Chapter 3 - Theory of Operation

3.1 Overview

The SimMatrix co-simulation process comprises three different stages; design assembly, partitioning and co-simulation (Figure 3-1).

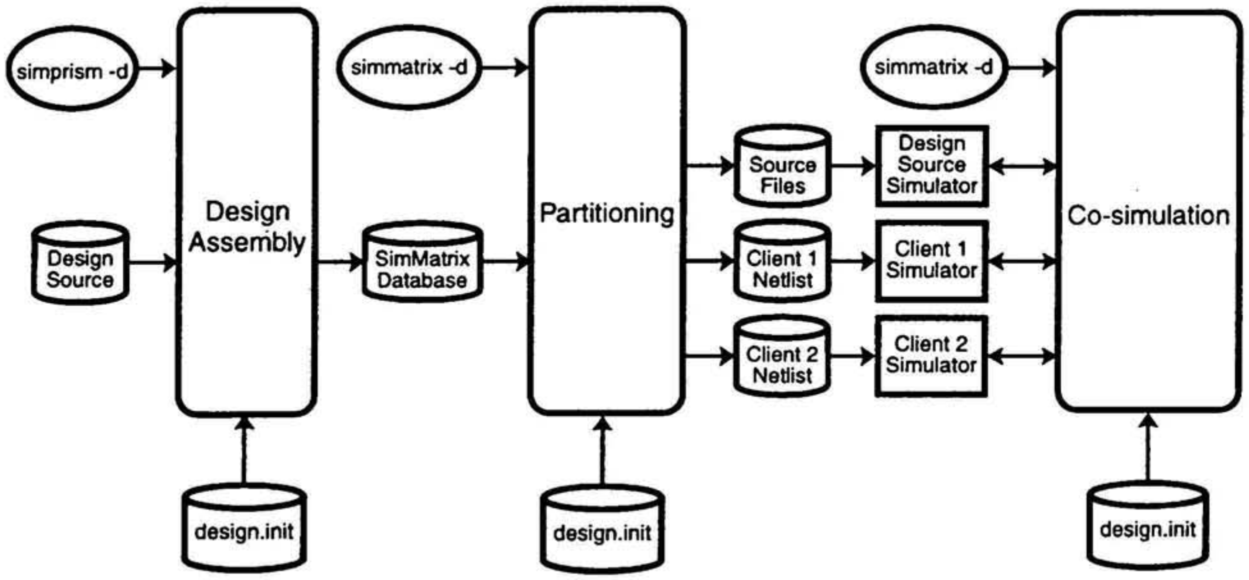


Figure 3-1. Overview of Co-simulation Processes

The design assembly stage involves compiling native design source files and extracting a hierarchical design representation from the design source. This extracted hierarchical design representation is written into the SimMatrix internal database file. The design assembly process is handled by a separate program module, called *SimPrism*.

The partitioning stage implements user specified partition rules to define which segments of the overall design, stored in the SimMatrix database file, are to be simulated by which of the clients involved in the co-simulation session. For each design segment partitioned to a particular client, the SimMatrix partitioner writes out a netlist in the format understood by the client. The client uses this netlist to simulate its portion of the design during co-simulation.

If the client uses the same file format as the design source, then the partitioner does not generate a new netlist, but uses those original design source files that are applicable to the partition to be simulated, instead.

The co-simulation stage initializes and runs all of the client simulators (including the design source simulator, if applicable) on their respective design partitions until a boundary event (mixed net, probe or breakpoint), interrupt, or session termination command is encountered. When a boundary event (spanning simulators) is encountered, SimMatrix synchronizes the simulators and distributes boundary event information (such as states, currents, or voltages) across simulators. A user-programmable state translation table provides for consistent signal representation between simulators. SimMatrix arbitrates state changes on boundary events to determine which simulator drives a net.

Synchronization is required so that signal state changes are propagated across all simulators at the same point in time. SimMatrix provides user-selectable synchronization schemes that can be used to optimize performance.

The event transfer process continues until the user interrupts or terminates the co-simulation session. When the user interrupts the co-simulation session, new commands can be injected into the co-simulation that take effect when the co-simulation session is resumed. When the user terminates a co-simulation session, all intermediate files that were created are deleted and all of the simulators being used are terminated.

3.2 SimMatrix Usage Model

SimMatrix co-simulation controls consist of user commands placed in the *design*`.init` file and commands entered directly into any of the client windows displayed during an interrupt to a co-simulation session. Commands placed in the *design*`.init` file are sourced when a co-simulation session is invoked. Commands entered directly into a client window are executed when the co-simulation session resumes.

User commands controlling the design assembly and partitioning processes must be implemented via the *design*`.init` file (they cannot be issued from a command line prompt) because they determine how client netlists are generated, prior to co-simulation. While all of the user commands controlling co-simulation can be implemented through the *design*`.init` file, only a subset of these commands can be issued from a command line prompt.

Except for design assembly related commands, which are mandatory, all commands are optional and up to the discretion of the user. Proper usage of the design assembly, partitioning and co-simulation related commands is discussed in Chapter 4.

3.2.1 Invoking the Co-simulation Processes

The three stages that define the overall co-simulation process; design assembly, partitioning and co-simulation (Figure 3-1) can be executed in various combinations, depending on how a co-simulation session is invoked.

3.2.1.1 Design Assembly

The processes comprising the design assembly stage are executed by a separate program module, called *SimPrism*.

Since the design assembly stage loads a design source into the SimMatrix database, it is usually only performed once, unless changes are made to the design source files. Design assembly is performed by executing the following command from a Unix command line prompt:

```
simprism -d design [+compile] [+extract]
```

The two aspects of design assembly, 1) compiling the design source files and 2) extracting the hierarchical design representation can be invoked separately or together, depending on command line options provided to the `simprism -d` command.

The `simprism -d` command sources the following commands from the *design*`.init` file:

```
simprism system compile_exec files
simprism extract source
```

The first line in the *design*`.init` file executes the native design source compiler to compile all of the relevant design source files that comprise the design to be co-simulated. The second line extracts a hierarchical image of the design source design representation and writes this into a SimMatrix internal design representation.

The first time a co-simulation is to be run all relevant design source files need to be compiled and the entire design hierarchy needs to be extracted. After that, if the original design source files are modified or added to, then those files and any other files that might have been affected by the change must be recompiled. If any of these design source file changes affect the structure of the design, then the design hierarchy needs to be re-extracted and re-written to the SimMatrix database. However, since the SimMatrix database only represents design structure, not behavior, if the design source changes only involved changes in behavior, then the design hierarchy does not have to be re-extracted.

NOTE

The `+compile` and `+extract` options will only be performed if the *design*`.init` file contains the compile related and extract related commands. If the *design*`.init` file does not contain the supporting commands, the processes cannot be executed.

3.2.1.2 Partitioning and Co-simulation

The partitioning and co-simulation executable consists of the same basic command followed by a different set of options (depending upon whether partitioning or co-simulation is to be executed). This command is issued twice; once for partitioning and once for co-simulation.

The first time a co-simulation session is run both the partitioning and co-simulation stages need to be executed. After that, the partitioning stage only needs to be executed if a change is made that affects a client netlist, e.g, how the design is partitioned. The command to execute partitioning and co-simulation is as follows:

```
simmatrix -d design [-compile]
```

The default for invoking SimMatrix is to run co-simulation without partitioning. In order to run partitioning, the `-compile` option must be specified. To run both partitioning and co-simulation, the `simmatrix -d` command needs to be issued twice.

To partition the design representation created in para. 3.2.1.1, the `simmatrix -d` command sources the following commands from the *design*`.init` file:

```
simmatrix_partition
import design.ext.db
```

The `simprism simmatrix_partition` command invokes the SimMatrix partitioner on the hierarchical design representation that was extracted in para. 3.2.1.1. This design is imported by the `import` *design*`.`*ext*`.db` command which supplies the design name and the design source type.

3.3 Design Assembly

Design assembly involves collecting all design components to be used, compiling the design and generating the hierarchical image. The design components are collected in accordance with standard commercial practices. The design is compiled using the design source specific compilation program. Following compilation, the hierarchical image is generated by executing the `extract` command.

3.3.1 Design Assembly Flow

The design assembly process requires that SimMatrix read in design information from the design source and create an internal hierarchical representation of the design source design in SimMatrix.

The *`compile`* program compiles all of the design source files into the binary files that are used by the *`database writer`* to generate the *`design`*`.mti.db` file for `simmatrix -compile`. The name of the compile program is specific to the format of design source files being used. The *`database writer`* is a custom integration program that enables a design representation from the design source to be scanned and rewritten into a standard hierarchical design representation used by SimMatrix. The *`design.ext`*`.db` file contains the complete SimMatrix hierarchical design representation (Figure 3-3). The *`extract`* command invokes the database writer, supplying it with the design source specific command line arguments needed to create the *`design`*`.mti.db` file.

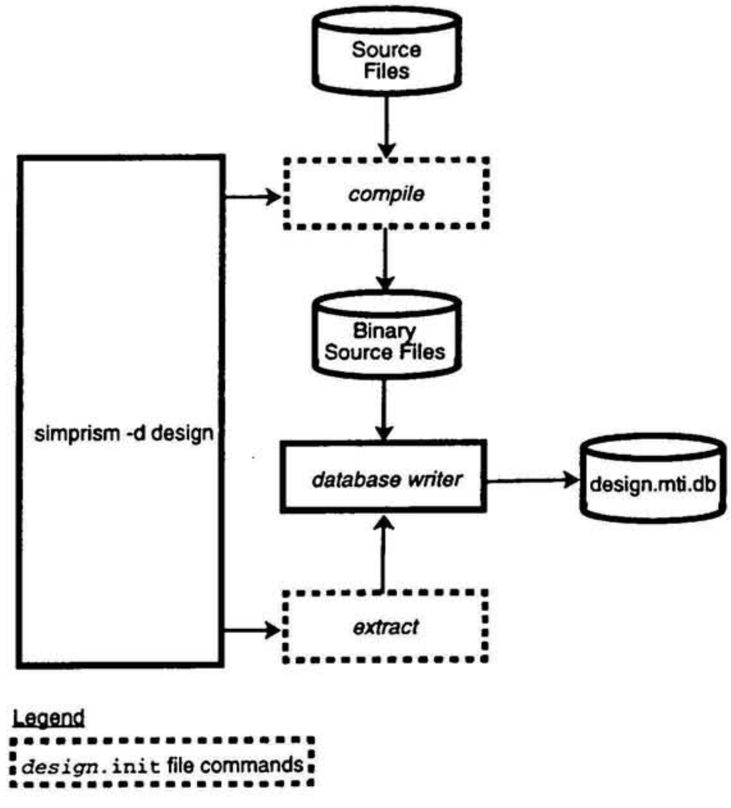


Figure 3-2. Design Assembly Process Flow Diagram

NOTE

The translation of design source design representation into the SimMatrix Data Base design representation is controlled by processes that were developed for the specific design source to SimMatrix integration being used. These processes are described in the SimMatrix Database Integration manual, PN IM002.

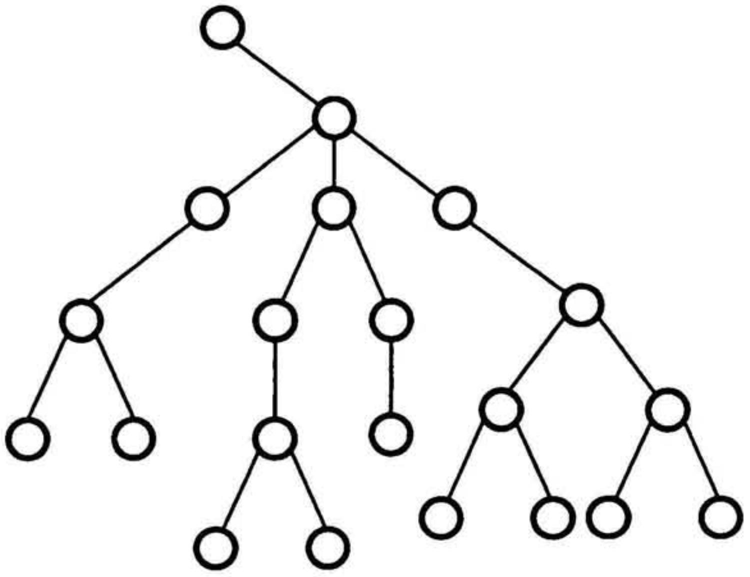


Figure 3-3. SimMatrix Database Design Representation (*design.ext*`.db`)

3.4 Partitioning

When SimMatrix is invoked by entering the `simmatrix -d` design command (Figure 3-4), the commands in the *design*`.init` are sourced by the `simmatrix -compile` program which launches SimMatrix processes to generate the appropriate client netlist file(s) for each integrated client. As shown in Figure 3-4, the sourced commands from the *design*`.init` file (shown enclosed by dotted lines) represent user inputs to the partitioning process.

Figure 3-4 illustrates the elements involved in the overall partitioning process. The SimMatrix partitioning tool automates the partitioning process by implementing partition rules that are annotated to native design source files (Appendix A) and/or placed in the *design*`.init` file. From this information, SimMatrix identifies all design blocks pertaining to each simulator, determines all nets that cross simulator boundaries (mixed nets), and automatically generates inter-simulator connectivity information for the integrated clients. SimMatrix then generates the appropriate netlist files to be simulated by each client.

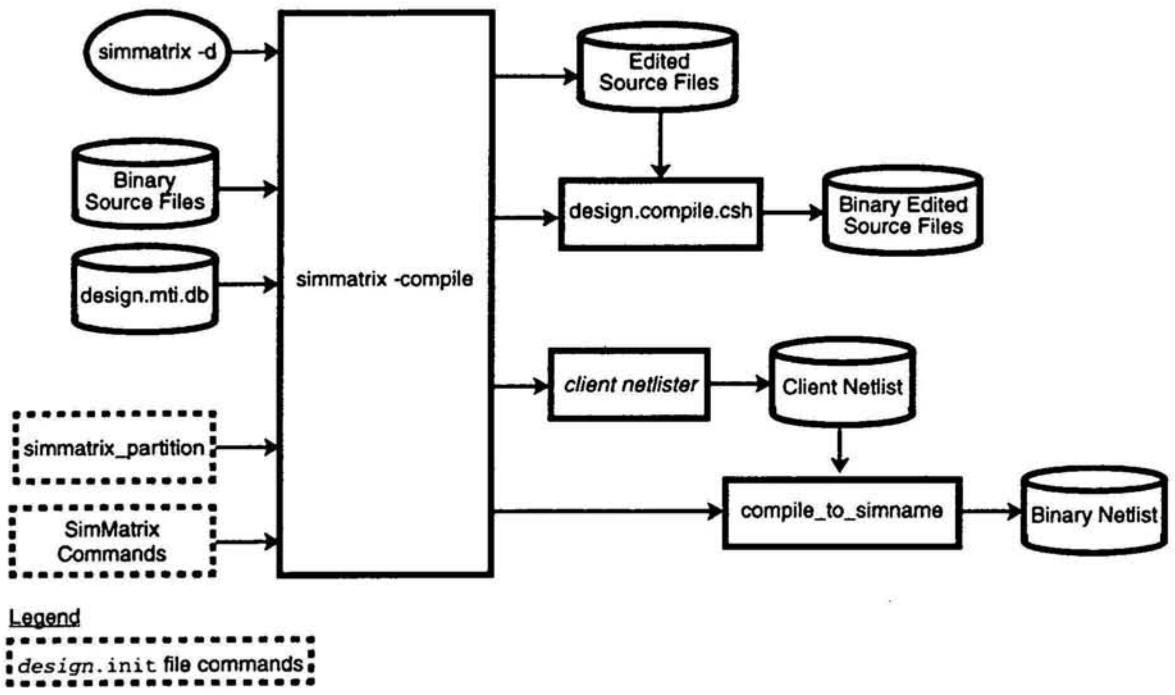


Figure 3-4. Partitioning Process Flow Diagram

The `simmatrix_partition` command invokes the `simmatrix -compile` program which begins the process of implementing the various SimMatrix commands in the *`design`*`.init` file. The most common SimMatrix command is `partition`, which is used to define which cells in the overall design are to be simulated by which client.

For any particular partitioning scenario, one of the client simulators involved in the co-simulation session (the primary client) must be designated as the owner of the top cell in the design being simulated (Simulator A in Figure 3-5). The `-top` argument to the `partition` command determines which simulator this is.

NOTE
The primary client generally runs in the environment of the design source and simulates using a set of edited design source files.

NOTE
The `partition` command includes options that address a wide variety of possible partitioning scenarios. These options are explained in more detail in para. 4.3.2.

As shown in Figure 3-1, the partitioning process can generate two different kinds of files for simulation, depending on the relationship between the client simulator and the design source. If the design source is written in the native language of the client simulator, then the partitioner re-uses and edits those design source files that are applicable to that segment of the overall design being partitioned. If the client simulator uses a language different from the design source, then SimMatrix generates a netlist for the client simulator in the netlist format recognized by the client simulator.

The `client netlister` is a custom integration program that enables a SimMatrix hierarchical design representation (created by the database writer) to be written into a hierarchical design representation format that is recognized by the client simulator.

NOTE
The translation of SimMatrix hierarchical design representation into a client netlist format is controlled by the processes that were developed for the specific client simulator to SimMatrix integration being used. These processes are described in the SimMatrix Integration Development Environment manual, PN IM001.

Some of the other SimMatrix commands that can be sourced are described in Chapter 4.

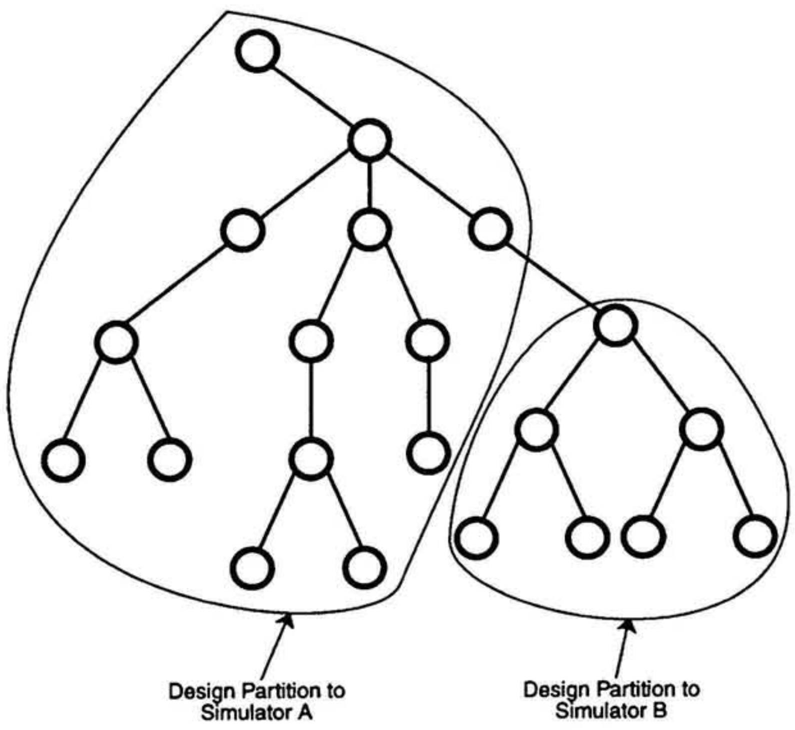

Design Partition to Simulator A
Design Partition to Simulator B

Figure 3-5. Possible Design Partitioning Scenario

Figure 3-6 shows the design representations that are created for two client simulators based on the partitioning rules shown in Figure 3-5. SimMatrix retains name space mapping in the design partitions by adding the necessary levels of hierarchy from the design partition to the top cell in the original unpartitioned design (Figure 3-3). This is only required for those clients not owning the top cell in the overall design.

Also, if additional levels of design hierarchy exist in the client representation, below what could be represented in the original design source, then that portion of the client netlist can be imported and attached (in native client netlist code) to the generated client netlist (Figure 3-7).

3.4.1 Shadow Generation for Exported Blocks

During partitioning SimMatrix creates a *shadow* representation of the design segments partitioned to all other simulators and attaches this to the design representation used by the primary client (Figure 3-6). The shadow design representation enables the primary client's environment to interact with the corresponding cells in the design segments partitioned to the other integrated simulators. This interaction depends on the capabilities of the primary client's environment and can include such things as dropping probes, monitoring waveforms, inputting stimulus, etc.

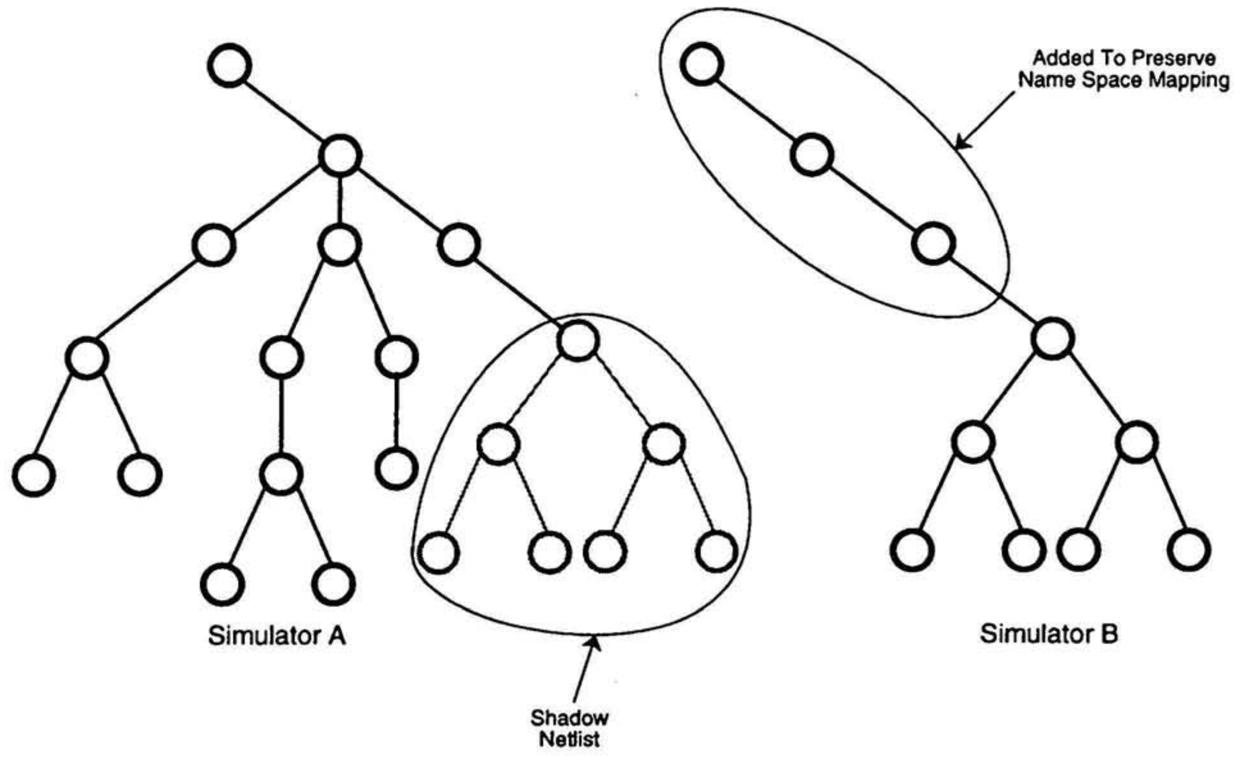


Figure 3-6. Shadow Generation for Exported Block

3.4.2 *Placeholder Generation for Imported Blocks*

The partitioning function allows for the design representation of cells from within the partitioned block to be imported from an outside library, rather than being determined by the overall design representation. For example, if a particular cell is instantiated from a library in the client simulator (Simulator C), the partitioning function can be specified to *import* the description of the cell from the imported library instead of the overall design representation (Figure 3-7). This way the netlist created for the client only instantiates the specific cell, not the netlist hierarchy underneath it.

Any design representation in the primary client below the imported block is pruned from the primary client and a placeholder is created in the primary client for the cell designated as an import. Since imported blocks have no design representation in the primary client, they are not directly accessible for interaction with the primary client's environment.

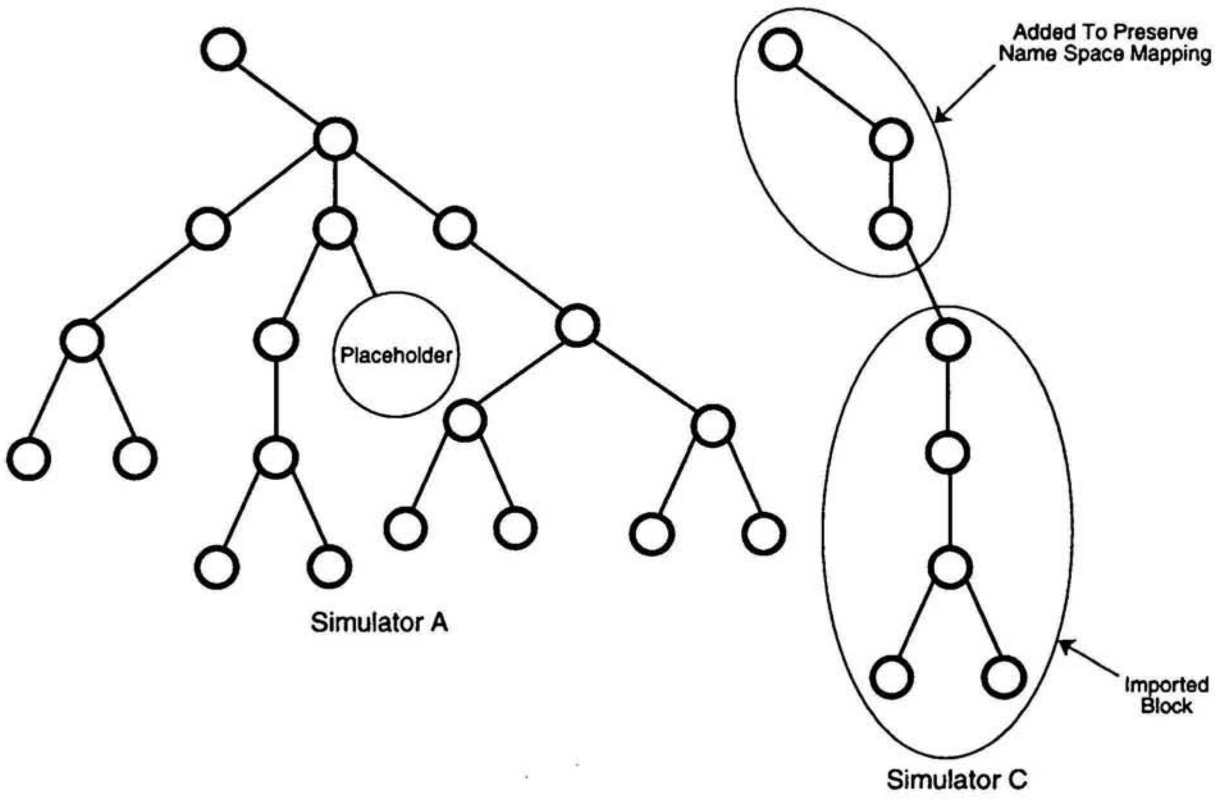


Figure 3-7. Placeholder Generation for Imported Block

3.4.3 Distributed Simulation

Partitioning a design involves specifying the design object to be partitioned and the target simulator that will simulate that object. Specifying the target simulator involves defining the client simulator as well as the host computer and cpu process that the client will be running on. The client simulator taken in conjunction with the host computer and cpu process define the target solver for a particular design partition.

There are primarily two types of designs that are good candidates for distributed simulation:

- large designs with partitions exceeding available memory.
- designs using slower, high accuracy simulators, e.g., SPICE.

The concept of a solver as the target for a design partition provides the vehicle that enables design partitions to be simulated on different hosts and even different cpu processes within a host. This enables large designs to be segmented for simulation across a network of hardware systems. It also enables a partition being simulated by a slower, high accuracy simulator to be performed on a faster machine.

Figure 3-8 illustrates a complex partitioning scenario that utilizes all of the parameters for specifying a target solver (client, host, cpu process). During the partitioning, SimMatrix creates a client socket interface for each solver so that it can communicate back and forth with each partition. As can be seen from the illustration, different client simulators can be run on different process within the same host.

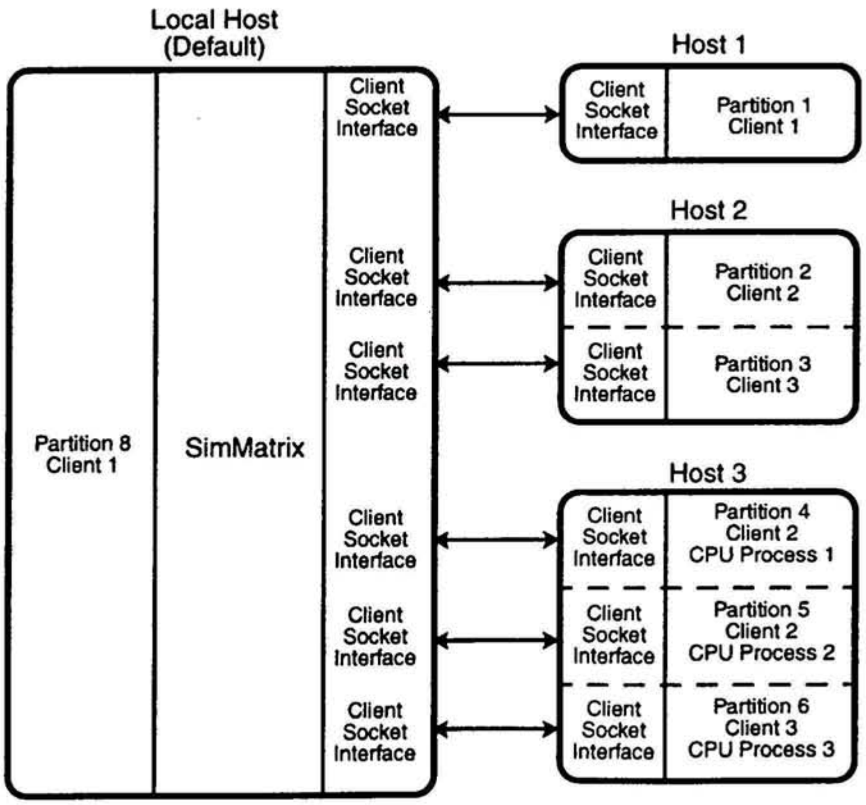


Figure 3-8. Partitioning for Distributed Simulation

Chapter 5 provides information on how to obtain maximum performance using distributed simulation.

3.4.4 *Legal Partitions*

SimMatrix supports virtually any partitioning scheme that the user can devise. This includes nested design partitions (donuts) and isolated partitions (islands) (Figure 3-9).

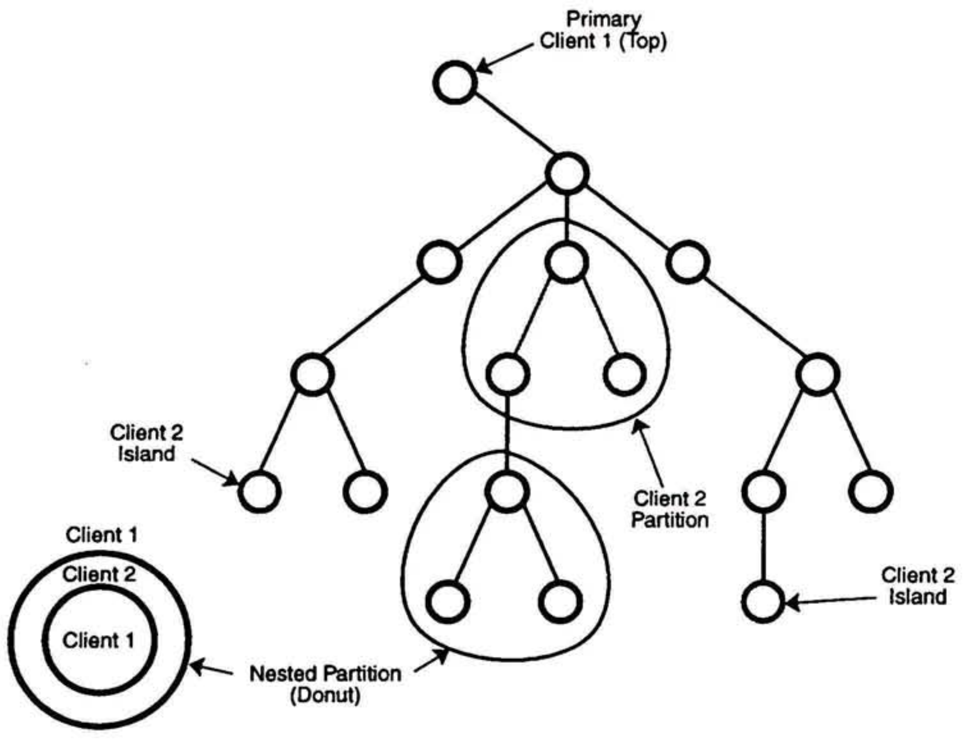


Figure 3-9. Legal Partitions

3.4.5 *Conflicts*

The client integration to SimMatrix handles conflicts between simulators regarding such issues as object names, design representation, and representation of nets and ports. If problems revolving around these issues do occur, please consult your integration engineer.

3.5 Co-simulation

After successfully partitioning the design in preparation for co-simulation, the user can invoke SimMatrix to perform co-simulation (para. 3.2.1.2). The co-simulation process breaks down into initialization and runtime stages. Figure 3-10 shows a typical functional flow for a co-simulation session.

3.5.1 Initialization Stage

The initialization stage goes through the following processes:

- Initialize client
- Parse netlist
- Registering mixed nets, probes, and breakpoints.
- Initialize simulator state
- Find initial starting point

A simulation session begins with SimMatrix instructing each client simulator initializing. SimMatrix then instructs each client to open one or more netlist files created by the netlist writer, and parse their contents. Upon completion of the parsing function, SimMatrix instructs each client to register all mixed nets, probes, and breakpoints in its design partition.

NOTE
Mixed nets must be registered during the initialization stage whereas probes and breakpoints can be registered during the initialization stage or they can be injected during the runtime stage (para. 3.5.2.1).

SimMatrix then instructs each client to initialize its circuit partition to some initial operating point. This resets each client's notion of time to 0, releases all forced values on probes, and initializes the state of every net. In digital simulators, the initial state of every net is usually defined by the simulator. For analog simulators, the initial state of every net results from finding a stable dc operating point for the circuit.

Each simulator performs at least one initialization cycle. At the end of this initialization, the resulting states of mixed nets are exchanged between simulators. After any resultant state changes have been made and the dc solution settles, SimMatrix begins to coordinate the active co-simulation session.

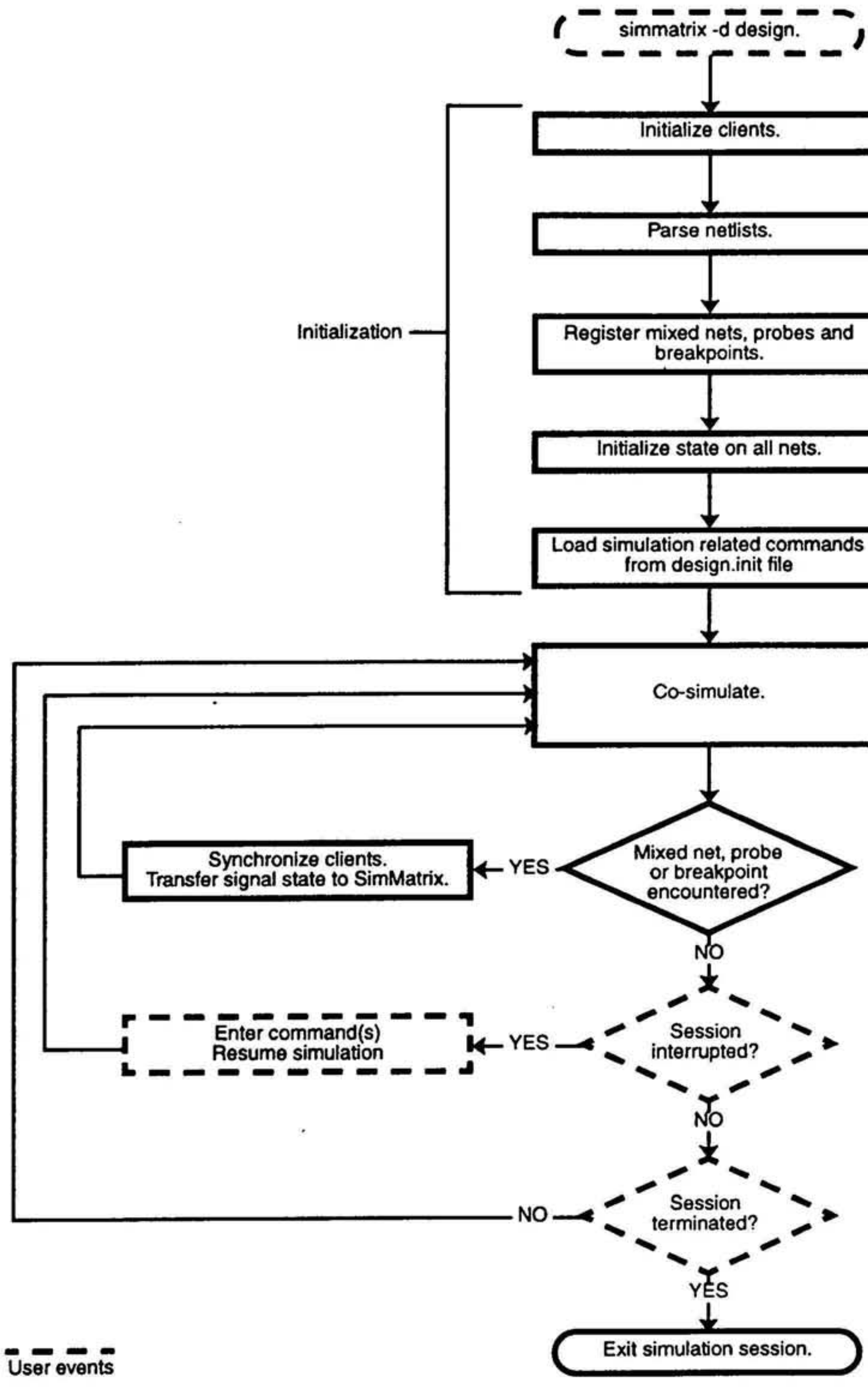

simmatrix -d design.
Initialization
Initialize clients.
Parse netlists.
Register mixed nets, probes and breakpoints.
Initialize state on all nets.
Load simulation related commands from design.init file
Co-simulate.
Mixed net, probe or breakpoint encountered?
YES
Synchronize clients. Transfer signal state to SimMatrix.
NO
Session interrupted?
YES
Enter command(s) Resume simulation
NO
Session terminated?
NO
YES
Exit simulation session.
User events

Figure 3-10. Co-simulation Flow

3.5.2 Runtime Stage

After initialization, the runtime stage starts up as a regular simulation session in the environment of the primary client (the client owning the top cell of the overall design). When SimMatrix detects a mixed net to a design segment partitioned to another simulator, it transfers control of the co-simulation session to that simulator. During the co-simulation session, only the User Interface (UI) of the controlling simulator is active.

While the co-simulation session is running, each client simulates that portion of the overall design partitioned to it until a boundary event (mixed net) is encountered. When a mixed net (spanning simulators) is encountered, SimMatrix synchronizes the simulators and distributes boundary event information (such as states, currents, or voltages) across simulators. A user-programmable state translation table provides for consistent signal representation between simulators. SimMatrix arbitrates state changes on mixed nets to determine which simulator drives a mixed net.

Synchronization is required so that boundary event information occurs in all simulators at the same point in time. SimMatrix manages the synchronization of integrated simulators based on an internal time tick of one femtosecond (1fS). SimMatrix also provides for the selection of an *event-based* synchronization scheme to optimize performance.

The mechanism used to gather and transfer boundary events varies from one simulator to another, and can occur through direct memory transfer (subroutine or shared memory implementation) or through inter-process communication (over Ethernet), if multiple workstations are being used. SimMatrix automatically selects the highest performance communication vehicle available for each simulator. The simulation session appears identical regardless of the communication vehicle utilized.

The event transfer process continues through the entire simulation session, pausing only if the designer wishes to interrupt and interact with the co-simulation session. When the simulation session terminates, all intermediate files that were created are deleted and all of the simulators being used are terminated.

If no breakpoints are triggered, simulation time advances until the specified interval elapses, at which point the simulation terminates and control is returned to the user.

3.5.2.1 Boundary Event Processing

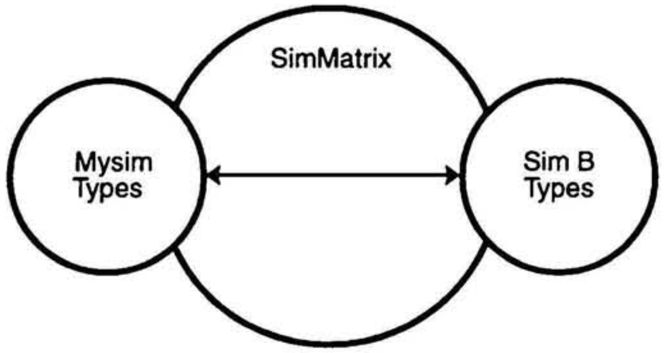
Signal states are communicated to and from SimMatrix when boundary events (mixed nets, probed nets, or breakpoints) are encountered. Mixed nets are defined during the netlisting process and are registered during *initialization* of the co-simulation session (para. 3.5.1). Probed nets and breakpoints can be registered during initialization or incrementally after a co-simulation session begins.

Every time a signal (mixed net, probe or breakpoint) is communicated to or from SimMatrix, a state translation function is executed.

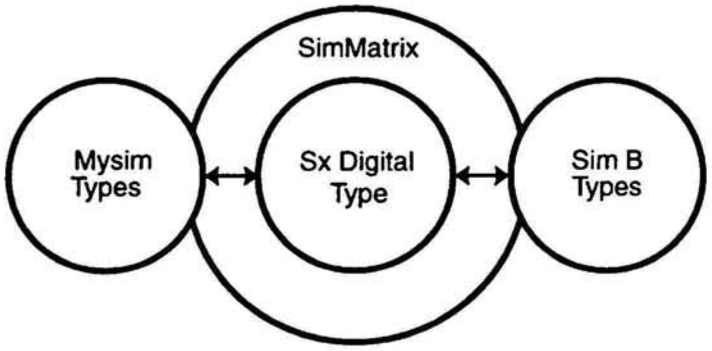
When a client is simulating and it encounters a mixed net, probe or breakpoint, it drives the net by forcing a new state on that net and passes the value to SimMatrix, so that SimMatrix can propagate the change in state of the net to other simulators.

3.5.2.1.1 State Translation

Signal state representation, while consistent within any one particular simulator, varies between diverse simulators, especially if one of the simulators is digital and the other is analog. As such, it is critical that signal states being passed between diverse simulators (on mixed nets) be as consistent as possible. This consistent signal state representation is achieved by point-to-point signal mapping between simulators, or by mapping to the SimMatrix intermediate type, Sx Digital (Figure 3-11).



Point to Point Type Conversion



Type Conversion Using SimMatrix Abstract Data Types

Figure 3-11. Type Conversions using Point to Point and SimMatrix Intermediate Types

The point to point type conversion produces the best co-simulation performance because only a single conversion process needs to be executed. Converting to SimMatrix intermediate types requires a dual conversion process from Mysim to the intermediate types and then from the intermediate types to another simulator. The benefit of converting a simulator's types to the SimMatrix intermediate types is that it will be compatible with any other simulator whose types have also been converted to the SimMatrix intermediate types. In such a case, a direct point to point conversion path does not have to be available between integrated simulators. The default type mappings can be changed by using the `remap_type` command (Table 4-1).

3.5.2.1.2 Mixed Nets

Mixed nets are registered as `in`, `out`, or `bidirectional` types. An `in` mixed net means that the state of that mixed net in the client gets changed to something else via SimMatrix. An `out` mixed net means that the client provides the source of a state change to an associated mixed net via SimMatrix. A bidirectional mixed net can both be changed and be the source of a change from/to SimMatrix.

Signal flow direction between components connected by mixed nets is the same as signal flow direction between those components in the original design source. If SimMatrix cannot determine signal direction during partitioning, it will be assigned to be bi-directional.

3.5.2.1.3 Probes

Probes are normally placed by a user to collect simulation (waveform) data for display on the screen. SimMatrix monitors all probed nets and report any state/voltage changes occurring at the probed net in a similar manner as the reported state changes for outgoing or bidirectional mixed nets.

3.5.2.1.4 Breakpoints

Breakpoints are used to detect certain state conditions within the design, and typically to take some action, such as issuing an interrupt, when the state has been detected. When a breakpoint is encountered, the breakpoint reports a state change to SimMatrix in a similar way the state changes are reported for mixed and probed nets.

3.5.2.2 Synchronization

When a boundary event occurs between integrated simulators, SimMatrix synchronizes the integrated simulators so that they are at the same point in time. This enables data to be reliably exchanged between integrated simulators.

SimMatrix uses two types of synchronization schemes; 1) look-ahead and 2) optimistic. Look-ahead synchronization is used for digital simulation and optimistic synchronization is used for analog simulation. In a mixed simulation (digital and analog co-simulators) environment, a *look-ahead* or *optimistic* synchronization will take place, depending on which simulator contains the *synchronizing boundary event*. If it is the digital simulator, the analog simulator will employ *optimistic* synchronization to synchronize to it. If it is the analog simulator, the digital simulator will employ *look-ahead* synchronization to synchronize to it.

3.5.2.2.1 Look-ahead Synchronization

Look-ahead synchronization allows co-simulators to execute asynchronously until a boundary event (requiring synchronization) is reached. Between synchronization points, each co-simulator operates sequentially with the other simulators, the sequence being determined by which simulator has the next event to be processed.

SimMatrix arbitrates between the integrated clients to determine which simulator has will have the next event to be processed and instructs that simulator to simulate until it's next event.

This process repeats for the next closest *next event* (among all connected simulators), until a boundary event is encountered. When a boundary event is encountered, all simulators, except the one that advanced to the next (boundary) event, will advance to the time of the boundary event, and all simulators will be synchronized to the same time.

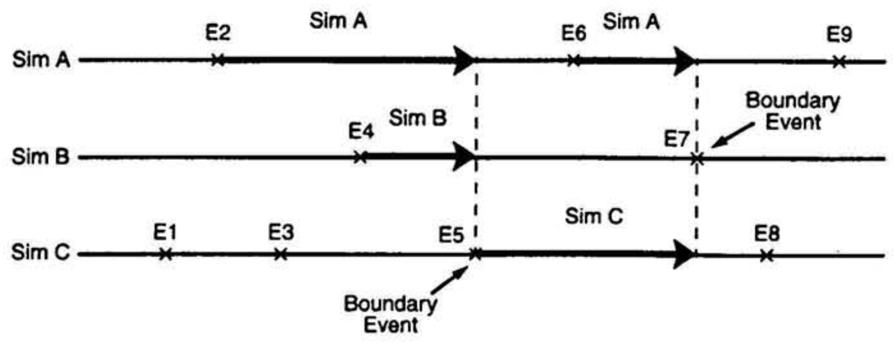


Figure 3-12. Look-Ahead Synchronization

3.5.2.2.2 Optimistic Synchronization

During optimistic synchronization, each of the co-simulators advance (in parallel) through next events until they encounter a boundary event. The times of each simulator boundary event are provided to SimMatrix for arbitration so that SimMatrix can determine which simulator encountered a boundary event at the earliest time. The time of the earliest encountered boundary event becomes the *synchronizing boundary event*, i.e., the time to which all the other co-simulators must synchronize.

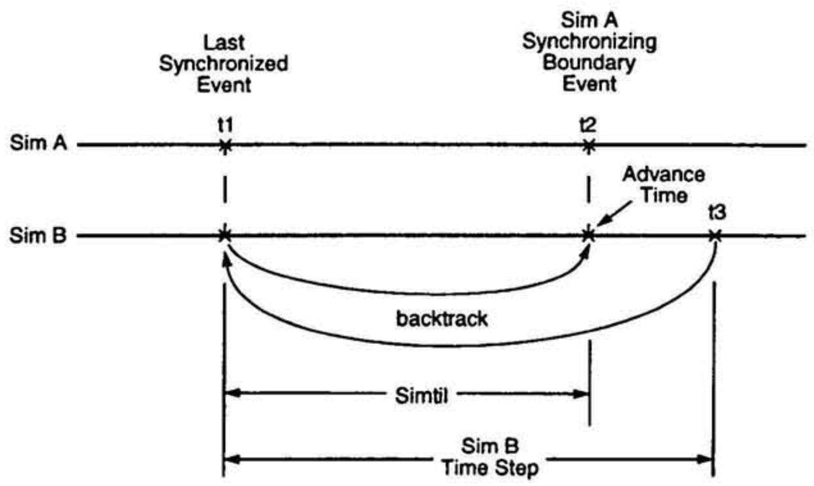


Figure 3-13. Optimistic Synchronization

The client with a later occurring boundary event is instructed to backup to its last event that occurred prior to the *synchronizing boundary event*. The co-simulators must then simulate from the point in time backtracked to, to the same time as the *synchronizing boundary event*.

3.5.2.2.3 Event-Based Synchronization

In some cases, a design partition exists on a synchronous boundary between two cells where communication is important only when a clock occurs. For example, a pipelined multiplier may generate many intermediate results on its output before a clock latches a final result. Normally, boundary event traffic and synchronizations are required for each of the intermediate results, even though the data has no immediate value, until the clock latches the final result.

In this situation, you might be able to reduce backplane overhead by using event-based synchronization, which specifies the signal (event) to synchronize on. Event-based synchronization reduces the number of synchronizations, but has no affect on event traffic.

SimMatrix provides the `sync_signal` command (para. 4.3.3.2) that can be used to set the SimMatrix synchronizations to occur during specific boundary events.

3.5.2.3 Interrupt Handling

An interrupt can be issued at any time during a co-simulation session from the controlling client environments. An individual solver can be interrupted *asynchronously* by using it's native interrupt facility (usually <CTRL C>) from within its own control window, or *synchronously* by issuing the SimMatrix `pause` command from any of the other clients control window to the target client.

Both types of interrupts disable all other client environments and suspend the co-simulation session until the interrupted client resumes the co-simulation session.

Asynchronous interrupts retain simulation control for the interrupted client. Synchronous interrupts propagate the interrupt from one client to the other and simulation control is retained by the interrupting client.

Asynchronous interrupts can be issued by a controlling simulator at any time during a co-simulation session.

A synchronous interrupt is immediately propagated to the targeted client, where it is serviced as soon as it is safe to do so.

Two conditions can cause a delay before an interrupt; 1) the targeted client might process interrupts only at designated times, and 2) co-simulator synchronization might delay the servicing of the interrupt until the client is active.

During an interrupt, the interrupted client can issue new simulator commands that will be propagated into the co-simulation session once the co-simulation session is resumed. Those commands can be issued directly to the client controlling the co-simulation session, or they can be propagated to any other client involved in the co-simulation session.

Resuming from an interrupt (whether asynchronous or synchronous) gives control back to the interrupting clients environment.

3.5.2.4 Debugging

SimMatrix preserves the User Interface (UI) and debugging systems used by a client, but they are only active when that client has control of the simulation. As a result, if a synchronous interrupt is issued from another client, then the issuing client has control of the simulation, and graphical waveform functions for the interrupted client will be disabled until it has control of the simulation.

SimMatrix debugging commands to add/delete breakpoints and probes, and to view nets and ports can be issued to any client in the co-simulation session.

When encountered, breakpoints issue a synchronous interrupt to the client in which the breakpoint occurred.

The design environment of the primary client can be used to examine the overall design hierarchy and signal states for those portions of the overall design exported to another client. Signals that span multiple simulator boundaries can be debugged in the environment of any affected client.

3.5.2.5 Injecting a Stimulus (Driving a Mixed Net)

A client can use its native signal stimulus facility (or external command files) to inject (drive) a stimulus value onto any net in its own design partition hierarchy. This includes nets that have access to mixed nets and therefore could get propagated into another client's design partition. In order for an injected stimulus to drive an associated mixed net, the injected stimulus must follow the signal design flow (directionality) of the associated mixed net.

3.5.2.5.1 Sample Signal Injection

A sample signal injection is as follows. In the design shown in Figure 3-14, the entire design is simulated in *client_1,* except for cell B and its sub-hierarchy which is simulated in *client_2.* The signals crossing the boundary are DATA (bi-directional), CONTROL (input to cell B, in *client_2*) and DATA_RDY (output from *client_2*). In this design, you can drive the DATA signal from either environment because it is bidirectional. However, the CONTROL signal can only be driven from the *client_1* environment, and the DATA_RDY signal can only be driven from the *client_2* environment. If you drive the CONTROL signal from the *client_2* environment, the changes will be seen in the *client_2* environment but will not be propagated into the *client_1* environment. Similarly, any stimulus on the DATA_RDY signal in the *client_1* environment will not be propagated to cell B in the *client_2* environment.

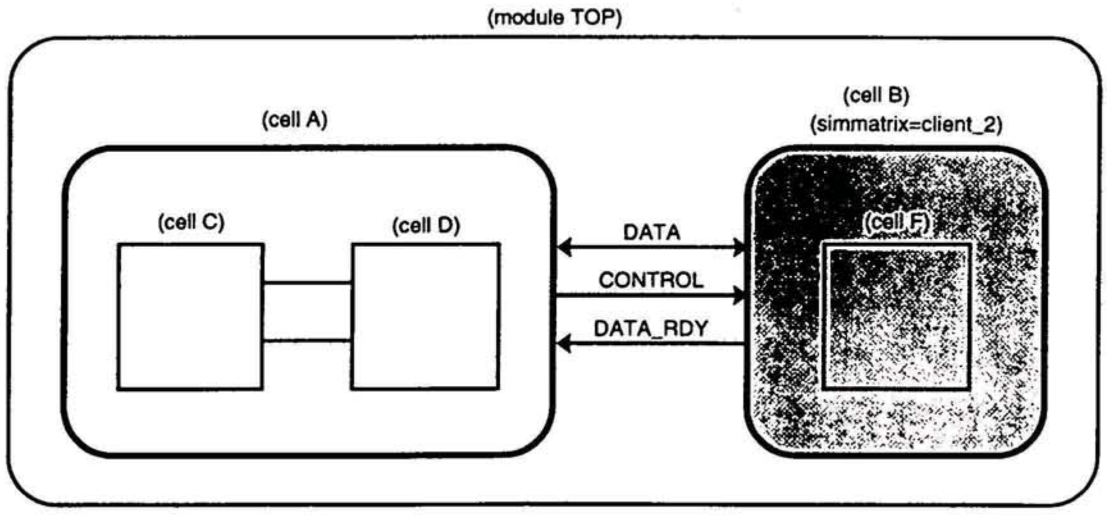


Figure 3-14. Diagram of Two Blocks with Stimulus

This situation can be circumvented by changing CONTROL and DATA_RDY into bidirectional signals. This can be done by adding the `add_mixed_net` command into the *`design`*`.init` file as follows:

```
add_mixed_net -dir inout CONTROL

add_mixed_net -dir inout DATA_RDY
```

These changes take effect after re-partitioning the design. This changes the direction of the CONTROL and DATA_RDY signals from unidirectional to bidirectional. Note that there is a small performance penalty associated with changing a unidirectional signal into a bidirectional signal since events must now propagate into both directions.

3.5.3 Terminating a Co-simulation Session

A co-simulation session can be terminated by shutting down SimMatrix or any of the client simulators. When SimMatrix recognizes that any of the simulators involved in the co-simulation session has terminated, it instructs all other simulators to terminate the session in an orderly manner.