

Functional Requirements Specification: Standard Co-Emulation Modeling Interface (SCE-MI)

**John Stickley and Duaine Pryor with feedback contributions
from the SCE-API Technical Group**

Copyright (c) 2001 Ikos Systems, Inc. all rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy, recording or otherwise without the prior written permission of the copyright holder. This document is under NDA with the SCE-API consortium.

Ikos Systems, Inc.
79 Great Oaks Blvd.
San Jose, CA 95119

SCE-API
Standard Co-Emulation API
Consortium

Abstract

This document describes a proposal for a standard C/C++ modeling interface for emulators and other verification platforms. The interface provides multiple communication channels between software models running on a host workstation and a device under test (DUT) represented as a structural hardware netlist running on a verification platform such as an emulator. The channel end points are presented as message ports to the DUT model and as message port proxies to the software models.

**SCE-API Version 1.0
Revision 1.9**

1.0 Introduction

1.1 Purpose

There is an urgent need for the EDA industry to meet the exploding verification requirements of SoC design teams. While the industry has delivered verification performance in the form of a variety of emulation and rapid prototyping platforms, there remains the problem of connecting them into SoC modeling environments while realizing their full performance potential. Existing standard verification interfaces were designed to meet the needs of design teams of over 10 years ago. A new type of interface is needed to meet the verification challenges of the next 10 years. This document is a proposal for a multichannel communication interface that addresses these challenges. This new interface must cater to the needs of both emulation end users and emulation suppliers.

1.1.1 Emulation Customer Problems to be Solved by SCE-API

- All emulators on the market today have proprietary APIs. The proliferation of APIs makes it very difficult for software-based verification products to port to the different emulators, thus restricting the solutions available to customers. This also leads to low productivity and low return on investment (ROI) for emulator customers who build their own solutions.
- The emulation “APIs” which exist today are oriented to gate-level and not system-level verification.
- The industry needs an API which takes full advantage of emulation performance.

1.1.2 Emulation Suppliers Problems to be Solved

- Customers are reluctant to invest in building applications on proprietary APIs.
- Traditional simulator APIs like programmable language interface (PLI) and VHDL PLI slow down emulators.
- Third parties are reluctant to invest in building applications on proprietary APIs.

1.2 Scope

This document is directed to members of the *Standard Co-Emulation API* (SCE-API) technical committee. The committee members represent a consortium of EDA customers and EDA tool suppliers dedicated to the creation of a breakthrough high-performance common emulation API. The founding participants in the consortium are Aptix, CoWare, Ikos Systems, Mentor Graphics, ST Microelectronics, Synopsys, and TransEDA.

The scope of this document shall be restricted to what is specifically referred to herein as the *Standard Co-Emulation API: Modeling Interface* (SCE-MI). This modeling interface is intended to be one of possibly several parts making up the whole SCE-API standard. In the future, SCE-API may be expanded to include additional parts to handle such features as debug and control, code coverage, etc.

This document is intended to describe an overall use model of the SCE-MI, a tutorial illustrating this use model, and a formal functional specification of the SCE-MI API itself. Future revisions of the document will continue to reflect the feedback from members of the SCE-API technical committee. It is hoped that the document will undergo a relatively quick evolution phase which culminates with a common industry standard that can be used by simulation and emulation tool vendors to provide plug-and-play high speed verification solutions to end users on SoC design teams.

1.3 Revision History

Revision #	Date	Revised by	Summary of revision
1.0	12-11-00	Duaine Pryor	Initial draft based on discussions at SCE-API kickoff meeting that occurred in Oct. 2000 in Paris, France.
1.2	1-15-01	John Stickley	Revised document to incorporate feedback and action items from second SCE-API meeting on Dec 14-15, 2000 in Cupertino, CA.
1.3	2-2-01	John Stickley	Revised document to incorporate feedback from e-mail review with SCE-API committee after the revision 1.2 release. Also completed Routed tutorial section.
1.4	2-16-01	John Stickley, Susan Ross	Incorporated comments from committee received during SCE-API phase 1 ratification process. Added minor corrections and converted hand drawn figures to electronic drawings.
1.5	5-6-01	John Stickley	Corrections to SceMiParameters accessors, added section on controlled reset semantics and clarification of controlled clock semantics, added typedefs for callback function pointer types, added port width accessors to port proxies, changed VHDL macro data types from <code>std_ulogic(_vector)</code> to <code>std_logic(_vector)</code> .
1.6	7-20-01	John Stickley	Changes based on feedback to 1.5 changes, added bit accessors to message data object, changed version format from <code>major.minor.minor</code> to <code>major.major.minor</code> .
1.7	10-19-01	John Stickley	Added clarification to how the <code>g()</code> function works with <code>::ServiceLoop()</code> , added note about memory allocation and deletion, added <code>::ReplaceBinding()</code> calls <code>::SceMiMessagePort[In/Out]Proxy</code> classes, added info message callback handler support, added clarification about parameters database structure, added clarification about input ready semantics.
1.8	11-14-01	John Stickley	Incorporated feedback, corrections to above.
1.9	2-26-02	John Stickley	Added support for “don’t care duty cycle”, clarifications about duty cycles, clock ratios. Changed default duty cycles on SceMiMacros. Added explanation about deriving clock ratios from clock frequencies. Added section about multi-clock alignment. Minor modifications to the SceMiIC info message handler structure.

1.4 Definitions, Acronyms, and Abbreviations

A formal glossary of terms used in this document can be found in the appendices. Some of the acronyms referenced in the document are defined here.

BCA - *Bus Cycle Accurate* model - sometimes used interchangeably with RTL model

BCASH - CoWare style *Bus Cycle Accurate SHell* model

BFM - Bus Functional Model

DUT - Device or Design Under Test

HDL - Hardware Description Language such as Verilog or VHDL

ISS - Instruction Set Simulator

RTC - Register Transfer Level C model

SCE-API - Standard Co-Emulation API

SCE-MI - Standard Co-Emulation API - Modeling Interface

UT or UTC - Untimed or untimed C model

1.5 References

1. **Functional Specification for SystemC 2.0** - *Synopsys, Inc.; CoWare, Inc.; Frontier Design, Inc.*
2. **CoWare N2C User Manual** - *CoWare, Inc.*
3. **IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std 1364-1995)** - *IEEE Computer Society*
4. **IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1993)** - *IEEE Computer Society*

1.6 Overview

This specification describes a modeling interface that provides multiple channels of communication that allow software models describing system behavior to connect to structural models describing implementation of a device under test (DUT). Each communication channel is designed to transport untimed *messages* of arbitrary abstraction between its two end points or “ports” of a channel.

These message channels are not meant to connect software models to each other but rather to connect software proxy models to message port interfaces on the hardware side of the design. The means to interconnect software models to each other must be provided by a software modeling and simulation environment such as SystemC which is beyond the scope of this document.

Although the *software side* of a system can be modeled at several different levels of abstraction including untimed, cycle accurate, and even gate level, the focus of the SCE-MI is to interface purely untimed software models with an RTL or gate level DUT. This can be summarized with the following recommendations regarding the API:

- Do not use it to bridge event based or sub-cycle accurate simulation environments
- It is possible but not ideal to bridge cycle accurate simulation environments
- It is ideal for bridging an untimed simulation environment with a cycle accurate simulation environment

There are many references in the document to SystemC as the modeling environment for untimed software models. This is because, although SystemC is capable of modeling at the cycle accurate RTL abstraction level, it is also considered ideally suited for untimed modeling. As such, it has been chosen for use in many of the examples in this document.

1.6.1 Performance Goals

While the *software side* of the described interface is generic in its ability to be used in any C/C++ modeling environment, it is optimized to be compatible with SystemC (see reference [1]). Similarly, its *hardware side* is optimized to prevent undue throttling of an emulator during a co-modeling session run.

Throughout this document the term *emulation* or *emulator* is used to denote a structural or RTL model of a DUT running in an emulator, rapid prototype, or other simulation environment including software HDL simulators.

That said however, the focus of the design of this interface is to avoid communication bottlenecks that might become most apparent when interfacing software models to an emulator as compared to interfacing them to a slower software HDL simulator or even an HDL accelerator. Such bottlenecks could severely compromise the performance of an emulator which is otherwise very fast. Although some implementations of the interface can be more inefficient than others, there should be nothing in the specification of the interface itself that renders it inherently susceptible to such bottlenecks.

For this reason, the communication channels described herein are *message* or *transaction* oriented rather than *event* oriented with the idea that a single message over a channel originating from a software model can trigger dozens to hundreds of clocked events in the hardware side of the channel. Similarly, it can take thousands of clocked events on the hardware side to generate the content of a message on a channel originating from the hardware that is ultimately destined for an untimed software model.

1.6.2 Document Layout

The document is divided into the following three major chapters:

- Chapter 2.0: An overall description and use model for the SCE Modeling Interface (SCE-MI).
- Chapter 3.0: A tutorial that shows how the use model is used in a simple application.
- Chapter 4.0: A formal functional specification of the API itself.

1.6.3 Document Conventions

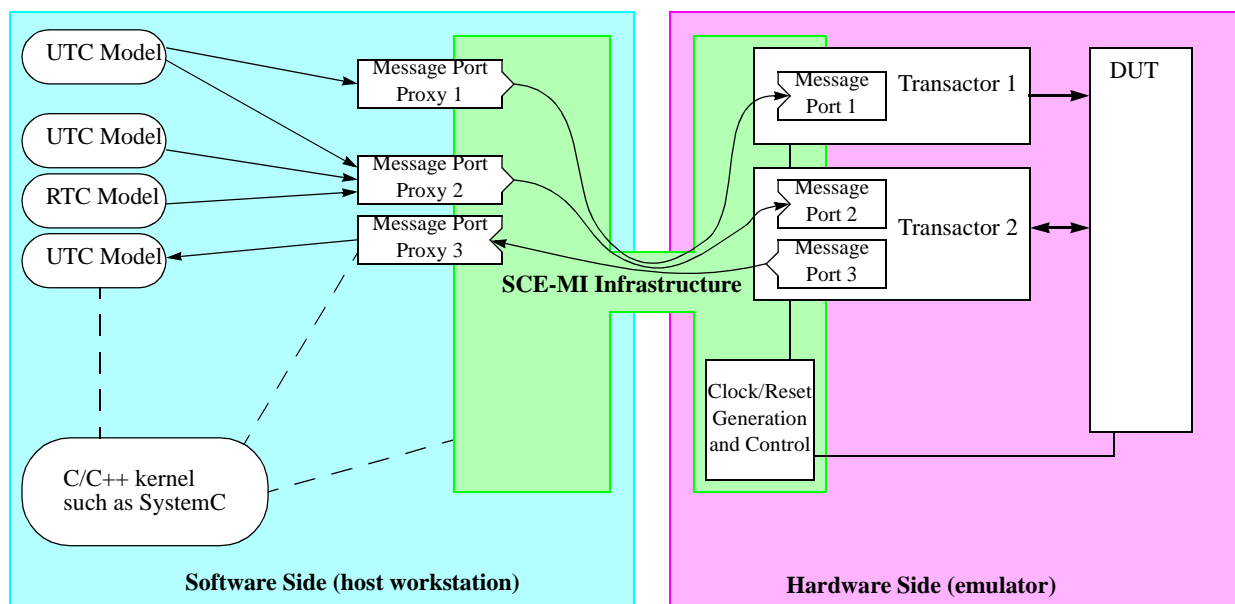
- Any references to actual literal names that might be found in source code, identifiers that are part of the API, file names, and other literal names are represented in `courier` font.
- Key concepts words, or phrases are often *italicized*. Also, there is a good chance that any *italicized* text may have a definition in the glossary, so the reader is directed there for a more detailed meaning of a term.

2.0 Overall Description and Use Model

2.1 High Level Description

The diagram in Figure 1 shows a high level view of how SCE-MI interconnects untimed software models to structural hardware *transactor* and DUT models.

FIGURE 1. High Level View Of Run Time Components



The SCE-MI provides a transport infrastructure between the emulator and host workstation sides of each channel, that interconnects *transactor* models in the emulator to C (untimed or RTL) models on the workstation. Again it is important to note that for purposes of this document, the term emulator can be used interchangeably with any simulator capable of executing RTL or gate-level models, including software HDL simulators. These interconnects are provided in the form of message channels that run between the *software side* and the *hardware side* of the SCE-MI infrastructure. Each message channel has two ends. The end on the software side is called a *message port proxy* which is a C++ object that gives API access to the channel. The end on the hardware side is a *message port* macro that is instantiated inside a transactor and connected to other components in the transactor. Each message channel is either an input or an output channel with respect to the hardware side.

It is important to note that message channels are not unidirectional or bidirectional busses in the sense of hardware signals, but are rather more like network sockets that use message passing protocols. It is the job of the transactors to serve as *abstraction gaskets* that decompose messages arriving on input channels from the software side into sequences of cycle accurate events that are to be clocked into the DUT. For the other direction of flow, transactors recombine sequences of events coming from the DUT back into messages to be sent via output channels to the software side.

In addition, the SCE-MI infrastructure provides clock (and reset) generation and shared clock control using handshake signals with the transactor. This allows the transactor to “freeze” *controlled time* while performing message composition and decomposition operations.

2.1.1 Support for Multi-Threaded Environments

The SCE-MI is designed to couple easily with multi-threaded environments such as SystemC, yet it also functions just as easily in single-threaded environments such as simple C programs. A special *service loop* function (see section 4.3.3.6 on page 68) is provided by the SCE-MI that can be called from an application to give the SCE-MI infrastructure an opportunity to service its communication channels. It is inside this function that queued up input messages can be sent to the hardware side and arriving output messages can be dispatched to the appropriate software models.

While there is no thread specific code inside the service loop function or elsewhere in the SCE-MI, in a multi-threaded environment this function is designed to be called periodically from a dedicated thread so that the interface is automatically serviced while other threads are running.

In a single-threaded environment, the service loop function can be “sprinkled” throughout the application code at strategically placed points to allow it to frequently yield CPU cycles to the SCE-MI infrastructure so that it can service its messages channels.

2.2 Users of the Interface

A major goal of this specification is to address the needs of three target audiences, each with a distinct interest in their use of the interface. The target audiences are as follows:

- End user
- Transactor implementor
- SCE-MI infrastructure implementor

2.2.1 End User

The *end user* is interested in quickly and easily establishing a bridge between a software testbench that can be composed of high level, *untimed*, algorithmic software models, and a hardware DU that can be modeled at the RTL, cycle accurate level of abstraction.

While end users might be aware of the need for a “gasket” that bridges these two levels of abstraction, they want the creation of these *abstraction bridges* to be as painless and automated as possible. Ideally, the end users are not required to be familiar with the details of SCE-MI API.

Rather, on the *hardware side*, they might wish to rely on the *transactor implementor* (see next section) to provide predefined *transactor* models that can directly interface to their DUT. This would remove any requirement for them to be familiar with any of the SCE-MI hardware-side interface macros (see section 4.1 on page 46) except possibly the `SceMiClockPort` macro whose interface is easy to understand because all it really does is furnish a clock and a reset.

Similarly, on the *software side*, the end users can also rely on the transactor implementors to furnish them with *plug-and-play* software models, custom tailored for a software modeling environment such as SystemC. Such models would encapsulate the details of interfacing to the SCE-MI software side and, might present a fully *untimed*, easy to use interface to the rest of the software testbench.

2.2.2 Transactor Implementor

The transactor implementor is familiar with the interface presented by the SCE-MI but is not concerned with its implementation.

The job of the transactor implementor is to provide plug-and-play transactor models on the *hardware side* and proxy models on the *software side* that might be used by *end users* to easily bridge their untimed software models with their RTL-represented DUT. Additionally, the transactor implementor can supply *proxy models* on the software side that provide untimed “sockets” to the transactors.

Using the models is like using any other vendor supplied, stand-alone IP models and the details of bridging not only two different abstraction levels but possibly two different verification platforms (such as SystemC and an emulator), is completely hidden within the implementations of the models. They must be distributed with appropriate object code, netlists, RTL code, configuration files, and all necessary documentation.

2.2.3 SCE-MI Infrastructure Implementor

The SCE-MI infrastructure implementor is interested in furnishing a working implementation of an SCE-MI that runs on some vendor supplied verification platform (probably their own). Any distributed product release from the infrastructure implementor will necessarily include both the *software side* and the *hardware side* components of the SCE-MI. For such a release to be compliant, it must conform to all the requirements set forth in this specification.

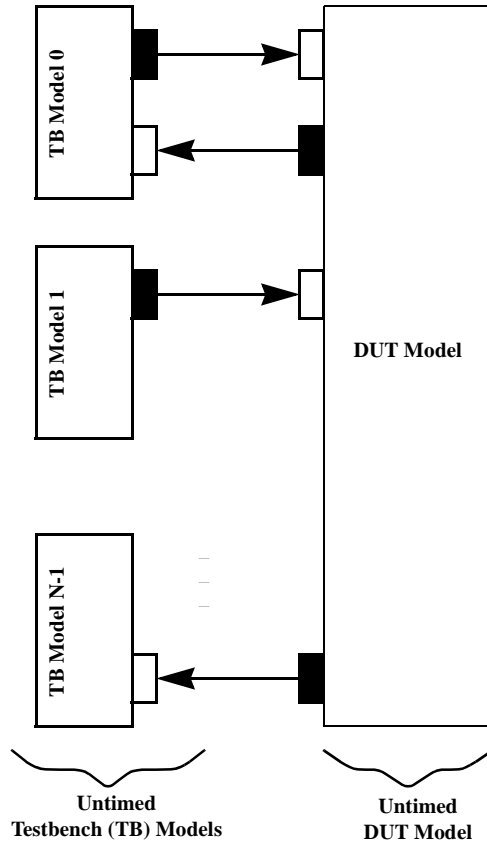
2.3 Bridging Levels of Modeling Abstraction

The central goal of this specification is to provide an interface designed to bridge two modeling environments, each of which supports a different level of modeling abstraction.

2.3.1 Untimed Software Level Modeling Abstraction

Suppose we have a testbench consisting of several, possibly independent models that stimulate and respond to a DUT at different interface points. Figure 2 depicts such a system. Such a configuration can be used to test a processor DUT that has some communications interfaces that can include an ethernet adapter, PCI interface, and a USB interface. The testbench can consist of several models that independently interact with these interfaces, playing their protocols and exchanging packets with them. These packets might be recoded as *messages* with the intent of verifying the processor DUT’s ability to deal with them. The system in Figure 2 might initially be implemented fully at the *untimed* level of abstraction using the SystemC software modeling environment.

FIGURE 2. Untimed Software Testbench and DUT Models



Now suppose that we ultimately want to create cycle accurate RTL models of the DUT model and eventually synthesize it to gates that can be verified on a high speed emulation platform. Once we do this however, we might still want to test them with the unaltered, untimed testbench models. Doing so requires a way of somehow bridging the untimed level of abstraction to the *bus cycle accurate (BCA)* level.

2.3.2 Cycle Accurate Hardware Level Modeling Abstraction

Suppose now that we take the purely untimed system shown in Figure 2, “pry apart” the direct coupling between the testbench models and the untimed DUT model, and insert an *abstraction bridge* in order to bridge the still untimed system testbench model to what is now an emulator resident, RTL-represented DUT. Figure 3 depicts this new configuration.

This bridge consists of a set of *DUT proxy* models, *SCE-MI message input and output port proxies*, a set of *message channels* which are transaction conduits between the software simulator and the emulator, *message input and output ports*, and a set of user implemented *transactors*.

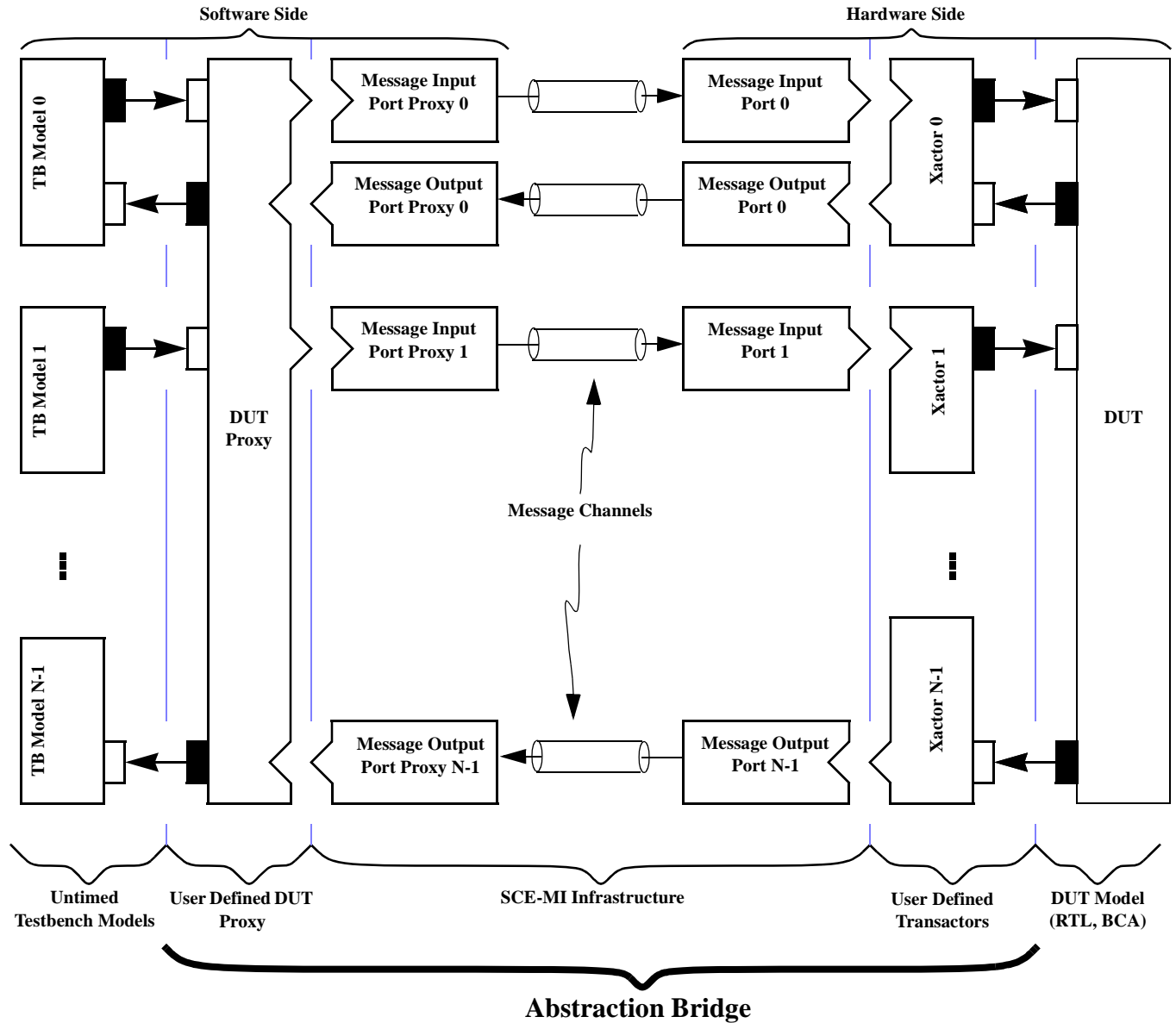
The SCE-MI infrastructure performs the task of serving as a transport layer that guarantees delivery of *messages* back and forth between the *message port proxy* and *message port ends* of each channel. Messages arriving on input channels are presented to the transactors through *message input ports*. Similarly, messages

arriving on output channels are dispatched to the *DUT proxy* software models via *message output port proxies* that present them to the rest of the testbench as if they had come directly from original untimed DUT model as depicted in Figure 2. In fact, the testbench models do not know that the messages have actually come from and gone to a totally different abstraction level.

The DUT input proxies have the job of accepting untimed messages from various C models and sending them to the message input port proxies for transport to the hardware side. The DUT output proxies establish callbacks that monitor the message output port proxies for arrival of messages from the hardware side. In other words, the SCE-MI infrastructure *dispatches* these messages to the specific DUT proxy models to which they are addressed.

Taking this discussion back to the context of users of the interface described in section 2.2 on page 7, the *end user* only has to know how to interface with the DUT proxy models on the software side of Figure 3 with the transactor models on the hardware side; whereas, the *transactor implementor* authors the proxy and transactor models using the SCE-MI message port and clock control components between them, and provides those models to the end user.

FIGURE 3. Multichannel Abstraction Bridge Architecture



2.3.3 Messages and Transactions

At this point, it makes sense to describe in more detail what a *message* is and how it is used in an untimed software modeling environment. In the discussion above, the implied meaning of a message is that it is a unit of data of arbitrary size and abstraction to be transported over a channel.

In a purely untimed modeling environment, messages are not associated with specific clocks or events. Rather, they can be considered arbitrary data types ranging in abstraction from a simple bit, boolean or integer, on up to something as complex as a C++ class or even some aggregate of objects.

It is in this form that messages can be transported either *by value* or *by reference* over abstract ports between fully untimed software models of the sort described in Figure 2 and, in substantially more detail in reference [1].

However, before messages can be transported over an SCE-MI message channel, their content must be serialized into a large bit vector by the DUT proxy model. Conversely, after a message arrives on a message output channel and is dispatched to a DUT output proxy model, it might be *de-serialized* back into an abstract C++ data type. At this point, it is ready for presentation at the abstract output ports (to use SystemC terminology) of the DUT proxy to the connected software testbench models.

Meanwhile, on the hardware side, a message arriving on the message input channel can trigger many dozens to hundreds of clocks of event activity. It is the job of the transactor to perform this decomposition of message data content to sequences of clocked events that are presented to the DUT hardware model inputs. Conversely, for output messages, it might be the job of the transactor to accept possibly hundreds to thousands of clocked events originating from the DUT hardware model, and assemble them into serialized bit streams that are sent back to the software side for de-serialization back into abstract data types.

Note that, for the most part, the term *message* can be used interchangeably with *transaction*. However, in some contexts, *transaction* can be thought of as including infrastructure overhead content in addition to user payload data (and handled at a lower layer of the interface) whereas, the term *message* denotes only user payload data.

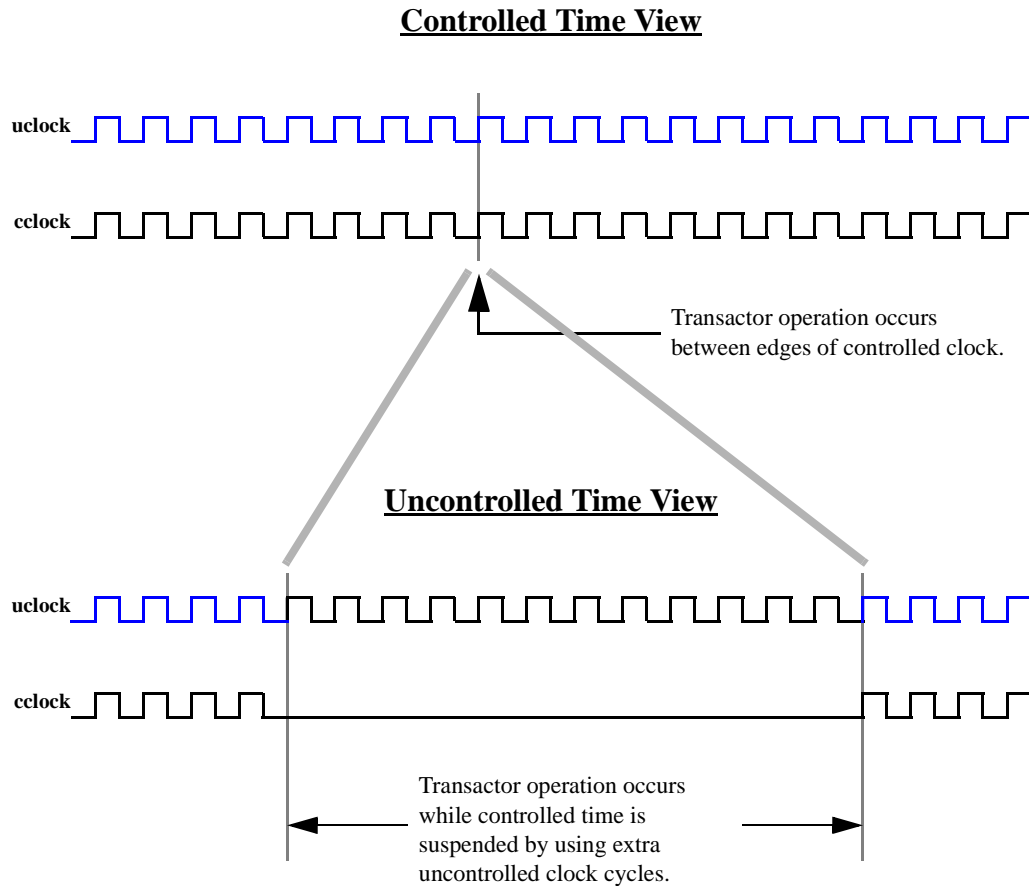
2.3.4 Controlled and Uncontrolled Time

One of the implications of the transactor's job of converting between message bit streams and clocked events is that it might need the ability to "freeze" controlled time while performing these operations such that the *controlled clock* that feeds the DUT is stopped long enough for the operations to occur.

If one visualizes the transactor operations strictly in terms of controlled clock cycles, they appear between edges of the controlled clock. This is depicted in the controlled time view shown in Figure 4. But if they are shown for all cycles of the *uncontrolled clock*, the waveforms would appear more like the *uncontrolled time* view shown in Figure 4. In this view, the controlled clock is suspended or disabled and the DUT is "frozen in controlled time."

Suppose a system has multiple controlled clocks (of possibly differing frequencies) and multiple transactors controlling them. Any one of these transactors has the option of stopping any clock. If this happens, all controlled clocks in the system stop in unison. Furthermore, all other transactors that did not themselves stop the clock, must still sense that the clocks were globally stopped and continue to function correctly even though they themselves had no need to stop the clock. In this case, they might typically idle for the number of uclocks during which the cclocks are stopped as illustrated in Figure 4.

FIGURE 4. Controlled and Uncontrolled Time Views



In the SCE-MI use model, the semantics of clock control can be described as follows:

- Any transactor can instruct the SCE-MI infrastructure to stop the controlled clock and thus cause controlled time to freeze.
- All transactors are told by the SCE-MI infrastructure when the controlled clock is stopped.
- Any transactor must function correctly if controlled time is stopped due to operations of another transactor, even if the transactor in question does not itself need to stop the clock.
- A transactor might need to stop the controlled clock when performing operations that involve decomposition or composition of transactions arriving from, or going to a message channel.
- The DUT is always clocked by one or more controlled clocks that are controlled by one or more transactors.
- All transactors are clocked by a free running uncontrolled clock that is provided by the SCE-MI hardware side infrastructure.

2.4 Work Flow

There are four major aspects of work flow involved in constructing a system verification with the SCE-MI environment. They are as follows:

- Software model compilation
- Infrastructure linkage
- Hardware model elaboration
- Software model construction and binding

2.4.1 Software Model Compilation

The models to be run on the workstation are compiled using a common C/C++ compiler or possibly obtained from other sources such as third party vendors in the form of IP, ISS simulators, etc. The compiled models are linked with the software side of the SCE-MI infrastructure to form an executable program.

2.4.2 Infrastructure Linkage

Infrastructure linkage is the process that reads a user description of the hardware, namely the source or *bridge* netlist that describes the interconnect between the transactors, the DUT, and the SCE-MI interface components, and compiles that netlist into a form suitable for emulation. Part of this compile process can involve adding additional structure to the bridge netlist, that properly interfaces the user supplied netlist to the SCE-MI infrastructure implementation components. Put more simply, the infrastructure linker is responsible for providing the core of the SCE-MI interface macros on the hardware side.

As part of this process, the infrastructure linker also looks at the parameters specified on the instantiated interface macros in the user-supplied bridge netlist and uses them to properly establish the dimensions of the interface. Among the parameters it analyzes are the following:

- Number of transactors
- Number of input and output channels
- Width of each channel
- Number of clocks
- Clock ratios
- Clock duty cycles

Once the final netlist is created, the infrastructure linker can then appropriately compile it for the emulation platform and accordingly convert it to a form suitable to run on the emulator.

2.4.3 Hardware Model Elaboration

The compiled netlist is downloaded to the emulator, elaborated, and prepared for binding to the software.

2.4.4 Software Model Construction and Binding

The software executable compiled and linked in the software compilation phase is now executed which causes all the software models to be constructed in the workstation process image space. Once construction takes place, the software models bind themselves to the message port proxies using special calls provided in the API. Parameters passed to these calls establish a means by which specific message

port proxies can *rendezvous* with its associated message port macro in the hardware. Once this binding occurs, the co-modeling session can proceed.

2.5 SCE-MI Interface Components

The SCE-MI run time environment consists of a set of interface components on both the *hardware side* and the *software side* of the interface, each of which provides a distinct level of functionality. They are introduced in this section, and each is discussed in more detail later in this document.

2.5.1 Hardware Side Interface Components

The interface components presented by the SCE-MI *hardware side* consist of a small set of macros that provide connection points between the transactors and the SCE-MI infrastructure. These compactly defined and simple to use macros fully present all necessary aspects of the interface to the transactors and the DUT.

From the point of view of this specification, these macros are simply represented as empty Verilog or VHDL models with clearly defined port and parameter interfaces. This is analogous to a software API specification that defines function prototypes of the API calls without showing their implementations. Future implementations of the interface can also accommodate modeling hardware in RTL C (a.k.a. RTC) using a C modeling language like SystemC, rather than RTL Verilog or VHDL. Such implementations would need to provide the RTC equivalents of the same macros defined in this specification.

Briefly stated, the four macros present the following interfaces to the transactors and DUT:

- Message input port interface
- Message output port interface
- Controlled clock and controlled reset generator interface
- Uncontrolled clock, uncontrolled reset, clock control logic interfaces

2.5.2 Software Side Interface Components

The interface presented by SCE-MI infrastructure to the software side consists of a set of C++ objects and methods which provide the following functionality:

- Version discovery
- Parameter access
- Initialization and shutdown
- Message input and output port proxy binding and callback registration
- Rendezvous operations with the hardware side
- Infrastructure service loop polling function
- Message input send function
- Message output receive callback dispatching
- Message input ready callback dispatching
- Error handling

In addition to the C++ object oriented interface, a set of C API functions is also provided for the benefit of pure C applications.

3.0 User's Guide and Tutorial

3.1 Hardware Side Interfacing

The hardware side interface of the SCE-MI consists of a set of parametrized macros which can be instantiated inside transactors that are to interact with the SCE-MI infrastructure. The macros are parametrized such that, at the point of instantiation, crucial parameters that will dictate the dimensions of the SCE-MI bridge to software can be easily specified by the user. It is the job of the *infrastructure linker* to learn the values of these parameters and customize implementation components inserted underneath the macros accordingly.

The following four macros fully characterize how the hardware side interface of the SCE-MI is presented to the transactors and the DUT:

- `SceMiMessageInPort` macro
- `SceMiMessageOutPort` macro
- `SceMiClockControl` macro
- `SceMiClockPort` macro

Any number of these macros can be instantiated as is required by the system. One `SceMiMessageInPort` macro must be instantiated for each required message input channel, and one `SceMiMessageOutPort` macro for each output channel. Message port macro bit-widths are parametrized at the point of instantiation.

Exactly one `SceMiClockPort` macro is instantiated for each defined clock in the system. This `SceMiClockPort` macro instance will, via a set of parameters, fully characterize a particular clock. The `SceMiClockPort` macro is instantiated at the top level and provides a controlled clock and reset directly to the DUT. The `SceMiClockPort` macro instance is named and assigned a reference `ClockNum` parameter that is used to associate it with one or more counterpart `SceMiClockControl` macros inside one or more transactors. The `SceMiClockControl` macro is used by its transactor for all clock controlling operations for its associated clock. These two macros are mutually associated by the `ClockNum` parameter and every `SceMiClockPort` macro must have a minimum of one `SceMiClockControl` macro associated with it.

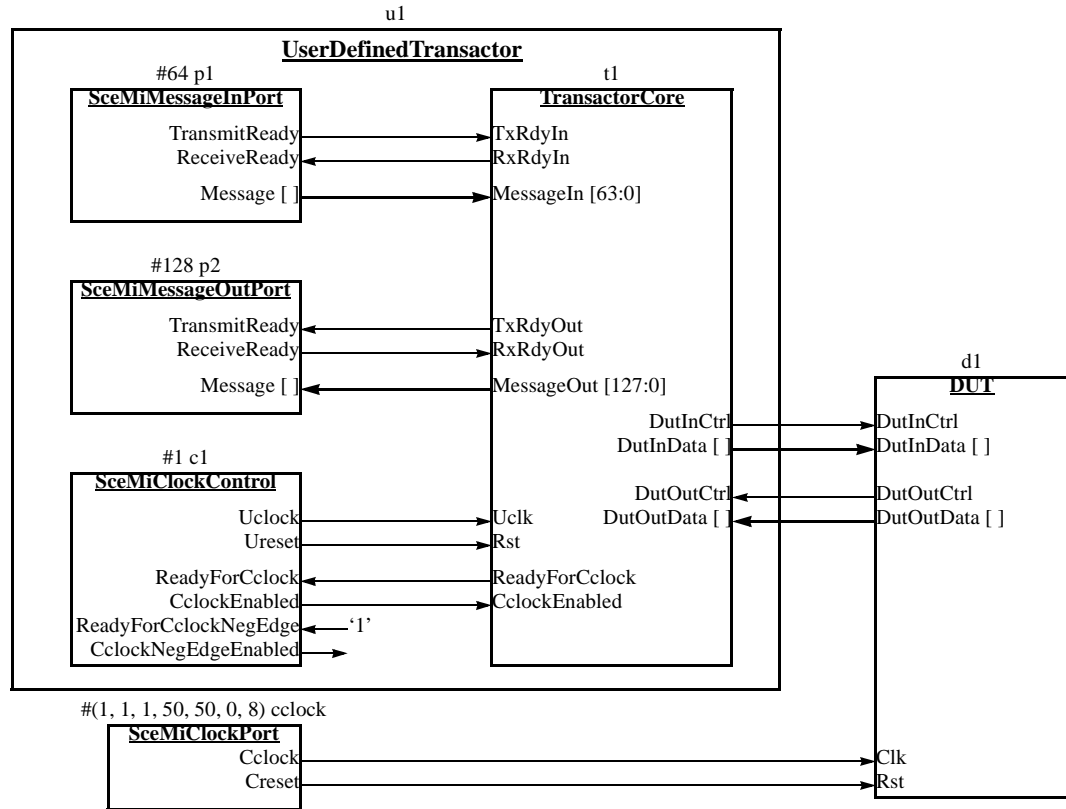
The infrastructure linker (not the user) is responsible for properly hooking up these, essentially empty, macro instances to the internally generated SCE-MI infrastructure and clock generation circuitry.

The following parameters specified at the points of instantiation of the macros fully specify the required dimensions of the SCE-MI infrastructure components (see 4.2.1 on page 59 for more details):

- Number of transactors
- Number of input and output channels
- Name and width of each channel
- Number of controlled clocks
- Name, clock ratio, and duty cycle of each controlled clock

Figure 5 shows a simple example of how a transactor and DUT might connect to the hardware side interface of the SCE-MI.

FIGURE 5. Connection of SCE-MI Macros on Hardware Side to Transactor and DUT



This example features a single transactor interacting with a DUT and interfacing to the software side through a `SceMiMessageInPort` and a `SceMiMessageOutPort`. In addition, it defines a single clock that is controlled by the transactor internally using the `SceMiClockControl` macro. This clock drives the DUT from the top level through a `SceMiClockPort` macro.

A key point that this example illustrates is that only the *transactor implementor* (see 2.2 on page 7) needs to be aware of all SCE-MI interface macros (except for the `SceMiClockPort`). Because the transactor encapsulates the message port macros and the `SceMiClockControl` macro, the *end user* only has to be aware of how to hook up to the transactor itself and to the `SceMiClockPort` macro.

The `SceMiClockPort` macro instantiation is where all the clock parameters are specified. The numbers shown (see Figure 5) in the component instantiation label as,

```
#(1, 1, 1, 50, 50, 0, 8) cclock
```

map to the parameters defined for the `SceMiClockPort` macro (see 4.1.4 on page 51). They are summarized here:

- `ClockNum = 1`
- `RatioNumerator = 1`
- `RatioDenominator = 1`

- `DutyHi = 50`
- `DutyLo = 50`
- `Phase = 0`
- `ResetCycles = 8`

Of these parameters, the `clockNum` parameter is used to uniquely identify this particular clock and also to associate it with its 1 or more counterpart `SceMiClockControl` macros which must be parametrized to the same `clockNum` value, in this case 1. In addition to learning the clock specification parameters, the *infrastructure linker* also learns the name of each clock by looking at the instance label of each `SceMiClockPort` instance, in this case `cclock`.

Similarly, message ports have a parametrized `PortWidth` parameter.

To summarize, the *infrastructure linker* learns the following specific information from analyzing this netlist:

- Single transactor called “Bridge.u1” (assuming top level module is called “Bridge”).
- Single “divide-by-1” controlled clock called “cclock”
- Controlled clock has a 1/1 ratio which, when enabled, is ideally (depending on implementation) the same frequency as the uncontrolled clock
- Controlled clock is parametrized to 50/50 duty cycle with 0 phase shift (a user can also specify a *don't care duty cycle* - see 4.1.4.1 on page 52 for details)
- Controlled reset is parametrized to 8 controlled clock cycles of reset
- Single `SceMiMessageInPort` called “p1”, parametrized to bit-width of 64
- Single `SceMiMessageOutPort` called “p2” parametrized to bit-width of 128

A more complicated example that involves two transactors and three clocks is shown in Appendix B: on page 90.

3.2 The Routed Tutorial

The `Routed` tutorial documents a real life example that uses the SCE-MI to interface between untimed software models modeled in SystemC, and hardware models of transactors and a DUT modeled in RTL Verilog. This tutorial tries to illustrate how the use model of the SCE-MI can be applied in a multi-threaded SystemC environment. It assumes some familiarity with the concepts of SystemC including *abstract ports*, *autonomous threads*, *slave threads*, module and port definition, and module instantiation and interconnect. Reference [1] has a description of these concepts.

3.2.1 What the Design Does

The `Routed` design is a small design that simulates air passengers traveling from `Origins` to `Destinations` by traversing various interconnected `Pipes` and `Hubs` in a `RouteMap`. In this design, the `Origins` and `Destinations` are the *transactors* and the `RouteMap` model is the *DUT*. Each `Origin` transactor interfaces to a `SceMiMessageInPort` to gain access to messages arriving from the software side. Each `Destination` transactor interfaces to a `SceMiMessageOutPort` to send messages to the software side. There is also an `OrigDest` module that has both an `Origin` and `Destination` transactor contained within it.

The “world” consists of these Origins,

Anchorage, Cupertino, Noida, SealBeach, UK, Waltham,

and these DestinationS,

Anchorage, Cupertino, Maui, SealBeach, UK.

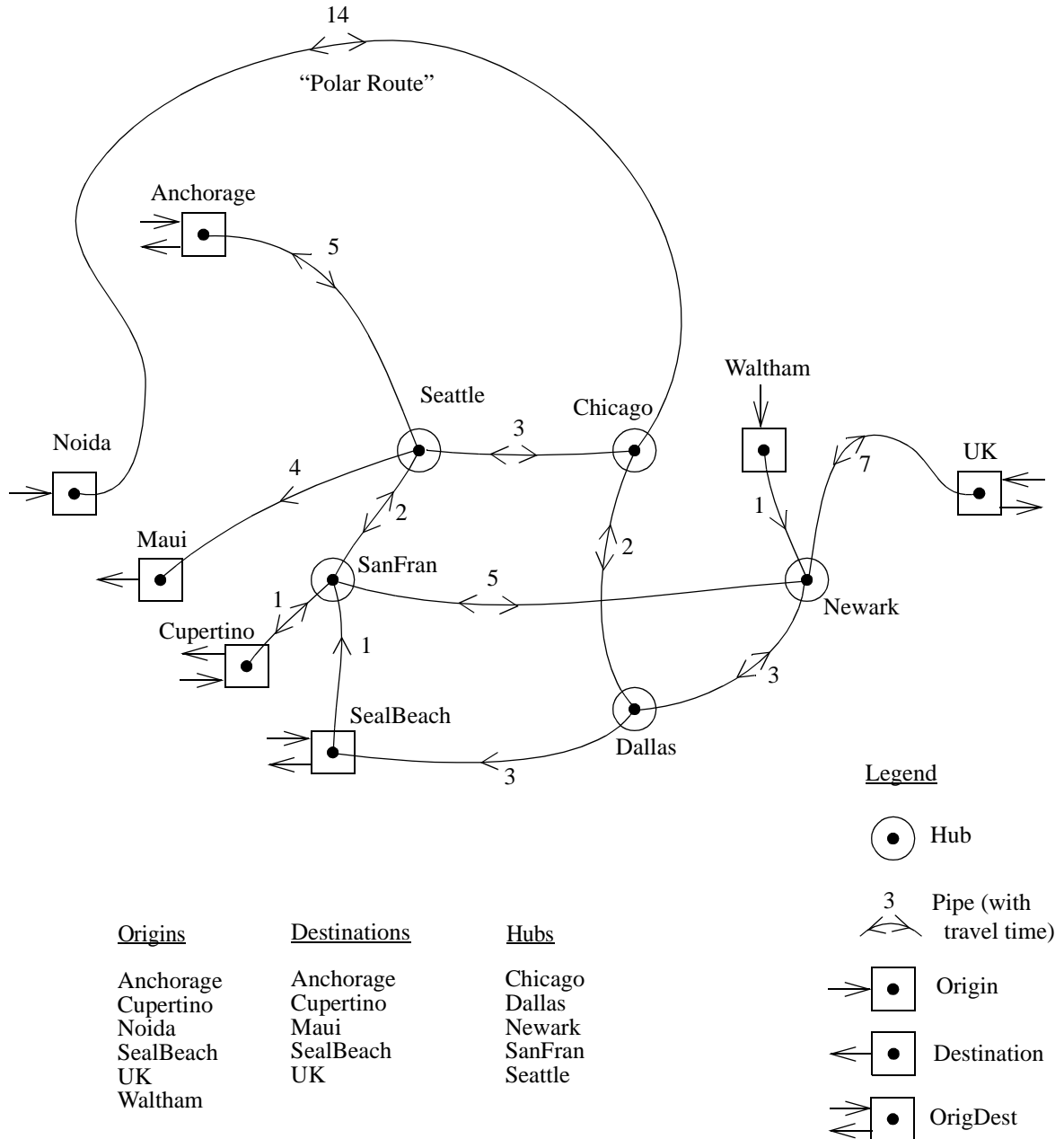
Travel from any Origin to any Destination is possible by traversing the RouteMap (DUT) containing the following Pipe interconnected Hubs,

Chicago, Dallas, Newark, SanFran, Seattle.

Each controlled clock cycle represents one hour of travel or layover time.

Figure 6 shows how the Routed world is interconnected.

FIGURE 6. The Routed World



The numbers shown by the directed arcs are the travel time (in hours) to travel the indicated Pipe. Layover time in each Hub is two hours.

The RouteMap is initialized by injecting TeachRoute messages for the entire system through the Waltham Origin transactor. Each TeachRoute message contains a piece of routing information

addressed to a particular `Hub` to load the route into its `RouteTable` module (see Figure 9 on page 25). Using this simple mechanism, the software side `RouteConfig` model progressively teaches each `Hub` its routes (via `Waltham`) so that it can, in turn, pass additional `TeachRoute` tokens to `Hubs` more distant from `Waltham`. In other words, by first teaching closer hubs, the `RouteMap` learns to pass routes bound for more distant hubs. This process continues until the entire mesh is initialized, at which point it is ready to serve as a backbone for all air travel activity.

After initiating the route configuration, the testbench then executes the itineraries of 4 passengers over a period of 22 days. Each itinerary consists of several legs, each with scheduled departure from a specified `Origin` and each with a specified `Destination`. The scheduled leg is sent as a message token to its designated `Origin` transactor. It is the job of the transactor to count the number of clocks until the specified departure time before sending the token into the `RouteMap` mesh.

3.2.2 System Hierarchy

The hierarchy of the whole system is textually shown in the following subsections.

3.2.2.1 Software Side Hierarchy

```
System
Testbench
Calendar <--> ClockAdvancer
Scheduler <--> OrigDest, Origin, Destination
RouteConfig
SceMiDispatcher
```

Note that the interactions shown between the `Calendar` and `Scheduler` software side models and the `OrigDest`, `Origin`, and `Destination` hardware side models. These interactions occur over *SCE-MI message channels*.

3.2.2.2 Hardware Side Hierarchy

Following is the hierarchy of the hardware side components instantiated under the `Bridge` netlist:

```
Bridge
  SceMiClockPort

  OrigDest anchorage, cupertino, sealBeach, UK
    Origin
      SceMiMessageInPort
      SceMiClockControl
    Destination
      SceMiMessageOutPort
      SceMiClockControl

  Origin noida, waltham

  Destination maui

  RouteMap
    Hub chicagoHub, dallasHub, newarkHub, sanFranHub, seattleHub
    Funnel
    Nozzle
    RouteTable

  Pipe

  ClockAdvancer
    SceMiMessageInPort
    SceMiMessageOutPort
    SceMiClockControl
```

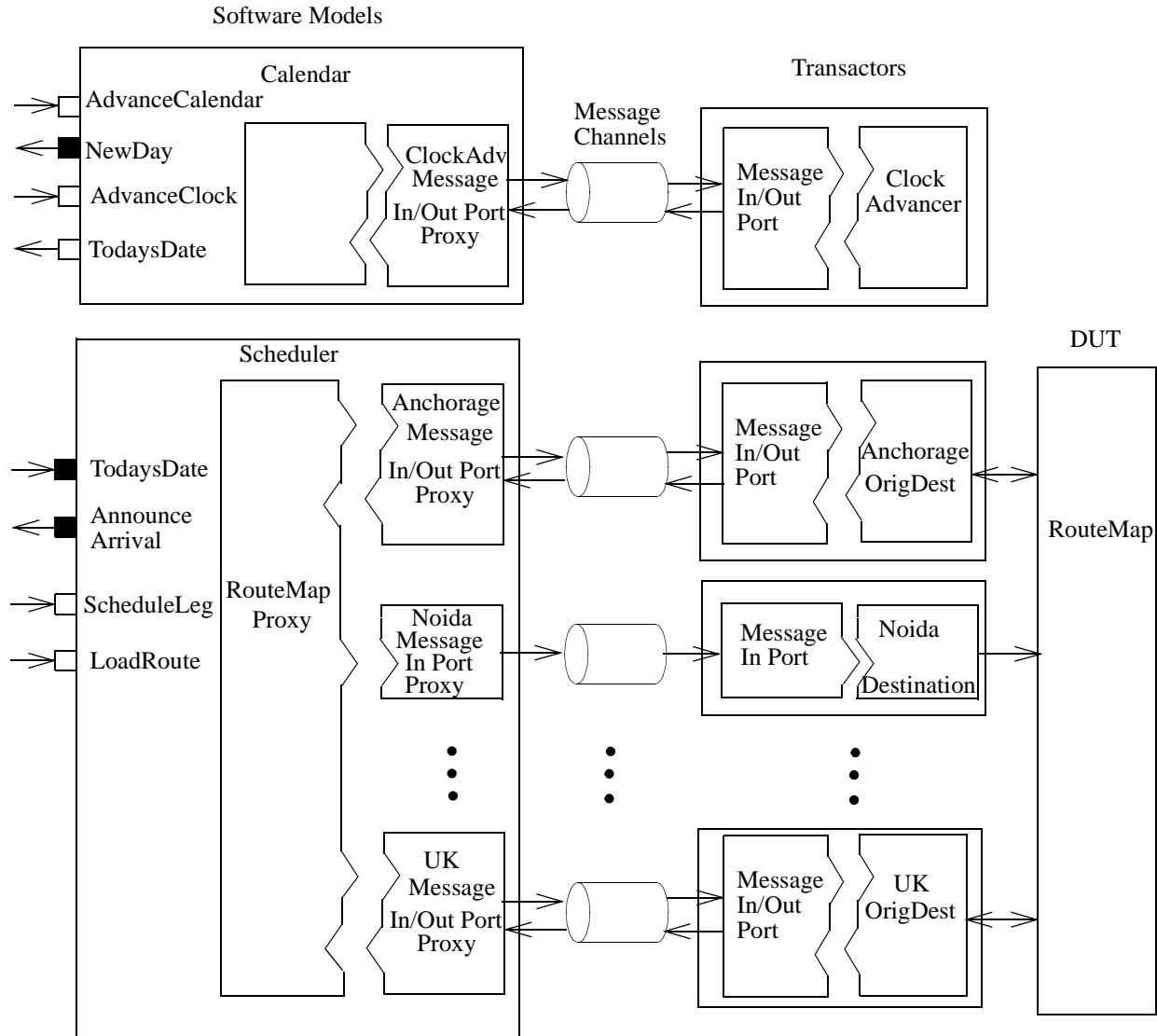
Notice that at the `Bridge` level only the `SceMiClockPort` macro, *transactor* components, and the DUT appear. Encapsulated within the `Origin` and `Destination` transactors are the `SceMiMessageInPort`, `SceMiMessageOutPort`, and `SceMiClockControl` macros. The `ClockAdvancer` transactor has both message input and output ports in addition to the required `SceMiClockControl` macro.

3.2.3 Hardware Side

3.2.3.1 Bridge

The bridge between the hardware and software side of the design is depicted in Figure 7. Notice that this diagram more or less follows the structure of the generalized *abstraction bridge* shown in Figure 3 on page 11. The design uses 13 message channels in all. Two (message input and output) for the `Calendar` <-> `ClockAdvancer` connection, six message input channels for the `Scheduler` <-> `Origin` connections, and five output channels for the `Scheduler` <-> `Destination` connections.

FIGURE 7. The Bridge



The two software models that interact with the hardware side are the `Calendar` model and the `Scheduler` model. These models encapsulate message port proxies which give them direct access to the message channels leading to the `Origin` and `Destination` transactors on the hardware side. These two software models are the only ones that are aware of the SCE-MI link. They converse with the other models through SystemC abstract ports.

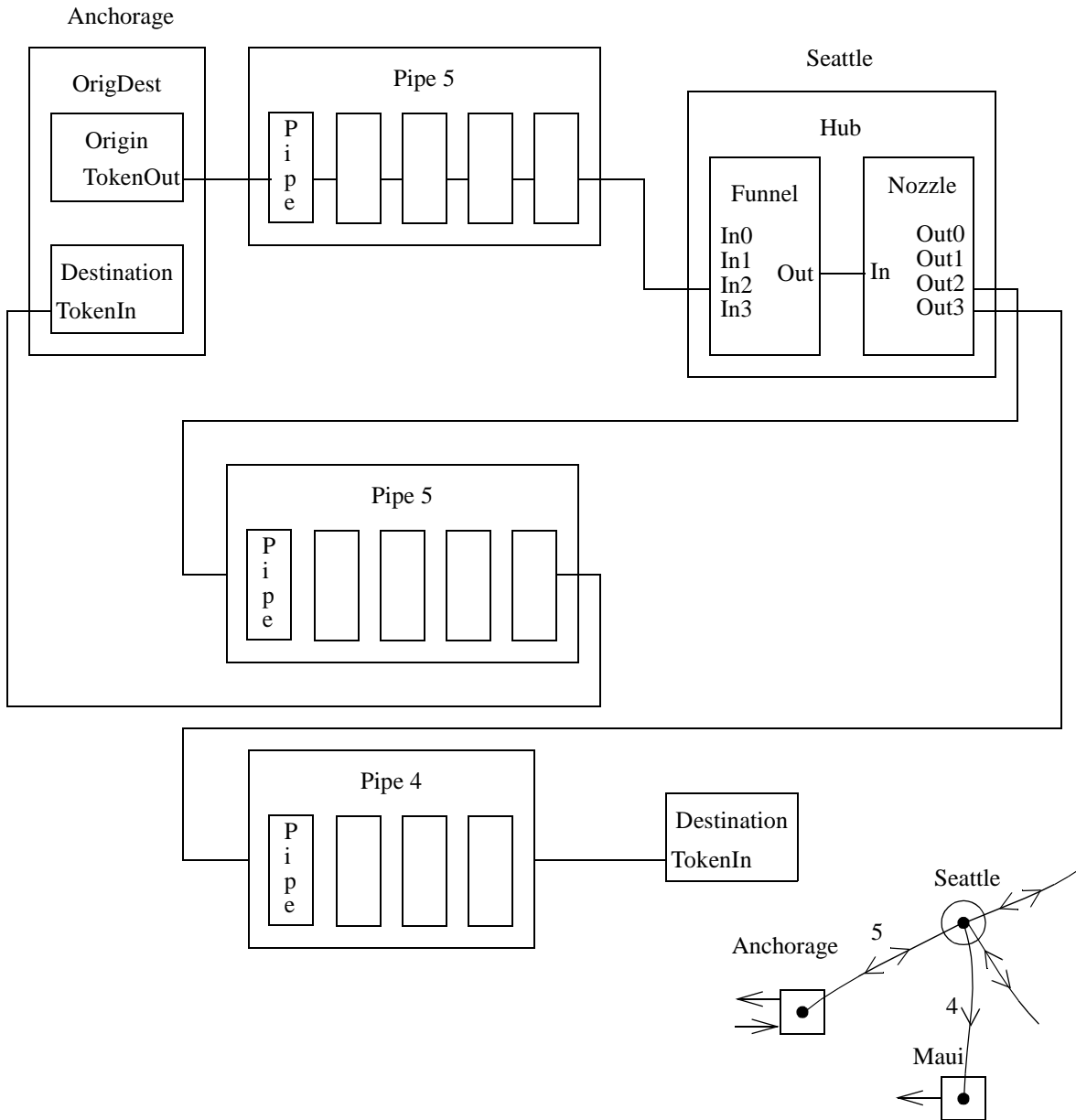
On the hardware side, there is a set of `Origin` and `Destination` transactors that service the message channels that interface with the `Scheduler` and route tokens to or from the DUT. Some locations, such as `Anchorage` and the `UK`, are both `Origin` and `Destination` (called `OrigDest`).

In addition, there is a `ClockAdvancer` transactor that interfaces directly with the `Calendar` model. The `ClockAdvancer` is a stand-alone transactor that does not converse with the DUT. Its only job is to allow time to advance a day at a time (see section 3.2.3.5 on page 28 for more details).

3.2.3.2 DUT and Transactor Interconnect

Figure 8 shows a representative portion of the `RouteMap` to show how it interconnects DUT components to form the `RouteMap` mesh.

FIGURE 8. DUT and Transactor Interconnect

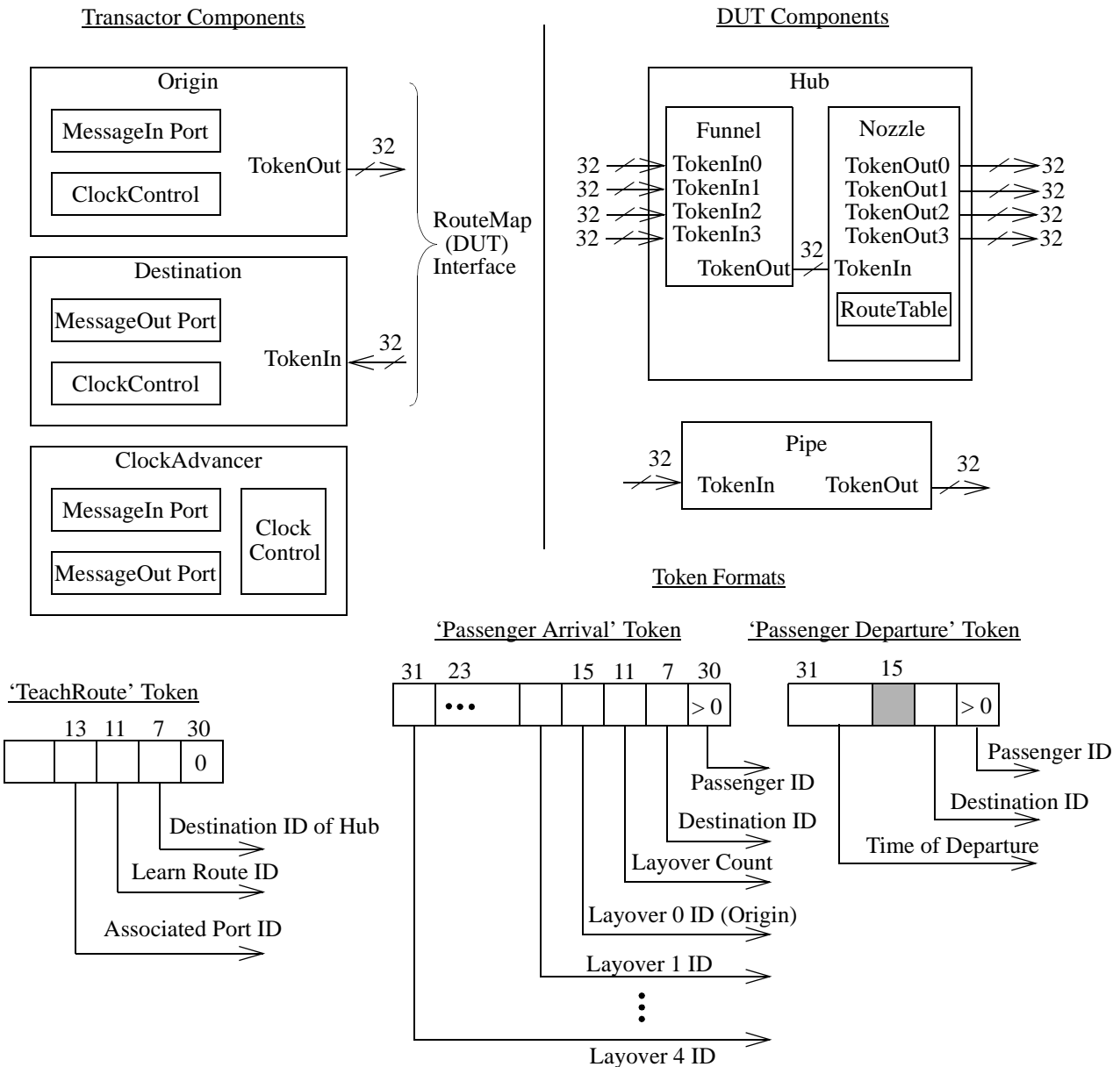


Pipes are inserted between two Hubs or between an Origin or Destination transactor and a Hub. Longer Pipes can be created by cascading primitive one hour Pipes to form the proper length. Each Pipe primitive represents one hour of travel (one clock). In this diagram, a Pipe4 model is inserted between the Seattle Hub and Maui Destination for a four hour flight leg. Since travel can occur in either direction between Anchorage and Seattle, a Pipe5 is inserted between them for each direction.

3.2.3.3 DUT and Transactor Components

Figure 9 shows the structure of the DUT and transactor components.

FIGURE 9. DUT and Transactor Components



Each `Origin` transactor contains a clock control macro and a message input port macro to receive departure tokens from the `Scheduler` on the software side. Each received token is passed to the `TokenOut` port when the scheduled departure time has matured. Although the `Origin` transactor has a clock control macro, it does not actively control the clock. Its only use of the clock control macro is to monitor the `ReadyForCclock` signal to know on which uclocks the cclock is active so that it can properly count cclocks until the scheduled departure time of a pending departure token.

Each `Destination` transactor contains a clock control macro and a message output port macro to send arrival tokens back to the `Scheduler` on the software side. The arrival tokens represent a passenger emerging from the `RouteMap` mesh and arriving at a `Destination` through its `TokenIn` port. See section 3.2.3.4 on page 26 for a detailed description of the `Destination` transactor. This transactor was chosen to be described in detail because it provides a simple example of clock control and message port interfacing.

Each token is a 32-bit vector signal. There are no handshakes in the system. Rather, the tokens are “self announcing.” Normally, 0s are clocked through the mesh so if, on any given cycle, a `Hub` or `Destination` senses a non-0 value on its input port, it knows it has received a token that needs to be processed.

Token formats are also shown in Figure 9. A departure token contains the passenger ID, destination ID, and scheduled time of departure. As the departure token travels through the mesh, it collects layover information consisting of the IDs of all the `Hubs` encountered before reaching its `Destination`, having been transformed into an arrival token. The arrival token then has a complete record of layover information that is passed back to the software side and displayed to the console.

A `Hub` consists of a `Funnel` that accepts tokens from up to four different sources, and a `Nozzle` that routes a token up to four different destinations. The `Nozzle` contains a small `RouteTable` that is initialized at the beginning of the simulation with routing information by receiving `TeachRoute` tokens.

3.2.3.4 The `Destination` Transactor: Interfacing with the DUT and Controlling the Clock

The `Destination` transactor accepts tokens arriving from a point-of-exit on the `RouteMap` and passes them to the message output port.

The `Destination` transactor uses clock control to avoid losing potentially successive tokens arriving from the `RouteMap` (through the `TokenIn` input) to this destination portal. It de-asserts the `readyForCclock` in the event that a token comes in, but the message output port is not able to take it because of tokens simultaneously arriving at other destination portals. This way, it guarantees that the entire `RouteMap` is disabled until all tokens are off-loaded from the requesting `Destination` transactors.

The Verilog source code for the `Destination` transactor is shown in the following listing:

```
module Destination ( TokenIn );
    input [31:0] TokenIn;
// {
    wire [3:0] destID;
    reg [31:0] message;
    reg transmitReady, readyForCclock;
    assign destID = TokenIn[7:4];

    SceMiMessageOutPort #32 messageOutPort(
        //Input                                     Output
        //-----                                     -----
        .TransmitReady(transmitReady),             .ReceiveReady(receiveReady),
        .Message(message) );

    SceMiClockControl clockControl(
        //Input                                     Output
        //-----                                     -----
        .ReadyForCclock(readyForCclock),           .Uclock(uclock),
                                                .Ureset(ureset),
                                                .CclockEnabled(cclockEnabled) );

    always@( posedge uclock ) begin // {
        if( ureset == 1 ) begin
            readyForCclock <= 1;
            message <= 0;
            transmitReady <= 0;
        end
        else begin // {
            // if( DUT clock has been disabled )
            //     It means that this destination transactor is waiting to
            //     unload its pending token and does not want to re-enable the
            //     DUT until that token has been off-loaded or else it might
            //     lose arriving tokens in subsequent DUT clocks.
            if( readyForCclock == 0 ) begin

                // When the MessageOutPort portal finally signals acceptance
                // of the token, we can re-enable the DUT clock.
                if( receiveReady ) begin
                    readyForCclock <= 1;
                    transmitReady <= 0;
                end
            end
            else if( cclockEnabled && destID != 0 ) begin
                message <= TokenIn;
                transmitReady <= 1;

                // if( token arrives but portal is not ready )
                //     Stop the assembly line ! (a.k.a. disable the DUT)
                if( receiveReady == 0 )
                    readyForCclock <= 0;
            end
        end // }
    end // }
endmodule // }
```

3.2.3.5 The `ClockAdvancer` Transactor: Controlling Time Advance

The `ClockAdvancer` transactor simply counts controlled clocks until the requested number of cycles has transpired, then sends back a reply transaction.

The Verilog source code for the ClockAdvancer is listed here:

```
module ClockAdvancer();
// {
  wire [31:0] advanceDelta, messageIn, messageOut;
  reg [31:0] cycleCount;

  assign receiveReadyIn = 1;
  assign advanceDelta = messageIn[31:0];
  assign messageOut = 0;

  SceMiMessageInPort #32 messageInPort(
    //Input          Output
    //-----
    .ReceiveReady(receiveReadyIn), .TransmitReady(transmitReadyIn),
    .Message(messageIn) );

  SceMiMessageOutPort #32 messageOutPort(
    //Input          Output
    //-----
    .TransmitReady(transmitReadyOut), .ReceiveReady(receiveReady),
    .Message(messageOut) );

  SceMiClockControl clockControl(
    //Input          Output
    //-----
    .ReadyForCclock(readyForCclock), .Uclock(uclock), .Ureset(ureset),
    .CclockEnabled(cclockEnabled) );

  always @( posedge uclock ) begin // {
    if( ureset ) begin
      transmitReadyOut <= 0;
      cycleCount <= 0;
      readyForCclock <= 0;
    end
    else begin // {
      // Received a clock advance command - Initialize cycle counter.
      if( transmitReadyIn && !transmitReadyOut ) begin
        cycleCount <= advanceDelta;
        readyForCclock <= 1;
      end
      // Decrement cycle count. When count gets down to 1,
      // prepare to send a response that the time has expired.
      if( readyForCclock && cclockEnabled ) begin
        if ( cycleCount == 1 ) begin
          transmitReadyOut <= 1;
          readyForCclock <= 0;
        end
        cycleCount <= cycleCount - 1;
      end
      if( receiveReadyOut == 1 )
        transmitReadyOut <= 0;
    end // }
  end // }
endmodule // }
```

Notice that the SceMiClockControl macro references the same cclock as that in the Destination transactor (i.e., uses the default ClockNum=1). This means that the ClockAdvancer and the Destination

transactor share in the control of the same cclock. In fact there is only one cclock in the entire system that is specified at the default 1/1 ratio.

Notice also that, although the `ClockAdvancer` handshakes with the message output port, the data that it sends is always 0. This is because the only thing that the software side needs from the `ClockAdvancer` is the cycle stamp which is automatically included in each message output response (see 4.3.5.3 on page 75).

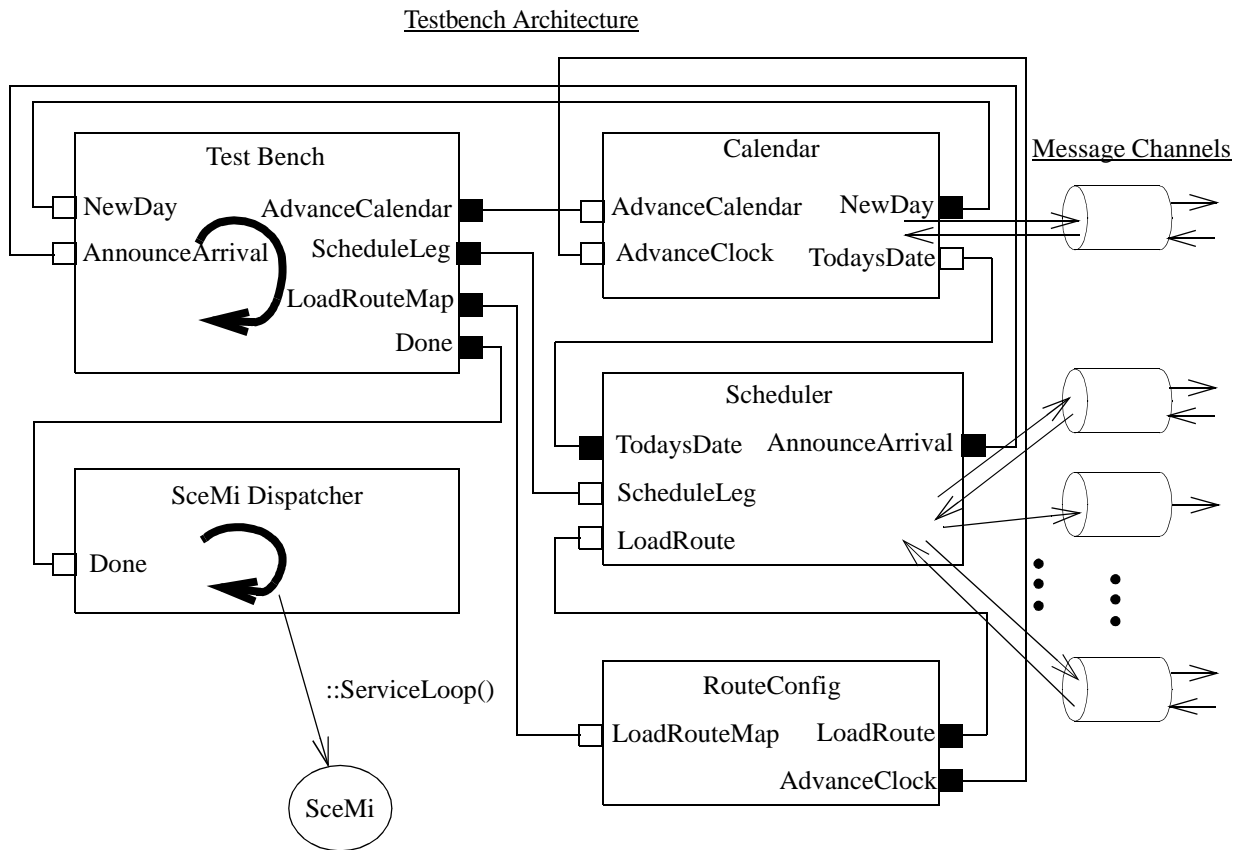
3.2.4 The Software Side

The software side of the Routed design is written completely in SystemC and C++. It is compiled as an executable program that links with the SCE-MI software side.

3.2.4.1 The `System` Model: Interconnect of SystemC Modules

The `System` model is the top level “software netlist” of SystemC modules (`SC_MODULE()`). It specifies the construction and interconnect of the component models as well. A block diagram of the `System` model is shown in Figure 10.

FIGURE 10. Interconnect of SystemC Models



Following is the source code for the System model:

```
SC_MODULE( System ){
    sc_link_mp<unsigned>                newDay;
    sc_link_mp<const Routed::ArrivalRecord *> announceArrival;
    sc_link_mp<unsigned>                advanceCalendar;
    sc_link_mp<const Routed::Itinerary *> scheduleLeg;
    sc_link_mp<>                        loadRouteMap;
    sc_link_mp<>                        done;
    sc_link_mp<>                        advanceClock;
    sc_link_mp<Routed::Date>            todaysDate;
    sc_link_mp<const Routed::Route *>    loadRoute;

    //-----
    // Module declarations
    Testbench      *testbench;
    Calendar        *calendar;
    Scheduler        *scheduler;
    RouteConfig     *routeConfig;
    SceMiDispatcher *dispatcher;

    SC_CTOR( System ){
        testbench = new Testbench( "testbench" );
        testbench->NewDay(          newDay );
        testbench->AnnounceArrival( announceArrival );
        testbench->AdvanceCalendar( advanceCalendar );
        testbench->ScheduleLeg(     scheduleLeg );
        testbench->LoadRouteMap(    loadRouteMap );
        testbench->Done(            done );

        calendar = new Calendar( "calendar" );
        calendar->AdvanceCalendar( advanceCalendar );
        calendar->AdvanceClock(    advanceClock );
        calendar->NewDay(          newDay );
        calendar->TodaysDate(      todaysDate );

        scheduler = new Scheduler( "scheduler" );
        scheduler->TodaysDate(     todaysDate );
        scheduler->ScheduleLeg(    scheduleLeg );
        scheduler->LoadRoute(      loadRoute );
        scheduler->AnnounceArrival( announceArrival );

        routeConfig = new RouteConfig( "routeConfig" );
        routeConfig->LoadRouteMap( loadRouteMap );
        routeConfig->LoadRoute(    loadRoute );
        routeConfig->AdvanceClock( advanceClock );

        dispatcher = new SceMiDispatcher( "dispatcher" );
        dispatcher->Done( done );
    }
};
```

SystemC interconnect channels are declared as `sc_link_mp<>` data types. These can be thought of as abstract signals that interconnect abstract ports. The parametrized data type associated with each `sc_link_mp<>` denotes the data type of the message that the channel is capable of transferring from an output abstract port to an input abstract port. Notice that the `todaysDate` channel is declared with a “by value” data type (i.e., `Routed::Date`) whereas some of the other channels such as the `announceArrival` are declared as “by reference” data types (i.e., `const Routed::ArrivalRecord *`). The former is less

efficient but safer because the message is passed by value and therefore there is no danger of the receiver corrupting the sender's data or worse, having the sender's data go out of scope leaving the receiver with a possibly dangling reference. However, passing messages by reference is more efficient but care must be taken in their use. Declaring them as `const` pointers helps alleviate some, but not all of the safety problems.

Module pointers are declared inside the `SC_MODULE(System)` object and constructed in its `SystemC` constructor (`SC_CTOR(System)`). After each child module is constructed, its abstract ports are mapped to one of the declared interconnect channels.

It is important to stress that `SystemC` channels, while conceptually the same, are distinctly different from `SCE-MI` message channels. Both types of channels pass messages, but `SystemC` channels are designed strictly to pass messages of arbitrary `C++` data types between `SystemC` modules. An entire simulation can be built of just software models communicating with each other. See reference [1] for more details about `SystemC` interconnect channels.

`SCE-MI` message channels have a completely different interface and are optimized for implementing abstraction bridges between a software subsystem and a hardware subsystem. In the use model presented in this example, their interfaces are encapsulated by `SystemC` models.

The thick round arrows in Figure 10 represent the `SystemC` *autonomous threads* contained in the `Testbench` and `SceMiDispatcher` modules. These two threads are the only autonomous threads in the system. All the other code is executed inside *slave threads*.

3.2.4.2 The `sc_main()` Routine and Error Handler

The following listing shows the `sc_main()` routine which is the top level entrypoint to the program. The `sc_main()` is required when linking to a SystemC kernel facility, but it is very much like a conventional `main()` C or C++ entrypoint and has the same program argument passing semantics.

```
static void errorHandler( void /*context*/, SceMiEC *ec ) {
    char buf[32];

    sprintf( buf, "%d", (int)ec->Type );
    string messageText( "SCE-MI Error[" );
    messageText += buf;
    messageText += "]: Function: ";
    messageText += ec->Culprit;
    messageText += "\n";
    messageText += ec->Message;
    throw messageText;
}

int sc_main( int argc, char *argv[] ){
    SceMi::RegisterErrorHandler( errorHandler, NULL );
    SceMi *scemi = NULL;

    try {
        SceMiParameters parameters( "./params.scemi" );
        scemi = SceMi::Init( 1, &parameters );

        System system( "system" ); // Instantiate the system.

        //-----
        // Establish proper bindings between the SCE-MI and the modules
        // that directly interact with it.
        system.dispatcher->Bind( scemi );
        system.calendar ->Bind( scemi );
        system.scheduler ->Bind( scemi );

        //-----
        // Kick off SystemC kernel ...
        cerr << "Let 'er rip !" << endl;
        sc_start(-1);
    }

    catch( string message ) {
        cerr << message << endl;
        cerr << "Fatal Error: Program aborting." << endl;
        SceMi::ShutDown( scemi );
        return -1;
    }
    catch(...) {
        cerr << "Error: Unclassified exception." << endl;
        cerr << "Fatal Error: Program aborting." << endl;
        SceMi::ShutDown( scemi );
        return -1;
    }
    return 0;
}
```

The first routine defined is the `errorHandler()`. This is the master error handling function that is registered with the SCE-MI. Whenever an error occurs, this function is called to format the message

before throwing a C++ exception. The exceptions are caught in the `catch { ... }` blocks at the end of the `sc_main()` routine where they are displayed before exiting the program.

Once the error handler is registered, the SCE-MI is initialized by calling `SceMi::Init()`. This method returns a pointer to an `SceMi` object that manages the whole SCE-MI software side infrastructure.

Next the `System` model described in the previous section is constructed. The constructor (`SC_CTOR(System)`) causes all of its child software models to get constructed by calling, in turn, their `SC_CTOR()` constructors.

Once the whole system is statically constructed, models that interface with SCE-MI are given the master `SceMi` object pointer so that they can access its methods, by calling special `::Bind()` accessor methods on those models.

Finally, the SystemC main kernel loop is initialized by calling the `sc_start()` function. The `-1` parameter tells it to go indefinitely until the program decides to end. How the program ends is explained in the next section.

3.2.4.3 The `SceMiDispatcher` Module: Interfacing with the SCE-MI Service Loop

The `SceMiDispatcher` module contains an *autonomous thread* that yields to the SCE-MI infrastructure so that it may service its message port proxies, by making repeated calls to the `SceMi::ServiceLoop()` method (see section 4.3.3.6 on page 68). By placing this logic on its own dedicated thread, other models in the system do not have to worry about yielding to the SCE-MI.

Following is the source code for the `ScemiDispatcher`:

```
SC_MODULE( ScemiDispatcher ){
    sc_slave<> Done;

    private:
    //-----
    // Context declarations
    Scemi *dScemi;

    //-----
    // Thread declarations
    void dispatchThread(); // Autonomous Scemi dispatcher thread
    void doneThread(){
        Scemi::ShutDown( dScemi );
        exit(0); // This is a normal exit.
    }
    public:
    //-----
    // Accessors
    void Bind( Scemi *scemi ){ dScemi = scemi; };

    SC_CTOR( ScemiDispatcher ){
        //-----
        // Thread bindings
        SC_THREAD( dispatchThread );
        sensitive << UTick;

        SC_SLAVE( doneThread, Done );
    }
};

void ScemiDispatcher::dispatchThread() {
    // This is all the dispatcher does
    // It just calls the Scemi dispatcher poll function and returns.
    for(;;){
        wait();
        dScemi->ServiceLoop();
    }
}
```

Between each call to the service loop, the autonomous thread yields to other threads in the system by calling the `wait()` function. Actually, the only other autonomous thread in the `Routed` system is the one in the `Testbench` model. Both of these threads are represented by the thick round arrows in Figure 10 on page 30.

The other job of the `ScemiDispatcher` is shut down the system when it detects a notification on its `Done` port that the simulation is complete. The `Done` *in*slave port is bound to the *slave thread*, `::doneThread()` on construction. The `Done` port is driven from its associated *out*master port on the `Testbench` module, so it is the `Testbench` that ultimately decides when the simulation is complete (see next section).

3.2.4.4 Application Specific Data Types for the Routed System

The following data types are defined in the `Routed.hxx` header file. They will be referenced throughout the subsequent discussion. They are data types that are specific to this application.

```
class Routed {
public:
    typedef enum Parameters {
        NumPassengers = 4,
        NumLocations = 12,
        MessageSize = 15
    };
    typedef enum PassengerIDs {
        Nobody,
        BugsBunny,
        DaffyDuck,
        ElmerFudd,
        SylvesterTheCat
    };
    typedef enum LocationIDs {
        // Location          Origin  Destination Hub
        // -----          -
        Unspecified,
        Anchorage, // 1: X      X
        Chicago, // 2:
        Cupertino, // 3: X      X
        Dallas, // 4:
        Maui, // 5: X
        Newark, // 6:
        Noida, // 7: X
        SanFran, // 8:
        SealBeach, // 9: X      X
        Seattle, // 10:
        UK, // 11: X      X
        Waltham // 12: X
    };
    typedef struct Itinerary {
        unsigned DateOfTravel;
        unsigned TimeOfDeparture;
        PassengerIDs PassengerID;
        LocationIDs OriginID;
        LocationIDs DestinationID;
    };
    typedef struct ArrivalRecord {
        PassengerIDs PassengerID;
        unsigned DateOfArrival;
        unsigned TimeOfArrival;
        unsigned LayoverCount;
        LocationIDs OriginID;
        LocationIDs LayoverIDs[4];
        LocationIDs DestinationID;
    };
    typedef struct Route {
        LocationIDs RouterID;
        LocationIDs DestinationID;
        unsigned PortID;
    };
    typedef struct Date {
        SceMiU64 CycleStamp;
        unsigned Day;
    };
};
```

```

    };
};

```

3.2.4.5 The Testbench Model: Main Control Loop

The Testbench model contains a SystemC autonomous thread that serves as the main driver for the Routed design. Its job is to look at the four passenger itineraries and schedule the legs in those itineraries on the appropriate dates and at the appropriate departure times by interacting with the Scheduler model.

The condensed source code for the passenger itinerary declarations for the Testbench model is as follows:

```

const Routed::Itinerary Routed::BugsesTrip[] = {
/*
On day,      at,                departs from,      enroute to,      */
{          2,          8, BugsBunny,      Anchorage,      Cupertino },
...
{          20,         10, BugsBunny,      SealBeach,      Maui      },
{          0,          0, BugsBunny,      Unspecified,    Unspecified } };

const Routed::Itinerary Routed::DaffysTrip[] = {
/*
On day,      at,                departs from,      enroute to,      */
{          1,          8, DaffyDuck,      Waltham,        Cupertino },
{          4,          2, DaffyDuck,      Cupertino,      SealBeach },
...
{          22,         7, DaffyDuck,      Cupertino,      Maui      },
{          0,          0, DaffyDuck,      Unspecified,    Unspecified } };

const Routed::Itinerary Routed::ElmersTrip[] = {
/*
On day,      at,                departs from,      enroute to,      */
{          3,          5, ElmerFudd,      SealBeach,      Anchorage },
...
{          23,         3, ElmerFudd,      Cupertino,      Maui      },
{          0,          0, ElmerFudd,      Unspecified,    Unspecified } };

const Routed::Itinerary Routed::SylvestersTrip[] = {
/*
On day,      at,                departs from,      enroute to,      */
{          1,          1, SylvesterTheCat,  Noida,          SealBeach },
{          4,          2, SylvesterTheCat,  SealBeach,      Cupertino },
...
{          20,         7, SylvesterTheCat,  Anchorage,      Maui      },
{          0,          0, SylvesterTheCat,  Unspecified,    Unspecified } };

static const char *passengerNames[] = {
    "Nobody", "BugsBunny", "DaffyDuck",
    "ElmerFudd", "SylvesterTheCat" };

static const char *locationNames[] = {
    "Unspecified", "Anchorage", "Chicago",
    "Cupertino", "Dallas", "Maui",
    "Newark", "Noida", "SanFran",
    "SealBeach", "Seattle", "UK", "Waltham" };

```

There are four passengers whose itineraries are given as lists of `Routed::Itinerary` records. Each record represents a leg of that passenger's journey consisting of a date of departure, time of departure, passenger, origin, and destination. The `passengerNames` and `locationNames` are strings use for printing of messages.

Following is the SystemC module definition (`SC_MODULE()`) for the `Testbench` model with its standard SystemC constructor (`SC_CTOR()`):

```
SC_MODULE( Testbench ){
    //-----
    // Abstract port declarations
    sc_master<>          LoadRouteMap;
    sc_master<>          Done;
    sc_outmaster<unsigned> AdvanceCalendar;
    sc_inslave<unsigned>  NewDay;

    sc_outmaster<const Routed::Itinerary *>  ScheduleLeg;
    sc_inslave<const Routed::ArrivalRecord *> AnnounceArrival;

private:
    //-----
    // Context declarations
    unsigned dNumMauiArrivals;
    unsigned dDayNum;
    const Routed::Itinerary *dItineraries[Routed::NumPassengers];

    //-----
    // Thread declarations
    void driverThread(); // Autonomous "master" thread.
    void newDayThread() { dDayNum = NewDay; }
    void announceArrivalThread();

    //-----
    // Helper declarations
public:
    SC_CTOR( Testbench ) : dNumMauiArrivals(0), dDayNum(0) {
        //-----
        // Thread bindings

        // This autonomous thread forms the main body of the Routed driver.
        SC_THREAD( driverThread );
        sensitive << UTick;

        SC_SLAVE( newDayThread, NewDay );
        SC_SLAVE( announceArrivalThread, AnnounceArrival );

        // Initialize itinerary pointers.
        dItineraries[0] = Routed::BugsesTrip;
        dItineraries[1] = Routed::DaffysTrip;
        dItineraries[2] = Routed::ElmersTrip;
        dItineraries[3] = Routed::SylvestersTrip;
    }
};
```

Following is the autonomous thread for the main driver loop:

```
void Testbench::driverThread(){
    LoadRouteMap(); // Signal RouteConfig model to begin configuration RouteMap.
    unsigned dayNum = dDayNum;
    AdvanceCalendar = 1; // Advance to day 1.

    for(;;){
        wait(); // Wait for day to advance (i.e., 'NewDay' arrives.)

        if( dayNum != dDayNum ){
            unsigned date, minDate = 1000;

            // Check itineraries to see if any passengers are
            // traveling today. If so, advance calendar to tomorrow
            // in case next leg of itinerary is tomorrow.
            for( int i=0; i<Routed::NumPassengers; i++ ){
                if( (date=dItineraries[i]->DateOfTravel) ){
                    if( date == dDayNum ){
                        cout << "On day " << setw(2) << dDayNum << " at "
                             << setw(2) << dItineraries[i]->TimeOfDeparture
                             << ":00 hrs, "
                             << passengerNames[dItineraries[i]->PassengerID]
                             << " departs "
                             << locationNames[dItineraries[i]->OriginID]
                             << " enroute to "
                             << locationNames[dItineraries[i]->DestinationID]
                             << endl;

                        ScheduleLeg = dItineraries[i]++;
                        minDate = dDayNum+1;
                    }
                    else if( date < minDate )
                        minDate = date;
                }
            }
            dayNum = dDayNum;
            AdvanceCalendar = minDate - dDayNum;
        }
    }
}
```

Before entering its main loop, the autonomous `::driverThread()` does two things. First, by signaling the `LoadRouteMap` outmaster port, it triggers the `RouteConfig` model to teach all the routes to the `RouteTables` of all the `Hubs` in the `RouteMap`. Each taught route that is injected to the hardware is staggered by one clock. These small one clock advances are achieved when the `RouteConfig` model signals the `AdvanceClock` port on the `Calendar` model. Passenger travel in the `RouteMap` is not possible until all the `Hubs` have been properly programmed with their routes.

Once all the routes have been taught to the `RouteMap`, the `Calendar` is advanced to day one. This will, in turn, cause the `Calendar` model to announce the arrival of day one via the `NewDay` inslave port. Once the day change has been detected, the `::driverThread()` then enters into a loop where it schedules any travel on the itineraries scheduled for the current day. If no travel is scheduled, it advances the `Calendar` to the first day on which travel is scheduled to occur. Legs of each itinerary are scheduled by sending the `Itinerary` record over the `ScheduleLeg` outmaster port to the `Scheduler` model which encodes it into a token and sends it to the hardware.

This operation continues for each leg of each itinerary until all passengers have traveled all legs of their trip and have finally arrived at the `Maui Destination`. This serves as the termination condition that is conveyed to the `SceMiDispatcher` model by signaling the `Done` outmaster port (see code for `::announceArrivalThread()` below). Upon receiving this notification, the `SceMiDispatcher` model gracefully shuts down the SCE-MI and exits the program with a normal exit status.

The `Testbench` model also announces arrivals of passengers at their destinations as they occur. The `::announceArrivalThread()` slave thread detects an arrival by receiving an `ArrivalRecord` on its `AnnounceArrival` inslave port (which was sent from the message output port proxy receive callback in the `Scheduler`). It prints out the arrival information to the console. Following is the source code:

```
void Testbench::announceArrivalThread(){
    const Routed::ArrivalRecord *arrivalRecord = AnnounceArrival;

    cout << "On day " << setw(2) << arrivalRecord->DateOfArrival
         << " at " << setw(2) << arrivalRecord->TimeOfArrival << ":00 hrs,\n"
         << "      " << passengerNames[arrivalRecord->PassengerID]
         << " arrives in " << locationNames[arrivalRecord->DestinationID]
         << " from " << locationNames[arrivalRecord->OriginID]
         << " after layovers in,";

    for( unsigned i=0; i<arrivalRecord->LayoverCount; i++ )
        cout << "\n          "
             << locationNames[arrivalRecord->LayoverIDs[i]];
    cout << endl;
    // Check for termination condition.
    if( arrivalRecord->DestinationID == Routed::Maui &&
        ++dNumMauiArrivals == Routed::NumPassengers ){
        cout << "Everyone has arrived in Maui. We're done. Let's party !"
             << endl;
        Done(); // Signal the dispatcher that the simulation has ended.
    }
}
```


3.2.4.6 The Scheduler Module: Interfacing with Message Port Proxies

Following is the SystemC module definition and constructor for the Scheduler model:

```
SC_MODULE( Scheduler ){
    //-----
    // Abstract port declarations
    sc_inmaster<Routed::Date>                TodaysDate;
    sc_inslave<const Routed::Itinerary *>    ScheduleLeg;
    sc_inslave<const Routed::Route *>       LoadRoute;
    sc_outmaster<const Routed::ArrivalRecord *> AnnounceArrival;

private:
    //-----
    // Context declarations
    SceMiMessageData dSendData;
    SceMiMessageInPortProxy *dOriginAnchorage;
    SceMiMessageInPortProxy *dOriginCupertino;
    SceMiMessageInPortProxy *dOriginNoida;
    SceMiMessageInPortProxy *dOriginSealBeach;
    SceMiMessageInPortProxy *dOriginUK;
    SceMiMessageInPortProxy *dOriginWaltham;

    SceMiMessageOutPortProxy *dDestinationAnchorage;
    SceMiMessageOutPortProxy *dDestinationCupertino;
    SceMiMessageOutPortProxy *dDestinationMaui;
    SceMiMessageOutPortProxy *dDestinationSealBeach;
    SceMiMessageOutPortProxy *dDestinationUK;

    Routed::ArrivalRecord dArrivalRecord;

    //-----
    // Thread declarations
    void scheduleLegThread();
    void loadRouteThread();

    //-----
    // Helper declarations
    static void replyCallback( void *context, const SceMiMessageData *data );
    void announceArrival( SceMiU64 cycleStamp, SceMiU32 arrivalToken );

public:
    void Bind( SceMi *scemi );

    SC_CTOR( Scheduler )
        : dSendData(Routed::MessageSize),
          dOriginAnchorage(NULL),
          dOriginCupertino(NULL),
          dOriginNoida(NULL),
          dOriginSealBeach(NULL),
          dOriginUK(NULL),
          dOriginWaltham(NULL),
          dDestinationAnchorage(NULL),
          dDestinationCupertino(NULL),
          dDestinationMaui(NULL),
          dDestinationSealBeach(NULL),
          dDestinationUK(NULL)
    {
        SC_SLAVE( scheduleLegThread,    ScheduleLeg );
        SC_SLAVE( loadRouteThread,     LoadRoute );
    }
};
```

```
    }  
};
```

There are two slave threads defined in this model: the `::scheduleLegThread()` and the `::loadRouteThread()`. The `::loadRouteThread()`, though not described in detail here, is responsible for sending `TeachRoute` tokens into the `RouteMap` mesh via the `Waltham Origin` transactor when the `RouteMap` is first being configured at the beginning of the simulation. This thread is activated each time the `RouteConfig` module wants to teach a new route during its `LoadRouteMap` operation.

The `::scheduleLegThread()` is activated when the `Scheduler` receives `Routed::Itinerary` messages on its `ScheduleLeg` inslave port from the `Testbench` model. It sends those legs encoded as departure tokens across the message input channels to their designated `Origin` transactors. The `Scheduler` has pointers to each of the message input port proxies that are connected to `Origin` transactors. Each departure token is encoded with the passenger ID and destination ID from the `Routed::Itinerary` record. Following is the source code for the `::scheduleLegThread()`:

```
void Scheduler::scheduleLegThread(){  
    const Routed::Itinerary *leg = ScheduleLeg;  
  
    // Form a 'Passenger Departure' token based on the contents of the given  
    // 'Itinerary' record.  
    SceMiU32 passengerDepartureToken =  
        leg->PassengerID      |  
        (leg->DestinationID   << 4) |  
        (leg->OriginID        << 12) |  
        (leg->TimeOfDeparture << 16);  
  
    dSendData.Set( 0, passengerDepartureToken );  
  
    switch( leg->OriginID ){  
        case Routed::Anchorage: dOriginAnchorage->Send( dSendData ); break;  
        case Routed::Cupertino: dOriginCupertino->Send( dSendData ); break;  
        case Routed::Noida:     dOriginNoida     ->Send( dSendData ); break;  
        case Routed::SealBeach: dOriginSealBeach->Send( dSendData ); break;  
        case Routed::UK:        dOriginUK        ->Send( dSendData ); break;  
        case Routed::Waltham:   dOriginWaltham   ->Send( dSendData ); break;  
        default:  
            assert(0);  
    }  
}
```

The `Scheduler::Bind()` method is called prior to simulation from the `sc_main()` routine (see section Figure 3.2.4.2 on page 33). Here is where the SCE-MI message input and output port proxies leading to each of the `Origin` and `Destination` transactors are bound. Notice that for each of the output port

proxies, the output receive callback, `replyCallback()`, is specified in the binding structure. See section 4.3.3.5 on page 67 for more information about message output port binding.

```
void Scheduler::Bind( SceMi *scemi ){

    // Establish message input portals.
    dOriginAnchorage = scemi->BindMessageInPort( "anchorage", "TokenIn" );
    dOriginCupertino = scemi->BindMessageInPort( "cupertino", "TokenIn" );
    dOriginNoida      = scemi->BindMessageInPort( "noida",      "TokenIn" );
    dOriginSealBeach = scemi->BindMessageInPort( "sealBeach", "TokenIn" );
    dOriginUK         = scemi->BindMessageInPort( "uk",         "TokenIn" );
    dOriginWaltham   = scemi->BindMessageInPort( "waltham",   "TokenIn" );

    // Establish message output portals.
    SceMiMessageOutPortBinding binding = { this, replyCallback, NULL };
    dDestinationAnchorage = scemi->BindMessageOutPort(
        "anchorage", "TokenOut", &binding );
    dDestinationCupertino = scemi->BindMessageOutPort(
        "cupertino", "TokenOut", &binding );
    dDestinationMaui       = scemi->BindMessageOutPort(
        "maui",       "TokenOut", &binding );
    dDestinationSealBeach = scemi->BindMessageOutPort(
        "sealBeach", "TokenOut", &binding );
    dDestinationUK         = scemi->BindMessageOutPort(
        "uk",         "TokenOut", &binding );
}
}
```

The `Scheduler` is also responsible for processing of arrivals. Once the `Calendar` is advanced, arrivals can occur at any time over the course of 24 hours (i.e., 24 clocks). Each arrival token is sent by a `Destination` transactor, over a message output port to the `Scheduler`. The SCE-MI infrastructure dispatches the arriving messages to the `replyCallback()` function that was registered in the `::Bind()` method described above. The `replyCallback()` function, in turn, passes the message to the private `::announceArrival()` method described below. Following is the code for the `replyCallback()` function:

```
void Scheduler::replyCallback( void *context, const SceMiMessageData *data ){
    ((Scheduler *)context)->announceArrival( data->CycleStamp(), data->Get(0) ); }
}
```

The `::announceArrival()` method processes the arrival token. It converts the encoded arrival token to the `Routed::ArrivalRecord` data type, stamps it with `Today'sDate` (an output from the `Calendar`), and

sends it out through the `AnnounceArrival` outmaster port to the `Testbench` model which displays the arrival information to the console as follows:

```
void Scheduler::announceArrival( SceMiU64 cycleStamp, SceMiU32 arrivalToken ){
    Routed::Date todaysDate = TodaysDate; // Read today's date from Calendar

    dArrivalRecord.DateOfArrival = todaysDate.Day;
    dArrivalRecord.TimeOfArrival = cycleStamp - todaysDate.CycleStamp;
    dArrivalRecord.PassengerID   = (Routed::PassengerIDs)
                                   ( arrivalToken      & 0xf );
    dArrivalRecord.DestinationID = (Routed::LocationIDs)
                                   ( (arrivalToken >> 4) & 0xf );
    dArrivalRecord.OriginID      = (Routed::LocationIDs)
                                   ( (arrivalToken >> 12) & 0xf );
    dArrivalRecord.LayoverCount  = (arrivalToken >> 8) & 0xf ;
    assert( dArrivalRecord.LayoverCount < 5 );
    arrivalToken >>= 16;
    for( unsigned i=0; i<dArrivalRecord.LayoverCount; i++){
        dArrivalRecord.LayoverIDs[i] = (Routed::LocationIDs)
                                       ( arrivalToken & 0xf );
        arrivalToken >>= 4;
    }
    AnnounceArrival = &dArrivalRecord; // Arrival record is passed by reference.
}
```

3.2.4.7 The `Calendar` Module: Interfacing with the Clock Advancer

The `Calendar` model is responsible for advancing time on the `RouteMap` one or more days at a time. Once a set of scheduled departures has been programmed in each `Origin` transactor that has departures scheduled for a particular day, the `Calendar` allows the DUT to advance by 24 clocks (24 hours) or some multiple of 24 clocks if the next scheduled departure occurs more than one day from now. The `Calendar` advances time by sending a message to the `ClockAdvancer` transactor in the hardware which has direct control of the DUT clock via the `ClockControl` macro. The source code for the `Calendar` module is very similar in structure to that for the `Scheduler`; therefore, most of it is not shown here.

The `Calendar` model has two slave threads that respond to requests to advance time. The `::advanceCalendarThread()` responds to requests on the `AdvanceCalendar` port to advance a given number of days.

The `::advanceClockThread()` responds to requests to advance one clock at a time which occurs during `RouteMap` configuration to stagger the injection of each `TeachRoute` token by one clock. This method is as follows:

```
void Calendar::advanceClockThread(){
    dSendData.Set( 0, 1 );          // Tell ClockAdvancer to advance by 1 clock.
    dInputPort->Send( dSendData ); // Send message out on port proxy.

    // Pend until the cycle stamp gets updated by the output port proxy reply callback.
    SceMiU64 currentCycleStamp = dCycleStamp;
    while( dCycleStamp == currentCycleStamp )
        wait();
}
```

Notice that this method enters a loop that calls `wait()` to yield to the SystemC kernel. The reason for this is that it wants to guarantee that the clock has completed its advance before returning. By yielding to the SystemC kernel while it is waiting for this condition, the autonomous `SceMiDispatcher` thread (see section 3.2.4.3 on page 34) is naturally given a chance to service the message output ports. This is necessary to reach the condition that the `::advanceClockThread()` is waiting for, namely, for the `Calendar::dCycleStamp` data member to change value.

The `::dCycleStamp` changes value when the `ClockAdvancer` (on the hardware side) responds on its output port that it has completed its one clock time advance which, in turn, causes the `Calendar::replyCallback()` function to get called from the `SceMi::ServiceLoop()`. The `replyCallback()` function is as follows:

```
void Calendar::replyCallback( void *context, const SceMiMessageData *data ){
    ((Calendar *)context)->dCycleStamp = data->CycleStamp(); } }
```

The cycle stamp is updated directly from the `::CycleStamp()` method on the `SceMiMessageData` object. This reflects a count of elapsed controlled clock counts that had occurred from the beginning of the simulation to the time this message was sent from the hardware side. This is a convenient way for software to keep track of elapsed clock time in the hardware. Once the `::dCycleStamp` is updated, the `wait()` loop in the `::advanceClockThread()` described above, is released and the function can return.

Keep in mind the `::advanceClockThread()` and `replyCallback()` functions are being called under two different autonomous threads that each frequently yield to each other. The former is called from the autonomous `Testbench::driverThread()` and the latter is called ultimately from the `SceMi::ServiceLoop()` function that is called from underneath the autonomous `SceMiDispatcher::dispatchThread()`. This illustrates the clean interaction between a general multi-threaded application software environment and the SCE-MI service loop.

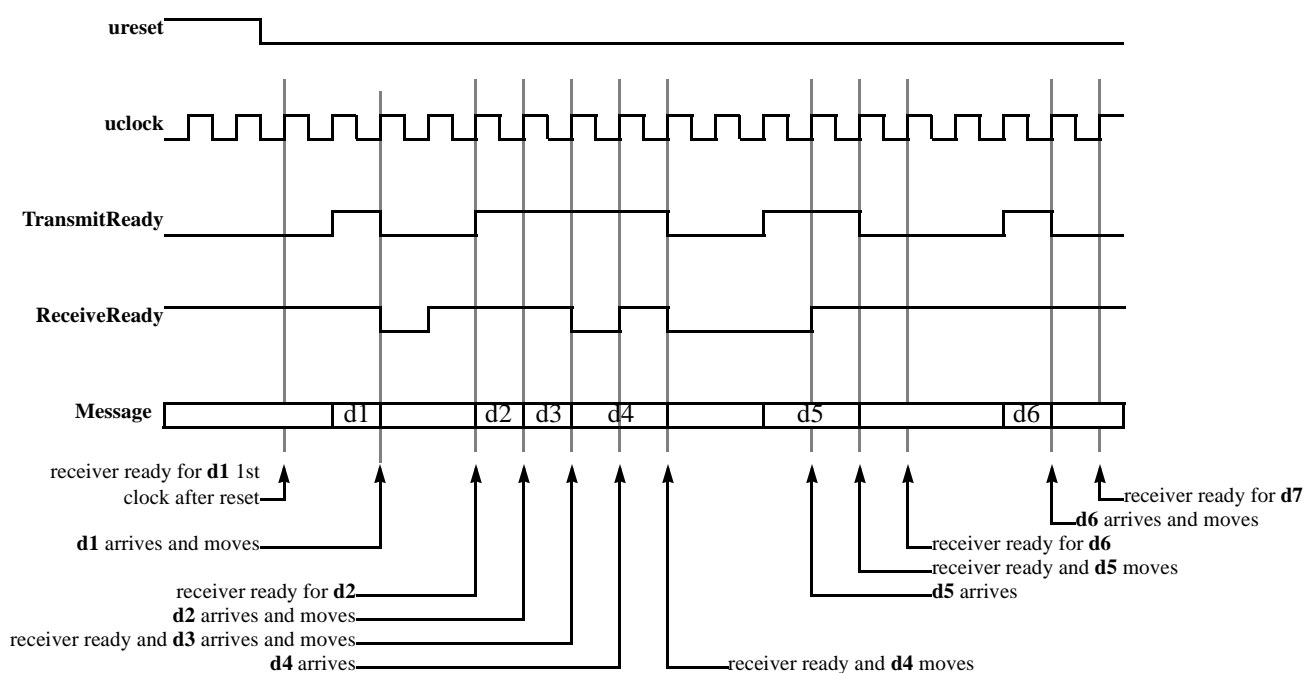
4.0 Formal Functional Specification

4.1 Hardware Side Interface Macros

4.1.1 Dual Ready Protocol

The message port macros on the hardware side use a general PCI-like dual ready protocol. Before going into detail about how these macros work, the dual ready handshake protocol that they use is explained in this section. The waveforms in Figure 11 depict several dual ready handshake scenarios.

FIGURE 11. Dual Ready Handshake Protocol



Briefly, the dual ready handshake works as follows. The transmitter asserts `TransmitReady` on any clock cycle when it has data, and de-asserts when it does not. The receiver asserts `ReceiveReady` on any cycle when it is ready for data, and de-asserts when it is not. In any clock cycle in which `TransmitReady` and `ReceiveReady` are both asserted, data “moves” meaning that it is taken by the receiver.

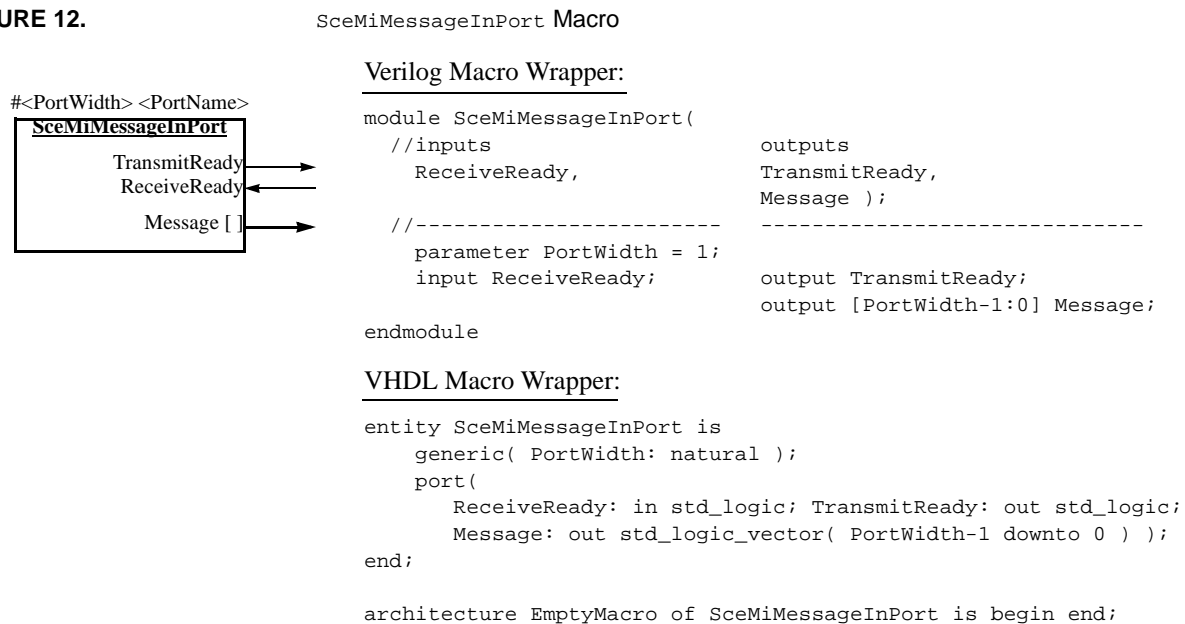
The dual ready protocol has the following two advantages:

- Signals are level-based; therefore, they are easily sampled by posedge clocked logic
- However, if both `TransmitReady` and `ReceiveReady` stay asserted, sequences of data can still move every clock cycle and therefore, the same performance can be realized as, for example, a *toggle* based protocol.

4.1.2 SceMiMessageInPort Macro

The `SceMiMessageInPort` macro presents messages arriving from the software side of a channel to the transactor. The macro consists of two handshake signals that play a dual-ready protocol and a data bus that presents the message itself. Figure 12 shows the symbol for the `SceMiMessageInPort` macro as well as Verilog and VHDL source code for the empty macro wrappers.

FIGURE 12.



4.1.2.1 Parameters and Signals

PortWidth

The message width in bits is derived from the setting of this parameter.

PortName

The port's name is derived from its instance label.

ReceiveReady

A value of one on this signal indicates that the transactor is ready to accept data from the software. By asserting this signal, the hardware indicates to the software that it has a location into which it can put any data that might arrive on the message input port. When a new message arrives, as indicated by the `TransmitReady` and `ReceiveReady` both being true, that location is consumed (see Figure 11 on page 46). When this happens, a notification is sent to the software side that a new empty location is available and this triggers an *input ready callback* to occur on the software side (see section 4.1.2.2 on page 48). The next section explains in detail when input ready propagation notifications are done with respect to the timing of the `TransmitReady` and `ReceiveReady` handshakes.

Note that it is possible for transactors not to utilize `ReceiveReady` and the input ready callback. If this is the case, the `ReceiveReady` input must be permanently asserted (i.e., “tied high”) and, on the software side, no input ready callback is registered. In this case, `TransmitReady` merely acts as a strobe for each

arriving message. The transactor must be designed to take any arriving data immediately as it is not guaranteed to be held for subsequent uclock cycles.

TransmitReady

A value of one on this signal sampled on any posedge of the uclock indicates that the channel has message data ready for the transactor to take. If `ReceiveReady` is not asserted, the `TransmitReady` remains asserted until and during the first clock in which `ReceiveReady` finally becomes asserted. During this clock data moves and, if no more messages have arrived from the software side, the `TransmitReady` is de-asserted.

Message

This vector signal constitutes the payload data of the message.

4.1.2.2 Input Ready Propagation

If the software client code registers an input ready callback when it first binds to a message input port proxy (see section 4.3.3.4 on page 66), the hardware side of the infrastructure must notify the software side each time it is ready for more input. Each time it is so notified, the port proxy on the software side makes a call to the user registered input ready callback. This mechanism is called *input ready propagation*. The prototype for the input ready callback is as follows:

```
void (*IsReady)(void *context);
```

When this function is called, a software model can assume that a message can be sent to the message input port proxy for transmission to the message input port on the hardware side. The `context` argument can be a pointer to any user-defined object, presumably the software model that has bound the proxy.

An important point to note is that it is totally up to the application to follow the protocol that if, the transactor is not ready to receive input, the software model should not do a send. A well behaved software model should know not to do a send if it has not received an input ready callback (assuming the input ready propagation mechanism is being deployed).

The SCE-MI infrastructure will not enforce this. So, the idea is that, when the transactor wants to say, "I'm ready for input", it will assert the `ReceiveReady` input to the `SceMiMessageInPort` which will cause an input ready callback to be called on the software side, the next time `::ServiceLoop()` is entered. The software model should then typically either set a flag that can be consulted later on outside `::ServiceLoop()`, indicating that the transactor is ready for input, or, in some cases, it may choose to do a send right then and there (i.e. inside the callback). The point is that a send should not be done by the software model until some time (either immediately or deferred) after the input ready callback has been received.

A second point to be made is that it should be fine for an application to service as many output callbacks as is desired while pending an input callback. In other words, the software model may have an outer loop that checks the status of an application defined *OKToSend* flag on each iteration and skips the send if the flag is false.

So suppose an application has an outer loop that repeatedly calls `::ServiceLoop()` and checks for arriving output messages and input ready notifications. Each callback function sets a flag in the context that the outer loop uses to know if an output message has arrived and needs processing, or an input port needs more input.

It is possible that, before an input ready callback gets called, that outer loop may have called `::ServiceLoop()` say 50 times each time resulting in an output message callback and the subsequent processing of that output message.

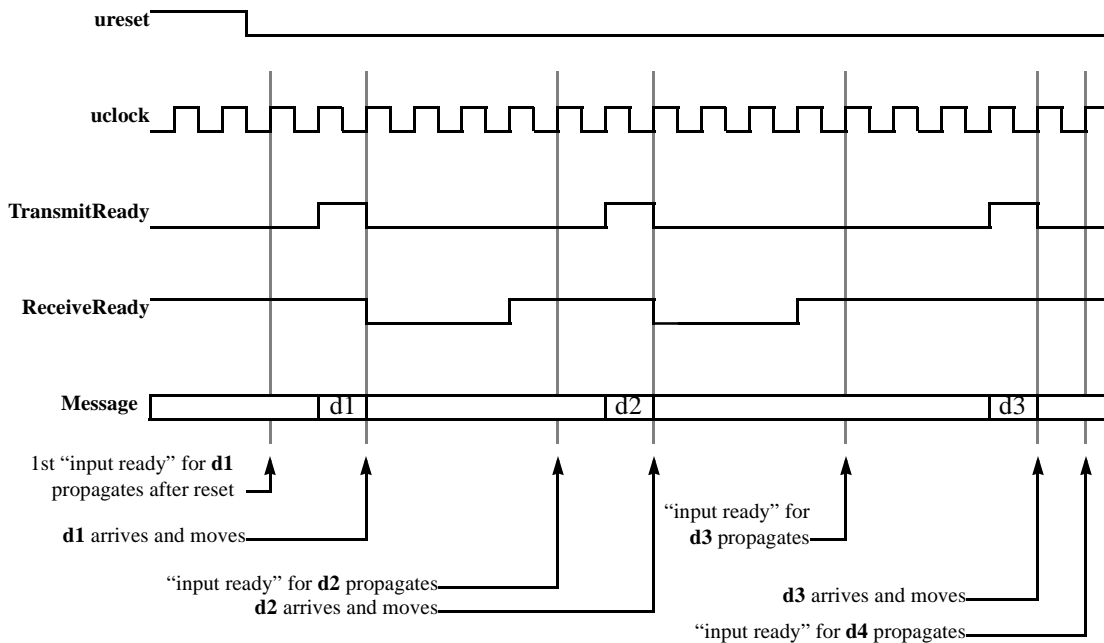
Finally, on the 51st time `::ServiceLoop()` is called, the input ready callback is called which sets the `OKToSend` flag in its context, after which the outer loop would detect the new flag status and initiate a send on that input channel.

The handshake waveforms in Figure 11 on page 46 are intended purely to illustrate the semantics of the dual ready protocol. There can be a couple of reasons why these waveforms might not be realistic in an actual implementation of a `SceMiMessageInPort` macro. First, if input ready propagation is enabled (because an optional callback was registered on the software side), the sender on the software side might expect input ready notifications before transmitting messages so that two back-to-back messages, and hence `TransmitReady` assertions on consecutive clocks might be impossible. Second, even if input ready callbacks were not registered on a given port, the timing of the physical layer of the SCE-MI bridge might be such that two successive transmissions are not possible unless the software end somehow batched consecutive message transmissions to the hardware.

All of this said however, the hardware in the transactor should be designed so as to anticipate any of the above scenarios whether or not they are likely to happen.

The waveforms shown in Figure 13 are typical of what one might see with input ready callbacks enabled. It shows four possible scenarios in which an input ready notification occurs.

FIGURE 13. SceMiMessageInPort Handshake Waveforms With Input Ready Propagation



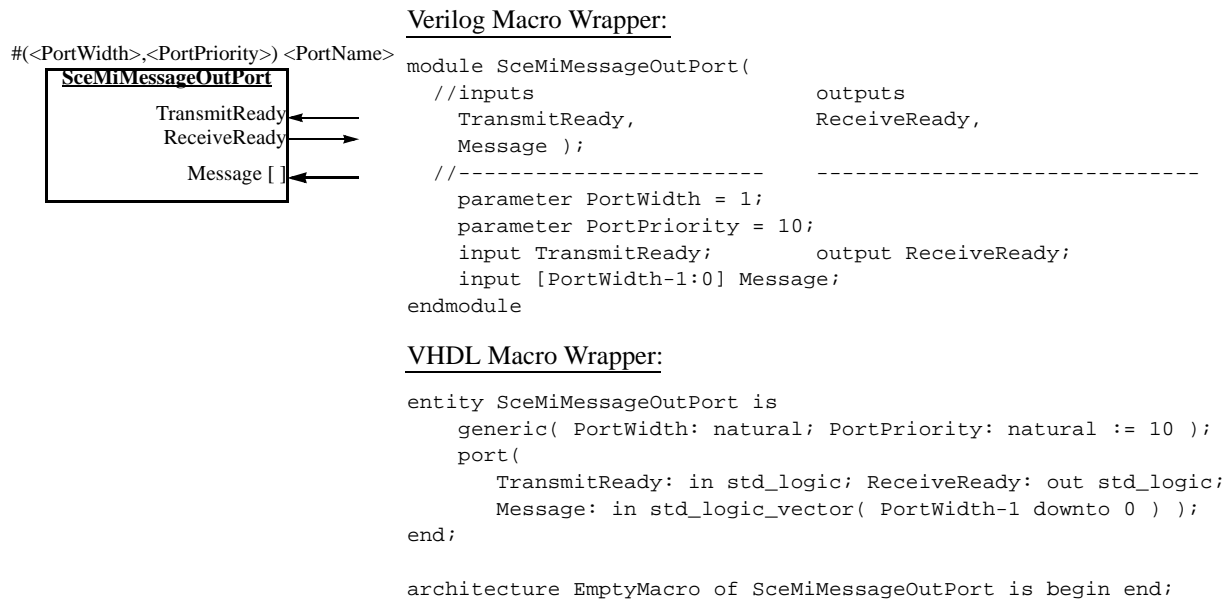
In the depicted scenarios, an input ready notification is propagated to the software if,

- `ReceiveReady` from transactor is asserted in the first clock following a reset or,
- `ReceiveReady` from transactor transitions from a 0 to a 1 or,
- `ReceiveReady` from transactor remains asserted in a clock following one in which a transfer occurred due to a assertions on both `TransmitReady` and `ReceiveReady`.

4.1.3 `SceMiMessageOutPort` Macro

The `SceMiMessageOutPort` macro allows transactors to send messages to the software side. Like the `SceMiMessageInPort` macro, it also uses a dual ready handshake except that in this case, the transmitter is the transactor and the receiver is the SCE-MI interface. A transactor can have any number of `SceMiMessageOutPort` macro instances. Figure 14 shows the symbol for the `SceMiMessageOutPort` macro as well as Verilog and VHDL source code for the empty macro wrappers.

FIGURE 14. `SceMiMessageOutPort` Macro



4.1.3.1 Parameters and Signals

PortWidth

The message width in bits is derived from the setting of this parameter.

PortPriority

The priority for determining which output messages are sent to the output channel first, should more than one arrive on the same uclock. See section 4.2.1 on page 59 for details on the meaning of this parameter.

PortName

The port's name is derived from its instance label.

TransmitReady

A value of one on this signal indicates that the transactor has message data ready for the output channel to take. If `ReceiveReady` is not asserted, the `TransmitReady` must remain asserted until and during the first clock in which `ReceiveReady` finally becomes asserted. During this clock data moves and, if the transactor has no more messages for transmission, it de-asserts the `TransmitReady`.

ReceiveReady

A value of one on this signal sampled on any uclock posedge indicates that the output channel is ready to accept data from the transactor. By asserting this signal, the SCE-MI hardware side indicates to the transactor that the output channel has a location into which it can put any data that is destined for the software side of the channel. In any cycle during which both the `TransmitReady` and `ReceiveReady` are asserted, the transactor can assume that the data moved. If, in the subsequent cycle, the `ReceiveReady` remains asserted, that means that a new empty location is available which the transactor can load any time by asserting `TransmitReady` again. Meanwhile, the last message data, upon arrival to the software side, triggers a *receive callback* on its message output port proxy (see section 4.3.7.1 on page 77).

Message

This vector signal constitutes the payload data of the message originating from the transactor, to be sent to the software side of the channel.

4.1.4 SceMiClockPort Macro

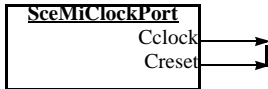
The `SceMiClockPort` macro supplies a controlled clock to the DUT. The `SceMiClockPort` macro is parametrized such that each instance of a `SceMiClockPort` fully specifies a controlled clock of a given frequency, phase shift, and duty cycle.

The `SceMiClockPort` macro also supplies a controlled reset whose duration is the specified number of cycles of the `cclock`.

Figure 15 shows the symbol for the `SceMiClockPort` macro as well as Verilog and VHDL source code for the empty macro wrappers.

FIGURE 15.

```
#(<ClockNum>,
<RatioNumerator>,
<RatioDenominator>,
<DutyHi>, <DutyLo>, <Phase>,
<ResetCycles>) <ClockName>
```



SceMiClockPort Macro

Verilog Macro Wrapper:

```
module SceMiClockPort(
//inputs                                outputs
//-----                                -----
parameter ClockNum=1;
parameter RatioNumerator=1, RatioDenominator=1;
parameter DutyHi=0, DutyLo=100, Phase=0;
parameter ResetCycles=8;
endmodule
```

VHDL Macro Wrapper:

```
entity SceMiClockPort is
generic( ClockNum: natural := 1;
RatioNumerator: natural := 1; RatioDenominator: natural := 1;
DutyHi: natural := 0; DutyLo: natural := 100;
Phase: natural := 0; ResetCycles: natural := 8 );
port( Cclock: out std_logic; Creset: out std_logic );
end;

architecture EmptyMacro of SceMiClockPort is begin end;
```

Note that all of the clock parameters have default values. In simpler systems where only one controlled clock is needed, exactly one instance of a `SceMiClockPort` can be instantiated at the top level with no parameters specified. This results in a single controlled clock with a ratio of 1/1, a *don't care duty cycle* (see discussion of duty cycle in section 4.1.4.3 on page 54), and a phase shift of 0. Ideally, this clock's frequency matches that of the uclock during cycles in which it is enabled.

The SCE-MI infrastructure always implicitly creates a controlled clock with a 1/1 ratio which is the highest frequency controlled clock in the system. Whether or not it is visible to the user's design depends on whether a `SceMiClockPort` with a 1/1 ratio is explicitly declared (instantiated).

In more complex systems that require multiple clocks, a `SceMiClockPort` instance must be created for each required clock. The clock ratio that is specified in the instantiation parameters always specifies the frequency of the clock as a ratio relative to the highest frequency controlled clock in the system (whose ratio is 1/1 whether explicitly or implicitly defined).

For example if a `cclock` is defined with a ratio of 4/1 this is interpreted as, "for every 4 edges of the 1/1 `cclock` there is only 1 edge of this `cclock`". This would be a "divide-by-four" clock.

4.1.4.1 Parameters and Signals

clockNum=1

This parameter is used to assign a unique number to a clock that is used to differentiate it from other `SceMiClockPort` instances. It shall be considered an error (by the infrastructure linker) if more than one `SceMiClockPort` instances share the same `clockNum`. The default `clockNum` is 1.

RatioNumerator=1, RatioDenominator=1

These parameters constitute the numerator and denominator respectively of this clock's ratio. The numerator always designates the number of cycles of the fastest 1/1 controlled clock that occur during the number of cycles of "this" clock specified in the denominator. For example, with

RatioNumerator=5 and **RatioDenominator=2** a 5/2 clock is specified which means that for every 5 cycles of the 1/1 clock that occur, only 2 cycles of this clock occur. The default clock ratio is 1/1.

DutyHi=0, DutyLo=100, Phase=0

The duty cycle is expressed with arbitrary integers that are normalized to their sum such that the sum of **DutyHi** and **DutyLo** represent the number of units for a whole cycle of the clock. For example, with **DutyHi=75** and **DutyLo=25** the high time of the clock is 75 out of 100 units or 75% of the period.

Similarly, the low time would be 25% of the period. The phase shift is expressed in the same units so that if **Phase=30**, that would mean that the clock should be shifted by 30% of its period before the first low to high transition occurs.

The default duty cycle shown in the macro wrappers in figure 15 above is a *don't care duty cycle* of 0/100 (see section 4.1.4.3 on page 54).

ResetCycles=8

This parameter specifies how many cycles of this controlled clock shall occur before the controlled reset transitions from its initial value of 1 back to 0.

ClockName

The clock port's name is derived from its instance label.

Cclock

This is the controlled clock signal that is supplied to the DUT by the SCE-MI infrastructure via this macro. This clock's characteristics is derived from the parameters specified on instantiation of this macro.

Creset

This is the controlled reset signal that is supplied to the DUT by the SCE-MI infrastructure via this macro.

4.1.4.2 Deriving Clock Ratios from Frequencies

Another way of to specify clock ratios is enter them directly as frequencies all normalized to the clock with the highest frequency. To specify ratios this way requires the following:

- make each ratio numerator equal to the highest frequency
- use consistent units for all ratios
- omit those units and simply state them as integers

For example, suppose a system has 100Mhz, 25Mhz, and 10Mhz, 7.5 Mhz, and 32kHz clocks. To specify the ratios, the frequencies can be directly entered as integers using kHz as the unit (but omitting it !):

100000 / 100000 - the fastest clock

100000 / 25000

100000 / 10000

100000 / 7500

100000 / 32

Users who like to think in frequencies rather than ratios can use this simple technique. It is also possible that an implementor of the SCE-MI API may wish to provide a tool to assist in deriving clock ratios from frequencies. Such a tool could allow a user to enter clock specifications in terms of frequencies and then generate a set of equivalent ratios.

In addition, the tool could post process waveforms (such as .vcd files) generated by the simulation in such a way that the defined clocks appear in the waveform display to be the exact same frequencies given by the user.

4.1.4.3 Don't Care Duty Cycle

The default duty cycle shown in the macro wrappers in figure 15 above is a *don't care duty cycle*. Users can specify that they only care about posedges of the cclock and do not care where the negedge falls. This is known as a *posedge active don't care duty cycle*. In such a case, the `DutyHi` is given as a 0. The `DutyLo` can be given as an arbitrary number of units such that the `Phase` offset can still be expressed as a percentage of the whole period (i.e `DutyHi+DutyLo`).

For example if a user specifies this combination:

```
DutyHi=0, DutyLo=100, Phase=30
```

The user is saying the following:

- I don't care about the duty cycle. Specifically I don't care where the negedge of the clock falls.
- If the total period is expressed as 100 units (0+100), the phase should be shifted by 30 of those units. This represents a phase shift of 30%.

Another example:

```
DutyHi=3, DutyLo=1, Phase=2
```

This says,

- I care about both intervals of the duty cycle. Duty cycle is 75%/25%.
- Phase shift is 50% of period (expressed as 3+1 units).

It is also possible to have a *negedge active don't care duty cycle*. In this case, the `DutyLo` parameter is given as a 0 and the `DutyHi` is given as a positive number > 0.

For example:

```
DutyHi=1, DutyLo=0, Phase=0
```

It means,

- I don't care about duty cycle. Specifically I don't care where the posedge of a clock falls.
- Phase shift is 0.

In any clock specification, it is considered an error if `Phase >= DutyHi + DutyLo`.

4.1.4.4 Controlled Reset Semantics

The `Creset` output of the `SceMiClockPort` macro shall obey the following semantics:

- `Creset` will start low (deasserted) and transition to high one or more uclocks later. It will then remain high (asserted) for a at least a minimum duration specified by the `ResetCycles` parameter adorning the `SceMiClockPort` macro. This duration is expressed as a number of edges of associated `Cclock`. Following the reset duration, the `Creset` will then go low (deasserted) and remain low for the remaining duration of the simulation. Some applications require 2 edged resets at the beginning of a simulation.
- For multiple cclocks, the reset duration must have a minimum length such that it is guaranteed to span the `ResetCycles` parameter of any clock. In other words, the controlled reset duration for all clocks must be, as a minimum,
`max(ResetCycles for cclock1, ResetCycles for cclock2, ...)`
- Some implementations may chose to use a reset duration that is larger than the quantify shown above if it is required to achieve proper alignment of multiple cclocks on the edges of the controlled reset as described in section 4.1.4.5 on page 55.
- During the assertion of `Creset`, `Cclock` edges shall be forced regardless of the state of the `ReadyForCclock` inputs to the `SceMiClockControl` macros. Once the reset duration completes, the `Cclock` will be controlled normally by the `ReadyForCclock` inputs.

Note: The operation of controlled reset just described provides the default controlled reset behavior generated by the `SceMiClockPort` macro. If more sophisticated reset handling is required it will have implemented with a specially written reset transactor that is used in lieu of the simpler controlled resets that come from the `SceMiClockPort` instances. For example, if a software controlled reset is required, an application would need to create a reset transactor that responds to a special software originated reset command that arrives on its message input port.

4.1.4.5 Multiple Cclock Alignment

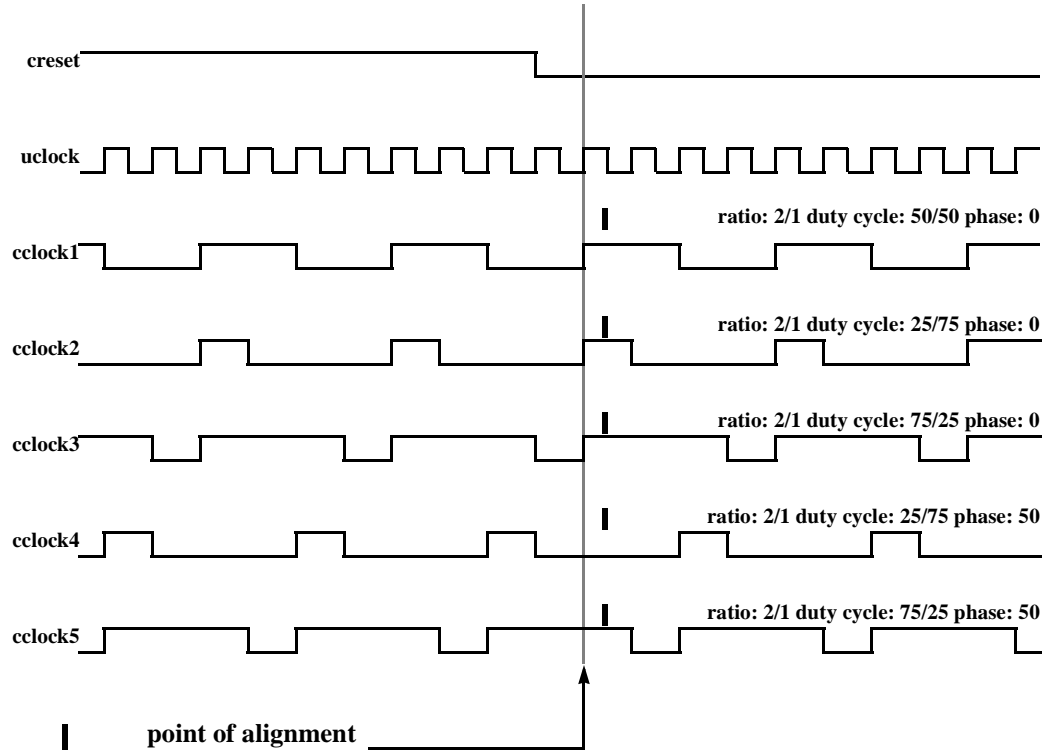
In general, all cclocks should align on the first rising uclock edge following the trailing edge of the creset. This uclock edge shall be referred to as the *point of alignment*. For cclocks with phases of 0 this means that rising edges of these clocks should coincide with the point of alignment. For cclocks with phases > 0 edges will occur some time after the point of alignment. Figure 16 shows an assortment of cclocks with the uclock and creset. It also shows how those cclocks behave at the point of alignment.

In the diagram, `cclock1`, `cclock2`, and `cclock3` have phases of 0 and therefore have rising edges at the point of alignment.

`cclock4` has the same duty cycle as `cclock2` but a phase shift of 50%. Therefore its rising edge occurs 2 uclocks (1/2 cycle) after the point of alignment. Its *starting value* at the point of alignment is still 0.

`cclock5` has the same duty cycle as `cclock3` but a phase of 50%. Again, its rising edge occurs 1/2 cycle after the point of alignment. But notice that its starting value at the point of alignment is 1. This can be alternatively thought of as an *inverted phase*. Anytime the phase is greater than the initial duty cycle interval, the starting value at the point of alignment will be a 1.

FIGURE 16. Multi-Clock Alignment

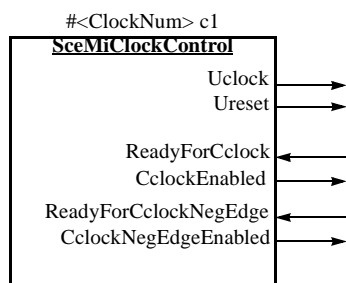


4.1.5 *SceMiClockControl* Macro

For every *SceMiClockPort* macro instance there must be at least one counterpart *SceMiClockControl* macro instance presumably encapsulated in a transactor. The *SceMiClockControl* macro is the means by which a transactor can control a DUT's clock and by which the SCE-MI infrastructure can indicate to a transactor on which *uclock* cycles that *controlled clock* have edges.

Figure 17 shows the symbol for the *SceMiClockControl* macro as well as Verilog and VHDL source code for the empty macro wrappers.

FIGURE 17. SceMiClockControl Macro



Verilog Macro Wrapper:

```

module SceMiClockControl(
    //inputs
    ReadyForCclock,
    ReadyForCclockNegEdge,
    //-----
    parameter ClockNum = 1;
    input ReadyForCclock;
    input ReadyForCclockNegEdge,
endmodule

    outputs
    Uclock, Ureset;
    CclockEnabled,
    CclockNegEdgeEnabled );
    -----
    output Uclock, Ureset;
    output CclockEnabled;
    output CclockNegEdgeEnabled;

```

VHDL Macro Wrapper:

```

entity SceMiClockControl is
    generic( ClockNum: natural := 1 );
    port(
        Uclock, Ureset: out std_logic;
        ReadyForCclock: in std_logic;
        CclockEnabled: out std_logic;
        ReadyForCclockNegEdge: in std_logic;
        CclockNegEdgeEnabled: out std_logic;
    );
end;

architecture EmptyMacro of SceMiClockControl is begin end;

```

For each `SceMiClockPort` defined in the system, at least one corresponding `SceMiClockControl` macro must be instantiated in one or more transactors. In addition to providing uncontrolled clocks and resets, this macro also provides handshakes that provide explicit control of both edges of the generated cclock.

4.1.5.1 Parameters and Signals

ClockNum=1

This is the only parameter given to the `SceMiClockControl` macro. This parameter is used to associate a `SceMiClockControl` instance with its counterpart `SceMiClockPort` instance that is defined at the top level. The default `ClockNum` is 1.

Note that associated with each instance of `SceMiClockControl` there must be exactly one instance in the system of `SceMiClockPort`. But for each instance of `SceMiClockPort`, there can be one or more instances of `SceMiClockControl`. A `SceMiClockControl` instance identifies the `SceMiClockPort` it is associated with by properly specifying a `ClockNum` parameter matching that of its associated `SceMiClockPort`.

Uclock

This is the uncontrolled clock signal generated by the SCE-MI infrastructure.

Ureset

This is the uncontrolled reset generated by the SCE-MI infrastructure. This signal is high at the

beginning of simulated time and transitions to a low an arbitrary (implementation dependent) number of uclocks later. It can be used to reset the transactor.

Note: The uncontrolled reset must have a duration spanning that of the longest controlled reset (`CReset` output from each `SceMiClockPort` - see section 4.1.4.4 on page 55) as measured in uclocks. This guarantees that all DUTs and transactors properly wake up in an initialized state the first uclock following expiration of the last controlled reset.

ReadyForCclock

This input to the macro indicates to the SCE-MI infrastructure that a transactor is willing to allow its associated DUT clock to advance. The transactor de-asserts this signal when it needs to perform operations during which the DUT must be frozen. One of the most useful applications of this feature is to perform complex algorithmic operations on the data content of a transaction before presenting it to the DUT.

If this input to one of the `SceMiClockControl` instances that are associated with a given controlled clock is deasserted, the next posedge of that cclock will be disabled. In reacting to a `ReadyForCclock` of a slower clock, the infrastructure must not prematurely disable posedges of other faster clocks that occur prior to the last possible uclock preceding the edge to be disabled. In other words, that edge is disabled “just in time” so as to allow faster clock activity to proceed until the last moment possible. Once the edge is finally disabled, all posedges of all controlled clocks are also disabled.

ReadyForCclockNegEdge

Similarly, for negedge control, if this input to one of the `SceMiClockControl` instances that are associated with a given controlled clock is deasserted, the next negedge of that clock will be disabled. In reacting to a `ReadyForCclockNegEdge` of a slower clock, the infrastructure must not prematurely disable negedges of other faster clocks that occur prior to the last possible uclock preceding the edge to be disabled. In other words, that edge is disabled “just in time” so as to allow faster clock activity to proceed until the last moment possible. Once the edge is finally disabled, all negedges of all controlled clocks are also disabled.

Support for explicit negedge control is needed because without it, transactor logic that only cares about controlling posedge clocks could not inadvertently disable the next negedge of a clock when it only intended to disable the next posedge of a clock. Transactors that do not care about controlling negedges (such as the one shown in Figure 5 on page 17) should tie this signal high.

CclockEnabled

This macro output signals the transactor that on the next posedge of uclock there is a posedge of the controlled clock. The transactor can thus sample this signal to know if a DUT clock edge occurs. The SCE-MI infrastructure looks at the `ReadyForCclock` from all the transactors and asserts `CclockEnabled` only if they are all asserted. This means that any transactor can stop all the clocks in the system by simply de-asserting `ReadyForCclock`.

Note: for a *negedge active don't care duty cycle* (see 4.1.4.3 on page 54), since the user does not care about the posedge, the `CclockEnabled` will always be 0.

CclockNegEdgeEnabled

This signal works like `CclockEnabled` except that it indicates if the negedge of a controlled clock occurs on the next posedge of the uclock. This can be useful for transactors that control double pumped DUTs. Transactors that do not care about negedge control can ignore this signal.

Note: for a *posedge active don't care duty cycle* (see 4.1.4.3 on page 54), since the user does not care about the negedge, the `CcLockNegEdgeEnabled` will always be 0.

4.2 Infrastructure Linkage

The contents of this section is strictly the concern of the *infrastructure implementor* class of user, as defined in section 2.2.3 on page 8. *End users* and *transactor implementors* can assume that the operations described herein are automatically handled by the *infrastructure linker*.

As described in section 2.4.2 on page 14, *infrastructure linkage* is the process that analyzes the user's bridge netlist on the hardware side and compiles it into a form suitable to run on the emulator. This may involve expanding the interface macros into infrastructure components that are added to the existing structure as well as to generate parameter information that is used to bind the hardware side to the software side. In order to determine this information, the infrastructure linker analyzes the netlist and searches for instances of the SCE-MI hardware side macros, read the parameter values from those instances, and generate a parameters file that can be read during software side initialization to properly bind message port proxies to the hardware side.

It can also be typical for the infrastructure linker to provide options either in the form of switches and/or an input configuration file that allow a user to pass along or override implementation specific options. A well crafted infrastructure linker however, should maximize ease-of-use by transparently providing the end user with a suitable set of default values for implementation specific parameters, so that most, if not all of these parameters need not be overridden.

4.2.1 Parameters

The following set of parameters define the minimum set that is needed for all implementations of the SCE-MI standard. Specific implementations might require additional parameters.

Number of Transactors

The number of transactors shall be derived by counting the number of modules in the user's design that contain `SceMiClockControl` macros. It shall be assumed that any module that is to be officially considered a transactor must have at least one `SceMiClockControl` instance immediately inside it.

Transactor Name

The transactor name shall be derived from the hierarchical path name to an instance of a module that qualifies as a transactor as per the above definition. Naturally, if there are multiple instances of a given type of transactor, they shall be uniquely distinguished by their instance path names. The syntax used to express the path name must be that of the HDL language that the bridge netlist is expressed in.

Number of Message Input or Output Channels

The infrastructure linker derives the number of message input and output ports by counting instances of the `SceMiMessageInPort` and `SceMiMessageOutPort` macros.

Port Name

The name of each port shall be derived from the relative instance path name to that port relative to its containing transactor module. For example, if the full path name to a message input port macro instance is, using Verilog notation, `Bridge.u1.tx1.ip1` and the transactor name is `Bridge.u1.tx1`, then the port name is `ip1`. If an output port is instantiated one level down from the input port and its full path is

`Bridge.u1.tx1.m1.op1`, then its port name is `m1.op1` since it is instantiated a level down relative to the transactor root level.

The full pathname to a port can be derived by concatenating the transactor name to the port name (with a hierarchical separator inserted between).

Message Input or Output Port Width

The width of a port in bits shall be derived from the `PortWidth` parameter defined in the message port macro. This width defaults to 1 but is almost always overridden to a significantly larger value at the point of instantiation.

Message Output Port Priority

The priority of a message output port shall be derived from the `PortPriority` parameter defined in the `SceMiMessageOutPort` macro. For certain implementations, it might be useful as a “hint” to the infrastructure linker to decide which output ports should be serviced first, should they happen to present message data on the same uclock, and are implemented over a number of “physical message channels” that is less than the limitless number of virtual message channels. To some users this might be important. For those who do not care, the default value of 10 does not need to be overridden and need not be specified in the instantiation statement.

With some exceptions, the output port priority generally follows the semantics of the UNIX `nice` command as follows:

- $0 < \text{allowed priority values} < 20$
- Default priority value is 10
- The lower the number the higher the priority
- Output port priority 0 is reserved for internal use by the infrastructure
- For message output ports with the same priority number, their relative priority is undetermined and strictly an artifact of infrastructure linker implementation

Number of Controlled Clocks

This number shall be derived by counting all instances of the `SceMiClockPort` macro.

Controlled Clock Name

The name of a controlled clock is derived from the instance label (not path name) of its `SceMiClockPort` instance, necessarily instantiated at the top level of the user’s bridge netlist and unique among all instances of `SceMiClockPort`.

Controlled Clock Ratio

The clock ratio is determined from the `RatioNumerator` and `RatioDenominator` parameters of the `SceMiClockPort` macro. The `RatioNumerator` designates the number of cycles of the fastest 1/1 controlled clock that occur during the number of cycles of “this” clock specified in `RatioDenominator`. See 4.1.4 on page 51 for more details about the clock ratio.

Controlled Clock Duty Cycle and Phase

The duty cycle is determined from the `DutyHi`, `DutyLo`, and `Phase` parameters of the `SceMiClockPort` macro. The duty cycle is expressed as a pair of arbitrary integers, `DutyHi` and `DutyLo` interpreted as follows: if the sum of `DutyHi` and `DutyLo` represents the number of units in a period of the clock, then `DutyHi` represents the number of units of high time and `DutyLo`, the number of units of low time.

Similarly, `Phase` represents the number of units the clock is phase shifted relative to the reference 1/1 cclock. It is also possible for a user to specify a *don't care duty cycle*. See 4.1.4 on page 51 for more details about the duty cycle and phase.

Controlled Reset Cycles

The duration of a controlled reset expressed in terms of cclock cycles is determined from the `ResetCycles` parameter of the `ClockPort` macro.

4.2.1.1 Parameter File

A parameter file should be automatically generated by the infrastructure linker after analyzing the user supplied netlist and determining all the parameters identified in the previous section. The parameter file can be read by the software side of the SCE-MI infrastructure to facilitate binding operations that occur after software model construction.

Because it is automatically generated, content and syntax of the parameter file is left to specific implementors of the SCE-MI. The content itself is not intended to be portable. However, on the software side, a parameter access API must be provided by the infrastructure implementor that conforms to the specification in 4.3.4 on page 70. This access block must support access to a specifically named set of parameters required by the SCE-MI as well as an optional, implementation specified set of named parameters.

All SCE-MI required parameters are read-only because their values are automatically determined by the infrastructure linker by analyzing the user supplied netlist. Implementation specific parameters can be read-only or read-write as required by the implementation.

4.3 Software Side Interface - C++ API

To gain access to the hardware side of the SCE-MI, the software side must first initialize the SCE-MI software side infrastructure and then bind to port proxies representing each message port defined on the hardware side. Part of initializing the SCE-MI involves instructing the SCE-MI to load the parameter file that was generated by the infrastructure linker. The information in the parameter file gives the SCE-MI software side the means to establish rendezvous with the hardware side in response to port binding calls from the user's software models. Rendezvous during port binding is achieved primarily by name association involving transactor names and port names.

Note that clock names and properties identified in the parameter file are of little significance during the binding process although this information is procedurally available to applications that might need it through the parameter file API (see section 4.3.4 on page 70).

Access to the software side of the interface is facilitated by a number of C++ classes that are as follows:

```
class SceMiEC
class SceMi
class SceMiParameters
class SceMiMessageData
class SceMiMessageInPortProxy
class SceMiMessageOutPortProxy
```

4.3.1 Primitive Data Types

In addition to C data types such as `integer`, `unsigned`, and `const char *`, many of the arguments to the methods in the API require unsigned data types of specific width. To support these, the following data types are placed in the header files for the API:

```
typedef unsigned int SceMiU32;
typedef unsigned long long SceMiU64;
```

4.3.2 Miscellaneous Interface Issues

4.3.2.1 class `SceMiEC` - Error Handling

Note that most of the calls in the interface take an `SceMiEC * ec` as the last argument. Because the usage of this argument is consistent for all methods, error handling semantics are explained in detail in this section rather than documenting error handling for each method in the API.

Error handling in SCE-MI was designed to be flexible enough to either use a traditional style of error handling where an error status is returned and checked with each call, or a callback based scheme where a registered *error handler* is called when an error occurs.

```
enum SceMiErrorType {
    SceMiOK,
    SceMiError
};

struct SceMiEC {
    const char *Culprit;
    const char *Message;
    SceMiErrorType Type;
    int Id;
};

typedef void (*SceMiErrorHandler)(void *context, SceMiEC *ec);

static void
SceMi::RegisterErrorHandler(
    SceMiErrorHandler errorHandler,
    void *context );
```

This method registers an optional error handler with the SCE-MI that is to be called in the event that an error occurs.

When any SCE-MI operation encounters an error, the following procedure is followed:

- If the `SceMiEC *` pointer passed into the function was non-NULL, the values of the `SceMiEC` structure are filled out by the errant call with appropriate information describing the error, and control is returned to the caller. This can be thought of as a *traditional* approach to error handling

such as one might find in C applications. It is up to the application code to check the error status after each call to the API and take appropriate abortive action if an error is detected.

- Else if the `SceMiEC *` pointer passed to the function is `NULL` (or nothing is passed since the default is `NULL` in each API function), and an error handler was registered, that error handler is called from within the errant API call. The error handler is passed an internally allocated `SceMiEC` structure filled out with the error information. In this *error handler callback approach*, the user-defined code within the handler can initiate abort operations. If it is a C++ application, a `catch` and `throw` mechanism might be deployed. A C application can simply call the `abort()` or `exit()` function after printing out or logging the error information.
- Else if the `SceMiEC *` pointer passed to the function is `NULL`, and no error handler is registered, an `SceMiEC` structure is constructed and passed to a default error handler. The default error handler attempts to print a message to the console and to a log file and then calls `abort()`.

It is important to note that this error handling facility only supports irrecoverable errors. This means that if an error is returned through the `SceMiEC` object, either via a handler or a return object, that there is no point in continuing with the co-modeling session. Any calls that support returning a recoverable error status should return that status using a separate, dedicated return argument.

A second point to note is that the `Message` text filled out in the error structure is meant to fully describe the nature of the error and can be logged or displayed to the console verbatim by the application error handling code. The `Culprit` text is the name of the errant API function and can optionally be added to the message that is displayed or logged.

Because every API call returns either a success or fatal error status, and because the detailed nature of errors is fully described within the returned error message, the `SceMiErrorType` enum has only two values pertaining to success (`SceMiOK`) or failure (`SceMiError`). The `SceMiEC::Type` returned from API functions to the caller can be either of these two values depending on whether the call was a success or a failure. However the `SceMiEC::Type` passed into an error handler will, by definition, always have the value `SceMiError`, since otherwise the error handler would not have been called in the first place.

The optional `Id` field can be used further classify different major error types, or even to tag each distinct error message with a unique integer identifier.

4.3.2.2 class `SceMiIC` - Informational Status and Warning Handling (Info Handling)

The SCE-MI also provides a means of conveying warnings and informational status messages to the application.

Like error handling, *info handling* is done with callback functions and a special structure that is used to convey the warning information.

```
enum SceMiInfoType {
    SceMiInfo,
    SceMiWarning,
    SceMiNonFatalError
};

struct SceMiIC {
    const char *Originator;
    const char *Message;
    SceMiInfoType Type;
    int Id;
};

typedef void (*SceMiInfoHandler)(void *context, SceMiIC *ic);

static void
SceMi::RegisterInfoHandler(
    SceMiInfoHandler infoHandler,
    void *context );
```

This method registers an optional info handler with the SCE-MI that is to be called in the event that a warning or informational status message occurs.

When any SCE-MI operation encounters a warning or wishes to issue an informational message, the following procedure is followed:

- If an info handler was registered, that info handler is called from within the API call that wants to issue the warning. The info handler is passed an internally allocated `SceMiIC` structure filled out with the warning information. In this *info handler callback approach*, the user-defined code within the handler can convey the warning to the user in a manner that is appropriate for that application. For example, it can be displayed to the console, or logged to a file, or both.
- Else if no info handler is registered, a `SceMiIC` structure is constructed and passed to a default, implementation defined error handler. The default error handler can attempt to print a message to the console and/or to a log file in an implementation specific format.

It is important to note that the `Message` text filled out in the error structure is meant to fully describe the nature of the info message and can be logged or displayed to the console verbatim by the application's warning and info handling code. The `Originator` text is the name of the API function that detected the message and can optionally be added to the message that is displayed or logged.

The `SceMiInfoType` is an extra piece of information that indicates if the message is a warning or just some informational status. An additional category called `SceMiNonFatalError` can be used to log all error conditions leading up to a fatal error. The final fatal error message should always be logged using a `SceMiEC` structure and `SceMiErrorHandler` function so that an abort sequence is properly handled (see section 4.3.2.1 on page 62). In addition, the info message can optionally be tagged with a unique identifying integer specified in the `Id` field.

4.3.2.3 Memory Allocation Semantics

Regarding memory allocation semantics of the SCE-MI API, the following rules apply:

- *Anything constructed by the user is the user's responsibility to delete.*

- *Anything constructed by the API is the API's responsibility to delete.*

So this means that any object, such as `SceMiMessageData`, that is created by the application using that object's constructor, must be deleted by the application when it is no longer in use. Some objects, such as `SceMiMessage[In/Out]PortProxy` objects, are constructed by the API then handed over to the application as pointers. Those objects must not be deleted by the application. Rather, they are deleted when the entire interface is shut down during the call to `SceMi::ShutDown()`.

Similarly, non-null `SceMiEC` structures that are passed to functions are assumed to be allocated and deleted by the application. If a NULL `SceMiEC` pointer is passed to a function and an error occurs, the API allocates the structure to pass to the error handler and therefore is responsible for freeing it.

4.3.3 class `SceMi` - SCE-MI Software Side Interface

This is the singleton object that represents the software side of the SCE-MI infrastructure itself. Operations global to the interface are performed using methods of this class.

4.3.3.1 Version Discovery

```
static int  
SceMi::Version(  
    const char *versionString );
```

This method allows an application to make queries about the version prior to initializing the SCE-MI that gives it its best chance of specifying a version that it is compatible with. A series of calls can be made to this function until a compatible version is found. With each call, the application can pass version numbers corresponding to those it is familiar with and the SCE-MI can respond with a version handle that is compatible with the queried version. This handle can then be passed onto the initialization call described in the next section.

If the given version string is not compatible with version of the SCE-MI that the application is interfacing to, a -1 is returned. At this point, the application has the option of aborting with a fatal error or attempting other versions that it might also know how to use.

This process is sometimes referred to as *mutual discovery*.

```
versionString
```

This argument is of the form "`<majorNum>.<majorNum>.<minorNum>`" and can be obtained by the application code from the header file of a particular SCE-MI installation.

4.3.3.2 Initialization

```
static SceMi *  
SceMi::Init(  
    int version,  
    SceMiParameters *parameters,  
    SceMiEC *ec=NULL );
```

This call is the constructor of the SCE-MI interface. It gives access to all the other global methods of the interface.

The return argument is a pointer to an object of class `SceMi` on which all other methods can be called.

version

This input argument is the version number returned by the `::Version()` method described in the previous section. An error results if the version number is not compatible with the SCE-MI infrastructure being accessed.

parameters

This input argument is a pointer to the parameter block object (`class SceMiParameters`) that is initialized from the parameter file generated by the infrastructure linker. See section 4.3.4 on page 70 for a description of how this object is obtained.

4.3.3.3 Shutdown

```
static void
SceMi::Shutdown(
    SceMi *sceMi,
    SceMiEC *ec=NULL );
```

This is the destructor of the SCE-MI infrastructure object that must be called when connection to the interface is to be terminated. This call is the means by which graceful decoupling of the hardware side and the software side is achieved. It is also during this call that termination (`close()`) callbacks registered by the application are called.

4.3.3.4 Message Input Port Proxy Binding

```
SceMiMessageInPortProxy *
SceMi::BindMessageInPort(
    const char *transactorName,
    const char *portName,
    const SceMiMessageInPortBinding *binding,
    SceMiEC *ec=NULL );
```

This call searches the list of input ports learned from the parameter file that is generated during infrastructure linkage for one whose names match the `transactorName` and `portName` arguments. If one is found, an object of `class SceMiMessageInPortProxy` is constructed to serve as the proxy interface to that port and the pointer to the constructed object is returned to the caller to serve all future accesses to that port. It is considered an error if no match is found.

transactorName, portName

These arguments uniquely identify a specific message input port in a specific transactor on the hardware side to which the caller wishes to bind. These names must be the path names (described in 4.2.1 on page 59) expressed in the syntax of the HDL language that the hardware side bridge netlist is expressed in.

binding

The binding argument is a pointer to an object defined as follows:

```
struct SceMiMessageInPortBinding {
    void *Context;
    void (*IsReady)(void *context);
    void (*Close)(void *context);
};
```

whose data members are used for the following:

Context

The application is free to use this pointer for any purposes it wishes. Neither `class SceMi` nor `class SceMiMessageInPortProxy` interpret this pointer other than to store it and pass it when calling either the `IsReady()` or `Close()` callbacks.

IsReady()

This is the function pointer for the callback that is to be called whenever an input ready notification has been received from the hardware side. This call signals that it is okay to send a new message to the input port. If this pointer is given as a `NULL`, the SCE-MI assumes that this port does not need to deploy input ready notification on this particular channel. See 4.1.2.2 on page 48 for a detailed description of the input ready callback.

Close()

This is a termination callback function pointer. It is called during destruction of the SCE-MI. This pointer can also be optionally specified as `NULL`.

4.3.3.5 Message Output Port Proxy Binding

```
SceMiMessageOutPortProxy *
SceMi::BindMessageOutPort(
    const char *transactorName,
    const char *portName,
    const SceMiMessageOutPortBinding *binding,
    SceMiEC *ec=NULL );
```

This call searches the list of output ports learned from the parameter file that was generated during infrastructure linkage for one whose names match the `transactorName` and `portName` argument. If one is found, an object of `class SceMiMessageOutPortProxy` is constructed to serve as the proxy interface to that port and the handle to the constructed object is returned to the caller to serve all future accesses to that port. It is considered an error if no match is found.

transactorName, portName

These arguments uniquely identify a specific message output port in a specific transactor on the hardware side to which the caller wishes to bind. These names must be the path names (described in 4.2.1 on page 59) expressed in the syntax of the HDL language that the hardware side bridge netlist is expressed in.

binding

The binding argument is a pointer to an object defined as follows:

```
struct SceMiMessageOutPortBinding {
    void *Context;
    void (*Receive)(
        void *context,
        const SceMiMessageData *data);
    void (*Close)(void *context);
};
```

whose data members are used for the following:

Context

The application is free to use this pointer for any purposes it wishes. Neither `class SceMi` nor `class`

`SceMiMessageOutPortProxy` interpret this pointer other than to store it and pass it when calling either the `IsReady()` or `Close()` callbacks.

Receive()

This is the function pointer for the receive callback that is to be called whenever an output message arrives on the port. This callback is required on every output port proxy so it is considered an error if this function pointer is given as a NULL. See 4.3.7.1 on page 77 for more information about how receive callbacks process output messages.

Close()

This is a termination callback function pointer. It is called during destruction of the SCE-MI. This pointer can also be optionally specified as NULL.

4.3.3.6 Service Loop

```
typedef int (*SceMiServiceLoopHandler)( void *context, bool pending );

int
SceMi::ServiceLoop(
    SceMiServiceLoopHandler g=NULL,
    void *context=NULL,
    SceMiEC *ec=NULL );
```

This is the main workhorse method that yields CPU processing time to the SCE-MI. In both single-threaded and multi-threaded environments, calls to this method gives the SCE-MI an opportunity to service all its port proxies and check for arriving messages or messages that are pending to be sent. It is also inside this call that the SCE-MI dispatches any input ready or receive callbacks that might be needed. The underlying transport mechanism that supports the port proxies must respond in a relatively timely manner to messages enqueued on the input or output port proxies. Since these messages cannot be handled until a call to `::ServiceLoop()` is made, it is advisable that applications call this function frequently.

The return argument is the number of output messages that arrived and were processed since the last call to `::ServiceLoop()`.

g()

If `g` is NULL, check for transfers to be performed and dispatch them, returning immediately afterwards. If `g` is non-NULL, enter a loop of performing transfers and calling `g()`. When `g()` returns 0, return from the loop. When `g()` is called, an indication of whether there is at least one message pending is made with the pending flag. The `context` argument to `g()` is the pointer which is passed as the `context` argument to `::ServiceLoop()`.

context

Context argument to be passed to the `g()` function.

4.3.3.6.1 Example of Using the `g()` Function to Return on Each Call to `::ServiceLoop()`

There are several different ways to use the `g()` function.

Some applications might like to force a return from the `::ServiceLoop()` call after processing each message. The `::ServiceLoop()` call will always guarantee that a separate call is made to the `g()`

function for each message processed. In fact, it is possible to force `::ServiceLoop()` to return back to the application once per message by having the `g()` function return a 0.

So even if all `g()` does is return 0 as follows,

```
int g( void /*context*/, bool /*pending*/ ){ return 0; }
```

the application will force a return from `::ServiceLoop()` for each processed message.

Note, in this case, the `::ServiceLoop()` will not block because it also returns even if no message was found (i.e. `pending == 0`). So basically, `::ServiceLoop()` will return no matter what in this case. No messages or 1 message.

4.3.3.6.2 Example of Using the `g()` Function to Block `::ServiceLoop()` Until Exactly 1 Message

An application can use the `g()` function to put `::ServiceLoop()` into a blocking mode rather than its default polling mode. The `g()` function can be written to cause `::ServiceLoop()` to block until it gets 1 message then return on the message it got:

This is done by making use of the `pending` argument to the `g()` function. This argument simply indicates if there is a message to be processed or not. Here is how it would be written in this case:

```
int g( void /*context*/, bool pending ){  
    return pending == true ? 0 : 1 }  
}
```

What this does it to block until a message occurs then return on processing the first message.

4.3.3.6.3 Example of Using the `g()` function to Block `::ServiceLoop()` Until At Least 1 Message

Alternatively, suppose the application would like `::ServiceLoop()` to block until at least 1 message occurs, then return only after all the currently pending messages have been processed.

To do this the application can define a `haveProcessedAtLeast1Message` flag as follows:

```
int haveProcessedAtLeast1Message = 0;
```

Call `::ServiceLoop()` giving the `g()` function and this flag's address as the context:

```
...
haveProcessedAtLeast1Message = 0;
sceMi->ServiceLoop( g, &haveProcessedAtLeast1Message );
...
```

Now define the `g()` function as follows:

```
int g( void *context, bool pending ){
    int *haveProcessedAtLeast1Message = (int *)context;
    if( pending == 0 )
        // If no more messages, kick out of loop if at least
        // one previous message has been processed, otherwise
        // block until the first message arrives.
        return *haveProcessedAtLeast1Message ? 0 : 1;
    else {
        *haveProcessedAtLeast1Message = 1;
        return 1;
    }
}
```

In conclusion, depending on precisely what type of operation of `::ServiceLoop()` is desired, the `g()` function can be tailored accordingly.

4.3.4 class `SceMiParameters` - Parameter Access

This class provides a generic API that can be used by application code to access the interface parameter set described in section 4.2.1 on page 59. It is basically initialized with the contents of the parameter file that is generated during infrastructure linkage. It provides accessors that facilitate the reading and possibly overriding of parameters and their values. All SCE-MI required parameters are read-only because their values are automatically determined by the infrastructure linker by analyzing the user supplied netlist. Implementation specific parameters can be read-only or read-write as required by the implementation. All parameters in a `SceMiParameters` object must be overridden before that object is passed to the `SceMi::Init()` call to construct the interface (see 4.3.3.2 on page 65). Overriding parameters afterwards has no effect.

4.3.4.1 Parameter Set

While the format of the parameter file is implementation specific, the set of parameters required by the SCE-API and the methods used to access them must conform to the specifications described in this section. For purposes of access, the parameter set shall be organized as a database of *attributed objects*, where each object instance is decorated with a set of attributes expressed as name/value pairs. There can be zero or more instances of each object kind. The API shall provide a simple accessor to return the number of objects of a given kind, and read and write accessors (described below) to allow reading or overriding attribute values of specific objects.

The objects in the database are composed of the set of necessary interfacing components that interface the SCE-MI infrastructure to the application. For example, there is a distinct object instance for each message port and a distinct object instance representing each defined clock in the system. Attributes of each of the objects then represent, collectively, the parameters that uniquely characterize the dimensions and constitution of the interface components needed for a particular application.

So, for example, a system that requires 1 input port, 2 output ports and 2 distinct clocks is represented with 5 objects parametrized such that each port object has name and width attributes, each clock object has ratio, duty cycle attributes, etc. These objects and their attributes precisely and fully describe the interfacing requirements between that application and the SCE-MI infrastructure.

The following table gives the minimal, predefined set of objects and attributes required by the SCE-MI. Additional objects and attributes can be added by implementations. For example, there may be a single, implementation specific object representing the entire SCE-MI infrastructure facility itself. The attributes of this singleton object might be the set of implementation specific parameters that an implementor of the SCE-MI needs to allow the user to specify. For more details of attribute meanings, see section 4.2.1 on page 59.

Object Kind	Attribute Name	Attribute Value Type	Meaning
MessageInPort	TransactorName	String	Name of the transactor enclosing the message input port.
	PortName	String	Name of the message input port.
	PortWidth	Integer	Width of the message input port in bits.
MessageOutPort	TransactorName	String	Name of the transactor enclosing the message output port.
	PortName	String	Name of the message output port.
	PortWidth	Integer	Width of the message output port in bits.
	PortPriority	Integer	Priority of the message output port.
Clock	ClockName	String	Name of the clock.
	RatioNumerator	Integer	Numerator (“fast” clock cycles) of clock ratio.
	RatioDenominator	Integer	Denominator (“this” clock cycles) of clock ratio.
	DutyHi	Integer	High cycle percentage of duty cycle.
	DutyLo	Integer	Low cycle percentage of duty cycle.
	Phase	Integer	Phase shift as percentage of duty cycle.
	ResetCycles	Integer	Number of controlled clock cycles of reset.
ClockBinding	TransactorName	String	Name of the transactor that contributes to the control of this clock.
	ClockName	String	Name of the clock that this transactor helps control.

For simplicity, values can be either signed integer or string values. More complex data types can be derived by the application code from string values. Each attribute definition of each object kind implies a specific value type.

4.3.4.2 Parameter Set Semantics

Although the accessors provided by the `SceMiParameters` class directly provide the information given in the above table, other implied parameters can be easily derived by the application. Following are some of the implied parameters and how they are determined:

- `ClockBinding` objects indicate the total number of transactor - clock control macro combinations. From the `ClockBinding` objects, one can ascertain the number of distinct contributors to the control of a given clock, as well as the number of distinct transactors in the system.
- The number of transactors in the system is determined by counting the number of distinct `TransactorName`'s encountered in the `ClockBinding` objects.
- The number of controlled clocks is determined by reading the number of `Clock` objects (using the `::NumberOfObjects()` accessor described below).
- The number of input and output ports are determined by reading the number of `MessageInPort` and `MessageOutPort` objects, respectively.

In addition, the following semantics characterize the parameter set:

- Transactor names are absolute hierarchical path names, and must conform to the syntax of the HDL language that the bridge netlist is expressed in.
- Port names are relative hierarchical path names (relative to the enclosing transactor), and must conform to the syntax of the HDL language that the bridge netlist is expressed in.
- Clock names are identifiers, not path names, and must conform to identifier naming syntax of the HDL language that the bridge netlist is expressed in.

4.3.4.3 Constructor

```
SceMiParameters::SceMiParameters(  
    const char *paramsFile,  
    SceMiEC *ec=NULL );
```

The constructor constructs an object containing all the default values of parameters and then overrides them with any settings it finds in the specified parameter file. All parameters, whether specified by the user or not must have default values. Once constructed, parameters can be further overridden procedurally.

paramsFile

This is the name of the file generated by the infrastructure linker that contains all the parameters derived from the user's hardware side netlist. This name can be either a full pathname to a file or pathname relative to the local directory.

4.3.4.4 Destructor

```
SceMiParameters::~SceMiParameters()
```

This is the destructor for the parameters object.

4.3.4.5 Accessors

```
unsigned int
SceMiParameters::NumberOfObjects(
    const char *objectKind,
    SceMiEC *ec=NULL ) const;
```

This accessor returns the number of instances of objects of the specified `objectKind` name.

```
int
SceMiParameters::AttributeIntegerValue(
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    SceMiEC *ec=NULL ) const;
```

```
const char *
SceMiParameters::AttributeStringValue(
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    SceMiEC *ec=NULL ) const;
```

These two accessors read return an integer or string attribute value.

```
void
SceMiParameters::OverrideAttributeIntegerValue(
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    int value,
    SceMiEC *ec=NULL );
```

```
void
SceMiParameters::OverrideAttributeStringValue(
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    const char *value,
    SceMiEC *ec=NULL );
```

These two accessors override an integer or string attribute value. It is considered an error to attempt to override any of the object attributes shown in the table in section 4.3.4.1 on page 70 or any implementation specific attributes designated as read-only.

The following argument descriptions generally apply to all the accessors shown above:

objectKind

Name of the kind of object for which an attribute value is being accessed. It is considered an error to pass an unrecognized `objectKind` name to any of the accessors.

index

Index of the instance of the object for which an attribute value is being accessed. It is considered an error if the `index` \geq the number returned by the `::NumberOfObjects()` accessor.

attributeName

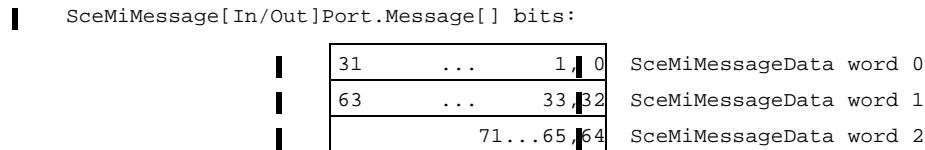
Name of the attribute whose value is being read or overwritten. It is considered an error if the `attributeName` does not identify one of the attributes allowed for the given `objectKind`.

value

Returned or passed in `value` of the attribute being read or overridden respectively. Two overloaded variants of each accessor are provided. One for string values and one for integer values.

4.3.5 class `SceMiMessageData` - Message Data Object

The class `SceMiMessageData` represents the vector of message data that can be transferred from a `SceMiMessageInPortProxy` on the software side to its associated `SceMiMessageOutPort` on the hardware side, or from a `SceMiMessageOutPort` on the hardware side to its associated `SceMiMessageOutPortProxy` on the software side. The message data payload is represented as a fixed length array of `SceMiU32` data words large enough to contain the bit vector being transferred to or from the hardware side message port. For example if the message port had a width of 72 bits, the following diagram shows how the those bits would be organized in the data array contained inside the `SceMiMessageData` object:

**4.3.5.1 Constructor**

```
SceMiMessageData::SceMiMessageData(
    const SceMiMessageInPortProxy &messageInPortProxy,
    SceMiEC *ec=NULL );
```

Construct a message data object whose size matches the width of the specified input port. The constructed message data object can only be used for sends on that port (or another of identical size) or an error will result.

4.3.5.2 Destructor

```
SceMiMessageData::~SceMiMessageData()
```

Destruct the object, free data array.

4.3.5.3 Accessors

```
unsigned int  
SceMiMessageData::WidthInBits() const;
```

Return the width of the message in terms of number of bits.

```
unsigned int  
SceMiMessageData::WidthInWords() const;
```

Return the size of the data array in terms of number of `SceMiU32` words.

```
void  
SceMiMessageData::Set( unsigned int i, SceMiU32 word, SceMiEC *ec = NULL );
```

Set word element `i` of array to `word`.

```
void  
SceMiMessageData::SetBit( unsigned int i, int bit, SceMiEC *ec = NULL );
```

Set bit element `i` of the message vector to 0 if `bit == 0`, 1 otherwise. It is an error if `i >= ::WidthInBits()`.

```
void  
SceMiMessageData::SetBitRange(  
    unsigned int i, unsigned int range, SceMiU32 bits, SceMiEC *ec = NULL );
```

Set `range` bit elements whose LSB's start at bit element `i` of the message vector to the value of `bits`. It is an error if `i+range >= ::WidthInBits()`.

```
SceMiU32  
SceMiMessageData::Get( unsigned int i, SceMiEC *ec = NULL ) const;
```

Return the word at slot `i` in the array. It is an error if `i >= ::WidthInWords()`.

```
int  
SceMiMessageData::GetBit( unsigned int i, SceMiEC *ec = NULL ) const;
```

Return the value of bit element `i` in the message vector. It is an error if `i >= ::WidthInBits()`.

```
SceMiU32  
SceMiMessageData::GetBitRange( unsigned int i, unsigned int range, SceMiEC *ec = NULL )  
const;
```

Return the value of `range` bit elements whose LSB's start at `i` of the message vector. It is an error if `i+range >= ::WidthInBits()`.

```
SceMiU64  
SceMiMessageData::CycleStamp() const;
```

The SCE-MI supports a feature called *cycle stamping*. Each output message sent to the software side is stamped with the number of cycles of the 1/1 controlled clock elapsed since the beginning of emulation time. This provides a convenient way for applications to keep track of elapsed cycles in their respective transactors as the simulation proceeds. The returned value is an absolute, 64-bit unsigned quantity.

4.3.6 class `SceMiMessageInPortProxy`

The class `SceMiMessageInPortProxy` presents to the application a proxy interface to a transactor message input port.

4.3.6.1 Sending Input Messages

```
void  
SceMiMessageInPortProxy::Send(  
    const SceMiMessageData &data,  
    SceMiEC *ec=NULL );
```

This method sends a message to the message input channel. This message appears on the hardware side as a bit vector presented to the transactor via the `SceMiMessageInPort` macro (see 4.1.2 on page 47) instance bound to this proxy.

data

Message data object containing the message to be sent.

4.3.6.2 Replacing Port Binding

```
void ReplaceBinding(  
    const SceMiMessageInPortBinding* binding,  
    SceMiEC* ec=NULL );
```

This method replaces the `SceMiMessageInPortBinding` object originally furnished to the `SceMi::BindMessageInPortProxy()` call that created this port proxy object (see 4.3.3.4 on page 66). This can be useful for replacing contexts or input ready callback functions some time after the input message port proxy has been established.

binding

New callback and context information to be associated with this message input port proxy.

4.3.6.3 Accessors

```
const char *  
SceMiMessageInPortProxy::TransactorName() const;
```

This method returns the name of the transactor that is connected to the port. This is the absolute hierarchical path name to the transactor instance expressed in the syntax of HDL language that the netlist is expressed in.

```
const char *  
SceMiMessageInPortProxy::PortName() const;
```

This method returns the port name. This is the path name to the `SceMiMessageInPort` macro instance relative to the containing transactor expressed in the syntax of HDL language that the netlist is expressed in.

```
unsigned  
SceMiMessageInPortProxy::PortWidth() const;
```

This method returns the port width. This is the value of the `PortWidth` parameter that was passed to the associated `SceMiMessageInPort` instance on the hardware side.

4.3.6.4 Destructor

There is no public destructor for this class. Destruction of all message input ports shall automatically occur when the `SceMi::ShutDown()` function is called.

4.3.7 class `SceMiMessageOutPortProxy`

The class `MessageOutPortProxy` presents to the application a proxy interface to the transactor message output port.

4.3.7.1 Receiving Output Messages

There are no methods on this object specifically for reading messages that arrive on the output port proxy. Rather, that operation is handled by the receive callbacks. Receive callbacks are registered with an output port proxy when it is first bound to the channel (see 4.3.3.5 on page 67). The prototype for the receive callback is as follows:

```
void (*Receive)( void *context, const SceMiMessageData *data );
```

When called, the receive callback is passed a pointer to a class `SceMiMessageData` object (see section 4.2.1.1 on page 61) that contains the content of the received message. It is also passed the context pointer. The context pointer is typically a pointer to the object representing the software model that is interfacing to the port proxy. Proper usage of this callback is to process the data quickly and return as soon as possible. The reference to the `SceMiMessageData` is of limited lifetime and ceases to exist once the callback returns and goes out of scope.

Typically in a SystemC context, the callback might do some minor manipulation to the context object then immediately return and let a suspended thread resume and do the main processing of the received transaction.

Notice that no `SceMiEC * error status` object is passed to the call. This is because if an error occurs within the `SceMi::ServiceLoop()` function (from which the receive callback is normally called), the callback is never called in the first place and standard error handling procedures (see 4.3.2.1 on page 62) are followed by the service loop function itself. If an error occurs inside the receive callback, by implication it is an application error not an SCE-MI error and thus it is the responsibility of the application to handle the error by perhaps setting a flag in the context object before returning from the callback.

4.3.7.2 Replacing Port Binding

```
void ReplaceBinding(  
    const SceMiMessageOutPortBinding* binding,  
    SceMiEC* ec=NULL );
```

This method replaces the `SceMiMessageOutPortBinding` object originally furnished to the `SceMi::BindMessageOutPortProxy()` call that created this port proxy object (see 4.3.3.5 on page 67). This can be useful for replacing contexts or receive callback functions some time after the output message port proxy has been established.

binding

New callback and context information to be associated with this message output port proxy.

4.3.7.3 Accessors

```
const char *  
ScemiMessageOutPortProxy::TransactorName() const;
```

This method returns the name of the transactor that is connected to the port. This is the absolute hierarchical path name to the transactor instance expressed in the syntax of HDL language that the netlist is expressed in.

```
const char *  
ScemiMessageOutPortProxy::PortName() const;
```

This method returns the port name. This is the path name to the `ScemiMessageOutPort` macro instance relative to the containing transactor expressed in the syntax of HDL language that the netlist is expressed in.

```
unsigned  
ScemiMessageOutPortProxy::PortWidth() const;
```

This method returns the port width. This is the value of the `PortWidth` parameter that was passed to the associated `ScemiMessageOutPort` instance on the hardware side.

4.3.7.4 Destructor

There is no public destructor for this class. Destruction of all message output ports shall automatically occur when the `Scemi::ShutDown()` function is called.

4.4 Software Side Interface - C API

The SCI-MI software side also provides an ANSI standard C API. All of the subsections to follow will parallel those in the C++ API described starting in section 4.3 on page 61. It is possible to implement the C API as functions that wrap calls to methods described in the C++ API. The prototypes of those functions shall follow with a minimum of explanatory text. For full documentation about a function, see its corresponding subsection of section 4.3 on page 61.

4.4.1 Primitive Data Types

The C API has its own header file with the following minimum content:

```
typedef unsigned SceMiU32;
typedef unsigned long long SceMiU64;

typedef void SceMi;
typedef void SceMiParameters;
typedef void SceMiMessageData;
typedef void SceMiMessageInPortProxy;
typedef void SceMiMessageOutPortProxy;

typedef int (*ServiceLoopHandler)( void *context, int pending );

typedef enum {
    SceMiOK,
    SceMiError,
} SceMiErrorType;
typedef struct {
    const char *Culprit;
    const char *Message;
    SceMiErrorType Type;
    int Id;
} SceMiEC;
typedef void (*SceMiErrorHandler)(void *context, SceMiEC *ec);

typedef enum {
    SceMiInfo,
    SceMiWarning
} SceMiInfoType;
typedef struct {
    const char *Culprit;
    const char *Message;
    SceMiInfoType Type;
    int Id;
} SceMiIC;
typedef void (*SceMiInfoHandler)(void *context, SceMiIC *ic);

typedef struct {
    void *Context;
    void (*IsReady)(void *context);
    void (*Close)(void *context);
} SceMiMessageInPortBinding;
typedef struct {
    void *Context;
    void (*Receive)(
        void *context,
        const SceMiMessageData *data );
    void (*Close)(void *context);
} SceMiMessageOutPortBinding;
```

An application must include either the C API header or the C++ API header, but not both.

Note that because ANSI C does not support default argument values, the last `SceMiEC *ec` argument to each function must be explicitly passed when called, even if only to pass a NULL.

4.4.2 Miscellaneous Interface Support Issues

4.4.2.1 SceMiEC - Error Handling

```
void  
SceMiRegisterErrorHandler(  
    SceMiErrorHandler errorHandler,  
    void *context );
```

4.4.2.2 SceMiIC - Informational Status and Warning Handling (Info Handling)

```
void  
SceMiRegisterInfoHandler(  
    SceMiInfoHandler infoHandler,  
    void *context );
```

4.4.3 SceMi - SCE-MI Software Side Interface

4.4.3.1 Version Discovery

```
int  
SceMiVersion( const char *versionString );
```

4.4.3.2 Initialization

```
SceMi *  
SceMiInit(  
    int version,  
    const SceMiParameters *parameterObjectHandle,  
    SceMiEC *ec );
```

4.4.3.3 Shutdown

```
void  
SceMiShutdown(  
    SceMi *sceMiHandle,  
    SceMiEC *ec );
```

4.4.3.4 Message Input Port Proxy Binding

```
SceMiMessageInPortProxy *  
SceMiBindMessageInPort(  
    SceMi *sceMiHandle,  
    const char *transactorName,  
    const char *portName,  
    const SceMiMessageInPortBinding *binding,  
    SceMiEC *ec );
```

4.4.3.5 Message Output Port Proxy Binding

```
SceMiMessageOutPortProxy *  
SceMiBindMessageOutPort(  
    SceMi *sceMiHandle,  
    const char *transactorName,  
    const char *portName,  
    const SceMiMessageOutPortBinding *binding,  
    SceMiEC *ec );
```

4.4.3.6 Service Loop

```
int  
SceMiServiceLoop(
```



```
Scemi *scemiHandle,  
ScemiServiceLoopHandler g,  
void *context,  
ScemiEC *ec );
```

4.4.4 ScemiParameters - Parameter Access

4.4.4.1 Constructor

```
ScemiParameters *  
ScemiParametersNew(  
    const char *paramsFile,  
    ScemiEC *ec );
```

This function returns the handle to a parameters object.

4.4.4.2 Destructor

```
void
SceMiParametersDelete(
    SceMiParameters *parametersHandle );
```

4.4.4.3 Accessors

```
unsigned int
SceMiParametersNumberOfObjects(
    const SceMiParameters *parametersHandle,
    const char *objectKind,
    SceMiEC *ec );
```

```
int
SceMiParametersAttributeIntegerValue(
    const SceMiParameters *parametersHandle,
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    SceMiEC *ec );
```

```
const char *
SceMiParametersAttributeStringValue(
    const SceMiParameters *parametersHandle,
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    SceMiEC *ec );
```

```
void
SceMiParametersOverrideAttributeIntegerValue(
    SceMiParameters *parametersHandle,
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    int value,
    SceMiEC *ec );
```

```
void
SceMiParametersOverrideAttributeStringValue(
    SceMiParameters *parametersHandle,
    const char *objectKind,
    unsigned int index,
    const char *attributeName,
    const char *value,
    SceMiEC *ec );
```

4.4.5 SceMiMessageData - Message Data Object

4.4.5.1 Constructor

```
SceMiMessageData *
SceMiMessageDataNew(
    const SceMiMessageInPortProxy *messageInPortProxyHandle,
    SceMiEC *ec );
```

This function returns the handle to a message data object suitable for sending messages on the specified input port proxy.

4.4.5.2 Destructor

```
void  
SceMiMessageDataDelete(  
    SceMiMessageData *messageDataHandle );
```

4.4.5.3 Accessors

```
unsigned int  
SceMiMessageDataWidthInBits(  
    const SceMiMessageData *messageDataHandle );
```

```
unsigned int  
SceMiMessageDataWidthInWords(  
    const SceMiMessageData *messageDataHandle );
```

```
void  
SceMiMessageDataSet(  
    SceMiMessageData *messageDataHandle,  
    unsigned int i,  
    SceMiU32 word,  
    SceMiEC *ec );
```

```
void  
SceMiMessageDataSetBit(  
    SceMiMessageData *messageDataHandle,  
    unsigned int i,  
    int bit,  
    SceMiEC *ec );
```

```
void  
SceMiMessageDataSetBitRange(  
    SceMiMessageData *messageDataHandle,  
    unsigned int i,  
    unsigned int range,  
    SceMiU32 bits,  
    SceMiEC *ec );
```

```
SceMiU32  
SceMiMessageDataGet(  
    const SceMiMessageData *messageDataHandle,  
    unsigned int i  
    SceMiEC *ec );
```

```
int  
SceMiMessageDataGetBit(  
    const SceMiMessageData *messageDataHandle,  
    unsigned int i,  
    SceMiEC *ec );
```

```
SceMiU32  
SceMiMessageDataGetBitRange(  
    const SceMiMessageData *messageDataHandle,  
    unsigned int i,  
    unsigned int range,  
    SceMiEC *ec );
```

```
SceMiU64  
SceMiMessageDataCycleStamp(  
    const SceMiMessageData *messageDataHandle );
```

4.4.6 SceMiMessageInPortProxy - Message Input Port Proxy

4.4.6.1 Sending Input Messages

```
void  
SceMiMessageInPortProxySend(  
    SceMiMessageInPortProxy *messageInPortProxyHandle,  
    const SceMiMessageData *messageDataHandle,  
    SceMiEC *ec );
```

4.4.6.2 Replacing Port Binding

```
void SceMiMessageInPortProxyReplaceBinding(  
    SceMiMessageInPortProxy *messageInPortProxyHandle,  
    const SceMiMessageInPortBinding* binding,  
    SceMiEC* ec );
```

4.4.6.3 Accessors

```
const char *  
SceMiMessageInPortProxyTransactorName(  
    const SceMiMessageInPortProxy *messageInPortProxyHandle );  
  
const char *  
SceMiMessageInPortProxyPortName(  
    const SceMiMessageInPortProxy *messageInPortProxyHandle );  
  
unsigned  
SceMiMessageInPortProxyPortWidth(  
    const SceMiMessageInPortProxy *messageInPortProxyHandle );
```

4.4.7 SceMiMessageOutPortProxy - Message Output Port Proxy

4.4.7.1 Replacing Port Binding

```
void SceMiMessageOutPortProxyReplaceBinding(  
    SceMiMessageOutPortProxy *messageOutPortProxyHandle,  
    const SceMiMessageOutPortBinding* binding,  
    SceMiEC* ec );
```

4.4.7.2 Accessors

```
const char *  
SceMiMessageOutPortProxyTransactorName(  
    const SceMiMessageOutPortProxy *messageOutPortProxyHandle );  
  
const char *  
SceMiMessageOutPortProxyPortName(  
    const SceMiMessageOutPortProxy *messageOutPortProxyHandle );  
  
unsigned  
SceMiMessageOutPortProxyPortWidth(  
    const SceMiMessageInPortProxy *messageOutPortProxyHandle );
```

Appendix A: Glossary

Abstraction Bridge

A collection of *abstraction gasket* components that disguise a bus cycle accurate (BCA) RTL DUT model as a purely untimed model. The idea is that to the untimed testbench models, the DUT itself appears untimed (see Figure 2 on page 9) when, in fact, it is a disguised BCA model (see Figure 3 on page 11).

Abstraction Gasket

A special model that can change the level of abstraction of data flowing from its input to output and vice versa. For example, an abstraction gasket might convert an untimed transaction to a series of cycle accurate events. Or it might assemble a series of events into a single message. Examples of abstraction gaskets are CoWare or SystemC *BCASH models*, and Ikos *transactors*.

Behavioral Model

See definition for *untimed model*.

Bridge Netlist

The *bridge netlist* is the top level of the user supplied netlist of components making up the *hardware side* of a co-modeling process. The components typically found instantiated immediately under the *bridge netlist* are *transactors*, *DUT*, and `SceMiClockPort` macros. By convention, the top level netlist module supplied by the user to the *infrastructure linker* is often called `Bridge` and, for Verilog, is placed in a file called `Bridge.v`.

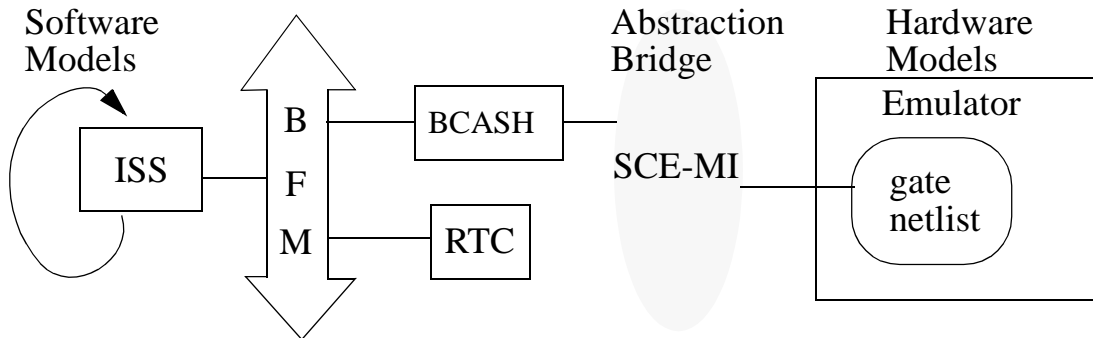
Co-Emulation

A shorthand notation for *co-emulation modeling* also known as *co-modeling*. See definition for *co-modeling*.

Co-Modeling

Although it has broader meanings outside this document, within this document co-modeling specifically refers to *transaction oriented co-modeling* in contrast to a broader definition of co-modeling that might include *event oriented co-modeling*. Also known as *co-emulation modeling*, transaction oriented co-modeling describes the process of modeling and simulating a mixture of software models represented with an *untimed* level of abstraction, simultaneously executing and inter-communicating through an *abstraction bridge*, with hardware models represented with the *RTL* level of abstraction, running on an emulator. The following diagram depicts such a configuration where the SCE-MI is being used as the abstraction bridge. See

section 1.4 on page 3 for meanings of acronyms. Another illustration can be seen in Figure 1 on page 6.



Controlled Clock (cclock)

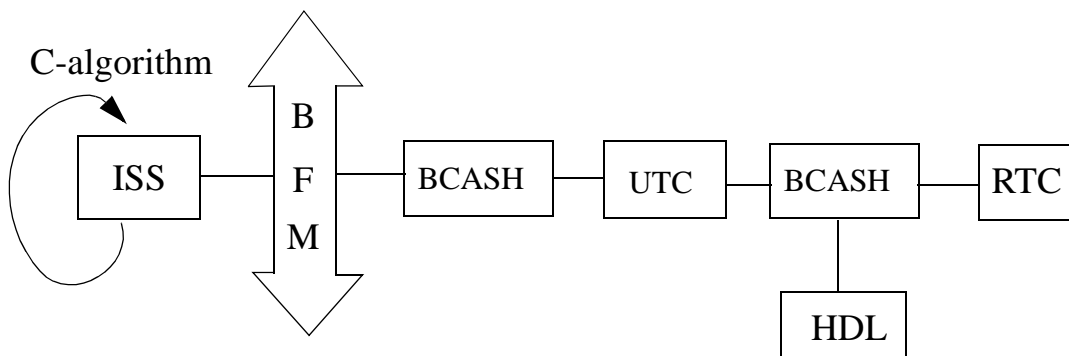
The clock that drives the DUT and can be disabled by any transactor during operations that would result in erroneous operation of the DUT if it is clocked. When performing such operations, any transactor can “freeze” *controlled time* long enough to complete the operation before allowing clocking of the DUT to resume. The term *cclock* is often used throughout this document as a synonym for *controlled clock*.

Controlled Time

Time which is advanced by the *controlled clock* and frozen when the *controlled clock* is suspended by one or more transactors. Operations occurring in *uncontrolled time* while controlled time is frozen appear between *controlled clock* cycles.

Co-Simulation

Execution of software models modeled with different levels of abstraction that interact with each other through *abstraction gaskets* similar to CoWare’s BCASH models. The following diagram illustrates such a configuration. See section 1.4 on page 3 for meanings of acronyms. The key difference between co-simulation and co-emulation is that the former typically couples software models to a traditional HDL simulator interface through a proprietary API whereas the latter couples software models to an emulator through an optimized transaction oriented message passing interface such as SCE-MI.



Cycle Stamping

A process by which messages are tagged with the number of elapsed counts of the fastest controlled clock in the hardware side of a co-modeled design.

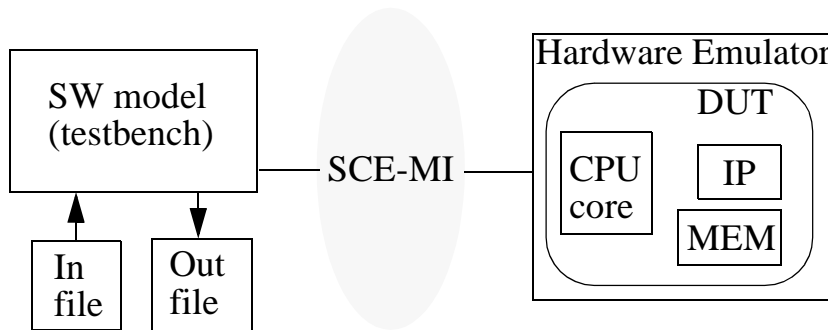
Don't Care Duty Cycle

A *posedge active don't care duty cycle* is a way of specifying a duty cycle where the user only cares about the posedge of the clock and does not care about where in the period the negedge falls, particularly in relation to other clocks in a functional simulation. In such a case, the `DutyHi` parameter is given as a 0. The `DutyLo` can be given as an arbitrary number of units that represent the whole period such that the `Phase` offset can still be expressed as a percentage of the period (i.e. `DutyHi+DutyLo`). See 4.1.4.1 on page 52 for more details.

A *negedge active don't care duty cycle* is a way of specifying a duty cycle where the user only cares about the negedge of the clock and does not care about where in the period the posedge falls, particularly in relation to other clocks in a functional simulation. In such a case, the `DutyLo` parameter is given as a 0. The `DutyHi` can be given as an arbitrary number of units that represent the whole period such that the `Phase` offset can still be expressed as a percentage of the period (i.e. `DutyHi+DutyLo`). See 4.1.4.1 on page 52 for more details.

DUT

Device or Design Under Test that might be modeled in hardware and stimulated and responded to by a software testbench through an *abstraction bridge* such as the SCE-MI.

**DUT Proxy**

Software model which acts as a “socket” that disguises an *abstraction bridge* to look to the rest of a software testbench like an untimed DUT model. If the abstraction bridge is the SCE-MI, the job of the DUT proxy is to encapsulate the *message port proxy* interfaces to the message channels in the interface. See Figure 3 on page 11 for an illustration of DUT proxies.

Hardware Model

Model of a block that has a structural representation (i.e., as a result of synthesis or a gate netlist generated by an appropriate tool) that is mapped onto the *hardware side* of a co-modeling process (i.e., an emulator or other hardware simulation platform). It could also be real silicon (i.e., a CPU core or memory chip) plugged into an emulator or simulation accelerator.

Hardware Side

See definition for *software side*.

Infrastructure Linkage Process

The process that reads a user description of the hardware, namely the source or *bridge* netlist that describes the interconnect between the transactors, the DUT, and the SCE-MI interface components, and compiles that netlist into a form suitable for executing in a co-modeling session. Part of this compile process can involve adding additional structure to the bridge netlist that properly interfaces the user supplied netlist to the SCE-MI infrastructure implementation components.

Macros

Implementation components provided by the hardware emulator vendor to implement the hardware side of the SCE-MI infrastructure. The required macros as defined in this specification are the `ScemiMessageInPort`, `ScemiMessageOutPort`, `ScemiClockControl`, and `ScemiClockPort` macros.

Message

A unit of data of arbitrary size and abstraction to be transported over a channel. Messages are generally not associated with specific clocked events but may trigger or result from, many clocks of event activity. Note that, for the most part, the term *message* can be used interchangeably with *transaction*. However, in some contexts, *transaction* could be thought of as including infrastructure overhead content in addition to user payload data (and handled at a lower layer of the interface) whereas the term *message* denotes only user payload data.

Message Channel

A two ended conduit of messages between the software and hardware sides of an *abstraction bridge*.

Message Port

The *hardware side* end of a *message channel*. A message port is the means by which *transactors* gain access to messages being sent across the channel to or from the *software side*.

Message Port Proxy

The *software side* end of a *message channel*. A message port proxy is the means by which *DUT proxies* or other software models gain access to messages being sent across the channel to or from the *hardware side*.

Negedge

See definition for *posedge*.

Posedge

Although this is a Verilog specific term that refers to the rising edge of a clock, it is used generically throughout this document to refer to the same thing. The same applies to *negedge* referring to the falling edge of a clock.

Service Loop

This is the method that provides a means for a set of software models running on a host workstation to yield access to the SCE-MI software side so that any pending input or output messages on the channels can be serviced. This method should be called frequently by the software, throughout the co-modeling session in order to avoid backup of messages and minimize the possibility of system deadlock. In multi-threaded environments, it makes sense to place the service loop call in its own continually running thread.

Software Model

Model of a block (hardware or software) that is simulated on the *software side* of a co-

modeling session (i.e., the host workstation). Such a model can be an algorithm (C or C++) running on an ISS or a hardware model that is modeled using an appropriate language environment such as CoWare, SystemC, or an HDL simulator.

Software Side

This term refers to the portion of a user's design which, during a co-modeling session, runs on the host workstation as opposed to the portion running on the emulator which is referred to as the *hardware side*. The SCE-MI infrastructure itself is also considered to have *software side* and *hardware side* components.

Structural Model

A netlist of *hardware models* or other *structural models*. Because this definition is recursive, it is implied that structural models have hierarchy.

Transaction

See definition for *message*.

Transactor

A form of *abstraction gasket*. The job of a transactor is to decompose an untimed transaction to a series of cycle accurate clocked events, or, conversely, to compose a series of clocked events into a single message. When receiving messages, transactors have the ability to "freeze" *controlled time* long enough to allow message decomposition operations to complete before presenting clocked data to a DUT. And when sending messages, they can freeze controlled time and allow message composition operations to complete before new clocked data is flooded in from a DUT.

Uncontrolled Clock (uclock)

Free running system clock internally generated by the SCE-MI infrastructure used only within *transactor* modules to advance states in *uncontrolled time*. The term *uclock* is often used throughout this document as a synonym for *uncontrolled clock*.

Uncontrolled Reset

This is the system reset internally generated by the SCE-MI infrastructure and used only with transactor modules. This signal is high at the beginning of simulated time and transitions to a low an arbitrary (implementation dependent) number of uclocks later. It can be used to reset a transactor. The controlled reset is generated exactly once by the SCE-MI hardware side infrastructure at the very beginning of a co-modeling session.

Uncontrolled Time

Time that is advanced by the *uncontrolled clock* even when the *controlled clock* is suspended (and *controlled time* is frozen).

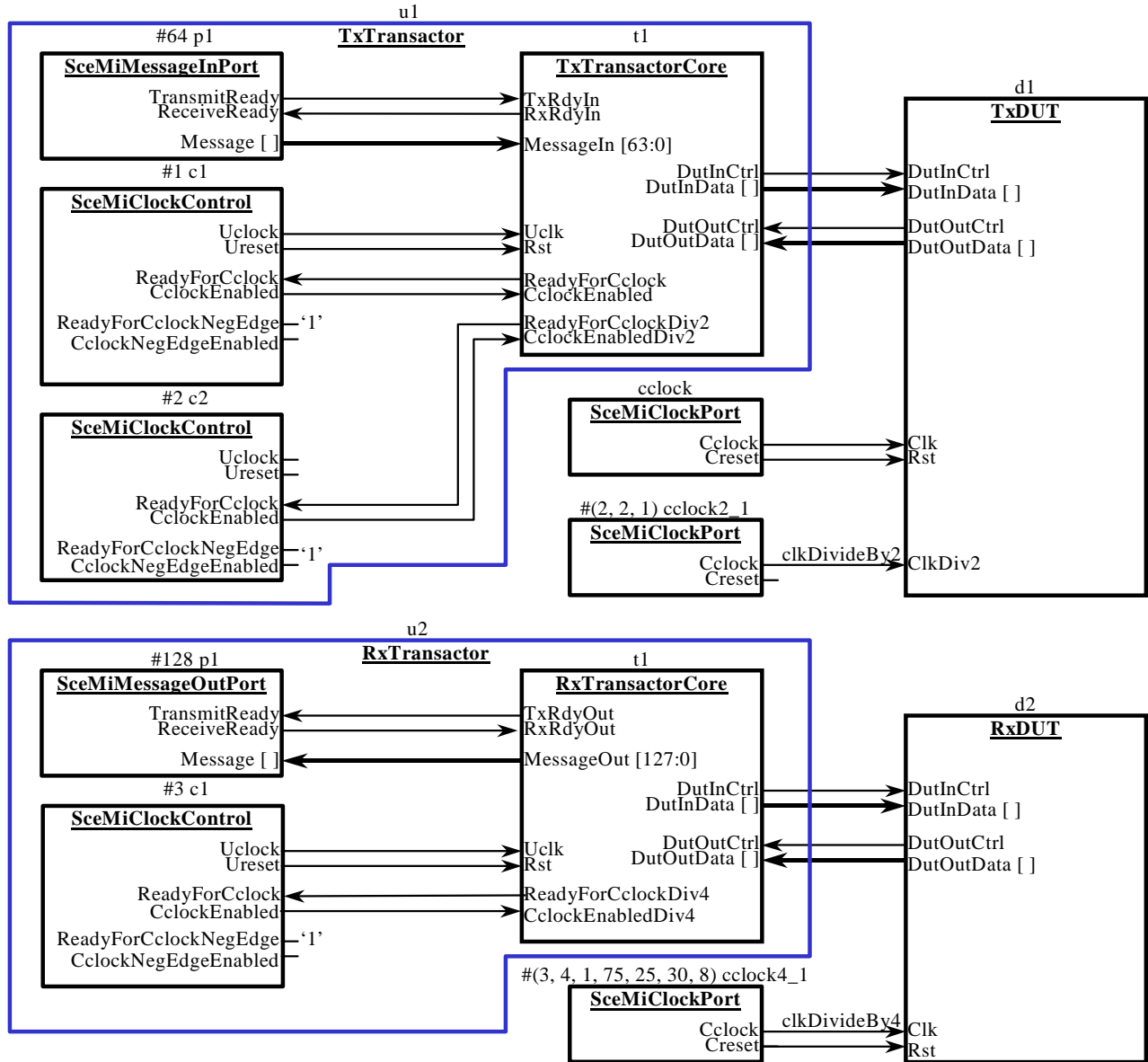
Untimed Model

A block that is modeled algorithmically at the functional level and exchanges data with other models in the form of messages. An untimed model has no notion of a clock. Rather, its operation is triggered by arriving messages and can, in turn, trigger operations in other untimed models by sending messages.

Appendix B: Multiclock Hardware Side Interface Example

The following diagram shows the top level structure of a simple multiclock, multitransactor example:

FIGURE 18. Multiclock, Multitransactor Example



This design demonstrates the following points:

- Three ClockPort instances define clocks named `cclock`, `cclock2_1`, and `cclock4_1`.
- Because no parameters are given with the `SceMiClockPort` instance `cclock`, all default parameters are used. This means that `cclock` has a `ClockNum=1`, a clock ratio of `1/1`, a *don't care duty cycle*, a

phase shift of 0, and the controlled reset it supplies has an active duration of 8 controlled clock cycles.

- The `cclock2_1` instance of `SceMiClockPort` overrides the first three parameters and leaves the rest at their default values. This means the `cclock2_1` has a `ClockNum=2`, a clock ratio of 2/1 (i.e., a “divide-by-2” clock), a duty cycle of 50%, a phase shift of 0, and an 8 clock cycle reset duration.
- The `cclock4_1` instance of `SceMiClockPort` has a `ClockNum=3`, a clock ratio of 4/1 (i.e., a “divide-by-4” clock), a duty cycle of 75%, a phase shift of 30% of the clock period, and an 8 clock cycle reset duration.
- The `TxTransactor` transactor model, named `Bridge.u1`, controls clocks `cclock` and `cclock2_1` since its `SceMiClockControl` macro instances have `ClockNum=1` and `ClockNum=2`, respectively.
- This `TxTransactor` model interfaces to a message input port called `p1` that is parametrized to a bit-width of 64.
- The `RxTransactor` transactor model, named `Bridge.u2`, controls clock `cclock4_1` since its `SceMiClockControl` macro instance has `ClockNum=3`.
- This `RxTransactor` model interfaces to a message input port called `p1` that is parametrized to a bit-width of 128.

The following listing shows some of the VHDL source code for the above schematic:

```
library ieee;
use ieee.std_logic_1164.all;
library SceMi;
use SceMi.SceMiMacros.all;

entity Bridge is end;
architecture Structural of Bridge is
    component TxTransactor is
        port(
            DutInCtrl: out std_logic;
            DutInData: out std_logic_vector(31 downto 0);
            DutOutCtrl: in std_logic;
            DutOutData: in std_logic_vector(31 downto 0) );
        end component TxTransactor;
    component TxDUT is
        port(
            DutInCtrl: in std_logic;
            DutInData: in std_logic_vector(31 downto 0);
            DutOutCtrl: out std_logic;
            DutOutData: out std_logic_vector(31 downto 0);
            Clk, Rst, ClkDiv2: in std_logic );
        end component TxDUT;
    component RxTransactor is
        port(
            DutInCtrl: out std_logic;
            DutInData: out std_logic_vector(31 downto 0);
            DutOutCtrl: in std_logic;
            DutOutData: in std_logic_vector(31 downto 0) );
        end component RxTransactor;
    component RxDUT is
        port(
            DutInCtrl: in std_logic;
            DutInData: in std_logic_vector(31 downto 0);
            DutOutCtrl: out std_logic;
            DutOutData: out std_logic_vector(31 downto 0);
            Clk, Rst: in std_logic );
        end component RxDUT;
    signal txDutInCtrl, txDutOutCtrl: std_logic;
    signal txDutInData, txDutOutData: std_logic_vector(31 downto 0);
    signal rxDutInCtrl, rxDutOutCtrl: std_logic;
    signal rxDutInData, rxDutOutData: std_logic_vector(31 downto 0);

    signal cclock, creset, clkDivideBy2, clkDivideBy4 cresetDivideBy4: std_logic;
begin
    u1: TxTransactor port map( txDutInCtrl, txDutInData, txDutOutCtrl, txDutOutData );
    d1: TxDUT          port map( txDutInCtrl, txDutInData, txDutOutCtrl, txDutOutData,
                                cclock, creset, clkDivideBy2 );
    cclock:  SceMiClockPort port map( cclock, creset );
    cclock2_1: SceMiClockPort
        generic map( 2, 2, 1, 50, 50, 0, 8 )
        port map( clkDivideBy2, open );
    u2: RxTransactor port map( txDutInCtrl, txDutInData, txDutOutCtrl, txDutOutData );
    d2: RxDUT          port map( txDutInCtrl, txDutInData, txDutOutCtrl, txDutOutData,
                                clkDivideBy4, cresetDivideBy4 );
    cclock4_1: SceMiClockPort
        generic map( 3, 4, 1, 75, 25, 30, 8 )
        port map( clkDivideBy2, open );
end;
```

Formal Functional Specification

```
library ieee;
use ieee.std_logic_1164.all;
library SceMi;
use SceMi.SceMiMacros.all;

entity TxTransactor is
  port(
    DutInCtrl: out std_logic;
    DutInData: out std_logic_vector(31 downto 0);
    DutOutCtrl: in std_logic;
    DutOutData: in std_logic_vector(31 downto 0) );
  end;
architecture Structural of TxTransactor is
  component TxTransactorCore is
    port(
      TxRdyIn: in std_logic;          RxRdyIn: out std_logic;
      Message: in std_logic(63 downto 0);
      DutInCtrl: out std_logic;
      DutInData: out std_logic_vector(31 downto 0);
      DutOutCtrl: in std_logic;
      DutOutData: in std_logic_vector(31 downto 0) );
      Uclk, Rst: in std_logic;
      ReadyForCclock: in std_logic; CclockEnabled: out std_logic;
      ReadyForCclockDiv2: in std_logic; CclockEnabledDiv2: out std_logic;
    end component TxTransactor;
  signal transmitReady, receiveReady: std_logic;
  signal message: std_logic_vector(63 downto 0);
  signal uclock, ureset: std_logic;
  signal readyForCclock, cclockEnabled: std_logic;
  signal readyForCclockDiv2, cclockEnabledDiv2;
begin
  t1: TxTransactorCore port map(
    transmitReady, receiveReady, message,
    DutInCtrl, DutInData, DutOutCtrl, DutOutData,
    uclock, ureset,
    readyForCclock, cclockEnabled, readyForCclockDiv2, cclockEnabledDiv2 );
  p1: SceMiMessageInputPort
    generic map( 64 )
    port map( transmitReady, receiveReady, message );
  c1: SceMiCclockControl
    port map( uclock, ureset, readyForCclock, cclockEnabled, '1', open );
  c2: SceMiCclockControl
    generic map( 2 )
    port map( open, open, readyForCclockDiv2, cclockEnabledDiv2, '1', open );
end;
```

Appendix C: VHDL SceMiMacros Package

The following package can be used to supply SCE-MI macro component declarations to an application. It is suggested that this package be compiled into a library 'SceMi' and that it be included in the application code as,

```
library SceMi;
use SceMi.SceMiMacros.all;
```

Here is the source code for the package:

```
library ieee;
use ieee.std_logic_1164.all;

package SceMiMacros is

    component SceMiMessageInPort
        generic( PortWidth: natural );
        port(
            ReceiveReady : in  std_logic;
            TransmitReady : out std_logic;
            Message       : out std_logic_vector( PortWidth-1 downto 0 ) );
        end component;

    component SceMiMessageOutPort
        generic( PortWidth: natural; PortPriority: natural := 10 );
        port(
            TransmitReady : in  std_logic;
            ReceiveReady  : out std_logic;
            Message        : in  std_logic_vector( PortWidth-1 downto 0 ) );
        end component;

    component SceMiClockPort
        generic(
            ClockNum          : natural := 1;
            RatioNumerator    : natural := 1;
            RatioDenominator  : natural := 1;
            DutyHi             : natural := 0;
            DutyLo             : natural := 100;
            Phase              : natural := 0;
            ResetCycles        : natural := 8 );
        port(
            Cclock : out std_logic;
            Creset  : out std_logic );
        end component;

    component SceMiClockControl
        generic( ClockNum: natural := 1 );
        port(
            Uclock,
            Ureset : out std_logic;
            ReadyForCclock : in  std_logic;
            CclockEnabled : out std_logic;
            ReadyForCclockNegEdge : in  std_logic;
            CclockNegEdgeEnabled : out std_logic );
        end component;
end SceMiMacros;
```