

IEEE Standard Interface for Hardware Description Models of Electronic Components

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

Approved 16 December 1998

IEEE-SA Standards Board

Abstract: The standard interface for hardware description models of electronic components is defined. The primary audiences of this standard are model developers and implementors of software supporting this interface.

Keywords: execution, intellectual property, model integration, model manager, Open Model Interface (OMI), simulation, simulator/model interoperability

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 24 November 1999. Printed in the United States of America.

Print: ISBN 0-7381-1555-X SH94716
PDF: ISBN 0-7381-1556-8 SS94716

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

(This introduction is not part of IEEE Std 1499-1998, IEEE Standard Interface for Hardware Description Models of Electronic Components.)

The Open Model Forum (OMF) was founded to solve the problem of logic model availability. Because several languages are used to develop such models, including VHDL, Verilog, and C, many models are not available for specific simulators. The OMF addressed this problem by defining a programming interface between models and simulators that allowed models developed in one language to be used with simulators that support other languages.

This programming interface, known as the Open Model Interface (OMI), was developed to provide a simulator independent interface for complex digital integrated circuit (IC) models. The interface was designed to improve model availability, streamline distribution, and reduce development costs. The existence of an open procedural interface to simulators permits OMI-compiled models to be used with any OMI-compliant simulator, regardless of the language in which the models are developed. This interface allows libraries of models developed in a variety of languages to be merged into one comprehensive, interoperable library. It preserves users' investments in model libraries because the same models can be used with various simulators as the users' simulation environments evolve and because model libraries can be passed easily between workgroups using different simulators.

The OMF Executive Committee transferred the OMI specification to the IEEE P1499 Working Group for consideration as the basis for a standard for integration of electronic hardware models. This standard is the result of the efforts of the working group in refining the baseline document.

The technical direction of the working group and the day-to-day activities of issue analysis and drafting of proposed wordings for the standard were the responsibility of the IEEE P1499 Technical Committee. Without the dedication and hard work of this group, it would not have been possible to complete this document. The technical committee consisted of the following persons:

Doug Dunlop, *Chair and Technical Editor*

Kathryn McKinley, *Technical Editor*

Frank Guerino
Kathy Lang

Steven R. McMaster
Viswanathan Ramakrishnan

John Stickley
Andrew R. Wilmot

At the time that this standard was completed, the following persons were members of the IEEE P1499 Working Group:

Gabe S. Moretti, *Chair*

Dennis B. Brophy
Douglas D. Dunlop
Frank Guerino
William A. Hobbs
Martin Lampard
Kathy Lang

Kathryn McKinley
Steven R. McMaster
Larry Melling
Jean P. Mermet
Stephen J. Peters
Viswanathan Ramakrishnan

Manohar Sambandam
Moe Shahdad
John Stickley
Raymond Tang
Cary Ussery
Andrew R. Wilmot

The following members of the balloting committee voted on this standard:

Peter J. Ashenden	Rita A. Glover	Sashi Obilisetty
Stephen A. Bailey	William A. Hanna	Serafin Olcoz
David L. Barton	Gregory A. Haynes	Quentin G. Schmierer
John Bastian	William A. Hobbs	Peter Schwarz
J. Bhasker	Jake Karrfalt	Moe Shahdad
William Billowitch	Stanley J. Krolikoski	Stuart Sutherland
Dennis B. Brophy	Oz Levia	Peter Ting
Kent Dalton	Francoise Martinolle	Jose A. Torres
Allen Dewey	Paul J. Menchini	Ajayharsh Varikat
Douglas D. Dunlop	Jean P. Mermet	Ronald Waxman
Peter Flake	Gabe Moretti	John Willis
Vassilios Gerousis		Andrew R. Wilmot

The final conditions for approval of this standard were met on 16 December 1998. This standard was conditionally approved by the IEEE-SA Standards Board on 8 December 1998, with the following membership:

Richard J. Holleman, *Chair*

Donald N. Heirman, *Vice Chair*

Judith Gorman, *Secretary*

Satish K. Aggarwal
Clyde R. Camp
James T. Carlo
Gary R. Engmann
Harold E. Epstein
Jay Forster*
Thomas F. Garrity
Ruben D. Garzon

James H. Gurney
Jim D. Isaak
Lowell G. Johnson
Robert Kennelly
E. G. "Al" Kiener
Joseph L. Koepfinger*
Stephen R. Lambert
Jim Logothetis
Donald C. Loughry

L. Bruce McClung
Louis-François Pau
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Hans E. Weinrich
Donald W. Zipse

*Member Emeritus

Noelle D. Humenick
IEEE Standards Project Editor

Contents

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	1
1.3	Structure and terminology of this standard.....	1
1.4	Semantic description.....	2
1.5	Front matter, annexes, index, examples, figures, and notes	2
1.6	Interface naming conventions	2
2.	References.....	3
3.	Introduction to OMI.....	3
3.1	OMI software architecture	3
3.2	OMI compliance	5
3.3	Creation of OMI models	5
4.	Model integration.....	6
4.1	Model integration software requirements	6
4.2	Model delivery	7
4.3	Characteristics of OMI applications	8
4.4	Classification of OMI simulators.....	9
4.5	Simulator/model interoperability	9
5.	Execution	10
5.1	Stages of execution	10
5.2	Bootstrap stage.....	12
5.3	Elaboration stage.....	17
5.4	Initialization stage.....	20
5.5	Simulation stage.....	21
5.6	Termination stage.....	21
5.7	Execution control	21
6.	Simulation	28
6.1	Simulation fundamentals	28
6.2	Event-driven simulation reference model.....	30
6.3	Cycle-based simulation reference model	31
6.4	Simulation cycle callbacks.....	33
6.5	Dynamic data transfer	36
6.6	Reverting to a previous state.....	40
6.7	Backannotation	42
7.	Model information	43
7.1	OMI information modeling methodology.....	44
7.2	The root object	47
7.3	Library objects	47
7.4	Models.....	47

7.5	Data objects.....	49
7.6	Data types.....	54
7.7	Timing definitions.....	61
7.8	Clocking specifications.....	62
7.9	Attributes.....	64
7.10	Model information access.....	65
8.	Programming interface fundamentals.....	71
8.1	Caller data.....	71
8.2	Errors.....	72
8.3	String buffers.....	75
8.4	Opaque references.....	77
8.5	Simulation time.....	77
8.6	Values.....	78
8.7	Logic systems.....	80
9.	Programming interface reference.....	85
Annex A	(informative) Predefined attributes and commands.....	143
Annex B	(informative) OMI headers.....	150
Annex C	(informative) Object diagrams.....	168
Index		185

IEEE Standard Interface for Hardware Description Models of Electronic Components

1. Overview

1.1 Scope

The intent of this standard is to accurately define the standard interface for hardware description models of electronic components. The primary audiences of this standard are model developers and implementors of software supporting this interface.

1.2 Purpose

This standard defines an open interface between models and simulators that enables model interoperability while protecting the model provider's intellectual property. This software interface is called the Open Model Interface (OMI). It permits an OMI-compiled model to be used with any OMI-compliant application that supports that class of OMI model, regardless of the programming language in which the model was developed.

1.3 Structure and terminology of this standard

This standard is organized into clauses, each of which focuses on a particular area of the definition of the specification. Within each clause, individual constructs or concepts are discussed in each subclause.

Each subclause describing a specific construct or concept begins with an introductory paragraph. A set of paragraphs providing more detail then follows. These paragraphs may provide additional semantic description, define portions of an information model, or describe the interface of an OMI function. Finally, each subclause may end with examples, figures, and notes.

1.4 Semantic description

The meaning of a particular construct or concept and any related restrictions are described with a set of narrative rules in which an italicized term indicates that the term is defined within the narrative.

Mandatory weight is indicated by use of the declarative verb form or by use of the word “shall.” A particular form of noncompliance may be described as *erroneous* or as an *error*. These terms are used in the semantic descriptions with the following meaning:

- *erroneous*: The condition described represents a form of noncompliance that an implementation is not required to detect or report. A condition is deemed erroneous only when it is either very difficult or impossible in general to detect the condition during the processing.
- *error*: The condition described represents a form of noncompliance that an implementation is required to detect and to report as an error.

1.5 Front matter, annexes, index, examples, figures, and notes

Prior to this clause are several pieces of introductory material; following the final clause are three annexes and an index. The front matter, annexes, and index serve to orient and otherwise aid the user of this standard but are not part of the definition of this standard.

Some subclauses of this definition contain examples, figures, and notes; with the exception of figures, these parts always appear at the end of a subclause. Examples are meant to illustrate the possible forms or uses of the construct described in this subclause. Figures are meant to illustrate the relationship between various constructs or concepts. Notes are meant to emphasize consequences of the rules described in the clause or elsewhere. In order to distinguish notes from the other narrative portions of the definition, notes are set as enumerated paragraphs in a font smaller than the rest of the text. Examples, figures, and notes are not part of the definition of the specification.

1.6 Interface naming conventions

The names introduced by the OMI follow these naming conventions:

- a) Names do not contain underscores.
- b) The first letter of a name is a lowercase letter, unless it is part of an acronym.
- c) Proper names and acronyms within a name are capitalized as they would be in normal text.
- d) Different words within the same name are distinguished by capitalizing the first letter of each word after the first word; therefore, subsequent letters are lowercase unless a letter corresponds to the first letter of a new word (or the name is a proper name or acronym, see above).
- e) All names declared in the outermost scope begin with the prefix `omi`.
- f) Type names end with the suffix `T`.

As a convenience to the reader, the `Courier` font is used to highlight items that are defined in the OMI programming interface.

2. References

This standard shall be used in conjunction with the following publications:

IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.¹

IEEE Std 1076-1993 (Revision of IEEE Std 1076-1987), IEEE Standard VHDL Language Reference Manual.

IEEE Std 1164-1993, IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164).

IEEE Std 1364-1995, IEEE Standard Description Language Based on the Verilog™ Hardware Description Language.

ISO/IEC 646:1991, Information technology—ISO 7-bit coded character set for information interchange.²

ISO/IEC 9899:1990, Programming languages—C.

OVI Standard Delay Format Specification, Version 2.1.³

3. Introduction to OMI

The OMI defines a protocol for interaction between a model and an application using that model. It defines a set of programming interfaces and corresponding rules for their use that enable model interoperability and limit access to internal model information.

3.1 OMI software architecture

An OMI *model* is a tailorable, functional representation of a device or system that is delivered in object code format. A model may encapsulate a simulatable specification, timing information, physical characteristics, and other data related to the device or system. A customized version of a model that has been incorporated into a hardware or system design is called a *model instance*. A single model may have any number of model instances associated with it.

A model may be used by a variety of applications. An *application* is a software tool that accesses one or more models. A *simulator* is a special kind of application that is capable of simulating the behavior of a hardware or system design. Throughout this standard, the use of the term “simulator” in a particular context implies that a simulation capability is required or assumed in that situation.

Privileged internal information that is associated with a model or an application is called *intellectual property*. The OMI model delivery mechanism protects intellectual property by enabling the delivery of models in object code format and by providing an interaction mechanism that allows a model and an application to control communication through calls to a small set of interface functions. The only information available about a model is the information at the port boundary and information relating to selected internal objects that the model supplier chooses to make visible to the external environment.

¹IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://standards.ieee.org>).

²ISO and ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iso.ch/>). They are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA (<http://www.ansi.org/>).

³OVI publications are available from Open Verilog International (OVI), 15466 Los Gatos Blvd, Suite 109-071, Los Gatos, CA 95032.

An OMI model is integrated into an application through a *model manager*, which coordinates the interaction between the application and the models under its control. Every model (and derived model instance) has a model manager associated with it. An application may interact with any number of model managers, as depicted in Figure 1.

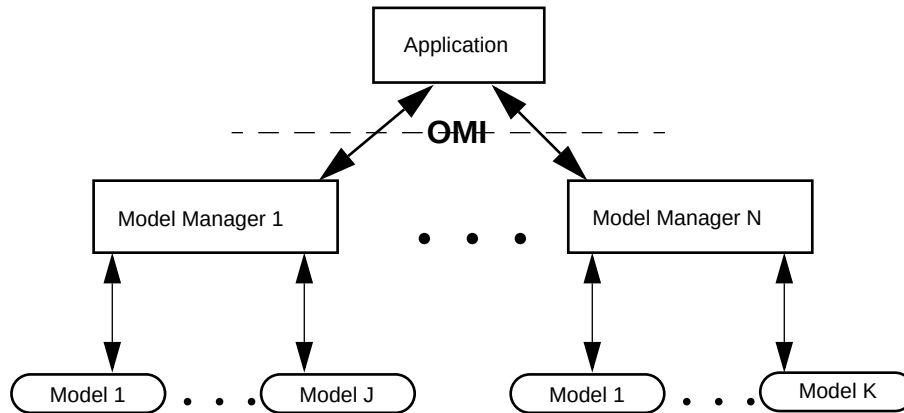


Figure 1—OMI model integration

A model manager performs a number of services. It provides general model packaging information (e.g., supplier, revision). It also handles information relating to the behavior of a model in a particular context, for example, by providing mechanisms for creating instances, setting parameter values and binding ports during instantiation, relaying data values, scheduling port updates, and exporting internal objects for external viewing.

All interaction between a model manager and its models is private. In addition to the essential OMI functionality described above, a model manager may supply its models with internal services, such as error handling or internal scheduling.

A model may be packaged separately or as a part of a *model library* that serves to collect together a set of models under the control of a common model manager, as illustrated in Figure 2. Libraries are a convenience for dealing with multiple models; a model is not required to exist in a library. The same model management functionality is required whether the model manager handles a single model, a model library, or a combination of models and libraries.

The OMI software elements are conceptually separate parts; however, a model supplier is not required to provide them as separate software components. A model manager may be directly embedded in the model itself or provided as one or more separately linkable objects.

The OMI interface is defined in the ANSI C programming language (see ISO/IEC 9899:1990⁴). The OMI follows an interactive protocol in the sense that function calls across the interface go in both directions. Due to this interactive nature, the OMI consists of two parts: one provided by the application and one provided by the model manager. This interface is defined in detail by the rest of this standard.

⁴Information on references can be found in Clause 2.

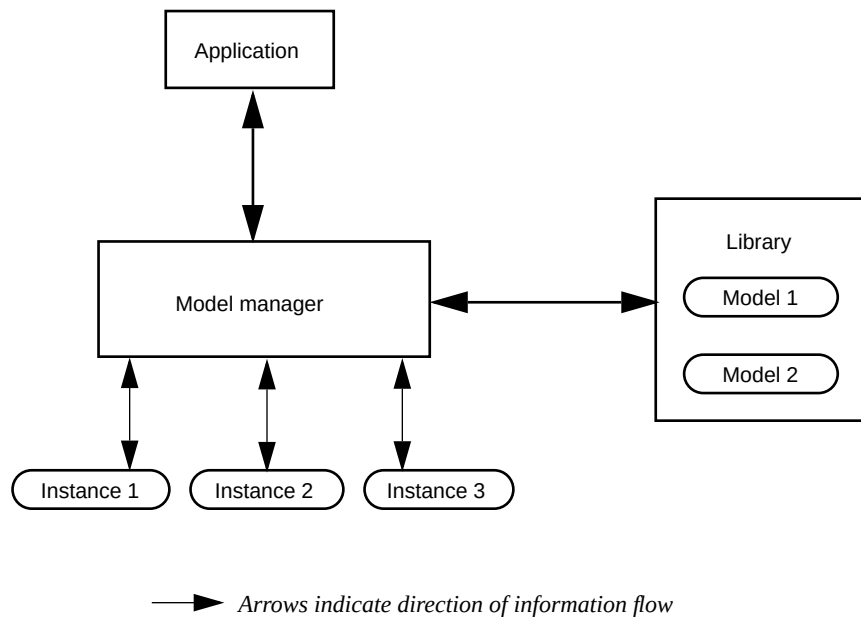


Figure 2—The basic OMI elements

3.2 OMI compliance

This standard defines compliance for models and applications in terms of adherence to relevant rules in the specification. It does not define a means of verifying compliance.

A model is *OMI-compliant* if it is associated with a model manager so that the model and model manager behave in accordance with the rules for these OMI elements as defined in this standard.

An application is *OMI-compliant* if it behaves in accordance with the rules for an application as defined in this standard.

This OMI recognizes a distinguished kind of application called an *OMI simulator*. An OMI simulator is an OMI-compliant application that supports a simulation session as described in 5.1.

NOTE—The above rule concerning the behavior of an OMI-compliant application implies that an OMI-compliant application must provide implementations of the required OMI routines and it also must use the routines on the other side of the interface in accordance with the OMI rules for these routines. Analogous remarks apply concerning OMI-compliant models.

3.3 Creation of OMI models

OMI-compliant models may be developed in a variety of languages. These languages may be modeling languages or general-purpose programming languages. They may be public or proprietary. A language that is used to develop an OMI model is called a *modeling language*. Regardless of the modeling language(s) used to construct a model, the model shall satisfy the basic interoperability requirements that are defined in 4.1.

A tool that processes a description of a model written in a high-level modeling language (such as a hardware description language) to produce an object code representation of that model is called a *model compiler*. Typically, models written in a general-purpose programming language do not require processing by a model compiler. Examples of OMI model creation scenarios are shown graphically in Figure 3.

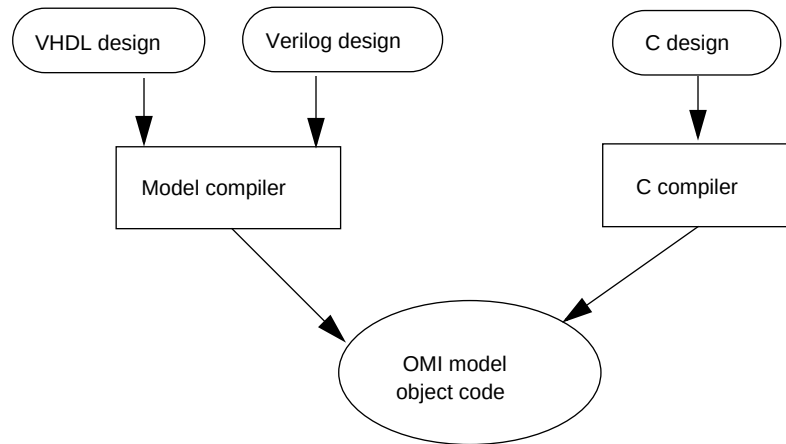


Figure 3—OMI model creation

NOTE 1—This standard does not define compliance for a model compiler.

NOTE 2—Although, in practice, a model compiler is not required for OMI models written in C, the development of C OMI models can be facilitated with a set of OMI-specific libraries.

NOTE 3—The OMI supports models compiled from IEEE Std 1364-1995 (Verilog) or IEEE Std 1076-1993 (VHDL) descriptions. It provides synchronization points and data exchange procedures that support a common subset of these hardware description languages. It is the responsibility of the Hardware Description Language (HDL) (i.e., VHDL or Verilog) based model compiler to map the activity at the model instance boundary to the OMI common execution model, perhaps imposing modeling restrictions at the boundary to insure that such a mapping is feasible for the implementation. Logic values corresponding to the logic systems of these hardware description languages are defined in 8.7.

4. Model integration

This standard defines a set of rules governing model/application interaction that insure that any OMI-compliant model operates correctly with any OMI-compliant application on a given platform.

4.1 Model integration software requirements

The OMI interfaces are defined in the C programming language. Other languages may be used to create OMI-compliant models or applications, but the developer must perform the conversions necessary to ensure ANSI Standard C compatibility.

Calls to OMI routines shall adhere to the host calling conventions.

It is erroneous for an object module associated with an OMI-compliant model to contain any global symbols other than the name of the `omiBootstrapRoutineT` function (see 9.5).

4.2 Model delivery

Although considerable flexibility is left to the model provider in its approach to delivering OMI models, this standard places several requirements on OMI model delivery. Specifically, the following two files are required when delivering OMI models:

- A dynamically loadable object file known as the *model manager object file*
- A text file that is known as a *bootstrap file*

The names of these files are determined by the model manager provider. A model manager provider may supply an installation script/procedure. This script/procedure shall be such that the model manager object file and the bootstrap file reside next to one another on the host file system. The full pathname to their common location is called the model manager *installation path*.

The model manager object file is the object-format representation of the model manager. The file may be a single object file or an object library file. In any case, the file shall be suitable as a dynamically loadable argument to the loader provided by the host operating system. If the model manager consists of multiple files, it is the responsibility of the model manager to link in the additional files. In other words, the application links in only the model manager object file, and the model manager shall link in any additional files (for itself or its models) as needed.

The bootstrap file contains two pieces of information needed by the application to bootstrap the associated model manager:

- The name of the model manager object file. The name shall be location-independent (i.e., it should not include any path information).
- The name of the C function that is the bootstrap routine (see 5.2) for the model manager.

Syntactically, the bootstrap file shall consist of exactly two lines. The first line consists of a sequence of zero or more blank or tab characters, the character sequence “MANAGER:” (the letters in upper case), followed by zero or more blank or tab characters, followed by a sequence of characters that gives the location-independent name of the model manager object file. The second line consists of a sequence of zero or more blank or tab characters, the character sequence “BOOTSTRAP:” (the letters in upper case), followed by zero or more whitespace characters, followed by a sequence of characters that gives the name of the C function that is the bootstrap routine for the model manager.

Example:

The following is an example bootstrap file:

```
MANAGER: vendorX_MM_1.0.so  
BOOTSTRAP: vendorX_bootstrap_1_0
```

NOTE 1—Once installed (placed on the host file system), the full pathname of the bootstrap file completely characterizes the model manager for integration purposes with any application. By using the information in the bootstrap file, an application can determine the exact location of the model manager in order to load it, as well as what routine to call to initiate communication with the model manager. The bootstrap file contents are meaningful to the human eye, serving as a sort of minimal documentation for the model manager.

NOTE 2—The mechanism by which an application becomes aware of the model managers it is to use and the mechanism by which an application locates the bootstrap file for a particular model manager are not defined in this standard.

NOTE 3—A model or model manager supplier would be wise to include a vendor-specific string in the name for its bootstrap function in order to avoid conflicts with the bootstrap function names of other suppliers.

4.3 Characteristics of OMI applications

In order to describe the way OMI applications and OMI models work together, it is first necessary to define two key characteristics of an OMI application. These two characteristics are called the *model boundary class* and the *simulation style* of the application.

4.3.1 Model boundary class

The *model boundary class* of an OMI application represents the richness in the object data types that the application environment supports at the model interface. The OMI defines four model boundary classes. These are called the *two-state*, *four-state*, *simple-logic*, and *unrestricted* model boundary classes. The most prominent restriction for each of these classes concerns the data types for the model ports that are supported, but each class also implies its own restrictions on the data types for model parameters and viewports. Tables 1 through 4 give the data type restrictions for each of the OMI model boundary classes. The data types themselves are defined in 7.6.

Table 1—Data type restrictions for the two-state model boundary class

Kind of interface object	Interface object shall be of one of these data types
Port	MVL2, array of MVL2
Parameter	Boolean, Integer, String
Viewport	MVL2, array of MVL2, array of array of MVL2, Integer

Table 2—Data type restrictions for the four-state model boundary class

Kind of interface object	Interface object shall be of one of these data types
Port	MVL2, array of MVL2, MVL4, array of MVL4
Parameter	Boolean, Integer, String
Viewport	MVL2, array of MVL2, array of array of MVL2, MVL4, array of MVL4, array of array of MVL4, Integer

Table 3—Data type restrictions for the simple-logic model boundary class

Kind of interface object	Interface object shall be of one of these data types
Port	Any logic type, an array of any logic type, Real, Integer
Parameter	Boolean, Integer, String, Real
Viewport	Any logic type, an array of any logic type, an array of array of any logic type, Integer, Boolean, Real, Time

Table 4—Data type restrictions for the unrestricted model boundary class

Kind of interface object	Interface object shall be of one of these data types
Port	Any OMI type
Parameter	Any OMI type
Viewport	Any OMI type

These model boundary classes are described here in increasing order of data type richness, and each class encompasses the preceding class, e.g., a simulator that supports the simple-logic model boundary class necessarily supports the four-state model boundary class.

In each of the model boundary classes, the model attributes are allowed to be of any OMI data type (that is, attribute data types are unrestricted).

4.3.2 Simulation style

The simulation style of an OMI application indicates whether the application environment provides a general, event-based simulation paradigm or is restricted to a simple clock-based model of simulation. The OMI *cycle-based* simulation style makes a restricted use of the OMI simulation interface. Specifically, the simulator's OMI routines that are relevant to delayed updates or event times are not used during cycle-based simulation and, in fact, do not need to be supplied by the simulator. Further, the simulation-time callbacks that are not specific to clock transitions are not used during cycle-based simulation and need not be supported by the simulator. The *event-driven* simulation style allows full use of the OMI simulation interface and, in particular, allows use of the simulator routines and callbacks that are not used in the cycle-based simulation style.

As with the model boundary classes, the OMI simulation styles are nested; specifically, an application environment that provides the event-driven simulation style necessarily provides the cycle-based simulation style.

4.4 Classification of OMI simulators

An OMI-compliant simulator shall support at least the two-state model boundary class and the cycle-based simulation style.

An *OMI event-based simulator* is a distinguished kind of OMI-compliant simulator that supports at least the simple-logic model boundary class and the event-driven simulation style.

4.5 Simulator/model interoperability

During the bootstrap process, the application indicates the model boundary class and the simulation style that it can support for the session (see 5.2). It is the responsibility of the model or model manager to diagnose any incompatibility between the requirements of the model or model manager and the model boundary class and simulation style indicated by the application.

A model may adapt or reconfigure itself based on the model boundary class and simulation style provided by the application during the bootstrap process. For instance, if the simulation style is cycle-based, a model may choose to present an “untimed” version of itself to the application. Similarly, if the model boundary class is two-state, a model that normally operates with four-state logic may choose to restrict itself to two-state logic for the purpose of the particular simulation session.

The simulation style and highest model boundary class that are established during the bootstrap stage (see 5.2) are especially significant because they may determine the nature of the model query information provided for a particular model. Model query information for a particular model at a given installation shall be the same across sessions that have the same simulation style and highest model boundary class.

Clearly the application characteristics described in 4.3 have implications for the kinds of OMI simulators with which a given OMI model will work. An OMI model that can operate within the two-state model boundary class and the cycle-based simulation style will necessarily work with any OMI simulator, since these represent the minimum level of support that OMI requires of a simulator. At the other extreme, an OMI model that requires the unrestricted model boundary class and the event-driven simulation style will work only with OMI simulators that support this most-demanding form of model. In between these two end points there exist intermediate levels of requirements imposed by models and corresponding degrees of support provided by simulators.

Any model that satisfies the requirements of the simple-logic model boundary class is of special significance since it will necessarily work with any OMI event-based simulator.

NOTE—The above rule concerning consistency of model query information across sessions guarantees that, for example, at a particular site, a “shell generator” OMI application and an OMI simulator will observe an identical model interface for a given OMI model when using the same simulation style and highest model boundary class.

5. Execution

An *OMI session* is conducted by an application and one or more model managers through calls to OMI routines provided by the participants. This activity is called *execution*. Although simulation is one of the primary kinds of execution supported by the OMI, execution is not limited to simulation. In fact, an OMI session may not involve simulation at all. For example, an OMI-compliant application may be a wrapper-generator that produces HDL shells for instantiating OMI-compliant models in HDL designs. Or, it may be a delay calculator that extracts timing information from OMI-compliant models when building a hierarchical timing model for an enclosing design.

The execution of an OMI session may involve a number of stages dedicated to specific activity: bootstrapping, elaboration, initialization, simulation, and termination. Although the application acts as the coordinator for an OMI session (determining when to advance to a new stage and maintaining the OMI simulation clock), generally the model manager determines when to synchronize with the application by registering and removing callbacks.

5.1 Stages of execution

An execution session proceeds in an orderly fashion. In other words, OMI routines cannot be called randomly, but shall be invoked according to an overall plan that divides execution into periods of activity called *stages*. Each stage of execution is focused on a particular kind of activity, such as model instantiation. Certain stages may be further decomposed into a series of steps according to an algorithm. For instance, simulation consists of a series of simulation cycles in which a number of steps are performed. The result is a partial ordering of execution activity in which certain activities are required to occur before or after others.

OMI sessions can be categorized by the sorts of activities that can or will take place. The application determines the nature of a session. In a *model query session*, library or model data are examined without creating any model instances, as a library browser might do. An *elaboration session* involves creating model instances (generally to extract certain information), without simulating their behavior. For example, a delay calculator might instantiate models in an elaboration session in order to generate backannotation data for later

use in simulation sessions. A *simulation session* not only creates model instances, it simulates their behavior as well.

The sessions described in the previous paragraph correspond to increasing levels of OMI activity. An elaboration session includes model query and elaboration activity; and a simulation session includes model query, elaboration, and simulation activity. The stages executed in an elaboration session are a superset of the stages executed in a model query session. Similarly, the stages executed in a simulation session are a superset of those executed in an elaboration session.

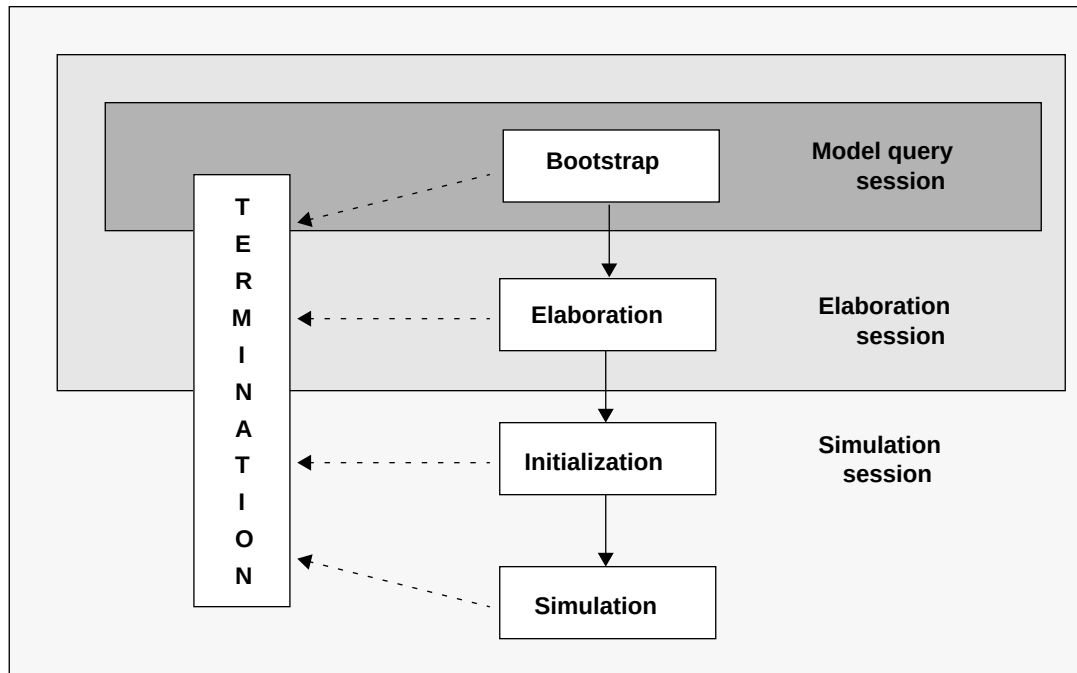


Figure 4—Execution of stages in OMI sessions

Figure 4 graphically presents the stages of execution, which are described briefly below. More detail can be found in the clauses pertaining to specific interface support for the different stages.

- Bootstrap**—*Bootstrapping* is the process of establishing communication between the application and the model manager. During the bootstrap stage, the application and model manager establish the nature of the session and perform setup activity, such as license checking. The bootstrap stage shall occur before any other stage.
- Elaboration**—*Elaboration* is the process of creating the model instances that will be manipulated during an OMI session. Elaboration shall occur after bootstrap and before initialization. Both elaboration and simulation sessions require an elaboration stage.
- Initialization**—*Initialization* is the process of establishing the initial values of data objects having state. Initialization shall occur after elaboration and before the simulation stage. It is a required stage for a simulation session.
- Simulation**—*Simulation* is the process of approximating the behavior of a system by exercising a set of model instances through application of stimulus and propagation of response. The simulation stage shall occur after initialization.

- e) **Termination**—*Termination* is the process of shutting down an OMI session. It may occur anytime after startup. Termination shall be the final stage of execution in any OMI session.

NOTE 1—The only stages that always occur in an OMI session are the bootstrap and termination stages.

NOTE 2—The OMI stages are concerned specifically with OMI activity. Other non-OMI activity of a different nature may be taking place concurrently with the OMI activity, as long as the rules pertaining to OMI activity are followed. For example, the bootstrap stage precedes elaboration from the perspective of a particular model manager; however, the application may in fact bootstrap a model manager upon encountering the first instance for that model manager during the application's broader design elaboration phase. Similarly, the OMI elaboration stage may take place in what is actually the beginning of simulation for an application, perhaps because the application system saves the elaborated representation of the "native" portion of a design to disk for repeated re-use and elaborates "foreign" models just prior to simulation.

5.2 Bootstrap stage

At startup time an application goes through a particular initialization protocol with each of the model managers it is using. This initialization protocol is called the *bootstrap process*. Achieved during the bootstrap process are the following goals:

- A specific OMI version is identified for the interaction between the application and the model manager.
- The application and the model manager each obtain the addresses of the routines that the other is required to provide as part of the OMI interface.
- Session-specific options are established.

It is expected that the OMI will evolve over time and that this evolution will lead to different recognized versions of the interface. The bootstrap process is performed in two steps to allow version incompatibility to be detected gracefully and in a version-independent way. Compatibility is established through a pair of startup routines that are provided by a model manager and called by the application. The `omiBootstrap` routine initiates an exchange about supported OMI versions, and the `omiBeginSession` routine completes the exchange by establishing an OMI version for the session and exchanging the information that each side needs in order to conduct a session for that particular version of the OMI.

5.2.1 Initiating communication

The application initiates communication with a model manager by invoking the model manager's `omiBootstrap` routine. The purpose of this routine is to provide the application with information about the OMI revisions that the model manager supports. This information is supplied in the form of a lower and upper bound defining a nonempty range of supported revisions.

The bootstrap routine also returns a pointer to the model manager's `omiBeginSession` routine, which the application calls to perform the second half of the bootstrap process, i.e., establish the basic characteristics of the OMI session. If the model manager is unable to continue for any reason, the bootstrap function shall return a status value indicating failure. The application shall not attempt to invoke the model manager's `omiBeginSession` routine if it receives an indication of failure from the bootstrap routine; all further communication with the model manager shall cease at that point.

Using the information provided by the bootstrap routine, the application shall select an OMI revision for the session. If the application is unable to find an OMI version supported by both the application and the model manager, then the application shall cease communication with that model manager without completing the bootstrap process.

Example:

The sample bootstrap routine below informs the application that the model managers supports a single OMI revision (in this case, the latest one):

```

omiStatusT VendorX_Bootstrap (
    /* output */   omiRevisionT           *earliestSupportedRevision,
    /* output */   omiRevisionT           *latestSupportedRevision,
    /* output */   omiBeginSessionRoutineT **beginSessionRoutine )
{
    if (earliestSupportedRevision) {
        *earliestSupportedRevision = omiLatestOMIRevisionTValue;
    } else {
        return omiError;
    }

    if (latestSupportedRevision) {
        *latestSupportedRevision = omiLatestOMIRevisionTValue;
    } else {
        return omiError;
    }

    if (beginSessionRoutine) {
        *beginSessionRoutine = VendorX_BeginSession;
    } else {
        return omiError;
    }

    return omiClear;
}

```

Programming interface references:

- See 9.79 for omiRevisionT.
- See 9.59 for omiLatestOMIRevisionTValue.
- See 9.5 for omiBootstrapRoutineT.

5.2.2 Beginning a session

The basic characteristics of an OMI session are fully established by the omiBeginSession routine. The following information is exchanged when beginning an OMI session:

- The OMI revision under which the session will be conducted
- The nature of the OMI session
- A reference simulation style
- The highest model boundary type supported by the application
- The precision for representing time values during the session
- The timing mode (min/typ/max) for the session
- The installation path where the model manager object file resides
- The names of the application and the model manager
- The caller identifiers of the application and the model manager
- The entry points of the required OMI routines provided by the application
- The entry points of the rest of the required OMI routines provided by the model manager

The begin session routine has a pair of arguments. The first argument is an indication of the OMI revision for the session. The second is a pointer to a structure of type `omiSessionArgumentST`; the remainder of the information characterizing the session is exchanged through the fields of the session arguments structure rather than directly through a set of arguments. Some fields are set by the application before invoking the `omiBeginSessionRoutine`; the remaining fields are set by the model manager before returning. The nature of the information exchanged is described in more detail below.

The OMI revision selected by the application shall belong to the range of revisions supported by the model manager. Both the application and the model manager shall interpret the second argument to the begin session routine as a pointer to the `omiSessionArgumentST` structure for the selected OMI revision.

The application shall inform the model manager of the sort of activity that can take place in the session by selecting a session type (e.g., `omiSimulation`). It is an error for the application or the model manager to attempt to engage in activity that is not allowed in that sort of session.

The application shall provide a reference simulation style to the model manager, indicating the broadest category of simulation support provided by the application environment. If the session is not a simulation session and the notion of a simulation style is not meaningful in this particular application, then the value `omiNoSimulation` shall be supplied. If the simulation style specified by the application is not supported by the model manager, then the model manager shall raise an `omiUnsupportedSimulationStyle` error.

The application shall indicate to the model manager the highest model boundary class that the application supports. If the model boundary class specified by the application is not supported by the model manager, then the model manager shall raise an `omiUnsupportedBoundaryClass` error. If the specified model boundary class is not compatible with the specified simulation style, then the model manager shall raise an `omiInvalidArgument` error. Simulation styles, model boundary classes, and compatibility issues are discussed in 4.3.1.

The application shall select a time precision in the form of a time unit and an integral scale. This precision defines the resolution for time values exchanged during the session. The representation of time is further discussed in 8.5. To assist an application in selecting an appropriate time precision, a model may have predefined attributes associated with it to provide information regarding the time precision that it requires (see A.1).

The application shall select a timing mode (min/typ/max) for the session if the session is an event-driven simulation session. If the session is not an event-driven simulation session or the model manager does not support min/typ/max timing, then this specification shall be ignored by the model manager.

The application shall inform the model manager of the installation location, i.e., the directory on the host file system where the model manager object file resides. The model manager may (but is not required to) access this location to retrieve information needed to conduct the rest of the session.

The application and the model manager shall each provide the other with a string representation of its name. That representation shall not be a null string or an empty string.

The application and the model manager shall each assign a unique *caller identifier* to the other (see 9.13 for details on caller identifiers). Thereafter, the assigned caller identifier shall be passed as a part of the caller data to any OMI routine provided by the OMI implementation that assigned that identifier.

The application and the model manager shall each provide a *function tray* containing pointers to the OMI routines that it supports. The function tray is a fixed structure of function pointers that shall remain valid for the duration of the OMI session. Each shall use the function pointers of the other to make calls during execution. Functions that are not supported shall have null (zero) entries. The function tray entry for a

required routine shall not be null. The nature of the session and the simulation styles that an application or model manager supports determine which routines shall be provided in an OMI function tray; see 9.2 and 9.60 for details on the required routines for the application and the model manager, respectively.

During the `omiBeginSession` call, the model manager may perform setup activity, such as license checking, and make calls to certain OMI routines. OMI activities that may be performed by an `omiBeginSession` function shall be limited to callback registrations (see 5.7.1), invocation option queries (see 5.2.3), error handling (see 8.2), and termination requests (see 5.7.2).

A model manager may be asked to support simultaneous sessions with different applications (see example below). Support for multiple applications is not required. A model manager that does not support this feature shall raise an `omiOneApplicationOnly` error if more than one application attempts to begin a session with it.

An OMI simulator shall support sessions involving multiple model managers.

Example:

A multiple application situation may arise when an OMI model is used in another OMI model and is also instantiated directly by the top-level application, as shown in Figure . In this case Model Manager X is acting both as a model manager and as an application, and Model Manager Y is supporting two applications.

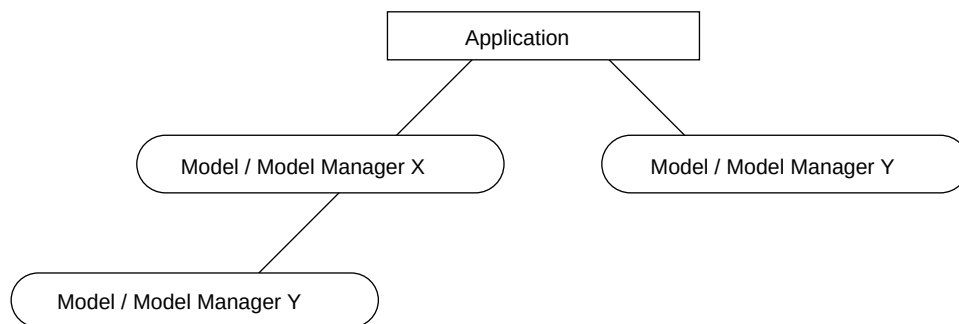


Figure 5—Two applications for Model Manager Y

NOTE 1—The session arguments are exchanged through a structure rather than a set of arguments to shelter the bootstrap process from OMI version dependencies. Information in the session arguments structure may change over time without affecting the interfaces of the `omiBootstrap` and `omiBeginSession` routines, enabling the same bootstrap process to span OMI revisions and successfully insure compatibility.

NOTE 2—The simulation style may be meaningful even if the application is not a simulator. For instance, an HDL shell generator application might provide an `omiCycleBasedSimulation` value for the simulation style because the shells will be used in a cycle-based simulation environment and this information could prove useful to the model manager. The session type always determines the nature of an OMI session, and in this example the session type provided would be `omiModelQuery`.

NOTE 3—The function trays do not include the anonymous routines that may be provided through the callback registration mechanism (see 5.7.1).

NOTE 4—The application and model manager names are provided for informational purposes. For instance, the names may be used in displays or error messages.

NOTE 5—The model manager installation path supports separate representation of models and model managers. In practice, a model manager may serve a specific domain of models that are compatible with it. Typically, a given vendor might supply a model manager along with a set of models designed to execute under that model manager. In such cases the model manager and the models are likely to be physically separate entities. Therefore, a model manager may need to learn about the models that it is responsible for managing in a given session, perhaps through a model registry located in the installation path. As with all interaction between a model manager and its models, this particular interaction is not defined in this standard. Furthermore, a model manager is free to use a mechanism other than its installation path to locate information about itself or its models. For instance, it may use an operating system environment variable to locate installation-related information.

NOTE 6—If a model manager does not support multiple applications, then it may be limiting its use in other OMI models. It is expected that large model suppliers will provide model managers that support multiple applications; such support may not be as common among providers of single models. Since the OMI does not define how OMI model packaging is accomplished, a model manager is not required to support OMI models that contain other OMI models. Such support is provided entirely at the discretion of the vendor providing the model manager and packaging service.

Programming interface references:

- See 9.3 for `omiBeginSessionRoutineT`.
- See 9.79 for `omiRevisionT`.
- See 9.90 for `omiSessionArgumentsT`.
- See 9.91 for `omiSessionKindT`.
- See 9.13 for `omiCallerIdT`.
- See 9.2 for `omiApplicationFunctionTrayT`.
- See 9.60 for `omiManagerFunctionTrayT`.

5.2.3 Invocation options

A model or model manager may allow an OMI session to be customized through special *invocation options* that it defines and recognizes. For example, a model manager might accept as an option the location of a results directory where simulation data for its model instances will be collected during the session. An invocation option may be a simple directive (e.g., telling it to run in interactive mode) or an option with an argument (e.g., telling it to take stimulus from a particular file).

An application shall provide a mechanism for accepting invocation options. The application is not required to interpret the options or know which, if any, of the model managers in the session understand those options. It merely has to recognize that they are OMI invocation options and then make them available to any model manager that inquires about them. The fact that an application provides this mechanism does not preclude a model manager from providing its own invocation option entry mechanism.

An invocation option is identified by a string called an *option keyword*. It is case-sensitive. This keyword shall be restricted to alphanumeric characters and underscores. The optional argument to an invocation option shall be a string that contains printable, noncontrol ASCII characters. The application shall define how this string is entered and how it is delimited.

The application shall establish invocation options at the beginning of a session, prior to the call to the `omiBeginSession` routine. They shall remain constant throughout the session. A model manager can obtain information about the invocation options for a session through the `omiScanOptions` routine.

Example:

Use of an invocation option might look like the following:

```
simulate +VendorX_results=/net/test/test123 testBench ...
```

The simulator might define its rules to say that

- a plus indicates the beginning of an OMI invocation option, which is terminated by white space
- an equal character signals an argument to that option
- if it is necessary to embed white space, then the argument must be enclosed in double quotes

The VendorX model manager might obtain its invocation information by the following OMI call:

```
omiCallerDataT  CD = { ... };
omiBooleanT     found;
omiBufferT      directoryName = { ... };
...
omiScanOptions ("VendorX_results", &found, &directoryName, &CD);
```

From that call, the following information would be returned:

```
found                omiTrue
directoryName.buffer  "/net/test/test123"
```

NOTE 1—This standard does not define how invocation options are input to the application. For example, input may be accomplished through command-line options, a graphical dialog box, an input file, an operating system environment variable, etc. The end user is responsible for using the entry mechanism in a manner that produces the string expected by the model or model manager.

NOTE 2—This standard does not require any particular invocation options for an OMI session; however, a model manager may require the use of certain invocation options that it defines. In other words, invocation options are always optional from the OMI application's perspective, but they are not necessarily optional from the model manager's perspective.

NOTE 3—A model/model manager supplier would be wise to include some vendor-specific string in an option keyword to avoid conflicts with the option keywords of other suppliers.

Programming interface references:

See 9.84 for `omiScanOptionsRoutineT`.

5.3 Elaboration stage

The elaboration stage is the period devoted to the creation of model instances. A model is *instantiated* to create a unique, customized instance by establishing the values of its design parameters and binding its ports to the environment. Elaboration is the only stage during which model instances are created.

An instance of a particular OMI model is created when the application invokes the model manager's `omiCreateInstance` routine. A data channel is created when the application supplies its own private identifier(s) for the net and/or driver that it has associated with an OMI port. These identifiers will be used by the model manager during simulation to denote different aspects of the port connection, e.g., to get the new value of a net or to schedule a change on a driver created by the OMI port association.

5.3.1 Model instantiation

A model instance is created through a multi-step process in which parameter and network connectivity information is exchanged and unique representations of the objects at the model instance boundary are created. The mechanics of instantiation are performed by several model manager routines, which shall be invoked by the application in the following order:

- a) The application shall call `omiCreateInstance` to initiate the instantiation of a particular model.
- b) The application shall call `omiSetParameter` for each parameter default value that it wants to override.
- c) The application shall call `omiElaborateInstance` to elaborate the instance.
- d) The application shall call `omiConnectPort` for each port that it wants to connect.
- e) The application shall call `omiEndPortConnect` to signal the end of the port connections.

During this process the model interface is described to the application through the model query mechanism (see 7.10); and the instance is described to the model manager through the arguments of the instantiation routines provided by the model manager. The OMI does not support simultaneous instantiation; it is erroneous for an application to interleave calls to these routines for different instances.

Each invocation of `omiCreateInstance` shall create a unique instance of the specified model, which shall be associated with the model manager supplying the `omiCreateInstance` routine. An application shall not invoke an `omiCreateInstance` routine in a phase other than the elaboration phase.

Parameters are set using the parameter definitions of the model as references. `omiSetParameter` shall be called at most once for each parameter definition, in an order determined by the application. The model manager shall use the default value for any parameter that is not explicitly set by the application.

The parameter values take effect when the instance is elaborated. The `omiElaborateInstance` routine shall elaborate each definition of the model according to the rules in 5.3.2, creating unique representations of the data objects (e.g., ports, parameters, viewports) at the model instance boundary. The application shall not attempt to access a data object before the associated instance has been elaborated.

A port is connected using the defined port objects of the instance as references. `omiConnectPort` shall be called at most once for each defined port in an order determined by the application. To make the connection, the application supplies net and/or driver information based on the port direction and the number of atomic elements associated with the port. A port shall be either entirely connected or entirely unconnected; in the case of a nonatomic port (i.e., a port that is manipulated by element; see 7.5.2.1), all of the elements shall be connected if any of them are. The model manager shall be responsible for insuring a well-behaved instantiation in the event that a port is not connected.

Processing the port connections ends when the `omiEndPortConnect` routine is called. The instantiation process is complete upon the return of this routine.

NOTE 1—Parameters are set with respect to the parameter definition alone. When connecting a port, an application must refer to both the elaborated port (for completely characterized data type information) and its corresponding port definition (for direction and atomicity information).

NOTE 2—A data object is not created by a call to `omiSetParameter` or `omiConnectPort`; it is created by the call to `omiElaborateInstance`.

NOTE 3—A port, including its nonatomic elements, is connected by a single call to `omiConnectPort`.

Programming interface references:

- See 9.21 for `omiCreateInstanceRoutineT`.
- See 9.92 for `omiSetParameterRoutineT`.
- See 9.31 for `omiElaborateInstanceRoutineT`.

See 9.18 for `omiConnectPortRoutineT`.

See 9.32 for `omiEndPortConnectRoutineT`.

5.3.2 Elaborating definitions

A data object is created from a definition object through a process called *elaboration*. As a part of this process, the model manager associates a completely characterized data type with the data object and performs the internal processing required to make handles to that data object available for subsequent model queries.

The general nature of a data object at the model instance boundary is determined by the model provider, who characterizes the data object through a *definition*. Definitions describe the model interface, i.e., telling the application which aspects of the interface are fixed (such as the directions of its ports) and which aspects may vary from instance to instance (such as the widths of certain ports). A definition is not itself a data object; it does not have a value. A parameter definition is not a parameter, a port definition is not a port, etc.

The application and the model manager use definitions as a reference when creating a new instance of a model. For example, the data type of a parameter definition tells the application what sort of values it may select for the corresponding parameter when customizing the model instance. Similarly, the direction indication in a port definition determines whether the application is required to supply net identifiers, driver identifiers, or both in order to connect the corresponding port.

Each instance of a model contains a data object corresponding to each definition in that model. These data objects are said to be *derived* from their definition counterparts. The data objects share the fixed characteristics with other instances of the same model. After the elaboration of a particular instantiation is complete (upon the return of the `omiElaborateInstance` routine), the application can access the instance-specific data objects.

One way in which a data object is characterized is by the set of values that it can assume. This information is conveyed through its data type. For most types of data objects, the set of allowed data values is completely fixed; and the definition and the data object have the same data type. In the case of arrays, the model provider may allow the application to determine the bounds or size of a data object's values on a per-instance basis. In this case the data types of the definition and the data object will be different but related, as described below.

The data type of a definition may be an incompletely characterized type. In this case, the storage type and element type of the value(s) are fixed, but the size and possibly the bounds of the value(s) are determined by the application on a per-instance basis. During the instantiation process, the application supplies the information necessary to complete the characterization of the data object's set of values. The model manager shall use this information to create a completely characterized data type for the data object, i.e., an `omiFixedArrayData` object with the same storage type and element type as the corresponding incompletely characterized data type.

If the object type of a parameter definition is an unconstrained array type, then the model manager shall select appropriate bounds for the corresponding parameter, which shall be consistent with the size of the value supplied by the application or the default value, whichever applies. A parameter definition shall not have a data type that is a parameterized data type. In other words, the bounds and size of a parameter shall not depend on the value of another parameter.

If the object type of a port definition is a parameterized array type, then the model manager shall use the instance parameter values to create a fixed array type that is consistent with those values. This fixed array type shall be the data type of the instance port. A port definition shall not have an unconstrained array type for its data type.

If the object type of a viewport definition is a parameterized array type, then the model manager shall use the instance parameter values to create a fixed array type that is consistent with those values. This fixed array

type shall be the data type of the instance viewport. A viewport definition shall not have an unconstrained array type for its data type.

NOTE—The data type of a data object is always a completely characterized type.

5.4 Initialization stage

Initialization is the stage during which OMI objects having state (i.e., ports, nets, drivers, viewports) get their initial values. At the end of this stage, the value of every OMI data object in the OMI session is defined, and an attempt to access a data value will yield a value that is appropriate for the data type of the object.

OMI initialization is a three-stage process in which

- Drivers associated with OMI output or bidirectional ports are initialized
- Nets associated with OMI input or bidirectional ports are initialized
- Ports and viewports are initialized

A model manager may choose to provide an initial value for a driver by invoking the `omiInitializeDriver` routine provided by the application. To do so, the model manager can register one or more callbacks for `omiDriverInitialization`. A model manager shall invoke the `omiInitializeDriver` routine at most once for each driver and then only during the execution of an `omiDriverInitialization` callback. If a model manager chooses not to initialize a driver, then the application shall initialize that driver to a value that is valid for its data type.

Prior to the execution of net initialization callbacks, the application shall establish valid initial values for all of the nets associated with OMI ports. Thereafter, the value of a net can be obtained by querying the application through the `omiGetNetValue` and `omiGetNetGroupValues` routines.

The initial values of ports and viewports are determined by the model manager. They shall be available to the application at the beginning of simulation, but not necessarily before then. The initial values can be obtained at the beginning of simulation by querying the model manager for the current values through the `omiGetDataValue` routine. The initial value of an OMI port is not necessarily the same as that of the associated net at the beginning of simulation.

NOTE 1—Initialization of a data object prior to simulation is not considered to be an update to that object; hence it does not trigger a callback.

NOTE 2—To obtain the initial value of a net as soon as it becomes available, a model manager can register an `omiNetInitialization` callback. From this callback, the model manager can make the appropriate net value queries. An attempt to obtain the value of a data object before it has been initialized will result in an error.

NOTE 3—A simulator may implement OMI driver initialization by scheduling an update to the net at time zero. The value of an OMI driver may influence the net either prior to execution of the net initialization callbacks or during the execution of the first simulation cycle.

Programming interface references:

See 9.51 for `omiInitializeDriverRoutineT`.

See 9.73 for `omiRegisterCallbackRoutineT`.

5.5 Simulation stage

Simulation is the execution of a series of simulation cycles in which OMI ports are updated and routines implementing OMI model behavior are executed. Interaction between the simulator and the model manager takes place through the execution of callbacks registered by the model manager (see 5.7.1). Model manager callbacks affect changes at the OMI port boundary by calling routines provided by the simulator to retrieve new values or schedule updates (see 6.5). Clause 6 provides a detailed description of the activities which may take place in an OMI simulation session, including the simulation stage.

5.6 Termination stage

The termination stage encompasses the activity required to gracefully shutdown an OMI session. Resources may be released (e.g., files closed), final messages emitted, etc. Termination may be initiated because of an explicit request (perhaps through a user interface), the occurrence of a fatal error, or normal completion due to the absence of further scheduled activity.

Prior to exiting, the application shall execute the termination callbacks of the model managers. No further communication between the application and the model managers shall take place after execution of the termination callbacks has occurred.

A model manager may request termination through a control request (see 5.7.2). Upon receiving a control request for termination, the application may choose to terminate the entire session or to terminate the session for that model manager alone while continuing execution with the remaining model managers. Regardless of the choice it makes, the application shall execute a termination stage for that model manager by executing its termination callback(s) and then ceasing communication with it.

5.7 Execution control

An OMI session has an execution state, which changes as execution proceeds. The current state at a given point in execution includes the notion of the current stage of execution, the current values of the data objects at the OMI model instance boundaries, and possibly the current simulation time.

The application is responsible for advancing the current state. If the session involves simulation, then it has a clock associated with it. That clock is advanced by the application, which is a simulator in this case. The application provides an `omiCurrentTime` routine, which provides the current simulation time.

The model manager may request certain changes in state, such as a reset, by making a control request. Upon receiving such a request, the application is responsible for initiating the change in state, possibly by notifying all of the model managers and asking them to perform certain tasks. The application may deny the request in some cases. Handling of control requests is discussed in detail in 5.7.2.

The OMI provides an automatic error handling facility, which may affect the flow of execution. Details about error handling are discussed in 8.2.

5.7.1 Callbacks

Dynamic interaction is achieved through the OMI by allowing the model manager to register and remove callbacks. The callback mechanism allows a model manager to choose the granularity with which it interacts with the application. A *callback* is a transfer of control from the application to a model manager, or vice versa, that occurs under a specified set of circumstances. A callback is executed through an anonymous callback routine that is provided when the callback is registered.

Although a model manager may provide a limited form of callback to the application to support access to viewport changes, such support is not required and shall not be relied upon to drive simulation. The other callbacks discussed in this subclause are callbacks that the application is required to provide to the model manager.

Figure 6 depicts the OMI execution model with the associated callbacks.

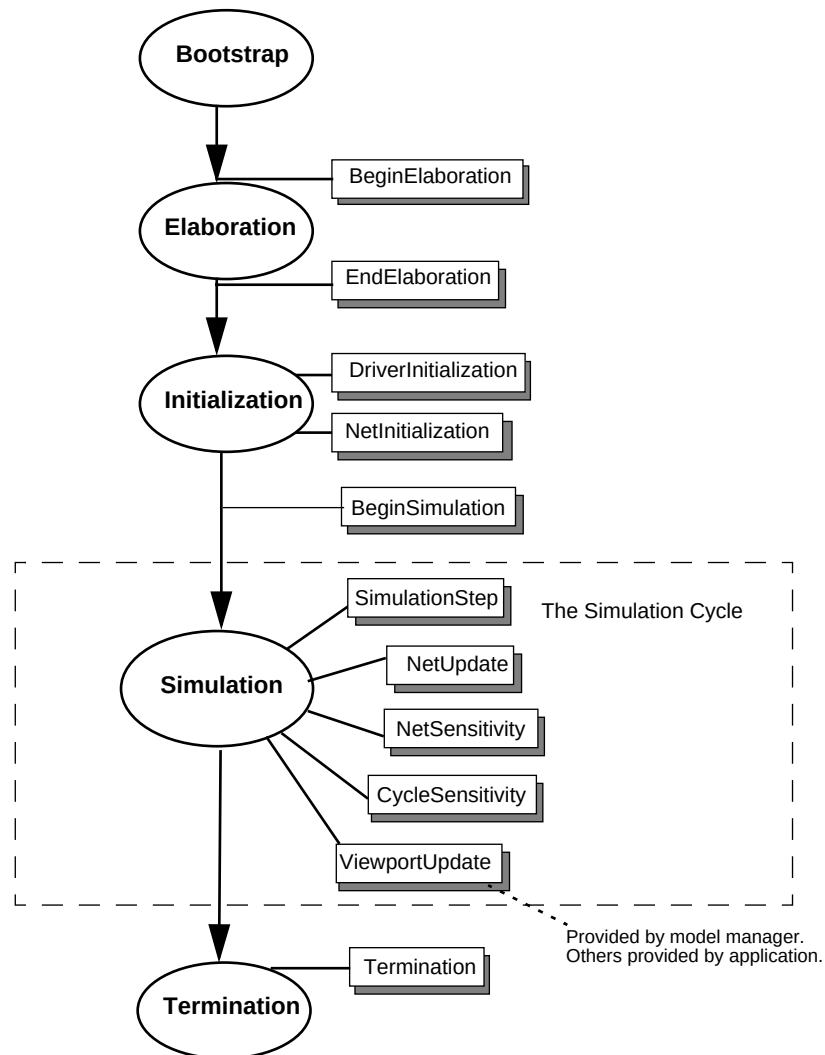


Figure 6—OMI execution model with callbacks

Two structures are involved in the registration and execution of callbacks. They are different structures, and they are controlled by different software components.

- The `omiCallbackDescriptionT` structure is provided during registration to describe to the callback provider exactly how the callback should be executed.
- The `omiCallbackInfoT` structure is provided during callback execution to give the callback recipient information pertaining to the callback and the current state.

A model manager or application requests a callback by registering the callback with a *callback reason*, i.e., an indication of the circumstances under which the callback is to occur. Callback reasons are defined for various execution stages and simulation events. Any number of callbacks may be registered for the same reason. Each registration yields a different, unique callback, even if the callback parameterization is exactly the same.

Callbacks are registered through the `omiRegisterCallback` routine. The callback description structure used to register the callback contains the callback reason, a pointer to the callback routine, private callback data, and any additional information required to further describe that kind of callback. The callback provider shall not assume that the callback description structure is valid after the registration routine returns.

A callback may be registered by the model manager from the following routines:

- `omiBeginSessionRoutineT`
- `omiCreateInstanceRoutineT`
- `omiConnectPortRoutineT`
- `omiElaborateInstanceRoutineT`
- `omiEndPortConnectRoutineT`
- A model manager callback routine other than a termination callback routine

A callback may be registered by the application from the following routines:

- An application callback routine
- An internal routine that is not one of the application's OMI routines

Private callback data may be provided when a callback is registered; this private data shall be returned when the callback is executed. For example, for a net update callback the model manager might choose to supply a structure containing net mapping information as the private callback data. The nature of the private data is determined by the software requesting the callback, and the callback provider shall not attempt to interpret it or rely upon it in any way.

Certain callbacks require special additional information to fully describe them. This information is provided through the *qualifier* field of the callback description structure. The *qualifier* field is a union, and each callback requiring additional information has a corresponding entry in that union for holding the required information. For instance, an `omiSimulationStep` callback has a `simulationStep` entry in this union; it denotes a structure containing a simulation time and an indication of when in the time step that callback should be executed.

A callback is executed by invoking the previously supplied callback routine with a single argument, i.e., a pointer to a structure of type `omiCallbackInfoT`. The structure contains

- A pointer to a representation of the current simulation time in the OMI format
- The callback provider's caller-specific data
- The callback recipient's private callback data
- Special additional state information for certain kinds of callbacks

The callback provider is responsible for all of the storage associated with this structure, which shall remain valid for the duration of the callback. The callback recipient shall not assume that the storage is valid after the return of the callback routine, nor shall it attempt to modify the structure or any of its constituents in any way.

Certain callbacks have additional state information associated with them, which is provided through the `special` field of the callback information structure. This field is a union with an entry for each callback for which special information is provided. For example, an `omiNetUpdate` callback has a `netUpdate` entry in the `special` union; it denotes the current value of the associated net.

Different kinds of callbacks occur at different points in execution; hence restrictions may be placed on the sort of activity that may take place in a particular kind of callback. Some of these restrictions are outlined in Table 5; additional restrictions relating to simulation cycle callbacks may be found in 6.4 and 6.5. Many of the callback descriptions refer to concepts defined in the simulation reference models described in 6.2 and 6.3.

Table 5—Restrictions on callback activities

Callback	Restrictions
<code>omiBeginElaboration</code>	Callback occurs at the beginning of elaboration, before any instances have been created for models under the control of that model manager. The current state and simulation clock have been initialized at this time.
<code>omiEndElaboration</code>	Callback occurs at the end of elaboration, after all instances have been created.
<code>omDriverInitialization</code>	Callback occurs at the beginning of initialization, before any nets/signals have been initialized. All instances have been created. The model manager can initialize its drivers at this time (see 5.4).
<code>omiNetInitialization</code>	Callback occurs after the application has established initial values for its nets (see 5.4). Net values are available at this time.
<code>omiBeginSimulation</code>	Callback occurs at the beginning of simulation, before any simulation cycle callbacks are executed. All data objects have valid values at this time.
<code>omiSimulationStep</code>	Callback occurs in a particular time step. The <code>processKind</code> qualifier (see 6.4) determines exactly when during the time step the callback is executed.
<code>omiNetSensitivity</code>	Callback occurs upon an edge or change on one or more nets. The <code>processKind</code> qualifier (see 6.4) determines when during the time step the callback is executed.
<code>omiCycleSensitivity</code>	Callback occurs upon an edge or change on one or more nets. The callback is executed after propagation in the cycle in which the edges or changes occurred.
<code>omiNetUpdate</code>	Callback occurs during network propagation when a specific net has been updated (the net will not necessarily have a different value).
<code>omiTermination</code>	Callback occurs just prior to the end of the session. No OMI routines shall be called from this sort of callback.
<code>omiViewportUpdate</code>	Callback occurs when the value of a viewport changes. This callback is the only form of callback available to the application.

The `omiSimulationStep`, `omiCycleSensitivity`, `omiNetSensitivity`, and `omiNetUpdate` callbacks are intended specifically for implementing simulation cycles. See 6.4 for a detailed discussion of these callbacks, how they relate to the OMI execution model, and the restrictions on the operations that they may perform.

NOTE 1—The restrictions on when callbacks may be registered are intended to ensure that callbacks cannot be registered during operations that are meant to be reading operations (such as the model query operations) or in the middle of a difficult and unrelated operation.

NOTE 2—Since the callback provider cannot rely on the persistence of the `omiCallbackDescriptionT` structure, the same structure may be reused to repeatedly register callbacks. Similarly, since the callback recipient cannot rely on the persistence of the `omiCallbackInfoT` structure, the callback provider can reuse the same structure when executing different callbacks.

Programming interface references:

- See 9.10 for `omiCallbackReasonT`.
- See 9.72 for `omiProcessKindT`.
- See 9.7 for `omiCallbackDescriptionT`.
- See 9.9 for `omiCallbackInfoT`.
- See 9.8 for `omiCallbackIdT`.
- See 9.11 for `omiCallbackRoutineT`.
- See 9.73 for `omiRegisterCallbackRoutineT`.
- See 9.75 for `omiRemoveCallbackRoutineT`.

5.7.2 Control requests

A model manager may directly affect the flow of execution by executing a control request. For instance, a control request may be used to revert to a previous state or to terminate execution of the model manager.

A control request for termination can be issued from any model manager routine. All other control requests can be issued only from callback routines.

In general, a call to `omiControlRequest` returns to the model manager with an indication as to whether the request is accepted or denied. An application may accept or deny any control request except for a request for termination, which it is required execute. Control shall not be returned to the model manager routine making the request if the request is for termination; model manager termination is described in detail in 5.6.

Most control requests occur immediately. A save or suspend request is a possible exception. If a model manager requests a save or suspend in the middle of a simulation cycle, then the requested operation is performed at the end of the current cycle. The request is still accepted in this case, and the model manager can then assume that it will take place at the end of the cycle.

If a control request other than termination is accepted and can be executed immediately, the following happens:

- a) The application's `omiControlRequest` routine returns control to the model manager, indicating acceptance. The application does not call any other model manager routines before returning control. In other words, from the model manager's perspective, the application returns immediately (although the application may in fact perform internal activity first).
- b) The model manager then returns control to the application (immediately, from the application's perspective) without calling any other application routines.
- c) The application executes the operation by invoking the appropriate routines of all of the model managers participating in the session.
- d) The application resumes execution at the appropriate point (possibly an earlier one).

Example:

An example illustrating the use of a control request for termination may be found in the error handling example in 8.2.

NOTE 1—The basic goal of the first two steps in the above sequence is to remove the call to `omiControlRequest` from the call stack before performing the requested operation.

NOTE 2—The fact that an application does not return from a termination request gives the model manager developer the option of using centralized error handlers rather than embedding error handling code throughout. A model manager error

handler typically contains a termination request for handling serious errors raised by the application. Similarly, an internal model manager routine for raising errors often contains a termination request for serious errors. Since the application does not return from a termination request and cannot deny such a request, the model manager does not have to include code to recover from errors that it considers to be fatal.

NOTE 3—The `omiSuspend` control request informs the simulator of a request for interactive input from a model user. A simulator is not required to support this capability, nor is suspension always appropriate (e.g., in a batch simulation). When suspending a simulation in response to a model manager request, the application should first finish the current simulation delta cycle. It should then “stop” at a steady state from which the user can observe a set of consistent simulation information (e.g., signal values, port values) and eventually resume simulation (e.g., through a “run” command).

Programming interface references:

See 9.19 for `omiControlT`.

See 9.20 for `omiControlRequestRoutineT`.

5.7.3 System exceptions

System exception handling shall be the responsibility of the application. The application shall detect and process system-addressing issues (e.g., segmentation violation, bus error) as well as user-initiated interrupts (e.g., control-C).

A model manager may install its own system exception handler; however, the application’s exception handler shall be restored before execution control is returned to the application. The model manager shall handle only the exceptions with which it can and should deal completely. Other exceptions shall be passed on to be handled by the application.

NOTE 1—A model manager that installs its own exception handler during model activity is required to reinstall the application’s exception handler when the model activity is complete. Thus, the application can assume that its exception handler is in place at all times and need not reinstall its exception handler after passing control to the model manager. Exception handling responsibilities have been assigned in this manner to minimize exception handler swapping and its performance cost.

NOTE 2—In some situations it may be reasonable for the model manager to catch and resolve a system exception. For example, some model developers may wish to catch ‘divide by 0’ exceptions in their models, deal with the problem and then proceed. Similarly, if the model manager implements a testbench with a graphical user interface (GUI), a control-C interrupt might be caught and processed by the model manager’s GUI. Situations such as these need not involve the application.

5.7.4 Debugging the model manager

In some cases a model manager developer may have access to the application software in order to debug a session involving OMI models. Since the model manager object file is dynamically loaded, the model manager is difficult to enter in a debugger. To make this task easier, the application shall supply a special routine that is intended to serve as an entry point for debugging purposes: `omiDebugEntry`. This routine shall do nothing. A call to this routine can be inserted anywhere in the model manager and used by the model manager developer to set a breakpoint in the model manager.

NOTE—The `omiDebugEntry` function is intended for debugging purposes. Calls to this routine should **NOT** be included in a model manager delivered to other parties. In this way, the debugging task for a model manager developer is simplified, because there will be no `omiDebugEntry` calls from other model managers with which to contend.

Programming interface references:

See 9.25 for `omiDebugEntry`.

5.7.5 Model manager commands

A model manager may provide special capabilities that enhance the usability of the models under its control. For instance, a model manager may allow dynamic control over simulation data collection (tracing) or allow a user to load the contents of a memory viewport from a particular file during initialization. The OMI provides access to special-purpose model manager operations through a generic command routine supplied by the model manager.

The command routine is an optional routine of type `omiCommandRoutineT`. A command is issued through a string argument to this routine. Another argument may be used to indicate whether the command applies to an instance, to all instances, or to the model manager itself. Depending on the nature of the command, a value may be returned through another argument.

The command facility allows the definition and use of dynamic operations that are not already explicitly supported by other OMI routines. The meaning and form of a command shall be determined by the model manager, and the model manager shall be responsible for interpreting the arguments to the command routine and performing or rejecting the requested operation.

A command may be issued by the application itself or indirectly by an end user of the model through a mechanism supplied by the application. An application is not required to provide a command entry mechanism for the end user, nor shall correct execution of a model depend upon the execution of model manager commands.

A number of predefined commands are described in A.2. This annex defines the syntax and meaning of a small number of commands that might be useful in OMI sessions; however, support for these commands is not required.

Examples:

An OMI-compliant simulator might provide an interactive command of the form

```
omiCommand [<instance_name> | <MM_name>] <command_string>
```

that allows an end user to direct commands to a specific model manager through the simulator's user interface. A use of this simulator command entry capability might look like

```
>omiCommand :dut/a/b "loadmem mem1 ./mem_file"
>/Vendor_X_MM/ :dut/a/b/memory successfully loaded from /user/joe/mem_file.
```

Command strings may be used to implement operations on a set of internal "state" variables that a model manager may define. Consider the command strings

```
"set debug_trace on"
"set debug_trace off"
"value debug_trace"
```

where the `value` command returns the current value of variable `trace` as a string message that may be displayed to the user by the simulator's user interface.

NOTE 1—It is expected that model manager commands will generally be interactive in nature. The command facility allows a model manager to take advantage of an OMI application's interactive interface if it has one. A model manager is not precluded from providing its own interface for dynamic control.

NOTE 2—The command facility is an opportunity for both the model manager and the OMI application to provide added value to the end user of a model: the model manager, by defining and supporting commands that enhance model control; and the OMI application, by providing a convenient mechanism for entering those commands and displaying results.

NOTE 3—The command facility is similar to the invocation option facility in that it provides a communication channel for functionality that is defined directly by the model manager rather than the OMI. The key difference is that invocation options are static in nature (they are established at the beginning of a session and remain constant thereafter), whereas commands are dynamic nature.

Programming interface references:

See 9.17 for `omiCommandRoutineT`.

6. Simulation

An OMI simulation session exercises OMI model instances through a protocol that defines how ports are updated and how values at the model instance boundary are exchanged. The OMI defines two basic types of simulation: event-driven simulation and cycle-based simulation. Requirements relating to simulation style support are discussed in 4.3.2.

To support model interoperability, the semantics of OMI simulation are defined in very general terms. A simulation reference model defines synchronization opportunities that are mapped to OMI callbacks. A simulator and model manager interact by registering callback routines and executing them when the specified conditions are met. An OMI simulation reference model is defined for each simulation style. All reference models support a variety of hardware description languages, including VHDL and Verilog.

A simulation session is distinguished from other kinds of OMI sessions by the fact that it includes initialization and simulation stages; however, this distinction is not the only one. Certain activity in other stages may take place only in a simulation session. For instance, backannotation during the elaboration stage may take place only in a simulation session. Similarly, the reset, save, and restore operations are specific to simulation sessions.

6.1 Simulation fundamentals

OMI simulation is described in terms of a system of behavioral elements called processes. An OMI *process* is a black box representing an independent thread of execution during simulation. The simulator knows nothing about the process other than the conditions under which it executes and the data changes that it produces. A process may be dedicated to a very low-level activity, such as computing the new value of a particular signal or executing a block of behavioral code. Alternatively, a process may be a powerful engine that performs a series of kernel activities or executes a whole set of internal parallel processes.

Communication between OMI processes takes place through data channels that are created by OMI port connections. Associated with each connection to an OMI input or bidirectional port is a *net* that represents the current value of the corresponding port. Associated with each connection to an OMI output or bidirectional port is a *driver* that represents an OMI model instance's scheduled contributions to the value of the associated net.

The set of nets connected to OMI ports constitute the *OMI network*. A net may have multiple drivers, not all of which necessarily represent OMI port connections. Hence the OMI does not define precisely how the value of a net connected to an OMI port is computed. It does, however, define when an OMI net is updated with respect to OMI execution and how a change in its value is made visible to the OMI processes.

The simulation stage consists of a series of *simulation cycles* in which the OMI network is updated and OMI processes are executed. The steps performed in the execution of a simulation cycle vary, depending on the style of simulation. The principal difference between cycle-based simulation and event-driven simulation lies in the role of simulation time and the absence of a time-based scheduling mechanism in cycle-based simulation. The event-driven reference model may be viewed as an extension of the cycle-based reference model in which simulation cycles are grouped into time steps, which serve as a reference for scheduling activity. The reference models for event-driven simulation and cycle-based simulation may be found in 6.2 and 6.3, respectively.

Nets are updated during *network propagation*, i.e., the process of recomputing the OMI network to reflect changes in nets or their drivers. During propagation, pending updates scheduled to take place at the current simulation cycle are removed from OMI drivers. Propagation may involve the execution of wire-tie, resolution, or type conversion functions to compute net values.

Regardless of the simulation style, network propagation takes place only once in a given simulation cycle. Scheduled updates are not visible prior to the network propagation step, and any changes to OMI ports scheduled by an OMI process after propagation take effect in a later simulation cycle (although not necessarily at a later simulation time), thereby insuring that the set of OMI processes executing at a given point in a simulation cycle share a consistent view of the OMI network.

Three basic kinds of OMI processes are recognized; they are differentiated primarily by when they are executed:

- New-time-step processes execute at the beginning of a time step, before network propagation in the first simulation cycle.
- Post-propagation processes execute after network propagation in a given simulation cycle.
- End-time-step processes execute at the end of a time step, just before simulation time is advanced; they shall not affect data in the current time step.

Because cycle-based simulation does not recognize time steps, the only form of process in a cycle-based simulation session is a post-propagation process.

The activation conditions for certain OMI processes are specified in terms of changes in net values. A process may be sensitive to any change in the value of a net or to a particular edge on that net. Type **omISensitivityT** defines this classification. An edge is a more restrictive value change that applies to nets of a logic type. Edge determinations ignore the strength aspect of a value. The only kind of sensitivity specification allowed for a net that is not of a logic type is **omIValueChange**. Table 6 defines the corresponding edge for each kind of logic value transition.

Table 6—Mapping of logic value transitions to edges

FROM	TO			
	0	1	X	Z
0	no edge	rising	rising	rising
1	falling	no edge	falling	falling
X	falling	rising	no edge	no edge
Z	falling	rising	no edge	no edge

A model manager creates OMI processes by registering certain kinds of callbacks (see 6.4 for details). The behavior of a given model instance may be implemented by a single process or many processes. In event-

driven simulation a single process may implement the behavior of several model instances. A process may read nets and modify drivers through the routines for dynamic data transfer described in 6.5.

NOTE—A model manager may further filter events through its callback routines to implement more stringent edge definitions (e.g., to implement IEEE Std 1164-1993 `rising_edge` and `falling_edge` semantics) if needed.

6.2 Event-driven simulation reference model

Event-driven simulation proceeds by executing a series of simulation cycles within a series of simulation times. A *simulation time step* is a time period that encompasses all of the activity that occurs between consecutive updates to the current simulation time. A simulation time step contains one or more simulation cycles. Multiple simulation cycles in a time step are caused by the scheduling of zero-delay updates during the execution of a simulation cycle. Figure 7 shows the flow of control during the execution of the simulation stage.

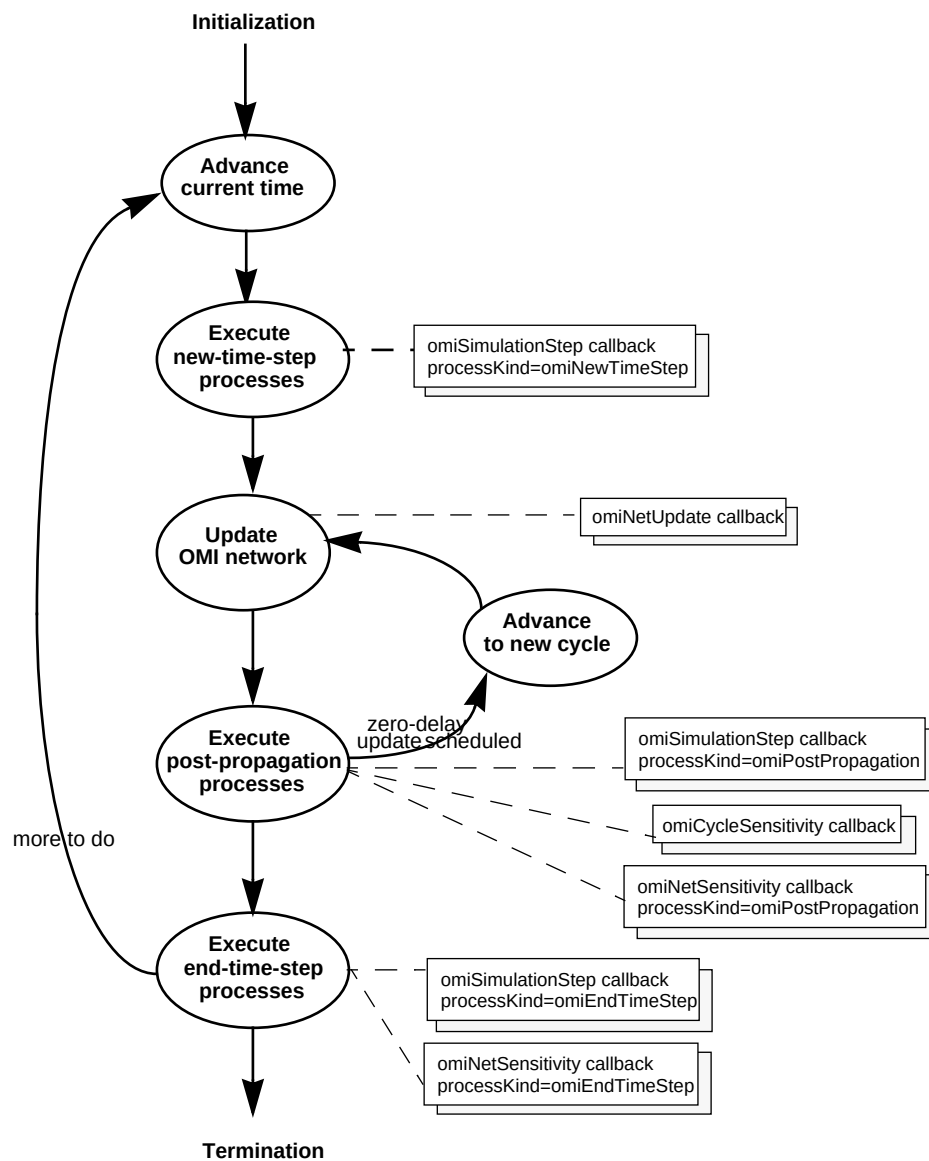


Figure 7—Event-driven simulation cycle with callbacks

Event-driven simulation proceeds according to the following algorithm, in which each iteration of Step a) is a different OMI time step, and each iteration of Step c) is a different OMI simulation cycle. Certain steps are annotated with references to the callbacks that are executed in that step. A process is activated in this context if it meets the terms for execution for the callback that implements it. For a net sensitivity callback, activation is contingent on one or more of a set of nets changing in that cycle or time step. For a simulation step callback, activation is contingent on the simulation time reaching a specified activation time.

- a) Advance simulation time.
- b) Execute activated new-time-step processes (`omiSimulationStep` callbacks for which `processKind=omiNewTimeStep`).
- c) Execute a simulation cycle.
 - 1) Update OMI network (`omiNetUpdate` callbacks are executed).
 - 2) Execute activated post-propagation processes scheduled for the current cycle (`omiSimulationStep` and `omiNetSensitivity` callbacks, for both of which `processKind=omiPostPropagation`, and `omiCycleSensitivity` callbacks).
 - 3) If zero-delay updates were scheduled in Steps 1) or 2), then return to Step 1).
- d) Complete time step by executing activated end-time-step processes scheduled for the current time step (`omiSimulationStep` and `omiNetSensitivity` callbacks for which `processKind=omiEndTimeStep`).
- e) If further activity has been scheduled for another time, return to Step a).

A model manager determines the granularity of its interactions with an event-driven simulator in a given time step. For instance, upon an input change, a model manager can choose to react after network propagation is complete for that simulation cycle, or after the OMI network has settled at the end of the time step. The mechanics for this interaction are provided through the callback capability described in detail in 6.4.

By selecting when in a given time step a process executes, a model manager determines how frequently the process executes, the state of the network that the process sees, and the nature of the actions that the process can perform. For example, an end-time-step process executes at most once per time step, it observes the final state of the OMI network for that time step, and it cannot schedule zero delay updates to OMI ports.

6.3 Cycle-based simulation reference model

Cycle-based simulation is a restricted form of simulation in which clock edges provide the foundation for the simulation execution algorithm. Rather than associating a specific delay with the execution of a particular process or a change in a particular signal, activity takes place with respect to changes in one or more clocks.

A cycle-based simulator is not timeless. Although execution does not advance through a time-based scheduling mechanism, a cycle-based simulator shall still select a time precision for exchanging (optional) clock period information and for reporting the current time as determined by its clock(s). The clock periods determine the current simulation time.

Clocks are binary devices that define two phases of execution relative to their state: “clock” and “not clock”. A phase change is represented by a rising or falling clock edge, which may trigger OMI cycle-based activity.

The cycle-based simulation algorithm is as follows:

- a) Advance/update simulation clocks as appropriate for their periods.
- b) Update OMI network.
- c) Execute activated OMI post-propagation processes (omiCycleSensitivity callbacks).
- d) Return to Step a).

Figure 8 illustrates this algorithm graphically.

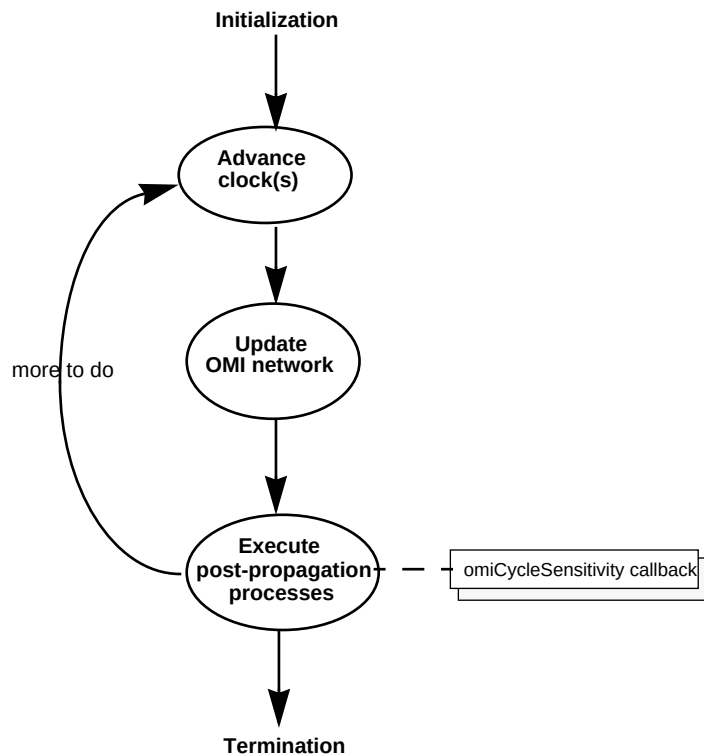


Figure 8—Cycle-based simulation cycle with callbacks

If the session is a cycle-based simulation session, then a clock domain specification shall be provided for each clock in the model. (Clock domain specifications are optional in an event-driven simulation session.) A clock domain identifies a clock and a set of ports that are dependent on that clock; special restrictions apply to clock ports (see 7.8). If clock domain specifications are provided, then they shall be complete. Each clock in the model shall have one, and each dependent of that clock shall be identified. If clock domain specifications are present, the simulator can assume that the absence of a dependency means that there is truly no dependency and that the cycle-based callbacks registered for an instance of the model will reflect the specified dependencies.

Cycle-based post-propagation processes are implemented with cycle sensitivity callbacks (see 6.4). An OMI cycle-based process may be activated (scheduled for execution during the current cycle) by a rising edge, a falling edge, or any change in value. A cycle-based process may be sensitive to both synchronous (clock) nets and asynchronous (nonclock) nets.

A model instantiated in a cycle-based simulation session shall not have transparent data paths. A transparent data path exists between an input and an output if a change in the input port's value directly affects the output port's value in the same cycle. A transparent data path typically occurs because the path between the input and output is purely combinatorial.

The OMI does not specify in detail how clocks in a model instance are coordinated with other clocks in the simulator. This coordination is implementation-defined; a simulator may even require assistance from the model user in this task. Furthermore, a simulator may choose not to support certain kinds of waveforms; for instance, a simulator may choose to accept only those clocks with 50% duty cycles.

In the absence of more detailed information about the relationships of ports to clocks and clocks to other clocks, the cycle-based simulator shall assume the worst case in performing its scheduling. To support optimal scheduling in multiple clock domain systems, a model may provide clocking information to the simulator on an optional basis. A model can provide three kinds of clocking specifications:

- A description of a clock waveform
- A description of clock/port relationships
- A description of clock value relationships

These specifications are described in more detail in 7.8.

NOTE 1—If no clocking information is provided by the model and no user interaction exists with the simulator to provide clocking information, then worst case scheduling by the cycle-based simulator should be expected. To prevent worst case scheduling, model suppliers are strongly urged to provide the clocking information with their cycle-based models.

NOTE 2—The same net can be connected to both clock and nonclock ports.

NOTE 3—A clock can be unconnected. The ability to leave a clock unconnected is particularly useful for scan clocks.

NOTE 4—An event-driven simulator is required to support models with timing and cycle-based models. However, an event-driven simulator is not responsible for providing a clock-generator capability for use with cycle-based models or for verifying that nets connected to clocks on cycle-based models satisfy clocking specifications. The OMI model user is responsible for connecting the model to nets in a manner that satisfies the clocking specifications for the model.

6.4 Simulation cycle callbacks

The OMI defines a small number of callbacks that are used to implement simulation cycles; these are the `omiSimulationStep`, `omiCycleSensitivity`, `omiNetSensitivity`, and `omiNetUpdate` callbacks. The `omiSimulationStep`, `omiCycleSensitivity`, and `omiNetSensitivity` callbacks may be used to implement OMI processes of the sort described in 6.1. These callbacks, when executed, are presented with a consistent view of the OMI network. Table 7 defines the execution semantics for each kind of callback. The `omiNetUpdate` callback, although executed during a simulation cycle, is provided to quickly record an individual net change rather than to view the OMI network. Each of these callbacks is discussed in more detail below. Callbacks in general are discussed in 5.7.1.

The `omiSimulationStep` and `omiNetSensitivity` callbacks may be executed at different points in a simulation cycle or time step. Hence each shall be qualified with an indication of exactly when the callback should take place. The point of callback execution is specified through the `processKind` field of the description structure; it is of the enumeration type `omiProcessKindT`, which defines values corresponding to the processes described in 6.1.

Table 7—Indications of when callbacks should occur

Indication	Explanation
omiNewTimeStep	Callback occurs at the beginning of a time step. The simulation clock has just advanced, and no OMI nets have been updated yet. Zero-delay updates scheduled by this sort of callback take place in the first simulation cycle of the time step.
omiPostPropagation	Callback occurs after all of the OMI nets have been updated for a particular simulation cycle. Zero-delay updates scheduled by this sort of callback take place in the next simulation cycle.
omiEndTimeStep	Callback occurs after all update activity for the specified time step has taken place, just before simulation time advances. No further updates to OMI data objects shall take place in the current time step.

An **omiSimulationStep** callback implements a process that is executed during a particular time step. The callback definition structure for a **simulation step** callback contains an **activationTime** field. Unless explicitly removed by the model manager, the callback shall be executed when the current simulation time reaches the activation time at the specified point in the execution of that time step.

The sensitivity callbacks (i.e., **omiCycleSensitivity** and **omiNetSensitivity**) implement processes that are activated by changes in the values of nets. A sensitivity callback shall be executed at most once in a given simulation cycle, even if more than one sensitivity condition activates that process for that cycle. If none of the net sensitivity conditions are met in a given cycle, then the callback shall not be executed at all during that cycle.

The **omiNetSensitivity** callback implements a process that is sensitive to changes on one or more nets. The callback description structure for an **omiNetSensitivity** callback contains a **nets** field that denotes one or more net identifiers constituting the sensitivity set for the callback. These nets shall be connected to ports that are either input or bidirectional. The **processKind** for this sort of callback shall be **omiPostPropagation** or **omiEndTimeStep**. If activated by a net sensitivity condition, then the callback shall execute at the specified point in the time step.

An **omiCycleSensitivity** callback implements a post-propagation process that is controlled by one or more nets and schedules activity with unit delay. It is erroneous for a cycle sensitivity callback to call a scheduling routine that takes a delay as an argument. Unlike other process callbacks, cycle sensitivity callbacks are associated with a particular instance; however, a simulator may ignore the instance information if it is not needed. Specifically, the use of nets and drivers with cycle-simulation callbacks is restricted as follows:

- Each net in the **nets** array of the callback description structure shall be connected to a port on the specified instance.
- A driver identifier provided in a call to a scheduling routine from a cycle sensitivity callback shall be associated with the instance with which the callback was registered.
- A net identifier provided in a call to **omiGetNetValue** or **omiGetNetGroupValues** from a cycle sensitivity callback shall be associated with the instance with which the callback was registered.

Cycle sensitivity callbacks shall be registered before the initialization phase begins.

The **omiNetUpdate** callback provides a mechanism for determining precisely which nets are changing and simultaneously obtaining their new values. **omiNetUpdate** callbacks are different from the other

simulation cycle callbacks because they do not present a stable view of the OMI network. They are executed during network propagation, hence the OMI network has not necessarily settled when a **net update** callback is executed. When an **omiNetUpdate** callback is registered, the callback description shall be qualified with the net identifier of the net of interest (the **net** field). When the callback is executed, the callback information structure contains a special **currentValue** field reflecting the current value of the net. Zero-delay updates scheduled by a net update callback take place in the next simulation cycle rather than the current one.

A simulator's implementation of net update callbacks shall be such that the following holds: if a net has a different value at the end of network propagation, then there shall be at least one net update callback for that net during that simulation cycle, and the current value in the last net update callback for that net in that cycle is the value observed by all post-propagation processes executed in that cycle. Depending on how the simulator implements network propagation, multiple net update callbacks may be executed for a particular net during a single network propagation phase. Furthermore, a simulator may choose to execute net update callbacks every time that it is updated or only when the net changes value; therefore, the current value is not necessarily different from the previous one.

The model manager may provide an analogous **omiViewportUpdate** callback to the application. A model manager may restrict the nature of a callback for a particular viewport. For instance, the model manager may allow a memory viewport to be monitored in its entirety, but not return the value of an entire memory viewport in the callback information structure. If a model manager imposes such a restriction on a callback, it shall indicate the restriction through the OMI error reporting mechanism when the callback is registered (e.g., through an **omiNoValueUpdate** warning).

NOTE 1—Cycle sensitivity callbacks are the only simulation cycle callbacks supported by a cycle-based simulator. They are used to implement purely functional models.

NOTE 2—Sensitivity callbacks are expected to be the primary vehicle for implementing net sensitive behavior since they provide a stable view of the OMI network. The current value of a net or set of nets may be obtained during a sensitivity callback by calling either **omiGetNetValue** or **omiGetNetGroupValues**.

NOTE 3—Some differences and similarities between net sensitivity callbacks and cycle sensitivity callbacks are as follows:

- Both may be used to implement a post-propagation process.
- Cycle sensitivity callbacks are always post-propagation processes; net sensitivity callbacks may also implement end-time-step processes.
- Cycle sensitivity callbacks are instance-specific; net sensitivity callbacks are not.
- Cycle sensitivity callbacks do not schedule with explicit delays; net sensitivity callbacks may schedule with or without explicit delays.
- Cycle sensitivity callbacks are supported by both event-driven and cycle-based simulators; net sensitivity callbacks are supported only by event-driven simulators.

NOTE 4—Once **omiEndTimeStep** processes begin execution, no further updates to OMI data objects are permitted. However, updates to OMI data objects in subsequent time steps may be scheduled from an **omiEndTimeStep** process.

NOTE 5—The net update callback is intended primarily to support debugging operations or to record specific net changes. A model manager cannot rely upon detecting glitches through an **omiNetUpdate** callback. A simulator is not required to execute **omiNetUpdate** callbacks exactly as the nets are updated during network propagation, although it may choose to do so. It may instead buffer the information and execute the callbacks at the end of its propagation phase. Maximum flexibility is provided to the simulator in implementing this callback to allow it to adopt the most efficient implementation possible.

Programming interface references:

- See 9.10 for `omiCallbackReasonT`.
- See 9.72 for `omiProcessKindT`.
- See 9.7 for `omiCallbackDescriptionT`.
- See 9.9 for `omiCallbackInfoT`.
- See 9.8 for `omiCallbackIdT`.
- See 9.11 for `omiCallbackRoutineT`.
- See 9.73 for `omiRegisterCallbackRoutineT`.
- See 9.75 for `omiRemoveCallbackRoutineT`.

6.5 Dynamic data transfer

The dynamic transfer of values across the ports of a model instance is achieved through connections to nets and drivers in the environment. Nets and drivers are represented by identifiers of type `omiNetIdT` and `omiDriverIdT`, respectively. They denote distinct aspects of a port connection for transferring values; nets are used to read a port, and drivers are used to write to a port. These identifiers are established by the application at elaboration time and remain constant throughout simulation. These identifiers are not handles and cannot be used to retrieve general information about the nets and drivers.

The application is responsible for creating a unique driver for each connection to an OMI output or bidirectional port. The model manager that is associated with the model instance is the only one that is allowed to schedule updates to that driver. It is erroneous for the application to associate the same OMI driver with more than one output or bidirectional port.

The application provides a number of routines that a model manager may call to read net values and schedule driver updates. Some variants allow a model manager to operate on individual nets or drivers or on groups of nets or drivers. Other variants allow a model manager to schedule updates with explicit delays or with an abstract unit delay. More information about these routines is provided in 6.5.1 through 6.5.4.

6.5.1 Reading net values

The application provides two routines for examining net values: `omiGetNetValue` retrieves the current value of a single net, and `omiGetNetGroupValues` retrieves the current values of a group of nets all at once. Special restrictions on the use of these routines from a cycle-sensitivity callback are discussed in 6.4.

A net value may also be implicitly obtained through the `omiNetUpdate` callback described in 6.4.

NOTE—Because a net may have other contributions to its value, the value of an OMI bidirectional port may differ from the value driven by the model instance at any given time.

Programming interface references:

- See 9.46 for `omiGetNetValueRoutineT`.
- See 9.45 for `omiGetNetGroupValuesRoutineT`.

6.5.2 Scheduling updates

An update to an OMI port is scheduled to take place with a certain delay, thereby placing the update in a particular simulation cycle. This delay is achieved by associating the update with a driver rather than directly modifying the value of the corresponding net.

The driver is manipulated by the model manager through the routines provided for that purpose by the application. Pending updates shall be removed from the driver by the application when they mature (i.e., when the simulation advances to the specified cycle or time and net values are being computed). This

circumstance is the only one under which the application shall modify an OMI driver without explicit direction from the model manager.

The OMI supports two basic kinds of updates: those with an explicit delay (e.g., 5 nanoseconds) and those with a unit delay. A *unit delay* is an abstract delay that has no time associated with it. An application may choose to implement a unit delay as a zero delay, or it may choose to implement it as some other constant delay. In any case, the effect of a unit delay update shall never be immediate; the new value shall take effect in a subsequent simulation cycle.

A model shall not mix these two forms of update. A model manager shall update all of the ports of a particular model instance either with unit delays or with explicit time-valued delays. The simulator may choose whether to verify this required consistency.

The application provides different routines for performing these two different kinds of scheduling. The routine `omiScheduleGroupUpdates` schedules unit delay updates to one or more drivers. This routine shall be used exclusively if a model has a `UsesUnitDelay` attribute (see A.1) and the value is true. The routines `omiScheduleDelayUpdate` and `omiScheduleDelayGroupUpdates` schedule explicitly delayed update(s) to a single driver or a group of drivers, respectively.

A unit delay driver has at most one pending update. The value in a unit delay update is placed on the driver, replacing a previously scheduled update if present.

The semantics for updating a driver with explicit delays are more complex. Conceptually, such a driver has a (possibly empty) sequence of pending updates associated with it, ordered by increasing simulation time. When scheduling an update to a port, the driver associated with that port is modified according to the semantics of the *delay mode*, which may cause previously scheduled updates to be cancelled by removing them from the driver. The delay mode semantics ensure that at any given time an OMI driver has at most one update scheduled for a particular time step.

Prior to modifying the driver for an update with an explicit delay, the application shall compute the absolute simulation time for the update by adding the specified delay to the current simulation time. If an overflow or numeric error occurs in the computation of the absolute time, then the update shall not be scheduled.

The scheduling delay modes in Table 8 are supported by an OMI-compliant simulator:

Table 8—Scheduling delay modes

Delay mode	Explanation
Inertial All	All pending updates are removed, and the new update is placed on the driver.
Transport	The new update is placed at the appropriate position in the sequence of pending updates, replacing a previously scheduled update for that time step if present. All updates scheduled to occur at a later simulation time are removed.
Pure Transport	The new update is placed at the appropriate position in the sequence of pending updates, possibly overriding a previously scheduled update for that time step. No other updates are removed from the driver.

Once OMI network propagation has begun, any zero-delay updates scheduled during that cycle shall take effect during the next simulation cycle. Similarly, any unit delay updates shall take place after propagation in a subsequent simulation cycle. See 5.5 for the definition of simulation cycles and time steps.

Pending updates may be explicitly cancelled by a model manager through the routine `omiCancelDriverUpdate`, which removes the update scheduled to occur at the specified absolute time. This routine is only applicable for event-driven simulation.

To support resolution of binary bidirectional ports, a model manager may use the `omiDisableDrivers` routine to indicate that it is not driving one or more bidirectional ports.

NOTE 1—The OMI does not allow “no delay” update scheduling, i.e., scheduling of updates that can or must be executed immediately. A newly scheduled update for a particular simulation time always replaces a previously scheduled update for that time. As a result, at most one OMI update to a particular driver can occur in a given simulation cycle.

NOTE 2—A purely functional model typically chooses to perform unit delay updates. A model may assist the simulator in its optimization by providing the optional predefined `UsesUnitDelay` attribute (see A.1) to indicate whether the model uses unit delay updates.

Programming interface references:

- See 9.26 for `omiDelayModeT`.
- See 9.87 for `omiScheduleDelayUpdateRoutineT`.
- See 9.86 for `omiScheduleDelayGroupUpdatesRoutineT`.
- See 9.88 for `omiScheduleGroupUpdatesRoutineT`.
- See 9.14 for `omiCancelDriverUpdateRoutineT`.

6.5.3 Group data operations

An application provides three routines that operate on groups of nets or drivers, allowing signal data to be passed in a single vector form. These routines are `omiGetNetGroupValues`, `omiScheduleGroupUpdates`, and `omiScheduleDelayGroupUpdates`. These routines share some common semantics that are defined below.

A group of nets is represented by the model manager as an array of net identifiers. Similarly, a group of drivers is represented by the model manager as an array of driver identifiers. The ordering of nets or drivers within a group array does not imply a processing order; it is erroneous for a model manager to rely upon a particular order for their processing. None of the identifiers in a group array shall be null references.

The number of group members shall be indicated by a count argument; it is erroneous for its value to be less than 1. The group `nets` or `drivers` array shall not be assumed to be null-terminated.

The data for a group value are exchanged through a value of type `omiGroupDataT`. Though the type of this value is a `void *`, it is actually an array. This `values` array shall not be assumed to be null-terminated. A direct positional correspondence exists between the items in the group `nets` or `drivers` array and the relative position of the corresponding value in the group value array. The element type of this array is interpreted in one of two ways, depending on the value of the `isPacked` argument, which indicates whether the scalar values are stored in the group value array using the smallest possible representation.

If the data are not packed, then the `values` argument shall be interpreted as an array of pointers to data value structures (`omiDataValueT **`), with the positional correspondence described above.

If the `isPacked` argument is true, then special restrictions apply to the nets or drivers. In this case, their data types shall define a structurally equivalent set of logic values of the same kind of logic. The rules below define equivalence among this set of data types. In the case of composite data types, these rules shall apply, recursively, to corresponding element data types as well.

- The net or driver data type shall be either a logic type or an array type.

- If a net or driver data type is a logic type, then it shall denote the same kind of logic as the other logic types.
- If corresponding net or driver data types are array types, then each shall have the same number of elements.
- The kind, storage type, and size of each corresponding net or driver data type shall be the same.

Finally, if the data are packed, the specified nets or drivers shall have the same atomicity. In other words, either all are atomic, or none is atomic. The application may choose either to assume adherence to these rules or to verify it.

If the data are packed, then the **values** argument is interpreted as an array of logic values, packed as tightly as possible (see Table 9).

Table 9—Logic value packing

Logic kind	Value size in bits
omiMVL2LogicData	1
omiMVL4LogicData	2
omi1164LogicData	4
omi1364LogicData	8

An application shall not assume that the storage associated with a group member array or group value is valid upon the return of one of these routines.

NOTE 1—The purpose of the packing capability is to allow very efficient transfer of a block of logic values. The application may use a tight loop to perform value processing since all values share the same characteristics. Though the packing capability does not reduce the space required for transferring **omi1364LogicData** values, it may still result in more efficient processing.

NOTE 2—It is not an error if a particular net or driver appears multiple times in the same group, although this practice is not particularly useful or efficient.

Programming interface references:

- See 9.48 for **omiGroupDataT**.
- See 9.45 for **omiGetNetGroupValuesRoutineT**.
- See 9.88 for **omiScheduleGroupUpdatesRoutineT**.
- See 9.86 for **omiScheduleDelayGroupUpdatesRoutineT**.

6.5.4 Viewport values

Viewport values are read and modified through handles that may be obtained by the application after a model instance is elaborated. Viewport handles may be obtained by iterating through the **omiViewports** relationship on the model instance or by a lookup operation using the viewport definition name and the model instance scope (see 7.10.6 for details on these methods).

The current value of a viewport may be obtained through one of two routines provided by the model manager: **omiGetDataValue** or **omiGetElementValue**.

Ports and viewports are not updated in the same way because they are different sorts of data objects. From an OMI perspective, viewports are variables rather than signals (although they may in fact be internal signals). Although certain kinds of viewports may have internal drivers, those drivers are not a part of the model instance boundary. Ports defined by a model provider are required for the correct operation of a model; a viewport is a debug or informational aid in using the model. Direct interaction between an application and a viewport is not required, and a model manager shall not rely on such interaction taking place. Similarly, an application shall not rely on the existence of a viewport for correct execution.

Updates to viewports cannot be scheduled in advance by the application. An application may, however, change the value of a viewport through a deposit if the model provider allows the viewport to be modified. A *deposit* is an asynchronous change in the current value of a viewport, requested by the application (see 9.27). Unlike a scheduled update, a deposit takes effect immediately.

A model manager provides an `omiDepositValue` routine that allows an application to deposit a particular value on a port or a viewport. The deposit capability is provided to enhance the usability of OMI-compliant models; deposits are not an essential part of model integration. For example, a deposit might be used to initialize a register viewport or to allow manipulation of the model instance boundary through an interactive user interface. It is erroneous for an application to use deposits to implement implicit drivers for port or viewports. It is erroneous for a model manager to rely on the effect of a deposit for correct simulation of its model instances.

A model manager may restrict the forms of access for a particular kind of viewport. For instance, it may require the application to manipulate a viewport representing a memory through individual memory words. Similarly, the model manager may require the application to manipulate other types of viewports in their entirety rather than through individual bits. Such restrictions shall be indicated to the application through an `omiElementAccessOnly` or `omiElementAccessNotSupported` error code when an unsupported form of access is attempted. Viewport manipulations through the following types of model manager routines might result in such an error:

- `omiGetDataRoutineT`
- `omiGetElementRoutineT`
- `omiGetElementValueRoutineT`
- `omiDepositValueRoutineT`
- `omiRegisterCallbackRoutineT`

An application may monitor a viewport by registering a callback on the viewport, as described in 6.4.

NOTE—Conceptually, a viewport should not be connected to anything in the application that represents a real piece of hardware; a port should be the point of connection in that case.

Programming interface references:

- See 9.27 for `omiDepositModeT`.
- See 9.28 for `omiDepositValueRoutineT`.
- See 9.39 for `omiGetDataValueRoutineT`.
- See 9.41 for `omiGetElementValueRoutineT`.

6.6 Reverting to a previous state

The OMI defines a small set of functions that may be used to revert a session to a previous state. The two basic forms of this sort of operation are reset and restore. A *reset* returns the session to its state at the beginning of the simulation stage. A *restore* returns the session to a previously saved simulation state that was created by a save operation; this saved state may be one from the current session or one from a prior session.

Support for reset, save, and restore is optional. Save and restore are companion operations; either both are supported or neither is supported. If the model manager does not support one or more of these activities, then the corresponding entry or entries in its function tray are null. If the application does not support one or more of these operations, then it will deny any control requests to perform the unsupported operation.

A reset, save, or restore operation shall be initiated by the application. If a model manager supports these operations, then it provides routines for the application to call. If the model manager wishes for one of these operations to take place, then it makes a control request asking the application to initiate the operation (see 5.7.2).

Callbacks that were in effect during the previous state shall be in effect after a reset or restore. Any currently active callbacks that were not in effect in that previous state are no longer in effect after a reset or restore operation. The callback identifiers for the callbacks that are no longer in effect are implicitly released when execution of the reset or restore operation begins.

NOTE—The reset operation returns execution to the point just after initialization and just before simulation cycle zero.

6.6.1 Reset

A reset returns a session to its state at the beginning of the simulation stage. A reset may take place any time after the elaboration stage. The current state presented to the model manager by the simulator during the reset operation itself may be the state at the point that the reset was initiated or the state at the beginning of simulation (that is, the state that is the goal of the reset operation). It is erroneous for a model manager to rely on a particular implementation choice.

The application executes a reset by invoking the `omiReset` routine of each model manager in the session. Simulation then resumes with execution of the `omiBeginSimulation` callbacks.

A reset occurs in the context of a single OMI session. In other words, a reset cannot be used to revert to an initial state in a previous session.

Programming interface references:
See 9.76 for `omiResetRoutineT`.

6.6.2 Save and restore

A restore operation occurs with respect to a previously saved state called a *checkpoint*. A checkpoint has a name that uniquely identifies the current state in a manner that is meaningful across OMI sessions. The syntax and interpretation of this name is determined by and understood by the application.

The application invokes the model manager's `omiSave` routine to initiate a save operation, creating a new checkpoint. An application can initiate a save at the following points in execution:

- At the beginning of simulation
- At the beginning of a simulation cycle
- At the end of a simulation cycle
- At the end of a time step

An application invokes the model manager's `omiRestore` routine to revert the session to a particular checkpoint. A restore can occur anytime after a session has begun.

The application and the model manager each shall be responsible for saving its own checkpoint data; however, the application shall select a location on the file system for storing the model manager's checkpoint

data. Through arguments to the `omiSave` routine, the application shall provide the model manager with a location of a nonexistent file on the host file system. The model manager can create this file as either a simple file or a directory, according to its needs. The save operation shall be such that the checkpoint name and checkpoint file location are the only information required by the model manager to identify and perform a restore operation.

It is erroneous to attempt to revert to a previous state involving a different design. Furthermore, it is erroneous to attempt to revert to a session of a different session type.

If the restore occurs in a different session from the save, then session-specific options may be different. The application and the model manager shall each verify that any difference in the options that it uses are acceptable for the restored state.

NOTE—A common approach to identifying a checkpoint is to use a file name supplied by the user. The OMI supports, but does not require, this approach to naming checkpoints.

Programming interface references:

See 9.83 for `omiSaveRoutineT`.

See 9.78 for `omiRestoreRoutineT`.

Since opaque references may be implemented as pointers, they shall not be assumed to be valid across sessions. Consequently, special steps shall be taken to preserve and restore this information in a memory-independent way so that save/restore will work properly. The routines `omiSaveReference` and `omiRestoreReference` are defined for this purpose. If an application or a model manager supports save and restore, then it shall provide an implementation of these two routines.

`omiSaveReference` converts an opaque reference to a form that can cross session boundaries, and `omiRestoreReference` converts this persistent form back to one that is usable during simulation in another session. The persistent form shall be the same across checkpoints within a single session; hence repeated calls to `omiSaveReference` are not needed for multiple checkpoints.

In performing a save or restore, the application and model manager shall use the other's conversion routines to save and restore the opaque references that are in effect. It is erroneous to invoke an `omiSaveReference` routine prior to the initiation of a save operation in that session. Similarly, it is erroneous to invoke an `omiRestoreReference` routine prior to the initiation of a restore operation in that session.

NOTE—Depending on how the software has implemented its opaque references and private data, these functions may actually return the same value because the value is already in a persistent form. For instance, these functions might return the same value if the model manager implements handles as array indices.

Programming interface references:

See 9.70 for `omiOpaqueKindT`.

See 9.82 for `omiSaveReferenceRoutineT`.

See 9.77 for `omiRestoreReferenceRoutineT`.

6.7 Backannotation

An OMI model may have configurable timing information (e.g., delays and constraints) associated with it. Default timing values may be modified prior to simulation through a process called *backannotation*. Backannotation shall take place only in an event-driven simulation session; support for this process in an event-driven simulation session is optional.

OMI timing information is identified and relayed using Standard Delay Format (SDF) syntax. Default timing information is associated with a model, and actual timing information is associated with each instance of that model. See 7.7 for a discussion of timing information and how it is represented in the OMI information model.

A model manager that supports backannotation shall provide a pair of routines to support that process. The `omiSetSDFVersion` routine allows the application to establish an SDF version for the communication of timing information. The `omiSetTimingData` routine changes the value of a particular piece of timing information.

Backannotation is not a separate OMI stage; backannotation operations may take place anytime between the beginning of elaboration and the beginning of initialization.

Prior to beginning the backannotation process, the application shall set the SDF version. If the model manager does not support the specified version, then it raises the appropriate error. During the course of the backannotation process, the application may change the SDF version by invoking the `omiSetSDFVersion` routine again. The specified version applies until the next call to this routine. This routine shall not be called after initialization begins.

Individual timing information is modified by a call to the `omiSetTimingData` routine. The application identifies the information by supplying an instance handle and an SDF string that describes the timing information and the new value. A boolean argument indicates whether to interpret the timing value as absolute or incremental (if this distinction is meaningful for the SDF construct). This routine shall not be called after initialization begins.

NOTE 1—Default timing information may be obtained by iterating through the timing definitions associated with a model. Actual timing information may be obtained by iterating through the timing definitions associated with a particular model instance. No OMI mechanism exists for obtaining timing information directly through some sort of name.

NOTE 2—Selection of min/typ/max timing is handled through the begin session routine (see 5.2.2).

Programming interface references:

See 9.61 for `omiMinTypMaxT`.

See 9.93 for `omiSetSDFVersionRoutineT`.

See 9.94 for `omiSetTimingDataRoutineT`.

7. Model information

As explained in the previous clause, the OMI defines an interface between an application and a model manager. A fundamental function performed by this interface is to allow an application to understand precisely what model information is being provided by a model manager. This understanding enables the application to correctly utilize the model manager in accomplishing its task.

The information provided to the application by the model manager is realized in the OMI as a set of *objects* that are manipulated through the programming interface. These objects are the basis for the communication that takes place between an application and model manager. As such, the different kinds of objects that are available and the information content of those objects needs to be precisely defined. The *OMI information model* is the means of specifying this definition in this standard.

The purpose of this clause is to describe not only *what* information is available to an application from a model manager but also *how* this information is accessed and manipulated. To explain the former, the OMI information model is presented and the meaning of this information is described; to explain the latter, the kinds of routines available for accessing the information are outlined (see in particular 7.10).

This clause begins with a description of the information modeling notation used in this standard. This description is followed by a sequence of subclauses that introduce the various kinds of OMI objects and then present the definition of these kinds of objects in the terms of the OMI information model. Finally, the approach for accessing the information is described.

Clause 3 introduces the reader to the concepts of a library and a model. Libraries and models are examples of kinds of objects that are defined in the OMI information model in this clause. Furthermore, Clause 3 indicates that a library serves to collect together a set of models; this association between a library and the models that it groups is an example of a relationship that is formalized in the OMI information model in this clause.

A number of the kinds of objects defined in this clause concern the interface that a model exhibits to an application. The model interface is the portion of the model that is used in integrating the model into the activity performed by the application (e.g., simulation). Of primary importance in this interface are *ports*, *parameters*, and *viewports*. Ports are the means by which signal information flows back and forth between the model and application during simulation. Parameters provide the ability to tailor model instances to fill particular needs when they are created. Viewports are internal (normally hidden) model objects that the model developer has chosen to expose to the application. Ports, parameters, and viewports are further examples of kinds of objects defined in the OMI information model in this clause.

A port is an example of an OMI *data object*. A data object is any kind of object that possesses certain common characteristics such as a name and a value. A data object also has an associated *data type* that defines the set of values the data object may assume. Data object and data type are examples of classes of objects that are defined in the OMI information model in this clause.

A final important concept defined in the OMI information model is that of an *attribute*. Attributes are a means for decorating certain OMI objects with additional information in order to describe the characteristics of an object beyond its basic set of information items. An attribute is defined in the OMI information model as a particular kind of data object.

7.1 OMI information modeling methodology

Much of the information provided to the application by the model manager through the OMI interfaces is formally defined in the OMI information model. The information-modeling notation used is based on a system of class definitions, object definitions, relationships, and properties and serves to formally relate the kinds of OMI objects to the interface routines that operate upon them.

7.1.1 Objects and classes

The OMI interfaces operate on *objects*, i.e., collections of information that describe a particular item, such as a port. The information that an object comprises is never accessed directly; instead, interface functions are available to return individual aspects of information about the object. Thus the users of an object are unaware of how an object is represented internally and the provider of an object has considerable freedom in implementing the required interface functions for the object.

This standard defines a number of different kinds of objects. An object has a tag that identifies the sort of item it represents and hence what sort of information is available about the object and what sort of operations may be performed upon the object.

Object definitions that share common aspects may be grouped into classes. A *class* serves as a notational convenience that allows characteristics (i.e., properties and relationships, see below) that are common across distinct object definitions or classes to be factored out and defined in a single place. When an object definition or class is specified, it is defined as having zero or more *parent classes*; in this case the object definition or class inherits the properties and relationships that are specified for each of its parent classes. Furthermore,