. An aspect is said to have variants if there are some different behaviors related to the same aspect. For example the process control aspect can have a discrete variant and a continuous variant. In the discrete variant the productive entity processes discrete units of material (like the material within a magazine or a single wafer carrier), while in the continuous variant the productive entity processes continuously as long as there is material to be processed.

6.3.14.1 Differences between productive entities are best modeled as parameters when possible in order to avoid an explosion of the number of variants, while at the same time trying to maintain the clarity of the model.

6.3.14.2 A complete specification of the behavior of a piece of productive entity should specify variants for these aspects that have them.

6.3.15 Where do aspects and variants come from? — They leverage on previous work done by the industry. GEM (Generic Equipment Model, SEMI E30) is a primary source for identifying generic aspects. Various SEMs (Specific Equipment Models) are a source for specific aspects and very likely for variants. Other SEMI standards like SEMI E-40 (Standard For Processing Management) cover other aspects neglected by GEM.

6.3.16 *Specifying Productive Entity Interfaces in a Factory* — In order to specify the interface of a productive entity it is necessary first to identify the aspects (or variants of these aspects) supported by the productive entity, then to specify the interfaces associated with each aspect. The productive entity interface specification is then the sum of the interface specifications for all participating aspects.

6.3.16.1 The interface of an aspect (or variants of these aspects) is the sum of its Queries, Commands, Event Notifications and Service Requests and thus the problem of defining productive entity interface is reduced to the problem of defining the interfaces of individual aspects (or variants of these aspects).

6.3.17 *Architecture* — The productive entity in a factory is viewed as a composition of its aspects. The total productive entity behavior is therefore represented by the sum of all its aspects representing these behaviors. Each aspect specifies a specific behavior of productive entity and provides an interface for incoming messages (queries and commands).

6.3.17.1 An aspect has a name. The productive entity can answer a reference to one of its aspects given the aspect name:

```
AspectInterface aspectNamed(in string
aspectName);
```

6.3.17.2 An AspectInterface is a virtual interface that represents a generic aspect. All aspect interfaces inherit from the generic AspectInterface. When an aspect has variants, the aspect interface itself is virtual, and all its variant interfaces inherit from it.

6.3.17.3 Additionally the productive entity answers a list of the names of all its aspects:

```
StringList allAspectNames();
```

6.3.17.4 The usual scenario for a client is to acquire a reference to the productive entity. It then acquires references to the productive entity aspects of interest by querying the productive entity. The client then invokes methods on the aspect interfaces as required.

6.3.17.5 The client only needs to acquire aspect interfaces once when it first establishes communication with the productive entity. From then on it caches the productive entity interface as well as the references to the productive entity aspects for further use. The procedure of acquiring references needs to be repeated only in case the references become invalid (due to productive entity restart for example).

6.3.18 *How To Add A New Aspect* — The definition of an aspect follows the following aspect definition pattern. It includes the following items:

6.3.18.1 *Aspect Name* — Each aspect has a name that identifies it.

6.3.18.2 *Aspect Description* — The aspect description explains the productive entity behavior covered by the aspect. If the aspect has been derived from an existing standard, the description includes a reference to this standard. It explains the concepts and the used terminology, adds state models and state transition diagrams if required, and explains the interactions between the productive entity and the factory elements related to the aspect. If the interactions require certain sequences of messages, they are also described (as use cases or interaction diagrams).

6.3.18.3 The description also includes all the side effects and exceptions that can occur as a result of the interactions related to the aspect.

6.3.18.4 If the aspect has any relation or effects related to other aspects, they are also described here.

6.3.19 *Variants* — If the aspect has variants, each variant is named and described.

6.3.19.1 *IDL* — Usually the IDL will include a special module for the aspect. The module includes data type definitions specific to the aspect and usually a single interface that provides the various queries and commands of the aspect. An aspect that has variants has an interface per variant. The interfaces for the variants can be inherited from a common (abstract) aspect interface.

6.3.19.1.1 An aspect that deals with service requests should also include an IDL definition for the server that provides the services. Which is to be done in a separate module.

6.3.19.2 *Aspect Definition Example* — Here the Control State Aspect has been chosen as an example for aspect definition.

6.3.19.2.1 *Aspect Name* — Control State

6.3.19.2.2 *Aspect Description* — The definition of the Control State Aspect is based on Generic Equipment Model (GEM) SEMI standard E30.

NOTE 2: The state diagram presented here is simpler than the one in SEMI E30, since the internal sub-states are irrelevant to the productive entity interface, and the concept of HOST OFFLNE is obsolete in a distributed factory environment

6.3.19.2.2.1 The control state model defines the level of cooperation between the productive entity and the factory. It also specifies how the operator interacts in the different levels of factory control.

6.3.19.2.2.2 The control state model provides the factory with three levels of control over the productive entity:

6.3.19.2.3 *OFFLINE* — In the OFFLINE state, operation of the productive entity is done by the operator. In the OFFLINE state the productive entity accepts a query to find out the current control state and the command to change its control state, but rejects all other queries or commands (raising the rejected exception). While in the OFFLINE state the productive entity produces no events and no service requests.

6.3.19.2.4 *LOCAL* — In the LOCAL state the productive entity is operated by the operator.

6.3.19.2.4.1 In the LOCAL state the productive entity answers all queries from the factory and allows the factory to execute a limited set of commands. The commands that are prohibited are those that cause movement or directly affect the process. The productive entity sends events and asks for services from the factory.

6.3.19.2.5 *REMOTE* — In the REMOTE state the productive entity is controlled by the factory. The factory has full access to all the necessary commands to operate the productive entity through the full process cycle in an automated manner. The degree of automation can vary from productive entity to productive entity and from factory to factory. Generally operators are required to intervene in setup operations, operator assist situations, etc. Therefore when in REMOTE state, even though theoretically fully under control of the factory, the productive entity should not restrict the operator from executing essential operations such as selecting a recipe, pausing or resuming the process, operator assists, material movement to/from the productive entity, initiating recipe download and other productive entity specific commands on a command by command basis as needed. At the very least the operator should be able to change the control state, actuate an emergency stop and interrupt processing (stop, abort or pause).

6.3.19.2.6 The following diagram depicts the productive entity Control State model.

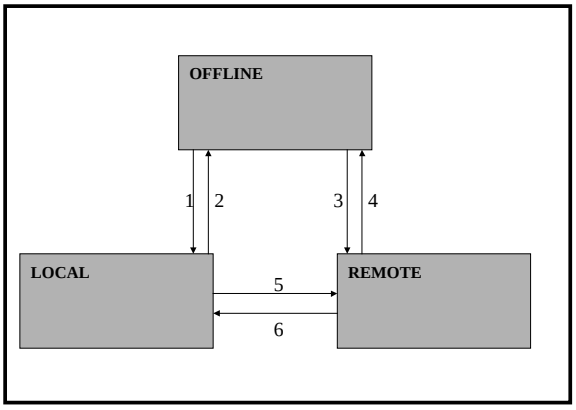


Figure 1

6.3.19.2.7 As can be seen from the diagram, transition from any state is allowed to the two others. The aspect provides the factory methods for querying the current state, for initiating a transition to any of the states, and for events when state transitions occur.

Variants : None

IDL

```
module ControlStateModule {
  // Type Definitions
  enum ControlState {OFFLINE, LOCAL,
  REMOTE);

       interface ControlState {
        // Queries
        ControlState getControlState()
      raises (EqBasicTypesModule::
      CommunicationFailure);

        // Commands
     void changeControlStateToOffline()
      raises (EqBasicTypesModule::
      CommunicationFailure);
     void changeControlStateToLocal()
      raises (EqBasicTypesModule::
      CommunicationFailure);
     void changeControlStateToRemote()
      raises (EqBasicTypesModule::
      CommunicationFailure);
       };
};
```

6.3.19.2.8 The control state aspect can post the following events:

```
ControlStateChangedToLocal
ControlStateChangedToRemote
controlStateChangedToOffline
```

6.3.20 *Interface Composition Mapping*

6.3.20.1 The DCOM interfaces do not support multiple inheritance. When multiple inheritance is used to extend functionality, the mapping is not very difficult. When multiple inheritance is used to "mix in" orthogonal behavior the mapping is more difficult. The CIM Framework interfaces that only use single inheritance provide the most reliable mapping. Interfaces that use multiple inheritance should follow the detailed mapping rules and ordering provided in the CORBA Interworking Architecture.

6.3.20.2 OLE Automation also has problems directly supporting the multiple inheritance of CORBA. The CORBA Interworking Architecture[4] provides detailed mapping rules for making the conversion where multiple inheritance is used.

6.3.21 *Identity Mapping*

6.3.21.1 CORBA and DCOM/OLE Automation have different notions of what object identity means. CORBA defines an object as a combination of the state and a set of operations that explicitly define the instance. An object reference is defined as a name that reliably and consistently denotes an instantiated object. A CORBA object exists until it is destroyed; its lifecycle is controlled by the server.

6.3.21.2 DCOM does not provide the same mechanism for identifying a particular object. DCOM objects are usually created when used and their state does not persist as an object instance. DCOM objects exist while they are referenced; their lifecycle is controlled by the client. This is true of OLE Automation objects as well.

6.3.21.3 The CORBA Interworking Architecture provides mapping solutions for managing the object lifecycle. The lifecycle issues should be minimized with the CIM Framework use of component managers to control object lifecycles. The implementation of DCOM lifecycle mapping should be encapsulated in the component manager.

6.3.22 *Naming*

6.3.22.1 As a mechanism to support initialization between collaborating components, the name of each component manager should be registered in a publicly available *namespace* along with the object reference for the component manager. The same name may be reused unambiguously as long as all occurrences belong to distinct namespaces. Other objects should be able to obtain a handle (object reference) to each component manager by utilizing the namespace through the operations of a naming service. A *trader* service may also be used to perform lookup of component managers and other objects based on well documented search criteria. Additional objects may also be registered in the namespace or trader as appropriate. The component manager serves as the namespace for the objects it manages, providing object references for named objects. Implementations should provide documentation on how to obtain available object references in either the namespace or the trader.

6.3.23 *Memory Management*

6.3.23.1 ORB implementations' memory allocation and deallocation services should handle all three types of OMG IDL parameter passing:

- *in* — Memory is caller-allocated and read-only. The caller is responsible for memory deallocation.
- *out* — Memory deallocation depends on the specific usage of the argument.
- *inout* — Memory allocation and deallocation depend on the specific usage of the argument.

6.3.23.2 The CIM Framework uses only the *in* parameter passing mechanism. Within operations that have had objects passed by reference, any modification of the object occurs by using the *in* parameter as a reference. Supplier provided extensions to the CIM Framework that use either the *out* or *inout* parameter passing mechanism should document the caller and callee responsibilities with respect to memory allocation and deallocation.

6.3.24 *Use of OMG IDL Module Packaging Constructs*

6.3.24.1 All IDL statements included as part of the specification of SEMI E81 should be contained within one or more CIM Framework defined modules.

6.3.24.2 *CIMFW Module Conventions*

6.3.24.2.1 Each CIM Framework sub-document that specifies IDL declarations should provide a full listing of the IDL statements in a compilable IDL file. This file may be presented as an appendix to the specification prior to final adoption and preparation for distribution with the standard. The IDL file should include module statements to enclose all IDL declarations for that specification.

6.3.24.2.2 IDL files (or appendices) should begin with a comment identifying the correct name of the file that contains the enclosed IDL specification. The following hypothetical example illustrates the form of this comment.

```
//File: CIMFactoryLabor.idl
//Part of the CIM Framework for the
Factory Labor Component
```

6.3.24.2.3 Every IDL appendix should contain the following statement identifying the top-level module that contains all CIM Framework declarations.

```
module CIMFW{

     ...

};
```

6.3.24.3 *Lower Level Modules within the CIM Framework Module*

6.3.24.3.1 The CIM Framework module should enclose second level modules that further package each partition of the CIM Framework specification as defined in SEMI E81. All IDL declarations should thus be scoped, first, to the CIM Framework, and second, to the specific component of the framework. The following example illustrates the positioning of elements of the Factory Labor component within a component module.

```
module CIMFW{

     module FactoryLabor{

          typedef ..., etc.

     };

};
```

6.3.24.3.2 Subsequent decomposition of CIM Framework specifications into a third level of module containment may be necessary in some places, but should be avoided where possible to keep fully qualified names from getting to an unworkable length.

6.3.24.4 *Conventions for CIM Framework Dependencies*

6.3.24.4.1 Each CIM Framework IDL file should include explicit statements identifying any other files in the CIM Framework specification set that contain modules that are referenced. These statements should be in the form of `#include` statements. In order to avoid circular references among related modules, a specification may need to partition a module into more than one file and include parts of the module at different points in the referencing file. This structure of IDL files is dependent on the specific implementation and which parts of the CIM Framework it implements and may be adjusted as needed to achieve successful compiles. The IDL files distributed with the standard should suggest a file structure to achieve a successful compile, but should not indicate that the file structure is specified as a part of the standard. The following example illustrates an include statement.

```
#include <CIMGlobal.idl>

module CIMFW{ ...
```

6.3.24.4.2 All references to elements of separate modules will then need to be fully qualified with the module scoping. For example, to reference a type defined in the CIMGlobal module, the reference should take a form similar to the following hypothetical example:

```
#include <CIMGlobal.idl>

module CIMFW{

     module FactoryLabor{ ...

          Global::MachineSequence
assignedMachines ( )

               raises
(Global::FrameworkErrorSignal);

     };

};
```

6.3.24.4.3 Fully qualified names may also be automatically generated if the target programming language compiler supports the “namespace” concepts. For example, the C++ standard uses namespace and Java uses package to support namespaces.

6.3.24.5 *Guard Statements*

6.3.24.5.1 To avoid the possibility of the same CIM Framework file being included more than once and thus causing multiple definition errors, each module should be preceded by the following type of guard statement.

```
#ifndef _CIM_FACTORY_LABOR_IDL_

#define _CIM_FACTORY_LABOR_IDL_

module ...

};

#endif // _CIM_FACTORY_LABOR_IDL_
```

6.3.24.5.2 The guard name should be designated by the string that begins and ends with an underscore and includes an all caps version of the filename with embedded underscores to separate the parts of the name. The guard statement should be documented for each module.

6.3.24.6 *Naming Modules and IDL Files*

6.3.24.6.1 All Module names for CIM Framework specifications are scoped within the CIMFW module and need not use redundant prefixing of the name with CIMFW. The names should be derived as closely as possible from the name of the CIM Framework specification they represent.

6.3.24.6.2 All Names should be composed of one or more words, abbreviations or acronyms concatenated together with capital letters used as delimiters between parts.

6.3.24.6.3 Names should be kept as short as possible while still providing understandable semantic associations for the subject module.

6.3.24.6.4 IDL File names should be based on the second level module contained within the CIMFW module.

6.4 *Exception Declarations*

6.4.1 Exceptions provide an alternative return mechanism for operations. When performing a normal return, control is returned to the point of invocation and the provided return values and output parameters are valid. Abnormal operation results raise an exception, which causes control to return to the defined exception handler and breaks the flow of control. Any data defined as part of the exception and provided by the called operation is valid and available to the exception handler. When an exception is raised, normal output parameters defined in the operation signature are not valid and are not available in the exception handler.

6.4.2 Exceptions are not communicated as an operation return code. An exception signifies that the post-conditions for successful operation completion have not been satisfied. If, on the other hand, the operation merely needs to communicate which one of multiple post-conditions were met, then the operation should provide this information in a return code or return structure as part of the normal operation completion.

6.4.3 Exception declarations in OMG IDL follow a *C* struct-like data structure with the keyword *exception* taking the place of *struct*. It contains attributes that can be used to pass information about an exception condition to a service requester. An exception is declared with an identifier (*ExceptionNameSignal*, the exception name), which is accessible as a value when the exception is raised, allowing the client to determine which exception has been received. Data values associated with the exception, if declared, are accessible to the client. The keyword *raises* is used in the operation definition to specify that a user-defined exception may be raised (or thrown in implementation terminology). The CIM Framework specifications assume that an operation may raise a CORBA-defined standard system exception, thus these exceptions are not specified.

6.4.4 The following conventions are used when defining exceptions:

Exceptions are not reserved for system or programming errors or failures, but should be included for any abnormal application behavior in the called service. This is consistent with CORBA usage of exceptions for application errors.

Exception descriptions are shown as OMG IDL comments similar to descriptions for services.

Return values should, if necessary, be implemented as fields in the exception definition.

All state transition services (e.g., *makeXXX* services) should include the *InvalidStateTransition-Signal* exception in their list of raised exceptions.

Services that perform a “find” or “lookup” function raise an appropriate *<ObjectType> NotFound-Signal* exception: a null return value is not an appropriate response. However, services that return a collection of objects do not raise an exception but simply return an empty collection.

Services that perform “add” functions raise a *<ObjectType>DuplicateSignal* exception for the case where the object to be added is already in the target collection or logical set. The exception includes a field containing a reference to the currently existing object.

Services that perform “remove” functions raise a *<ObjectType>NotAssignedSignal* exception for the case when the given object is not in the collection it was to be removed from or *<ObjectType> RemovalFailedSignal* exception if the object could not be removed.

Boolean query services should rarely raise an exception unless conditions are such that neither a true or false return value can be determined.

6.4.5 User-defined exceptions can be defined for any operation specified in IDL. Only user-defined exceptions that are defined and listed in the *raises* clause of an operation should be thrown. Interface-defined or other standard system exceptions may be thrown without using a *raises* clause on an operation. The data contained in the user exception should help the caller interpret and deal with the exception. Specific data can be defined for each user exception. Any additional information that assists in debugging situations should be sent to a tracing or logging facility.

6.4.6 Operations should throw standard system exceptions when an error condition clearly matches the defined exception. This ability should be used judiciously as the receiver of the exception may not be able to distinguish between a system-thrown or user-thrown exception. If a system exception does not clearly fit the situation at hand, then a user exception should be defined. Use of user-defined exceptions is part of the binding that should be considered for interoperability and substitutability.

6.4.7 The mapping defined for exceptions should support both the system exceptions and user exceptions. The CORBA model uses the concept of exceptions being raised to report error information. There should be exception specific data associated with the exception. The DCOM model provides error information by returning an HRESULT type. There is no facility for returning user-defined exception data.

6.4.8 The CORBA Interworking Architecture provides a mapping for the CORBA System Exceptions to the DCOM HRESULT values. The additional exception information for User Exceptions can be returned in an exception structure and added as another parameter to methods that include the *raises* keyword. The OLE Automation mapping provides for the use of a Pseudo-Automation Interface called a pseudo-exception. This is included in the interface as an additional out parameter.

6.4.9 The mapping of exceptions from CORBA to DCOM is very complex and requires an added parameter on many of the interfaces. The complete mapping rules are defined in the CORBA Interworking Architecture.

6.5 *Event Specification*

6.5.1 This guide uses a publish/subscribe model of events. Published events are sent to subscribers of the event in an asynchronous manner. The identity and quantity of subscribers are not known by the publisher of an event. The publisher is also known as the "supplier" and the subscriber known as the "consumer" of the event. Although the receipt of an event can alter the flow of control in the consumer, they should typically be used primarily as an information broadcast mechanism. Direct operation requests to another object should be used when affecting changes to critical flow of control to ensure message receipt by the intended recipient.

6.5.2 This guide suggests the use of an event delivery mechanism called "event channels." Event channels can provide a coarse grain filtering capability for events. Consumers can subscribe to a specific event channel in order to receive a particular event type. The event broker specification of this guide extends this event channel capability by adding features for locating event channel, registering for event delivery, and filtering the events of interest to minimize performance penalties when large numbers of events are present.

6.5.3 This guide supports creating, posting, and subscribing to events. It also provides the mechanisms to support subject-based addressing. When a consumer subscribes to events for a particular subject, it should be notified whenever an event for the subject is posted by any supplier.

6.5.4 The event delivery requirements are summarized as follows:

Suppliers do not know who the consumers of news events are; therefore, suppliers do not need to get a "handle" for them.

No response or answer is sent back from the consumer(s) to the supplier once the post is completed.

Message delivery is based entirely on message context (subject).

6.5.5 *Event Content*

6.5.5.1 The specification of event content should have two parts: a header and a body. The header should consist of information of a general nature regarding the event, such as the name of its subject (a subject string used for identification); an aging factor (for determining event effectiveness and may be site specific); priority; and any filtering information relevant to event delivery at a general level. The event body should contain: the actual event message; the original time of the event; and data relevant to filtering by the consumer of the event. The body should also contain any information required by the consumer not used in the filtering process (called "News"). The body should be extended with object references as required to facilitate communication with any objects associated with the event.

6.5.5.2 The event header is constructed by the supplier of the event and is used by an Event Notification System (ENS) to route the event to any consumer registering interest in the event. Note that the structure of the header may be specific to a particular ENS. The body of the event is constructed by the event supplier and is intended for use by the event consumer. Consumers of events express interest in an event type by passing the subject name and associated filter data along to the ENS. The ENS returns an original connection, called an event channel, to the consumer.

6.5.5.3 Name-value pairs are one mechanism that should be used to define data either in the header or body. Specific information within the body varies according to event type; issues such as allocation are implementation dependent.

6.5.6 *Subject String*

6.5.6.1 The event subject string should be defined as a multi-level hierarchy to assist in event classification and filtering. The levels designate the CIM Framework issuing the event component, the issuing interface within that component, and the event type. Each level should be delimited by a special character (e.g., a forward slash: "/"). An example of this syntax is */RecipeManagement/MachineRecipe/ParameterChanged*.

6.5.7 *Filter Data and News*

6.5.7.1 Filter data are attribute names, values, and operators that are specified by the consumer and are used by the filter subsystem to further qualify an event. News consists of additional attributes and values that are received by the consumer but not used in the filtering process and thus are not specified by the consumer. The consumer specifies the attributes, values, and operators upon which the event data is filtered. The actual filtering is performed after the supplier sends an event but prior to the consumer receiving the event. The filterable data should be well known and standardized. News may be used by the consumer to further filter the event, but the attributes and values are not standardized. Additionally, news may be used to convey the identity of the object generating the event to any consumer of the event.

6.5.7.2 An Event Broker is required to support subscription to an event channel that has the specified subject and supports filtering. The actual filtering mechanism is an implementation dependency. Filter data is passed to the Event Broker by the consumer to qualify the particular events that the consumer is interested in receiving. The filter data specifies filterable items in which the consumer is interested plus operators and operands to perform the filtering. The filtering sub-system should use the filter data to ensure that a specific event is passed to the consumer. The interface is *simple* and does not try to construct advanced logic to build the filter. The results of the filtering are *anded* together, such that the passed event should meet all of the filter criteria. The need to specify logical operators (e.g., "or") on the filter criteria or the use of query languages should be evaluated. Extensions to the filter structures would be required to support these additional capabilities.

NOTE 2: The location of the filtering subsystem is an implementation detail.

6.6 *Distributed Transactions*

6.6.1 Many operations defined in the CIM Framework are related to one another in complex ways that require multiple operations to be treated as a single unit of work. For example, grouping operations are combined with the ability to make an explicit decision to commit the aggregation of changes, or to abort all of the operations and return to the prior state. These grouping of operations are consistent with the familiar concept of a transaction (most commonly encountered in the context of database management systems). CIM Framework objects should be capable of participating in transactions as described below. However, the choice of how an object participates is implementation specific. For example, the implementer chooses the implementation technology and whether a change in object state is recoverable (that is, whether a change in state can be rolled back).

6.6.2 A transaction is a contract between two or more objects to perform some action based upon one or more requests in some context and having the ACID properties as follows:[6]

- Atomicity — State changes are atomic; either all happen or none happen. These changes include database changes, messages and events.
- Consistency — A correct transformation of state. Actions taken as a group do not violate any of the integrity constraints associated with the state.
- Isolation — Even though transactions execute concurrently, it appears to each transaction T that others executed either before T or after T, but not both.
- Durability — Once a transaction completes successfully (commits), its changes to state survive failures.

6.6.3 For example, consider a lot that starts processing in a piece of equipment. The states of the lot and the

6 J. Gray, Transaction Processing Concepts and Techniques, Morgan Kaufmann Publishers/Harcourt Brace and Co., 6277 Sea Harbor Dr., Orlando, FL, 1993.

equipment should change in order to accurately track the state of the factory. Coordinating the lot and equipment state changes as a transaction guarantees that factory state can be recovered accurately.

6.6.4 Transactions are created by a user of a service (the client) requesting an operation from a provider of a service (the server). To maintain the ACID properties of a transaction, the server should be able to return to the state prior to the request for the operation in the event of a failure. In this sense, the server should be recoverable. Failures are either software or hardware events that prevent the completion of the transaction. The server is a recoverable server if it is able to maintain the ACID properties when faced with a failure.

6.6.5 Transactions should be designed in a manner such that they do not span a period of interaction with an external entity such as a person using a GUI or a piece of equipment. Waiting for the response from an external entity can result in locks being held for multiple seconds, minutes, or longer. This can adversely affect other transactions by causing time-outs or deadlocks. These types of transactions can usually be split into multiple serially executed transactions with some small amount of extra revalidation of current states at succeeding transactions.

6.6.6 Coordinating the completion of transactions may involve multiple servers and may entail either committing the successfully completed transaction or rolling back the unsuccessful transaction. This activity imposes additional overhead on a system. Suppliers and consumers should assess the impact of transactions on system performance during component design and selection of transactional events.

6.6.7 Transactions may cause physical effects in the manufacturing system that cannot simply be rolled back in accordance with the ACID properties. Application and system designers should include ways to modify the logical view of the system to match the physical reality of the manufacturing floor if such a mismatch occurs.

6.6.8 Transactions can create CIM Framework events as state changes occur. As the transaction may not be committed at the point of event creation, the event should not be visible outside the transaction until the final commit point. The details of performing this task are implementation dependent. For example, the event announcing the completion of a lot at a processing step should not be published until the processing step completion transaction is committed. Otherwise, the event could be published, but the transaction subsequently rolled back, creating a system inconsistency.

6.6.9 Transactions should be able to be *nested*, thus providing the ability to define transactions within other transactions. These sub-transactions can generate additional sub-transactions, thus forming a hierarchy of transactions. In the spirit of maintaining the ACID properties, each sub-transaction can issue a commit or rollback for its piece of work. The results of the sub-transaction are only available to the parent transaction. The sub-transactions commit becomes permanent only after it issues a local commit and all ancestors commit. If the parent transaction does a rollback, all descendent transactions are rolled back regardless of any local commits.

6.6.10 There are two types of transactions widely supported for distributed object infrastructures. The OMA support for transactions is described in the OMG's Object Transaction Service (OTS). Microsoft provides support for transactions with its Microsoft Transaction Server (MTS) product. The OTS closely aligns with other standards such as The Open Group *Distributed Transaction Processing (DTP)* model.[7] Using industry standard protocols as a base, the OTS supports interfacing with products from the major database suppliers. Using the provided OTS interfaces and information about The Open Group standards, non-ORB supplied database interfaces could be developed to allow for interoperability with other cooperating transaction services. MTS also supports transactions with major database suppliers through use of The Open Group XA interface. Using the MTS Software Developer Kit, transaction support can be extended to other resources.

6.6.11 Combining heterogeneous components based on a combination of OTS and MTS is not straightfor-ward. Although both rely on the XA interface for dis-tributed transaction coordination, they are not designed to operate with each other. Both OTS and MTS hide the details of transactions from users. This makes either solution very convenient, but makes linking them together more difficult. For example, suppose a CORBA-based component adhering to OTS should interact with an MTS-based component. The scenario calls for the CORBA based component to use the XA interface to work with the MTS provided transaction coordinator (see Gray[8] for additional information on distributed transactions). However, in hiding XA complexity, OTS also hides the ability to readily specify the transaction coordinator (OTS does this behind the scenes). CIM Framework component developers and consumers should determine the relative need for distributed transactions spanning OTS and MTS against the additional complexity of developing a XA-based mechanism for combined OTS and MTS transactions.

7 The Open Group, Distributed TP: The XA+ Specification, Version 2, The Open Group, 11 Cambridge Center, Cambridge MA, 1994.

6.6.12 The ACID properties provided by either OTS or MTS may be used for CIM Framework transactions. Many implementation details are supplier specific; however, the major architectural principles have been described above.

6.7 *Component Management*

6.7.1 A component refers to a collection of related interfaces that form a coherent subsystem. Components may have a component manager to assist in the tracking and management of the instantiated interfaces (objects). The objects that are managed by a component manager are called managed objects.

6.7.2 *Component Level Interface*

6.7.2.1 Component managers provide services such as reporting on the collection of instances they manage and creating object instances. The component manager provides the following:

Object references to managed objects.

Collective queries for some aspect (usually a state) across all the objects it manages.

Services for:

Creating a managed object and returning a reference to it, or receiving an object reference to a newly created object. The component manager then "registers" the object reference.

Removing managed objects.

Finding managed objects.

6.7.3 *Component Manager Classification*

6.7.3.1 Component Managers are classified by their allowable number of instances.

6.7.3.2 *Unique Component Managers*

6.7.3.2.1 A unique component manager describes a component manager for which there is only one running instance in an MES implementation. Unique component managers are used when a single point of factory level control or focus is required. An example use of this pattern might be an interface within a Dispatcher component called DispatchingManager. This might be a unique component manager because multiple dispatching systems on the factory floor could create problems with work scheduling.

6.7.3.3 *Non-Unique Component Managers*

6.7.3.3.1 A non-unique component manager describes a component manager for which multiple instances may be running in an installed MES system. The component manager instances are derived from the same code base, but have separate instance data for each running instance. Non-unique component managers should be registered with the factory with some selection criteria in order for a requester to be able to obtain a handle to the correct instance. An example use of this pattern is the scenario in which several ProductManagers are employed within a production system.

6.8 *Architecture For Service Requests*

6.8.1 Services are implemented by a factory object that is the service provider. The productive entity in a factory invokes the service methods on that factory object.

6.8.1.1 For example, a recipe server could offer the following interface:

```
interface RecipeManagementServer {
     // upload a recipe
     void acceptRecipe(
          in string recipeName,
          in Recipe recipe);
     // download a recipe
     Recipe provideRecipe(in string
     recipeName);
};
```

6.8.1.2 The problem with service requests is that they require a flow from the productive entity to the factory object that provides the service. This kind of reverse flow contradicts the principle of layered architecture by which the productive entity is supposed to be a more primitive entity, unaware of factory objects, their locations and their structures. The trading service addresses this problem.

6.8.2 *Trading Service*

6.8.2.1 *Using Trading Service*

6.8.2.1.1 A trading helps clients to locate services. An object that must locate a service must know how to access the trading service.

6.8.2.1.2 A trading service relies on the description of the service itself, rather than any attribute relating to the server that provides the service (such as the name of the server). It must be able to describe the service it requires. The trading server locates a server that fulfills the required service profile.

NOTE 3: The trading service described here is based on the CORBA COS Trading Object Service, implementations of which are available from vendors of CORBA environments. The solution however does not require the full generality of the COS Trading Object Service, and can be viewed as a strict subset of the latter.

6.8.2.2 *The Trading Concept*

6.8.2.2.1 A trading scenario is based on a server that exports a service to a trader. The client then imports the service from the trader, receiving a reference to the server on which it can invoke the service.

6.8.2.2.2 This is depicted in the following diagram:

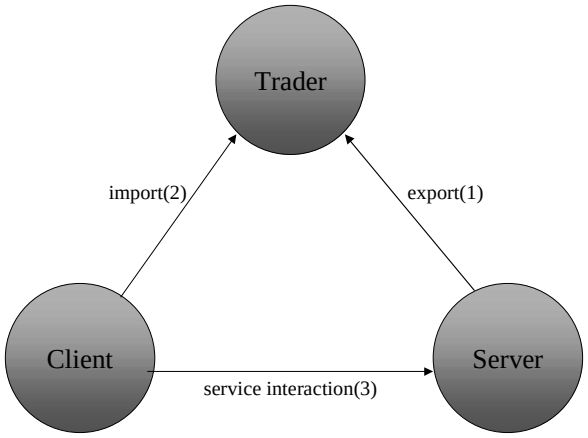


Figure 2

6.8.2.2.3 The diagram suggests that the server exports its service to the trader. In practice any object aware of the server and its provided services could assume this job. For example, a factory configuration object could be responsible for exporting all services to the trader.

6.8.2.3 *Trading Service Models*

6.8.2.3.1 The trader has to implement two interfaces:

- The Register interface allows other objects to register export service offers to the trader.

  NOTE 4: This interface is used e.g. by the service provider to inform the trader about the services the service provider offers.

- The Lookup interface allows other objects to lookup the trader for a required service.

  NOTE 5: This interface is used by e.g. the productive entity to find a service provider for a specific service.

6.8.2.4 *Export Use Cases*

6.8.2.4.1 The trader offers an interface named Register, that allows a server to register its services with the trader.

6.8.2.4.2 The IDL definitions given below are for illustrative purposes. They are extracted from the COS Trading Object Service specification [OMG]. For clarity, not all services are included here.

6.8.2.4.3 *Exporting a Service*

6.8.2.4.4 To export a service, the server uses:

```
OfferId export (
     in Object reference,
     in ServiceTypeName type,
     in PropertySeq properties
);
```

6.8.2.4.5 The server passes a reference to itself, and describes its offer by passing in a service type name and a list of properties of the service. The trader answers an offer id, through which the server can further manipulate its offer.

6.8.2.4.6 Factory service offers are described as follows:

6.8.2.4.6.1 *Service Type Name*

6.8.2.4.6.2 The type is a string naming the service itself. We are yet to agree upon the services supported by this specification. The following are obvious candidates:

- Recipe Service
- Wafer Map Service
- Fixtures Service[8]

6.8.2.4.6.3 *Properties*

6.8.2.4.6.3.1 Properties are used to characterize and specialize the service. A property is a name/value pair, where the name is a string naming the property, and the value specifies the property value offered by the server. The constraint language defined by the COS Trading Object Service [OMG] limits the type of values to the basic data types (such as numbers, chars, booleans and strings) and sequences of these.

6.8.2.4.6.3.2 The following properties are to be used for registering factory services:

- "Serviced Productive entities" — The value of this property is a list of the ids of the productive entities serviced by this server. This allows multiple servers of the same type to be installed, and partition the productive entity service among the available servers. Note that it does not require the server to know the productive entities it serves, since the server registration can be done by a factory configuration service.
- "Serviced Areas" — The value of this property is a string collection naming the factory areas serviced by the server. This property is another means of partitioning the service among multiple servers. The area could be a name of a cell controller if cellular manufacturing is practiced, or the name of any organizational unit implemented by the factory, and known to the factory configuration service.

6.8.2.4.7 *Withdrawing a Service*

6.8.2.4.7.1 To withdraw a registered service, the server (or the configuration service) uses:

```
void withdraw (
     in OfferId Id);
```

8 In back end fixtures is a generic name for durables and consumable materials

6.8.2.4.8 *Querying a Registered Service*

6.8.2.4.8.1 A server may query the trader the details of a registered service by passing in the offer id.

```
Struct OfferInfo {
     Object  reference;
     ServiceTYpeName  type;
     PropertySeq  properties;
};
OfferInfo describe (
     in OfferId  id);
```

6.8.2.4.9 *Modifying a Registered Service*

6.8.2.4.9.1 The server may modify the properties of a registered service. It may add new properties, delete existing properties, or modify the value of existing properties. This is done using the following method:

```
void modify (
     in OfferId  id,
     in PropertyNameSeq  del_list,
     in PropertySeq  modify_list
);
```

6.8.2.4.9.2 The properties named in the del_list are deleted. Properties in the modify_list that do not exist are added. Properties in the modify_list that exist, receive a new value.

6.8.2.4.9.3 The modify method can be used to support changes in the factory configuration, such as new productive entity being added or deleted, a productive entity being migrated from one cell to another, a new load balancing policy for the servers installed, etc.

6.8.2.5 *Import Use Cases*

6.8.2.5.1 *Importing a Service* — The trader offers an interface named Lookup that clients can use in order to locate a service:

```
void query (
in ServiceTypeName  type,
     in Constraint  constr,
     in Preference  pref,
     in PolicySeq  policies,
     in SpecifiedProps  desired_props,
     in unsigned long  how_many,
     out OfferSeq  offers,
     out OfferIterator  offer_itr,
     out PolicyNameSeq  limits_applied
);
```

6.8.2.5.1.1 The Query in Parameters — The “in” parameters are used by the client to specify the service it needs and the policies for searching it.

- The “type” parameter is key to the central purpose of trading. It specifies the name of the service type the client is interested in.
- The “constraint” guides the trader on how to select a server based on its registered properties. It is a string that describes the selection in some given constraint language. The typical constraints will select a server for a specific productive entity or for the area the productive entity belongs to. Both possibilities use the “in” operator for testing the inclusion of an element in a set. Some examples follow:

```
“DieAttachXYZ in ServicedProductive-
Entities”
“Cell22 in ServicedAreas”
```

6.8.2.5.1.2 “Preferences” specify how should a server be selected in case the query results in more than one answer. It is suggested that this parameter be ignored, which means that the default of first is always used.

6.8.2.5.1.3 The “policies” parameter guides the trader on how to choose a policy for performing the search. Search policies are a rather complicated issue, which can be ignored in the Simple Trader case.

6.8.2.5.1.4 The “desired_props” parameter instructs the trader which properties are to be returned as part of the answer (it does not affect the selection itself). This does not make much sense with the limited set of properties which has been defined, and can also be ignored (use none as the parameter value).

6.8.2.5.1.5 The “how_many” parameter is another way to restrict the number of answers. It is proposed that 1 always be used as the value of this parameter.

NOTE 6: Should areas be used as the selection criteria, the equipment must be aware of its area within the factory. This should be supported through the productive entity “Configuration” aspect.

6.8.2.5.1.6 *The Query out Parameters*

6.8.2.5.1.6.1 The query returns the selected servers in one of two forms: a collection of services or a reference to an iterator through which the returned servers can be obtained. The second method is designed for queries that may return a large number of offers. One can always assume that results are returned within the first out parameter (out OfferSeq offers), namely a sequence of offers. Furthermore, having specified 1 as the value of the “how_many” parameter, it is ensured that the answered sequence contains at most one element.

6.8.2.6 *Locating the Trading Service*

6.8.2.6.1 The productive entity locates the services it requires using a trading service.

6.8.2.6.2 A client can obtain a reference to the trading service by invoking the following method on the ORB:

```
Object resolve_initial_references (
     in ObjectId identifier)^
     raises (InvalidName);
```

Where:

the reserved name “TradingService” is passed as the identifier.

6.8.2.7 *A Usage Scenario Example*

6.8.2.7.1 Here is an example of a usage scenario.

6.8.2.7.2 In common factory practice, cellular manufacturing has a factory configuration service responsible (among other things) for exporting the factory services to the trading service. The following are some typical use cases:

1. The factory configuration service registers a wafer map server to server two cells named CellA and CellB.

   The factory configuration service obtains an initial reference to the trading service from the ORB:

   ```
   trader =
        orb.resolve_initial_refer
        ences("TradingService");
   The trader answers its Register
   interface:
        traderRegistry =
        trader.register_if();
   ```

   The factory configuration service builds properties as a single-itemed sequence of containing one name/value pair whose name is "ServicedAreas" and whose value is a sequence of the cell names { "CellA", , CellB"} . It then uses the Register interface of the Trading Service to register the service offer:

   ```
   traderRegistry.export(
        waferMapServer,"Wafer Map
        Service", properties);
   ```

2. The configuration manager informs a Die Attach equipment that it belongs to CellA.

   ```
   ProductiveEntities.setArea("CellA");
   ```

3. The Die Attach needs a wafer map.

   It obtains an initial reference to the trading service from the ORB:

   ```
   trader =
        orb.resolve_initial_refer
        ences("TradingService");
   ```

   The trader answers its Lookup interface:

   ```
   traderLookup = trader.lookup_if();
   ```

   The productive entity looks up the trading service for the wafer map service:

   ```
   traderLookup.query(
        "Wafer Map Service",
        "CellA in ServicedAreas",
        pref, policies, desired_props,
        1, preference, offers,
        offers_itr, limits_applied);
   ```

   The wafer map server is returned as the first element of the offers sequence. The productive entity may keep the reference to the wafer map server for future use.

4. The Die Attach can now invoke the service on the wafer map server:

   ```
   waferMapServer.getWaferMap(…);
   ```

7 Technical Architecture Conformance

7.1 Conformance is defined as "adherence to a standard or specification in the implementation of a product, process, or service." A conforming implementation should have an associated implementation conformance statement that details the capabilities that have been implemented. While recognizing that the CIM Framework is, by definition, not a complete specification of a MES, a guide for technical architecture defines conformance for each of its major requirements as follows.

7.2 *Distributed Object Communications Conformance*

7.2.1 The CIM Framework object model is based on the ability to issue service requests to a component object and to subscribe to events published by the component object. Component suppliers should explain how these two forms of communications are accomplished so consumers can assess the ease and feasibility of integrating a component into the factory MES. Example terminology specific to semiconductor manufacturing is provided for clarity. It does not preclude application specialization for other industries.

7.3 *Exception Conformance*

7.3.1 Alerting operation requesters of abnormal outcomes is essential for robust implementations. Component suppliers should explain how their implementations support raising specified system and user-defined exceptions.

7.4 *Event Specification Conformance*

7.4.1 Notification of asynchronous occurrences is a cornerstone of distributed event-driven application domains such as MES. Suppliers should explain how their components support event delivery, including the registration of event suppliers, event consumers and the provision for Event Broker features for event filtering.

7.5 *Distributed Transaction Conformance*

7.5.1 Ensuring consistent state changes among components is a key concern in the integration of a factory MES. Component suppliers should explain how their components support transactional units of work.

7.6 *Component Management Conformance*

7.6.1 Component suppliers should explain how object instances are managed. This includes how the object is identified, constructed, accessed, and destroyed (or flattened in the case of a persistent object). It also includes

mechanisms for query or lookup of specific managed object instances.

7.7 *General Rules for CIM Framework Conformance*

7.7.1 The following rules define the general expectations for technical conformance to any CIM Framework specification. Suppliers should provide documentation explaining any deviations from these general rules.

- All CIM Framework-defined operations for an interface should be supported.
- All exceptions and events for an interface should be supported.
- A component should use component manager interfaces for object instance creation and registration where these operations are specified.
- A component implementation should support all interfaces specified for that component.
- An application may not add states and transitions to the defined dynamic models that have external interfacing ramifications. The application may still further subdivide the states.
- A component implementation should explain how it supports substitutability. For example, it may support different degrees of substitutability between the following levels:
    - Strict — An application that supplies a CIM Framework component should be reconfigurable so it can use another supplier's implementation of that component. The application's interactions with the component are restricted to CIM Framework defined interfaces.
    - Weak — An application may use extended, pro-prietary, or private interfaces of a component. When the another supplier's implementation is substituted for an installed component, any components using the extended, proprietary, or private interfaces need to be reassessed and possibly modified. The use of the CIM Frame-work-defined interfaces does not change.

SEMI E97-0200A
PROVISIONAL SPECIFICATION FOR CIM FRAMEWORK GLOBAL DECLARATIONS AND ABSTRACT INTERFACES

This provisional specification 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 October 21 and December 15, 1999. Initially available at www.semi.org January 2000; to be published February 2000.

NOTE: This document was published twice during the February 2000 (0200) publishing cycle.

1 Purpose

1.1 This document defines the global declarations used by all other components of the CIM Framework and also specifies the common architecture patterns that serve to functionally integrate CIM Framework components. The material architecture defines functionality common to product management, durables management and consumables management components. The factory resource architecture defines relationships and common functionality of a variety of factory resources. The job architecture defines a factory-wide model for controlling factory jobs that drive a variety of manufacturing tasks. These specifications are separated into a distinct group to enable them to be specified once and then logically included or inherited wherever they are subsequently needed.

2 Scope

2.1 This specification provides the common interfaces required by Manufacturing Execution Systems to:

- Provide type definitions for common data structures to ensure consistent representation. These items include data types for common concepts such as coordinates, priorities, timestamps, and sequences of basic data types.
- Provide definitions for common exceptions used consistently throughout the CIM Framework.
- Provide the material architecture interfaces common to identifying, grouping, moving, locating and tracking material in the factory.
- Provide the factory resource architecture interfaces common to defining, organizing, tracking usage of and maintaining factory resources including equipment, sensors, durables, and people.
- Provide the job architecture interfaces common to creating, executing and managing work in the factory. The job architecture is specialized for material processing jobs, material transport jobs, resource maintenance jobs and factory jobs that drive product material through their process flows.

2.2 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 *Provisional Status*

3.1.1 This specification is designated as provisional due to known areas that need to be completed. The following items summarize the deficiencies of the provisional specification to be addressed before a subsequent ballot to upgrade it to full standard status.

3.1.2 The specification uses the IDL typedef “any” in several places. While this usage provides flexibility, it can have the effect of reducing interoperability due to differences in interpretation of the value provided by separate implementations that interact through a standard interface. The “any” typedefs should be replaced with explicit data types prior to upgrade from Provisional to full Standard status.

3.1.3 The definition of interfaces for retrieval of history associated with CIM Framework objects may need to be added to abstract interfaces in this document after the complete specification for the history facility within the CIM Framework Factory Services Component.

3.1.4 The specification of CIM Framework states reported through published state change events is currently based on a type definition for an enumeration of state values. There may be alternate representations for states that are better able to capture the semantics of nested and parallel states. The state representation used for the CIM Framework will be reviewed and possibly changed before upgrade to full standard status.

3.1.5 The Resource model defined in SEMI E81 includes several extensions that are not yet included in this specification. These extensions include composition of resources from other resources, capabilities associated with resources, and associations with tracking and maintenance functions for resources.

These extensions will be addressed before upgrade to full standard status.

3.1.6 The interfaces specified for MESFactory, Area, and ComponentManager are included in the E81 responsibilities for the Factory Component within Factory Operations. These interfaces will need to be moved to that component when it is considered in a future ballot.

4 Referenced Standards

4.1 *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 E81 — Provisional Specification for CIM Framework Domain Architecture

4.2 *Other Standards*

UML Notation Guide, Version 1.1, document number ad/97-08-05, Object Management Group[1]

ISO/IEC International Standard 14750 (also ITU-T Recommendation X.920) — Information Technology – Open Distributed Processing – Interface Definition Language [2]

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

5 Terminology

5.1 *abstract interface* — an interface specified only for inheritance rather than for implementation in order to standardize common features shared by all specializations of the interface.

6 Requirements

6.1 *Global Type Definitions*

6.1.1 This section describes data type definitions and exceptions that are global in nature. By "global" it is meant that they are designed to be used by any component within the CIM Framework. This specification does not address how they are implemented within a CIM Framework conformant application; that level of detail is within the realm of the development effort.

6.1.2 Global type definitions are specified as IDL declarations which may be referenced by any CIM Framework interface. The CIM Framework uses the keyword *typedef* to define aliases for basic object types, but with additional implied semantics. (e.g., the *units* typedef defines a *string*, whose contents conform to definitions found in SEMI E5). The keyword *struct* begins the type declaration for record structures composed of a collection of named and typed values. The third form of type definition is an enumerated type beginning with the reserved word *enum*. Enumerated types are used to declare a list of tokens that can be used as values of that type. Enumerated types are mainly used to denote the states of an object for communication in CIM Framework events. Finally, collections of values are declared with the keyword *sequence*. This kind of type definition may or may not imply significance to the ordering of the members in the sequence. Where no ordering constraint is mentioned, the elements of the *sequence* are not assumed to be in any meaningful order.

NOTE 2: In the following definitions, "//" or "/* .... */" delimits a comment.

NOTE TO THE READER: The comments in the following sections (described in NOTE 2, above) appear to immediately precede, rather than follow, the items they discuss. — SEMI Staff

6.1.3 All CIM Framework specifications will be declared within the context of the CIMFW module which spans all of the components of the CIM Framework. Within the CIMFW module, the global type definitions will be declared within a second-level module called Global.

---

1 UML Notation Guide v1.1 is available to the general public at http://www.omg.org/cgi-bin/doclist.pl, +1-508-820 4300, Object Management Group, Inc. , Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701.

2 ISO Central Secretariat, 1, rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland

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;
```

/* This type definition represents units for factory parameters, measurements. etc., and conforms to the SEMI E5 standard for representation of units. In that standard, the string contains a code representing a value of the units. For example, “ns” would mean nano-seconds; “A” for ampere; and “wfr” for wafer. */

```
typedef string Unit;

typedef string Units;
```

/* This type definition represents a sequence of *string* values. */

```
typedef sequence <string> StringSequence;
```

/* This type definition represents a sequence of *any* values. */

```
typedef sequence <any> AnySequence;
```

/* This type definition represents a sequence of *long* values. */

```
typedef sequence <long> LongSequence;
```

/* This enumerated type identifies event priorities and is used in each event definition. */

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

/* This enumerated type identifies the lifecycle states that an object may go through. It is used in event notifications of state changes. */

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

/* This enumerated type identifies the states of objects that can be reserved (Lot, Durable and Machine). It is used in event notifications of state changes. */

```
enum ReservationState {
        UndefinedReservationState,
        Reserved,
        UnReserved };
```

/* This enumerated type represents the SEMI E10 states for Machines and Support Resources. It is used in event notifications of state changes. */

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

/* *TimeT* is a *ulonglong* value (64 bits) that represents the number of 100 nanosecond increments that have passed since a base time (October 15, 1582 at 00:00, the Universal Time Representation which refers to time in Greenwich Mean Time). The specification for *TimeT* is: */

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

typedef ulonglong TimeT;
```

/* *TimeStamp* is mapped to the data type of *TimeT*. */

```
typedef TimeT TimeStamp;

typedef sequence <TimeStamp> TimeStampSequence;
```

/* The notion of a specific interval of time denoting a start time and an end time is represented as a struct called *IntervalT*. */

```
struct IntervalT {
        TimeT lower_bound ;
        TimeT upper_bound ;
};
```

/* *TimeWindow* is mapped to the data type *IntervalT*. */

```
typedef IntervalT TimeWindow;
```

/* Duration is mapped to the datatype TimeT. */

```
typedef TimeT Duration;
```

/* This structure is for the representation of a single schedule instance. It should be noted that “EndTime” should never proceed “StartTime.” */

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

/* The definition of a sequence of ResourceSchedules. This sequence is ordered in increasing time order and that order must be maintained in any manipulation of the sequence. */

```
typedef sequence <ResourceSchedule> ResourceScheduleSequence;
```

6.2 *Global Exception Declarations*

6.2.1 This section describes the standard CIM Framework exceptions that may be thrown by operations in any component.

Module: Global

/* This signal is raised when a lookup or find fails. */

```
exception NotFoundSignal { string errorMessage; };
```

/* This signal is raised when an add fails because an object already exists with the given identifier. Interfaces may also define and raise a <ObjectType>DuplicateSignal. */

```
exception DuplicateIdentifierSignal {
        string errorMessage;
        string duplicateIdentifier; };
```

/*This signal is raised when an invalid state transition request is made of an object. */

```
exception InvalidStateTransitionSignal {
        string errorMessage; };
```

/*This signal is raised when a “set” attribute contains a value out of range. */

```
exception SetValueOutOfRangeSignal {
        string errorMessage; };
```

/* This signal is raised when an incorrect TimePeriod is used. */

```
exception TimePeriodInvalidSignal {
        string errorMessage; };
```

/* This signal is raised when a Property name is not valid. */

```
exception InvalidPropertyNameSignal {};
```

/* This signal is raised when a a Property with this name is not defined. */

```
exception PropertyNotFoundSignal {};
```

/* This signal is raised when a Property is not supported. */

```
exception UnsupportedPropertySignal {};
```

/* This signal is raised when a Property is read-only and cannot be set. */

```
exception ReadOnlyPropertySignal {};
```

/* This signal is raised when no other defined signal matches the error condition. */

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

/* Definition of fields for FrameworkErrorSignal:

errorMessage — is a text field representing a description of the circumstances of the exception for use by developers in debugging the exception.

errorCode — is a numeric field representing the code for the given exception.

errorInformation — is any further debugging information related to the circumstances of the exception.

The errorCode has certain reserved values that are defined and standardized in the CIM Framework.

- 0000–0999 — reserved for the CIM Framework.
- 1000–1999 — reserved for extensions to the CIM Framework.
- 2000–2999 — reserved for specializations of CIM Framework interfaces.
- 3000–maximum — reserved for implementers. /*

/* This errorCode should be used for any operation where the supplier has chosen to not provide implementation, but needs to communicate to the user that nothing has happened as a result of this operation invocation. */

```
const unsigned long NOT_IMPLEMENTED = 0;
```

/* This errorCode should be used for any operation where the supplier assumes a specialization will implement this operation. If this exception is received, the user will realize that an interface has not been properly specialized. */

```
const unsigned long IMPLEMENTED_BY_SUBCLASS = 1;
```

/* This errorCode should be used for any operation where an unknown exception has been caught by the implementation and, rather than crashing, the implementation can map the “unknown” exception into this known exception. This probably does not aid in program debugging but does prevent program crashing. */

```
const unsigned long UNKNOWN_EXCEPTION = 2;
```

/* This errorCode should be used for any invocation where some unknown error has occurred that left the server object in an ambiguous state. */

```
const unsigned long COMPLETION_UNKNOWN = 3;
```

6.3 *Abstract Interface Type Definitions*

6.3.1 *Referenced Declarations*

6.3.1.1 The following declarations are not part of this specification, but are required for reference by Abstract Interface elements. These referenced declarations are defined in separate documents but are noted here as dependencies that appear in IDL compilations.

Module: EquipmentTracking

Interface: Machine

```
module EquipmentTracking {

interface Machine {}; // Stub

typedef sequence <Machine> MachineSequence;

exception MachineDuplicateSignal { };
```

```
exception MachineNotAssignedSignal { };

exception MachineRemovalFailedSignal { };

}; // module EquipmentTracking
```

Module: Labor

Interface: Person

```
module Labor {

interface Person {}; // Stub

typedef sequence <Person> PersonSequence;

exception PersonDuplicateSignal { };

exception PersonNotAssignedSignal { };

exception PersonRemovalFailedSignal { };

}; // module Labor
```

6.3.2 *Abstract Interface Declarations*

Module: AbstractIF

/* The following IDL interfaces will be fully defined in the sections below. They are declared here as forward references to support the sequence typedefs. */

```
interface Resource;

interface Material;

interface MaterialGroup;

interface JobSupervisor;

interface Job;

interface JobRequestor;
```

/* Type definitions for sequences of interfaces instances. */

```
typedef sequence <Resource> ResourceSequence;

typedef sequence <Material> MaterialSequence;

typedef sequence <MaterialGroup> MaterialGroupSequence;

typedef sequence <Job> JobSequence;

typedef sequence <JobSupervisor> JobSupervisorSequence;
```

6.4 *Resource Abstract Interface Group*

6.4.1 The Resource Abstract Interface Group provides a set of abstractions that are globally useful. Figure 1 is the Information Model for the Resource Abstract Interface Group.

Resource Abstract Interface Group

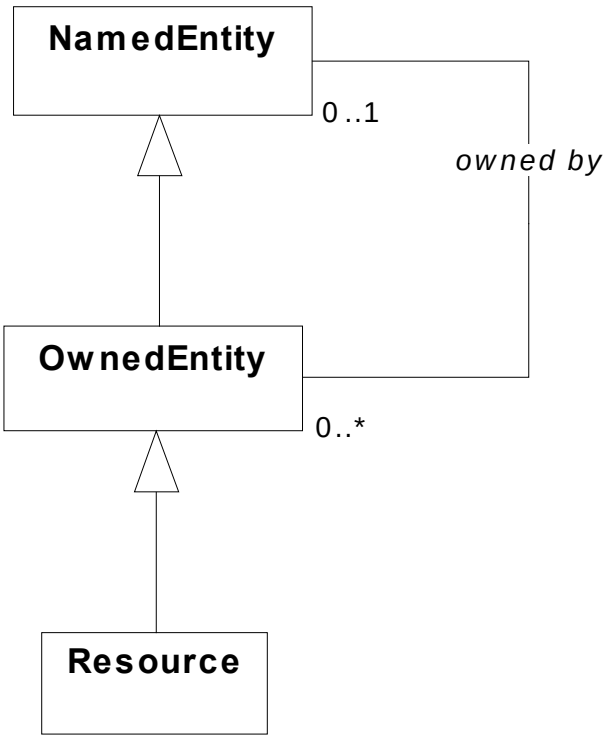


Figure 1
Resource Abstract Interface Group Information Model

6.4.1.1 All CIM Framework interfaces will inherit from one of the interfaces shown in Figure 1. NamedEntity provides the most basic naming functions. An OwnedEntity is a NamedEntity with functions supporting the concept of ownership. A Resource is an OwnedEntity which also takes an active role in product manufacturing.

6.4.2 *Named Entity Interface*

Module:	AbstractIF
Interface:	NamedEntity
Inherited Interface:	Implementation-dependent.
Description:	The abstract interface NamedEntity provides the concept of a named item. This allows for comparison and conversion of names via a standard object.
Exceptions:	None.
Published Events:	None.
Provided Services:	

```
interface NamedEntity {
```

/* Set and get the name. The NamedEntity interface does not specify scoping of names or enforce uniqueness of names. This could allow distinct instances of a NamedEntity to use the same string as a name. */

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

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

/* Tests the equality of the name with the name provided as an argument. */

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

}; // NamedEntity
```

Contracted Services: None.

Dynamic Model: None.

6.4.3 *Owned Entity Interface*

6.4.3.1 The concept of ownership in the CIM Framework relates to the hierarchical structure that may be defined where one object "owns" another object. This should not be confused with the business concept of ownership relating to an item's value as an asset.

Module: AbstractIF

Interface: OwnedEntity

Inherited Interface: NamedEntity

Description: The abstract interface OwnedEntity provides for the concept of an "owned" entity. There may be only one "owner" for each instance of an OwnedEntity. The OwnedEntity is able to communicate with the owner to request services, or forward information of interest. To build a "parts of" hierarchy, a series of ownerships can be established.

Exceptions: None.

Published Events: None.

Provided Services:

```
interface OwnedEntity : NamedEntity {
```

/* Set and get owner. */

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

NamedEntity getOwner ( )
        raises (Global::FrameworkErrorSignal);

}; // OwnedEntity
```

Contracted Services: None.

Dynamic Model: None.

6.4.4 *Resource Interface*

Module: AbstractIF

Interface: Resource

Inherited Interface: OwnedEntity

Description: Resource is an abstract inherited interface for any entity in the factory that takes an active role in advancing a product along its manufacturing life cycle (adds value). This includes the factory itself, personnel, production, planning and scheduling resources, and all of the machines used for processing, transporting, and storing materials. Resource provides a common set of services for monitoring and control. Resource uses the NamedEntity and OwnedEntity characteristics together to allow for the building of resource hierarchies.

There must be a clear division between the state of the Resource and the condition of the physical entity which the Resource represents. For instance, a Machine is a resource, but the fact that it is "Out of Service" may not mean the physical equipment is shutdown on the shop floor. In fact, the equipment may be operating in manual mode. The Resource state represents the availability of the Resource object to accept work for the factory system.

Exceptions: None.

Published Events: None.

Provided Services:

```
interface Resource : OwnedEntity {
```

/* Perform the startup activities for this Resource. Should be implemented by Resource specializations. */

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

/* Perform normal shutdown activities for this Resource. Normal is defined as allowing the Resource to complete any current activities and "gracefully" shutdown. */

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

/* Perform immediate shutdown activities for this Resource. Immediate is defined as aborting or terminating any current activities and stopping activity as soon as possible. This should be implemented by Resource specializations. */

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

/* Respond with the receiver's level in the Resource hierarchy. Resource specifies that each different type of Resource provide a "resourceLevel" identifier.

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

/* The following service provides name scoping for Resources. Resource name scoping makes use of the notion of "Resource level" and the ownership hierarchy. For example, unique identification of MachineResources within a Machine is possible, but to identify them outside the Machine additional information about their ownership will be required.

Thus: nameQualifiedTo ("Machine") sent to the ProcessResource named "Chamber" answers

"TestMachine>Chamber".

If the Machine was owned by a Factory named "TestFactory", then:

nameQualifiedTo ("Factory") sent to the ProcessResource answers

"TestFactory>TestMachine>Chamber"

where the ProcessResource has now been uniquely identified for the given Factory.

There is no limit to the number of levels that may be addressed this way. Based on the implementations of nameQualifiedTo (string); a name need not always be concatenated, if a particular Resource level is not applicable to identification. */

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

/* Returns the set of subordinate Resources for a given Resource. */

```
ResourceSequence subResources ( )
        raises (Global::FrameworkErrorSignal);
```

/* Answer if the resource is in service. In service means the resource is functional and ready to accept and perform its normal tasks. Derivatives of Resource are expected to expand this state (e.g., add sub-interfaces) that explicitly deal with such additional issues as capacity, “normal” work versus maintenance, etc. */

```
boolean isInService ( );
```

/* Answer if the resource is out of service. Out of service means the resource is unable to accept or begin new tasks. Previously begun tasks may continue in some cases. */

```
boolean isOutOfService ( );

}; // Resource
```

Contracted Services: None.

Dynamic Model:

Implementations of Resource may extend the state model by providing additional sub-states that are wholly contained within a state defined here. Extending the state model by the addition of state transitions is also an option for subtypes of Resource.

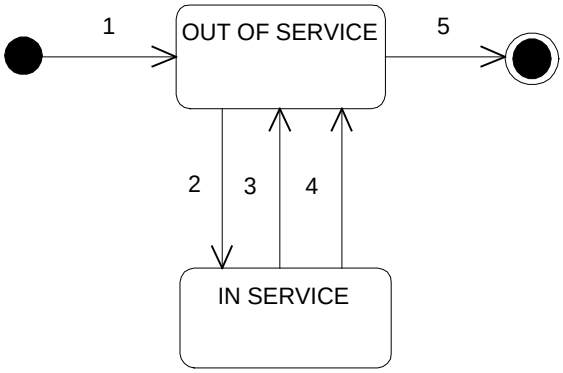


Figure 2
Resource Dynamic Model

Table 1 Resource State Definitions and Query Table

State	*Definition*	*Query for State via*
IN SERVICE	Resource is capable of interacting with other resources with its full service interface.	boolean isInService ( ); sent to the instance of Resource returns TRUE.
OUT OF SERVICE	Resource is not able to provide services	boolean isOutOfService ( ); sent to the instance of Resource returns TRUE.

Table 2 Resource State Transitions

#	*Current State*	*Trigger*	*New State*	*Action*	*Comment*
1	Non Existent	Object creation.	OUT OF SERVICE	None.	Default entry.
2	OUT OF SERVICE	startUp ( ) or object initialization.	IN SERVICE	Initiate Resource and subresources.	startUp() service initiates the trigger.
3	IN SERVICE	shutdownNormal ( ).	OUT OF SERVICE	Complete current execution of resources normally.	shutdownNormal() service initiates the trigger.
4	IN SERVICE	shutdownImmediate ( ). Take resource out of service.	OUT OF SERVICE	Stop execution of resource immediately.	shutdownImmediate() service initiates the trigger.
5	OUT OF SERVICE	Resource removed.	Non Existent	None.	

6.5 *Material Abstract Interface Group*

6.5.1 The CIM Framework uses a common architecture for identifying, grouping and locating materials. Any material has an identification, history, location, and associations to any material containers that contain it. Material can also be in multiple material groups that physically (same location or carrier) or logically (same lot, same process batch, same product family, etc.) associate material. Specializations of material include products, durables and consumables. Specializations of material groups include product groups, lots, and process groups.

Material Abstract Interface Group

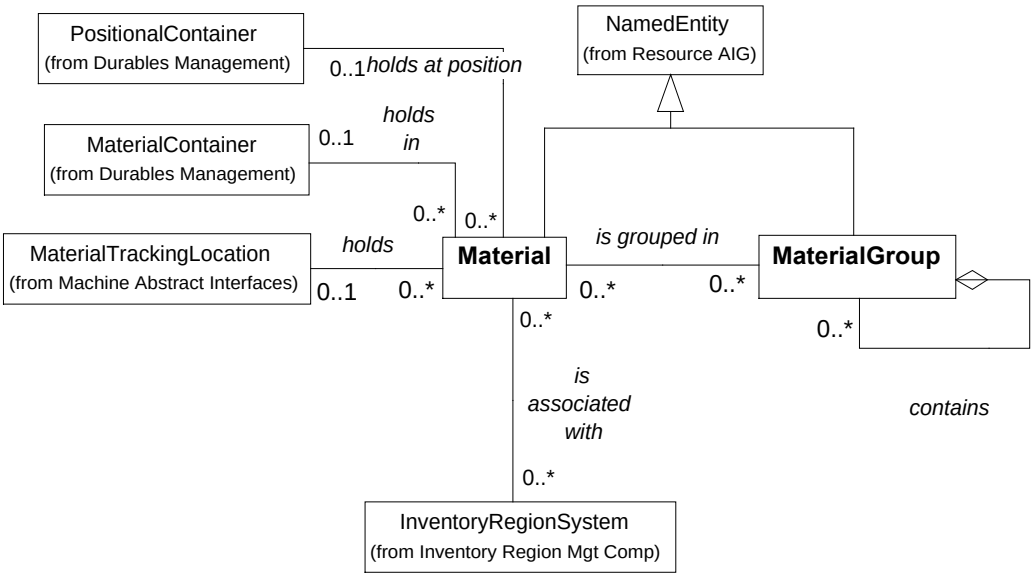


Figure 3
Material Abstract Interface Group Information Model

6.5.2 *Material Interface*

Module:	AbstractIF
Interface:	Material
Inherited Interface:	NamedEntity
Description:	Material is an abstract interface for physical items or substances that are required as inputs to the manufacturing process. This includes the product itself, consumables and durables used in the manufacturing process. It does not include the resources used to transform material into product.
Exceptions:	None.
Published Events:	None.
Provided Services:	

```
interface Material : NamedEntity {
```

/* Set and get the identifier for this material. The identifier is unique within the extent of all material for an MES Factory. */

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

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

/* Returns the material groups of which the receiver is a member. */

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

/* Answers whether the receiver is a member of the material group indicated by the argument. */

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

}; // Material
```

Dynamic Model: None.

6.5.3 *Material Group Interface*

Module: AbstractIF

Interface: MaterialGroup

Inherited Interface: NamedEntity

Description: MaterialGroup is the abstract interface for any aggregation of Material.

```
interface MaterialGroup : NamedEntity {
```

Exceptions:

/* Signals an attempt to add Material to the MaterialGroup that is already in the group. */

```
exception DuplicateMaterialSignal {Material aMaterial;};
```

/* Signals an attempt to add a MaterialGroup to a MaterialGroup that is already in the group. */

```
exception DuplicateMaterialGroupSignal {Material aMaterialGroup;};
```

/* Signals an attempt to remove Material that wasn't found in this MaterialGroup. */

```
exception MaterialRemovalFailedSignal {Material aMaterial;};
```

/* Signals an attempt to remove a MaterialGroup that wasn't found in this MaterialGroup. */

```
exception MaterialGroupRemovalFailedSignal {MaterialGroup aMaterialGroup;};
```

Published Events: None.

Provided Services:

/* Set and get the unique identifier for this group. */

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

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

/* Adds the argument MaterialSequence to the collection of Matrial held by the receiver. */

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

/* Adds the argument Material to the collection of Material held by the receiver. */

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

/* Removes the Material indicated from the MaterialGroup. Throws the exception if not found. */

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

/* Remove and return all Material from the MaterialGroup. */

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

/* Add the argument MaterialGroup to this MaterialGroup. */

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

/* Removes the argument MaterialGroup from this MaterialGroup. */

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

/* Remove and return all MaterialGroups from this MaterialGroup. */

```
MaterialGroupSequence removeAllMaterialGroups ( )
        raises (Global::FrameworkErrorSignal);
```

/* Returns the collection of material in this MaterialGroup. */

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

/* Returns all the MaterialGroups in this MaterialGroup. */

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

/* Returns the count of the items in the MaterialGroup. */

```
long count ( )
        raises (Global::FrameworkErrorSignal);

}; // MaterialGroup
```

Dynamic Model: None.

6.6 *Job Supervision Abstract Interface Group*

6.6.1 The Job Supervision Abstract Interface Group provides the abstractions common to creating, executing and managing a “job,” where a job can be defined as some system level operation which may be requested from the JobSupervisor. The job often spans a significant amount of time and multiple resources within the system. It is intended for specialization to provide specific job supervisors and jobs to provide system solutions.

Job Supervision Abstract Interface Group

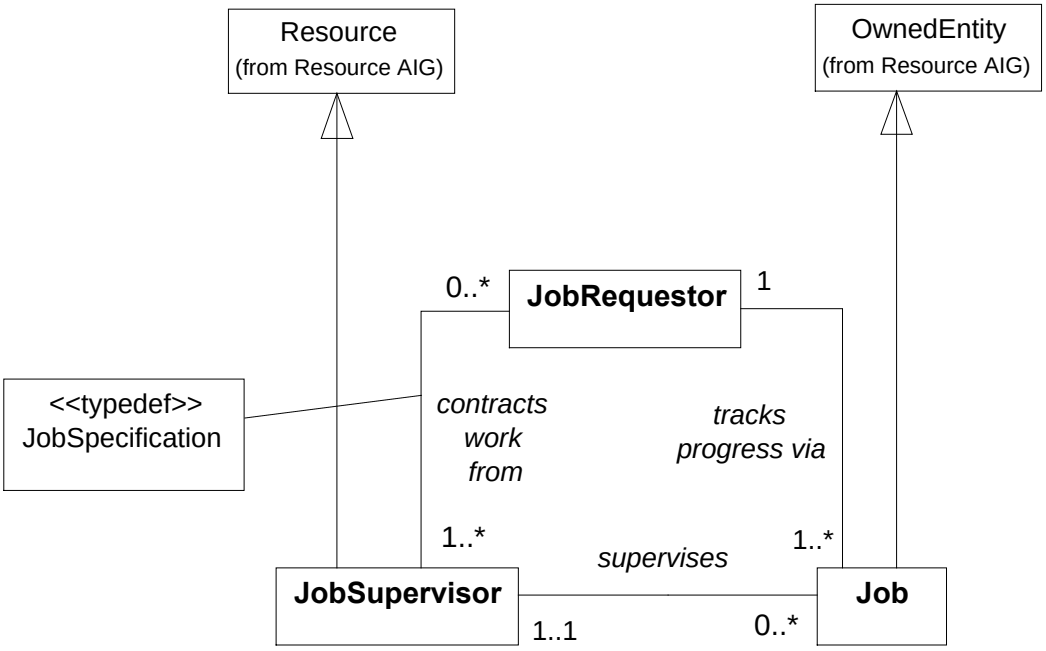


Figure 4
Job Supervision Abstract Interface Group Information Model

6.6.1.1 The basic Job Supervision Abstract Interface Group does not interact with other components, except to the extent that other components instantiate its interfaces. Figure 4 shows how the interfaces of Job Supervision relate to one another.

6.6.1.2 A Job Specification is a sequence of properties containing the parameters required to sufficiently define the work to be done. This sequence is passed by the JobRequestor to the JobSupervisor in the Job request message. See the JobSupervisor interface for more details.

6.6.1.3 JobSupervision levels are hierarchical. One level may accept a Job and delegate portions of that Job to lower levels. Jobs, however, are not purely hierarchical. A Job accepted by one JobSupervisor may be broken down, along with other Jobs of that component and reconstituted as needed to optimize the activities of the factory.

6.6.1.4 For example, a ProductRequest may ask for 15 wafers of a particular product. The ProductRequestManager may delegate a LotJob to Factory Operations with a Lot containing those 15 wafers and 10 more from a different ProductRequest. In the factory, this Lot may be split up and processed in smaller groups at various stages or, as scrap reduces the wafer count, combined with another small lot to create a more optimal process group. The Job Supervision implementation is allowed great latitude to optimize performance. Its only requirement is to fulfill the specification of the Job.

6.6.2 *JobSupervisor Interface*

Module: AbstractIF

Interface: JobSupervisor

Inherited Interface: Resource

Description: The JobSupervisor manages all the Jobs being performed by the component which implements it. It receives the requests for work, facilitates the creation of a Job for the task and returns (a reference to) that Job.

A JobSupervisor will have a well defined domain which it can call on to perform work. These may be CIM Framework Resources if it delegates the work, or internal resources if it performs the work itself. Only activity requests which can be accomplished within the domain of a JobSupervisor should be issued to/accepted by that JobSupervisor.

Jobs, as subtypes of NamedEntity, are named by the JobSupervisor in such a way that their name attribute may be used to query for the Job. Jobs from different JobSupervisors may have the same name.

The definition of the work to be performed is the JobSpecification, a sequence of name/value pairs (see "Properties" definition). Specializations of Job Supervision may require certain properties in the JobSpecification. Some commonly useful properties are defined in the following table. When possible, specializations should reuse these definitions. Specializations should also document the allowable and mandatory properties that are supported. See Job definition for more information.

```
interface JobSupervisor : Resource {
```

Type Definitions:

/* Type for returning results of Job execution. */

```
typedef Global::NameValueSequence Results;
```

JobSpecification Properties:

Name	*Value Type*	*Description*
"JobType"	string	The kind of Job to run. This is useful when a JobSupervisor can initiate more than one type of Job. String values reserved for this property are: ProductRequestJobType, LotJobType, AreaJobType, ProcessMachineJobType, TransportJobType, PMJobType, ControlStrategyJobType, AlgorithmJobType.
"Priority"	long	Integer value, ranges from 1 to 99, where 1 is the highest priority and 99 is the lowest.
"Deadline"	TimeStamp	The Job is expected to be completed no later than the specified value of the Deadline.

Exceptions:

/* Requested Job was rejected. */

```
exception JobRejectedSignal {
        string errorMessage; };
```

/* Requested Job was not found */

```
exception JobNotFoundSignal {
        string errorMessage;
        string missingJobName; };
```

Published Events:

/* Provide lifecycle event for tracking */

```
const string JobLifecycleSubject =
        "/JobSupervision/JobSupervisor/JobLifecycle";
```

/* The use of "name" here (and in all other events) indicates the string value for the name or identifier of the Job to which the event refers. Since filtering does not support object reference comparisons, the filtering must be on the "name" of the object. */

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

JobLifecycleFilters Properties:

Name	*Value Type*	*Description*
"Name"	string	The name of the Job.
"Lifecycle"	Global::LifecycleState	New lifecycle state.

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

Provided Services:

/* Request that work be done according to the referred specification. A Job which represents the work is returned for future reference. The post-condition for this operation is the specified Job successfully created. */

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

/* Request that work be done according to the Job specification. This operation blocks until the Job completes. Results generated by the Job are returned. The post-condition for this operation is a successful execution of the specified Job. This interface offers a lightweight alternative to ***requestJob*** that does not require that requestors support the JobRequestor interfaces (e.g., informJobStateChange operation). A Job may or may not be created by runJob, but if created, the Job may be accessed with the Job query operations of JobSupervisor. */

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

/* Ask whether the Job specified by the JobSpecification would be accepted for current or future (queued) processing if a requestJob or runJob message were issued now. */

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

/* Command to begin the pausing of all Jobs of this JobSupervisor which can be paused (e.g. Jobs that have not reached the Finished state). */

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

/* Command to "resume" all Jobs of this JobSupervisor which are currently Paused. */

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

/* This command aborts all the Jobs under the control of the JobSupervisor immediately, without regard to the impact of abruptly halting the Job. This service should be used with great caution. It may result in irrevocable change to factory or material state. */

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

/* This command stops all Jobs under control of the JobSupervisor, as quickly as possible. Stopping Jobs should not result in damage to the factory or material being processed. */

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

/* This service removes a Job which is in the terminated state. If required, persistent information about the Job may be captured in a history entry. Jobs may also be removed based on other archiving rules. */

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

/* Find a Job by name. A Job is a NamedEntity */

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

/* Find a queued Job by name. */

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

/* Find an active Job by name. */

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

/* Find a canceled Job by name. */

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

/* Find a finished Job by name. */

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

/* Return all the specified Jobs. The JobSequence may be empty. */

```
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
```

Contracted Services: None.

Dynamic Model: Inherited.

6.6.3 *Job Interface*

Module: AbstractIF

Interface: Job

Inherited Interface: OwnedEntity

Description: The Job interface represents a unit of work requested of an associated JobSupervisor and performed (or facilitated) by a factory entity. A Job generally results in some change of the overall factory state. How the entities that supply the Job and JobSupervisor interfaces actually perform the work (or delegation of work) is an implementation decision. A Job is expected (but not required) to take a non-zero time to perform and have a non-zero chance of refusal or failure. A Job may encapsulate a decomposition into a sequence of jobs/tasks/activities which are delegated to lower level job supervisors. The Job exists during the execution timeframe. The more persistent record of the Job should be maintained in a history entry.

```
interface Job : OwnedEntity {
```

Exceptions:

Published Events:

/* Any time the Job’s state changes. */