

# A Tale of Two Model Managers

Doug Dunlop and Kathy McKinley  
The Alta Group of Cadence Design Systems

## Abstract

*The OMI tool/model interface developed by the Open Model Forum (OMF) uses the concept of a "model manager". In its most general form, a model manager acts as an interfacing agent between a tool and a set of models that are used by that tool. In this paper we explore the role of a model manager in OMI model delivery and examine two possible model manager scenarios as detailed examples. The OMI interface is described in a companion paper [1].*

## 1. Introduction

The purpose of the Open Model Interface (OMI) is to enable interoperability between an EDA tool such as a simulator and a model of a device or system. OMI models are simulator and HDL independent, thereby eliminating the need for the end users of models to be tied to a particular set of tools. Further, the OMI interface protects the model developer's intellectual property since models are delivered in a compiled binary form. Recently, an IEEE standardization effort was launched to standardize the OMI interface. An overview of the interface can be found in a companion paper [1] and the full specification is available for download (see [2]).

The OMI specification utilizes the concept of a "model manager", a software component that acts as an interface element between an OMI-compliant EDA tool and a collection of related models to be used by that tool. A primary goal of OMI is flexibility with respect to model managers in order to accommodate gracefully a variety of model packaging/delivery scenarios. To meet this need the OMI interface provides the expressive power to allow considerable freedom in the architecture of a model manager and in the relationship it bears to the models it supports.

In this paper we discuss the role of a model manager in the OMI interface and explore two possible realizations of

an OMI model manager that are expected to be common in practice as OMI gains acceptance as standard model-delivery technology. These two realizations can be thought of as representing the ends in a spectrum of model manager possibilities. The first of these is a sophisticated model manager targeted to serve a collection of VHDL models and in which the model manager includes much of the functionality of a VHDL simulation engine. The second model manager discussed is a simple software element that supplements a cycle-based C model of a device such as a processor.

## 2. OMI model packaging

In OMI parlance, the term "application" is used for an EDA tool such as a simulator while the term "model" is used as a functional representation of a device or a system. Clearly applications work with models. The OMI goal is to promote interoperability between applications and models by defining an open, standard interface between the two.

So, what is a model manager? A model manager is a software element that is used in conjunction with one or more models to implement the model side of the OMI tool/model interface. A model manager as a distinct entity from a model is part of an important model-packaging technique that is recognized by the OMI. In particular, it may be useful for a model provider to distribute a number of models as a set or it may be useful for a model provider to allow a number of the models he has developed to participate in a given session with an application. An individual model typically requires support during execution for various operations. Examples of operations include reading values of signals, scheduling updates to signals, coordinating the execution of processes, etc. Some of these operations may involve interaction with the host simulator; others may not. But in any case when dealing with multiple models of a like kind it is convenient from an implementation point of view to factor out the

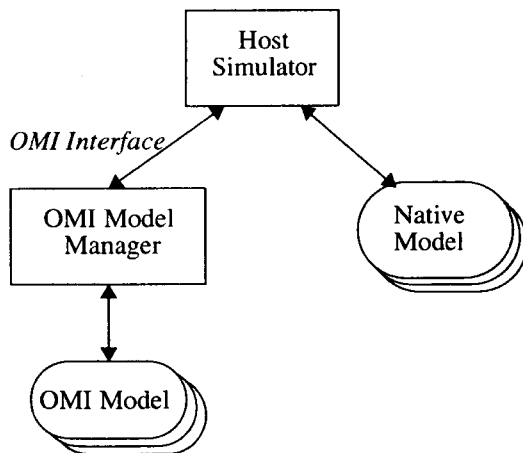
implementations of these operations and provide them once in a single software element. The OMI term “model manager” refers to such a software element.

It is important to understand that the OMI does not define what processing a model manager must perform vs. what processing a model must perform, or even if a model manager must be a distinct software element from a model. If a model manager and a model are distinct software elements, the OMI does not say anything about the interface between them. A model that works with one model manager will not in general work with a model manager developed by another party.

When an application is making use of OMI models there is dialog through the OMI interface that allows the application to understand what models and libraries of models are available from a particular model source. It is clearly convenient to consider the support for this that is delivered by the model supplier to be “model manager functionality” as opposed to “model functionality” but this distinction is largely conceptual and, in particular, from the point of view of the application, it is engaging in a dialog with a single entity throughout the entire session.

A common scenario involving use of OMI packaged models is shown in the following figure. The host simulator in the figure is working with a set of native models as well as one or more packaged models that interface to a particular OMI model manager.

**Figure 1: Common model manager scenario**



Note that other OMI model packaging scenarios are possible; in particular, there is no reason why there cannot be direct interaction between the host simulator and one of the packaged models.

OMI-compliant models may be developed in a variety of languages. These languages may be HDLs such as VHDL or Verilog or general-purpose programming languages such as C. C models may be written to directly implement the OMI interface or may implement the private

model interface of an OMI model manager designed to support a particular style of C modeling. For HDL models, a tool called a model compiler is often used to translate the HDL model into binary form, either to work through a private interface with a corresponding model manager or to interface directly to the application using OMI.

In the same way that a tool such as a simulator is typically designed to work with models in a particular native form, it must be expected that a model manager will assume models in a particular native form. As illustrations, a model manager for Verilog models will in practice be tied to a particular model compiler. Similarly, a model manager designed to work with C models will in practice work only with C models that exhibit a particular model interface. While these different model managers will interact with their models in dramatically different ways, what they have in common is that, when integrated with their models, they support the OMI interface and will function with any OMI-compliant simulator.

### 3. Model manager design considerations

In this section we provide background information on the OMI interface from the perspective of the designer of an OMI model manager. More specifically, we are describing the responsibilities of the software on the model side of the OMI interface, as well as the resources that this software may utilize in implementing its functions.

#### 3.1. Model manager function

The model manager has two primary responsibilities: it must provide routines that the application can use to extract model information, and it must implement the functionality of its models with respect to the OMI execution model.

**3.1.1. Model query.** In the OMI there are a set of routines called the “model query” functions that an application may call to extract various aspects of model-related information. Using these routines an application may determine the models that are available from a model manager in a given session and, for a given model, an application may determine the ports and parameters on the model and obtain various characteristics of those interface objects. The model query functions are also applicable for model instances, i.e., they may be used to extract information concerning model instances and the elaborated forms of ports and parameters.

The model query functions are based on the concept of a “handle”. A handle is the mechanism an application uses to refer to an object that the model manager is

responsible for. Example kinds of objects are models, model instances, parameters, ports, port elements, etc. With a handle to an object an application may use model query functions to obtain the properties of the object and it may use model query functions to traverse object relationships in order to obtain handles to other objects. The implementation of a handle is private to the model manager; an application does not know, for example, whether a handle is implemented as a pointer, array index or something else.

**3.1.2. Execution.** The execution of an OMI session involves a number of stages dedicated to specific activity: bootstrapping, elaboration, initialization, simulation, and termination. Although an OMI application does not have to be a simulator, simulation is the most important class of OMI-based activity. Much of the work required of a model manager is focused on preparing for and then performing simulation.

During the bootstrapping stage, the application and model manager establish the nature of the session and perform setup activity, such as license checking.

Instances of OMI models are created in the elaboration stage, during which an application invokes the model manager's `CreateInstance` routine to create a unique, customized instance. 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).

During initialization the initial values of objects having state (e.g. ports, nets, drivers) are established.

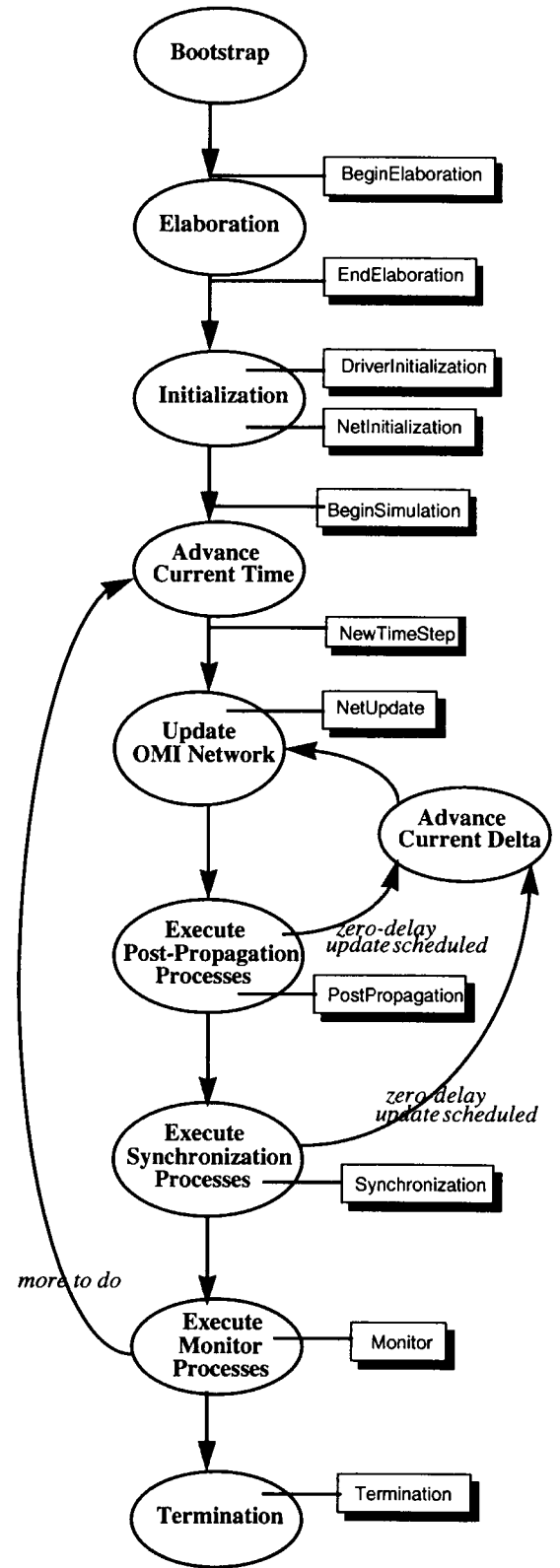
Simulation proceeds by executing a series of time steps, which in turn consist of the execution of one or more delta cycles — iterations of a multi-step cycle through which OMI nets are updated and processes evaluated. A time step ends with a monitor phase in which read-only activity is performed. Figure 2 below illustrates the OMI simulation algorithm in detail.

Clean-up activity (e.g. removal of temporary files) takes place during the termination stage.

### 3.2. Model manager resources

The OMI interface provides two basic forms of support to aid the model manager in accomplishing its tasks: a callback mechanism that it can use to customize its communication with the application, as well as a small set of application-provided routines that the model manager may call from its own routines.

**Figure 2: OMI execution model with callbacks**



**3.2.1. Callbacks.** 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 — it is generally the model manager that determines when to synchronize with the application by registering and removing callbacks.

A callback is a transfer of control from the application to a model manager that occurs at the request of the model manager. When a model manager registers a callback, it describes the conditions under which the callback is to take place, and provides an anonymous callback routine to invoke when the conditions are met. The model manager also supplies a pointer to private data that will be returned to it when the callback is executed. The callback mechanism gives the model manager considerable flexibility in devising its execution algorithm to best meet the needs of its models.

Callbacks are mapped to specific points in execution described by a reference model that supports a variety of event-based simulation systems. The OMI execution reference model is very similar to the VHDL execution model. Figure 2 shows the complete OMI execution reference model, annotated with boxes showing possible callback entry points.

An OMI process is a black box representing an independent thread of execution during simulation. It may be dedicated to a very low-level activity, such as computing the new value of a particular signal, or it may be a powerful engine that executes a whole set of internal parallel processes.

A model manager determines the granularity of its interactions with a simulator in a given time step. For instance, upon an input change, a model manager can choose to react: 1) immediately, 2) after network propagation is complete for that delta cycle, or 3) after the OMI network has settled for that time step (possibly in a later delta cycle) during a synchronization phase. By selecting when in a given time step a particular 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.

**3.2.2. Application-provided routines.** An OMI simulator provides a small set of routines for a model manager to use as building blocks in constructing its communication protocol.

**CurrentTime** — Provides current simulation time and next scheduled simulation time.

**ComposeTime** — Produces a time value in the OMI simulator's private time format from a number and a time unit.

**DecomposeTime** — Decomposes a value in the OMI simulator's private time format into a number and a time unit.

**SetParameter** — Provides the instance-specific value

for a particular model parameter.

**ConnectPort** — Provides connectivity information (net/driver) for a particular model port.

**InitializeDriver** — Takes the initial value of the driver for a particular port.

**ScheduleDriverUpdate** — Schedules an update for a particular driver with the specified delay.

**CancelDriverUpdate** — Cancels a previously scheduled update.

**GetNetValue** — Provides the current value of a particular net.

**RegisterCallback** — Registers a callback for the specified conditions.

**RemoveCallback** — Removes a previously registered callback.

**ControlRequest** — Allows the model manager to request a change in state (e.g. reset, termination).

## 4. A VHDL-based model manager

This section presents a high-level view of a model manager that operates on a compiled form of models derived from VHDL designs. The design presented here is not intended to be optimal, nor does it cover every detail that must be considered. It is intended to give the reader a feeling for what is involved in building this sort of model manager. This section begins with a likely model packaging strategy, followed by descriptions of how model query and execution might be handled.

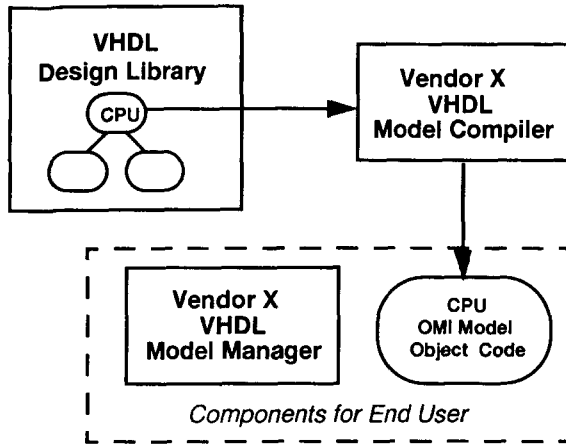
Like all model managers, the VHDL-based model manager has an intimate understanding of the internal data structures and execution algorithms on which its models rely. It may include a complete VHDL execution engine. It is not a generic VHDL model manager — it operates only on those models that are compiled into an expected internal form. It will not work with every OMI model derived from VHDL.

### 4.1. Model packaging

In this scenario, an OMI model is derived from a design in the developer's VHDL design library. The OMI representation is created by running the design through the model compiler that is bundled with the VHDL development environment. This process may involve elaboration or partial elaboration of the design in order to determine component bindings, etc. The model compiler produces an executable object code representation of the design that hides its internal details.

Figure 3 illustrates this packaging process. At the end user site, both the model and the model manager are required in order to use the OMI model.

**Figure 3: VHDL Model Packaging**



#### 4.2. Model query

The VHDL-based model manager provides access to all model and model instance information through special OMI objects that make the necessary information readily available. These objects are implemented as structures containing fields appropriate for the sort of item being represented. Some are created on demand (e.g. the object representing a model). Others are automatically created as a part of elaboration (e.g. the object representing an elaborated port) because they contain information essential for implementing OMI simulation.

An example of an essential OMI object might be one representing an elaborated port. Its primary function is to associate internal representations with their counterparts in the OMI application. It might contain references to definition information, the internal signal representation used by the internal engine, the net/driver identifier needed for communication with the OMI simulator, and bookkeeping information specific to OMI simulation.

An OMI handle could be implemented as a pointer to one of these objects, or as an index into an array of pointers to OMI objects, for an extra layer of data protection.

The information needed to implement model query is contained in the binary representation of the model and auxiliary memory or file representations created during execution. Since a model is not linked with the model manager in this scenario, the model manager provides a text-based model registry mechanism to allow the model manager to locate the models installed at a particular site.

#### 4.3. Execution

The OMI leaves many key decisions to the model manager designer, thereby supporting the design of optimal interactions with the OMI simulator. These include:

- How to map its internal activities to one or more

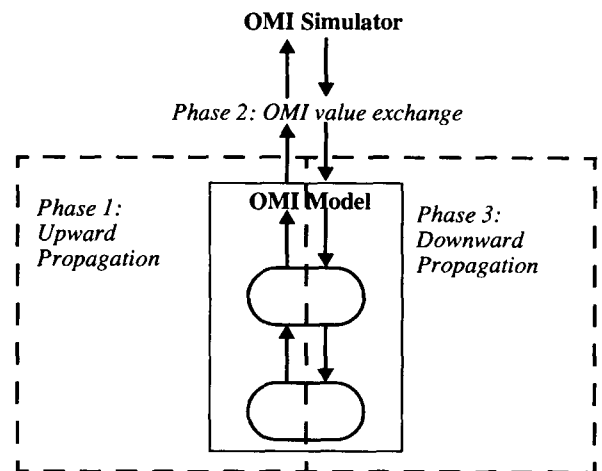
OMI processes

- How often, and when, to exchange values at the model boundary with the OMI simulator
- Whether to handle most of the scheduling on OMI output port drivers in the model manager, or ask the simulator to handle all of the scheduling

Major factors influencing these decisions include performance (e.g. how often do we want to pay the possible overhead of leaving the engine?) and convenience (do we want to embed OMI smarts in our internal engine, or implement the model manager as an external layer as much as possible?).

In this case, the VHDL-based model manager does not treat VHDL processes or instances as separate units that are mapped to individual OMI processes. Instead, the execution of distinct parts of the VHDL simulation cycle are mapped to different OMI processes.

**Figure 4: Signal value propagation**



In order to accomplish black box style cosimulation, the VHDL-based model manager splits the signal value propagation phase into three parts, as illustrated in Figure 4. First, signal values are propagated up the design hierarchy by computing driving values. Then, OMI output port driving values are provided to the OMI simulator, and the effective values of OMI input ports are obtained. Finally, signal values are propagated down the design hierarchy by computing effective values. This approach is used during both initialization and simulation.

In this scenario, the VHDL-based model manager chooses to do its own scheduling for ports, requesting to interact when new values take effect, and scheduling the new values immediately before the OMI network is evaluated.

To simplify this discussion, this model manager does not attempt to "run ahead" (i.e. execute as many time steps as possible without reaching the next scheduled OMI time), although such an approach would be more efficient.

**4.3.1. Setting up the session.** During the bootstrapping stage, the VHDL-based model manager registers some initial callbacks that are independent of the nature of the model instances that may be created during the session: `DriverInitialization`, `NetInitialization`, and `Termination`. Additional callbacks will be registered by these callback routines when they execute.

**4.3.2. Elaboration.** Most of the work performed by the model manager during elaboration is handled by the `CreateInstance` routine and the model query routines that it must provide. During the process of creating model instances, the application will typically use the model query facility for obtaining information needed for elaboration, e.g. to get the size and mode of a port. During this phase, the model manager will create and elaborate instances, as well as create the OMI objects needed to support model query and possibly simulation.

**4.3.3. Initialization.** Initialization is split across two callbacks: `DriverInitialization` and `NetInitialization`. The former performs post-elaboration activity and driver initialization, and the latter completes the VHDL initialization phase by computing the effective values (using the initial values of OMI nets obtained from the OMI simulator), registering input port sensitivities through `NetUpdate` callbacks, and executing the internal VHDL processes. At the end of initialization a callback is registered for the next time that the VHDL engine needs to run. If the engine is not scheduled to run again, then execution of the model manager will resume upon an OMI net update.

**4.3.4. Simulation.** Because of the way in which signal propagation is partitioned, the VHDL simulation cycle is implemented by three different callbacks: `NewTimeStep`, `NetUpdate`, and `PostPropagation`. They are described in more detail below.

The `NewTimeStep` callback performs the first part of the first simulation cycle in a time step. It handles activity previously scheduled for a future time — it is not a reaction to an OMI net update. This callback propagates signal values up the design hierarchy and informs the OMI simulator of any changes in OMI drivers. It then registers a `PostPropagation` callback for the current time step and returns control to the OMI simulator, giving it a chance to react to any changes. The rest of this simulation cycle, and any other simulation cycles that occur during this time step, are handled by one or more `PostPropagation` callbacks.

A `NetUpdate` callback simply records the fact that a particular port was updated during the cycle, and registers a `PostPropagation` callback for the current cycle if one is not already registered. The real processing of the update (obtaining the new value and propagating it if necessary) is performed during the `PostPropagation` callback. This

callback operates on an OMI port object that is passed into the callback as private data.

A `PostPropagation` callback performs the latter part of a VHDL simulation cycle. It executes either because an OMI input port was updated during the current cycle, or because internal activity was scheduled for this cycle. After synchronizing its internal clock, it retrieves any new OMI net values and performs the rest of the simulation cycle, beginning with downward propagation.

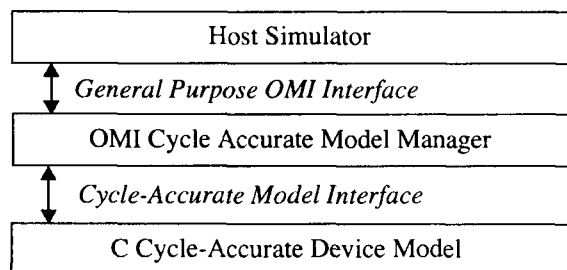
Before returning control to the OMI simulator, the `PostPropagation` callback prepares for the next simulation cycle. If zero delay activity was scheduled during process execution, it performs the first part of the next VHDL simulation cycle; the latter part of this delta cycle will be handled by a subsequent `PostPropagation` callback that executes after new OMI net values are obtained. Otherwise, it registers a callback for the next scheduled time step, if necessary.

## 5. An OMI model manager for a C cycle-accurate model

The OMI interface is by design general purpose in nature. It allows, for example, considerable flexibility with respect to the way processes may be executed, the way updates to ports may be scheduled, the way errors may be handled, the way ports and parameters may be typed, etc.

On the other hand, device models written for specific applications tend to follow a particular style that is focused on the particular kind of application. One important role of an OMI model manager is to act as a software element that allows special purpose kinds of models to exhibit the more general OMI interface so that they may be used with any OMI-compliant tool.

**Figure 5: OMI interface and model interface**



In this section we explore the idea of a model manager that is designed to work with a cycle-accurate model of a high-level hardware component such as a processor and memory. This arrangement is shown in Figure 5. By “cycle accurate” here it is meant that the model behavior faithfully reflects the values on the device

pins at clock cycle boundaries. Unlike the VHDL models discussed in the preceding section, the models here are assumed to be written in C.

### 5.1. The model properties file

One of our goals for our model manager of C cycle-accurate models is to incorporate into the model manager as much of the required OMI functionality as possible. A modular organization like this makes the task of maintaining or replacing the model simpler since the OMI-related functionality is written/validated once as part of the development of the model manager.

However, some of the OMI functionality we would like to include in our model manager is model-dependent processing. For example, we must provide a set of routines that support the OMI model query capability. One convenient way of providing these functions in an OMI model manager for a C model is for the model developer to document the static model characteristics separately in a "model properties file" which is used by the model manager in implementing the OMI model query capability. The model manager implementation may be one where certain portions of the model manager software are automatically generated off of a model properties file or one where portions of the model manager software operate off of a model properties file (or off data derived from that file).

In terms of information content, a model properties file includes the kind of information that would be present in a HDL model shell such as a VHDL entity declaration. In addition, however, application-specific information may also appear in a model properties file in order to facilitate the processing in the model manager. For example, our model manager for a C cycle-accurate model assumes that the model has an input port for the simulation clock and that the identification of this special port appears in the model properties file information used by the model manager.

### 5.2. Model interface

To be used by our model manager an integrated C model must exhibit a simple interface consisting of a small number of routines geared toward clock-driven simulation. The model must provide a `CreateInstance` routine to build a new instance of the model. The model may optionally provide an `Initialize` routine should it need to initialize the local state for an instance depending on the initial values of the nets the ports are connected to. The model provides an `ExecuteClockRising` and/or `ExecuteClockFalling` routine that implement the model behavior for the given transition on the clock.

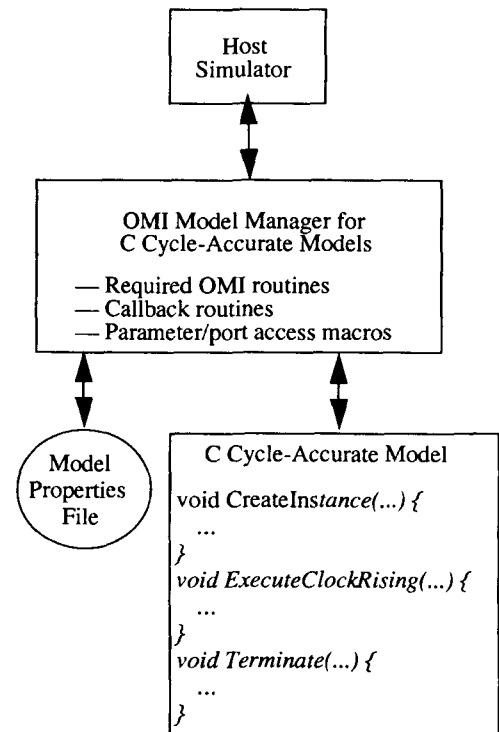
Finally, a model must provide `Terminate` routine that shuts down a model instance.

### 5.3. Model manager software

Our model manager for a C cycle-accurate model includes all the OMI routines that may be called directly by the host simulator. These include the model query routines described above as well as the bootstrap routines and the OMI routine for creating a model instance.

Also included in the model manager are a set of callback routines. These are called by the host simulator as requested by the model manager (see below). These callback routines are the drivers for the `Initialize`, `ExecuteClockRising`, `ExecuteClockFalling`, and `Terminate` routines provided by the model.

Figure 6: Model manager with integrated model



Finally, as a convenience to the model developer, our model manager for a C cycle-accurate model provides a simple set of macros for reading the values of parameters and input ports and for scheduling update transactions on output ports. The definition of these macros is based on the modeling needs of C cycle-accurate models. For example, the macros that schedule updates on output ports automatically use the OMI "unit delay" update scheduling functionality. Further, the macros for reading port values and scheduling port updates implement a particular semantics for unconnected ports (i.e., for the case where

the host simulator does not attach a net to a port on a model instance). Finally, the model manager macros are tailored based on the model properties file defined for the model. In particular, the macros to read port/parameter values give C types that correspond to the type of the port/parameter as given in the model properties file. The port update scheduling routines are tailored similarly.

For the macros that read port values efficiency and ease of use are important considerations. In the general case, determining the value of an input port requires invoking an OMI function provided by the host simulator. In order to minimize the need for this, it is helpful to track whether or not a given port has been read by the model in the current cycle. Once read, the current value of the port can be stored locally in the model manager and accessed directly for subsequent read operations on the port during that cycle. Further, for complex models such as those of processors and microcontrollers it seems unwise for the model manager to automatically pre-fetch values of input ports at the top of cycle because in many cases only a small subset of the ports would be read during the execution of that cycle.

## 5.4. Implementation overview

Basic to the implementation of our model manager is a C `ModelInstance` structure. There is a single `ModelInstance` structure definition for the model integrated into the model manager. This structure definition depends on the parameter/port interface of the model and in fact may be synthesized from the model properties file. The structures are created during elaboration as model instances are being created and are chained together for efficient implementation of operations on all instances known to the model manager.

The only argument to the model-provided routines discussed above is a pointer to the `ModelInstance` structure for the instance. The macros described above that provide access to parameters and ports assume they are being used in the model-provided routines since they use the `ModelInstance` pointer argument.

**5.4.1. Elaboration.** The model manager provides the OMI `CreateInstance` routine. This routine creates the `ModelInstance` structure and then engages in a dialog with the host simulator to obtain parameter values and net/driver information for ports. The parameter value (either the value supplied by the host simulator or the default value if the simulator did not provide a value) is stored in the `ModelInstance` structure as well as the net/driver information for ports. The OMI `CreateInstance` routine in the model manager calls the model-provided `CreateInstance` routine to allocate space for the model's internal

state for the instance and link this into the `ModelInstance` structure.

**5.4.2. Initialization.** During start-up the model manager registers an `DriverInitialization` and `NetInitialization` callbacks. The `DriverInitialization` callback routine uses information from the model properties files to assign initial driving values to the output ports for each instance. The `NetInitialization` callback routine invokes the model's `Initialize` routine if it provided one.

**5.4.3. Simulation.** During start-up the model manager registers an `NetUpdate` callback for the net attached to the clock on each model instance. The `NetUpdate` routine schedules an `Synchronization` callback. The `Synchronization` callback in turn invokes the appropriate model-provided `ExecuteClock...` routines for the model instances known to the model manager. Using the `Synchronization` callback allows the values of the nets to settle and guarantees that when the model behavior executes it sees a consistent state in the nets attached to an instance.

**5.4.4. Termination.** At start-up the model manager registers and `Termination` callback. This callback routine walks the list of model instances and invokes the model's `Terminate` routine for each instance.

## 6. Conclusion

Packaging and distributing models in compiled form is becoming an increasingly common way of delivering intellectual property. It is felt that the notion of a model manager as described in the OMI specification is a fundamental aspect of any comprehensive mechanism for distributing compiled simulation models. While the model manager concept is useful for delivery of models written in C it is critical for models written in an HDL and processed by a model compiler. In this paper we have discussed the role of a model manager in OMI and presented two concrete examples of OMI model managers.

## 7. References

- [1] "OMI — A Standard Model Interface for IP Delivery", Doug Dunlop and Kathy McKinley, IVC'97.
- [2] "OMI Model Interface Draft Standard Version 1.0", OMF Technical Team, see <http://www.cfi.org/OMF>.