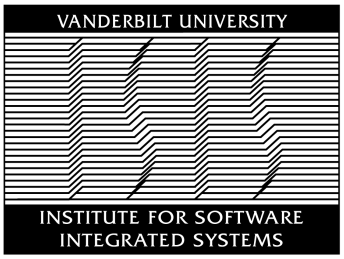
Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235



TECHNICAL REPORT

TR #: **ISIS-01-202**

Title: **High-Level Functional Simulation for Model-Based Embedded System Synthesis**

Author: **Brandon Eames, Sandeep Neema, Jason Scott, Ted Bapty**

Abstract

In the synthesis of embedded systems from models, the designer represents complex systems with domain-specific, multi-aspect, abstract models. In developing the models, the designer must make many high-level design decisions. The impact of these decisions are not readily predictable, given the complexity of the system. Consequently, the designer needs tools to assess these impacts and predict system performance, from a numerical/accuracy standpoint as well as performance. This report describes a tool for performing the numerical (functional) simulation of systems directly from the models.

The tool is integrated with the Model-Integrated Design Environment for Adaptive Computing Systems, and implements a critical part of the design flow. From multi-aspect models, a component-based data-flow computation is extracted and a Matlab program is synthesized. Matlab is used as a runtime environment to execute the program, due to its large user community and the number of existing tools and libraries. The results of the computation are visible to the designer, providing feedback in the system specification/design cycle.

KEYWORDS
Functional Simulation, Component-Based Design, Interface Synthesis, HW/SW Synthesis, FPGA, VHDL, Design Environment, Model-Integrated Computing.

ACKNOWLEDGMENTS
This work was sponsored by the Defense Advanced Research Projects Agency, Information Technology Office, under contract # DABT63-97-C-0020.

A HIGH-LEVEL FUNCTIONAL SIMULATION

When a designer creates a set of system models, he/she may want to simulate the proposed design before committing to a particular implementation path. A functional simulation is a representation of a system, which can be executed to verify the behavior or function of the models. By simulating a set of models before proceeding to component implementation, a designer can verify the semantics of each piece of the system, perhaps uncovering design details which would otherwise only be discovered much later, during component implementation or system integration. A further benefit of a functional simulation is that it empowers the designer to perform "back-of-the-envelope" calculations when starting a new design concept. The MatSim interpreter was introduced into the ACS toolset to generate a functional simulation from the algorithm models of an ACS system design.

MatSim generates an executable representation of the system structural models in the Matlab language. System designers often use the Matlab language and computational environment during initial stages of a design to perform algorithm development and simulation. Matlab is a common choice because of its powerful computational libraries, which provide many of the functions required in signal processing applications, as well as its data visualization capabilities. MatSim assumes that the system designer will provide a Matlab m-function representing each primitive component in a design to be simulated. These user-provided functions simulate the functionality provided by a particular component. MatSim will then generate the glue code to call the user-provided m-functions, scheduling them in the correct order. The generated glue code along, with the user-provided m-functions, forms an executable simulation that represents the system structural models.

MatSim generates a functional simulation for a single point design, as opposed to a design space. Before the designer invokes MatSim, he/she should invoke the desert interpreter to allow the selection of a point design from the modeled design space. When generating a functional simulation, the resource mapping is not taken into account. MatSim assumes that all components included in the point-design are to be simulated, and will have an associated Matlab-based simulation component, regardless of the resource category of the component.

Matlab Representation of Structural Models

MatSim is responsible for generating Matlab code that correctly represents the structural models of a system design. Structural models, as stated in Chapter II, are used to model the computations or algorithms of the system. The three types of structural models are ProcessingCompounds, ProcessingTemplates, and ProcessingPrimitives. Because MatSim is invoked on a single point design instead of on a design space, ProcessingTemplates carry no significance. A template represents a design alternative. When a design is selected from the design space, all possible alternatives are resolved,

and a template can be thought of as merely a placeholder for that alternative which it contains that was selected through design space exploration. In representing a ProcessingTemplate model, MatSim simply uses the selected alternative in the place of the template. Representing compounds and primitives, however, is not so simple.

ProcessingPrimitives

Each processing primitive represents a user-provided system component. The user is required to supply a Matlab m-function, which represents the behavior of the system component. This m-function will be invoked as part of the functional simulation. There are no limitations on the contents of the m-function; however, there is a correspondence between the function and the parameter signature, or prototype of the function. MatSim assumes that the ports of a model represent parameters of a function. Each input port represents an input parameter, and each output port an output parameter of the function. The name of the function is derived from the "Script/Component name" attribute of the ProcessingPrimitive model. Figure 1 depicts a ProcessingPrimitive model named DoCorrection. The model has two input ports, Correction and Position, and output port Out. The "Script/Component name" attribute has been set to "doCorrection." While MatSim assumes that the name of the user-supplied function is similar to the scriptname attribute, there is one minor difference. MatSim generates simulation components names by adding the letter "M" to the front of the scriptname, to avoid naming conflicts with components which are named according to the scriptname (as will be seen in Chapter IV). The function that simulates the DoCorrection component should therefore be named "MdoCorrection". The function MdoCorrection, shown in Figure 2, correctly represents the DoCorrection model of Figure 1.

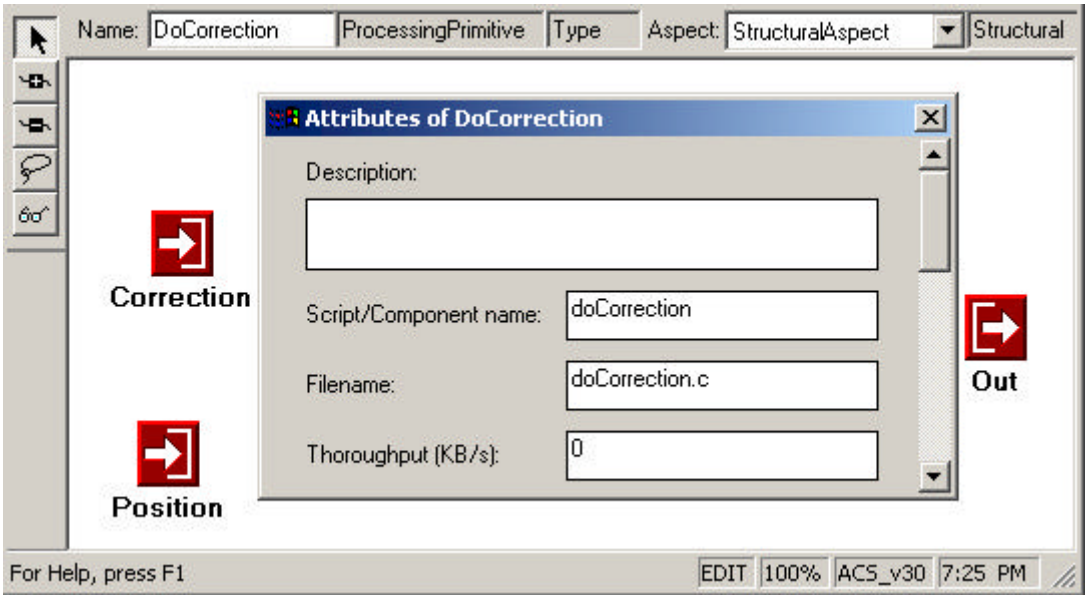


Figure 1. A ProcessingPrimitive model with ports and a scriptname, to be translated into a Matlab function

```
function [Out] = MdoCorrection(Correction, Position)
%function doCorrection to simulate the correction
% runtime component.

%P-constant set to 0.375 by experimentation
P_constant = 0.375;
Out = Position * Correction * P_constant;
```

Figure 2. User-provided function represented by the Primitive model in Figure 1.

The only requirement on the user when creating simulation components is that a component's signature matches the port interface of the model that represents the component. If a model has multiple ports, the order of the parameters is significant. The order of the parameters must match the numbering of the ports. When models are constructed, each data port is assigned a number. It is the modeler's responsibility to ensure that ports are numbered sequentially, and that all ports are numbered. Input ports are numbered independent of output ports. In the case of the DoCorrection model of Figure 1, the Correction port is input port number 0, and Position is input port number 1. This matches the code in Figure 2, where the Correction parameter appears first in the input parameter list, followed by Position. When these conventions are followed, the code generated by MatSim will invoke the primitive components properly, with the proper data.

ProcessingCompounds

A ProcessingCompound model represents a collection of components. MatSim generates a Matlab function to represent each compound model in the system. Every model in a system is represented by a Matlab function: templates are resolved through design-space exploration, a function representing each primitive is provided by the user, and MatSim generates a function for each compound. The model hierarchy translates directly to a set of Matlab functions. MatSim simply needs to ensure that the functions are called in the proper order, and that the parameter passing is performed properly.

Dataflow and Static Scheduling

Components in the ACS modeling language are modeled according to the dataflow formalism. This means that control flow in the execution graph is implied by the flow of data. A function does not execute until its inputs have been generated. A function only produces outputs when it executes. As data flows across the execution graph of a system, new functions are triggered. When MatSim generates code to represent the contents of a compound model, this implicit control flow must be translated into an explicit order in which functions execute. MatSim must generate the proper order in which to call the functions representing the contents of a compound. This proper order is a static schedule for the execution of the functions.

In order to produce the correct static schedule for a model, MatSim must resolve the data dependencies between functions in each compound. The data dependencies within a compound can be conveniently represented with a Directed Acyclic Graph

(DAG). Nodes of the graph represent the models contained in a compound. Edges in the graph represent a communication path between models. If two models have multiple paths connecting them (in the same direction), only one directed edge in the DAG is needed to show the dependency. Any communication paths in the models in the compound which connect directly to the ports of the compound need not be represented in the DAG, as it is assumed that input parameters to a function can always be read from, and output parameters can always be written to. The resulting DAG represents the data dependencies between the models contained in a parent compound. For example, Figure 3 depicts a compound model, GenCorrection, with two submodels, AcquirePosition, and DoCorrection. The DAG representing the data dependencies of this graph is fairly simple, and is depicted in Figure 4.

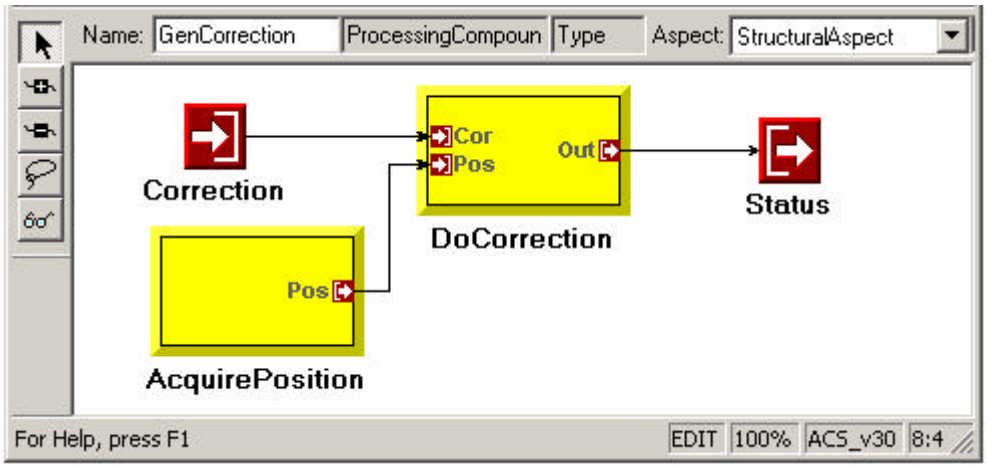


Figure 3. ProcessingCompound model GenCorrection, with submodels AcquirePosition and DoCorrection

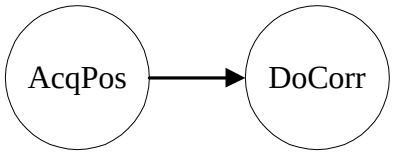


Figure 4. DAG representing the data dependencies of the GenCorrection model shown in Figure 3

Once the DAG representing the data dependencies of the components in a model has been formed, the static schedule may be derived through a topological enumeration algorithm. A topological enumeration attempts to find an ordering of the nodes of a directed graph such that each node that is selected would have an in-degree of zero if all edges which are sourced by nodes which have already been selected were removed from the graph. In the case of the DAG in Figure 4, the AcqPos node would be selected first, and then by default, DoCorr. DoCorr could not be selected first, because when the enumeration contains no nodes, there are no edges which could be removed from the graph, and DoCorr retains an in-degree of 1. In constrast, after AcqPos is included in the

enumeration, the only edge in the graph happens to be sourced by AcqPos and can be removed. DoCorr is left with an in-degree of zero and can be included in the enumeration. Figure 5 shows an example of a more complicated DAG and one possible topological enumeration.

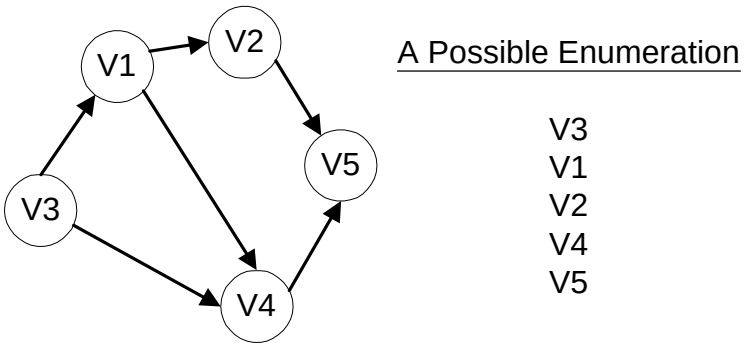


Figure 5. A more complex DAG, with one possible topological enumeration

The topological enumeration represents the static schedule for calling the functions which represent the models contained in a compound. In the case of Figure 4, AcqPos was selected first, so a call to the m-function representing the AcquirePosition primitive in Figure 3 will be called first, followed by the function representing DoCorrection. MatSim writes these two function calls to a file. This file becomes the function representing the GenCorrection compound.

Parameter Naming and Passing

As mentioned previously, the ports of a model represent the parameters of the model. This is no different for compound models. The input parameters of a function representing a compound model are named after the input ports of the model, likewise for the output parameters with the output ports. The input and output parameters of a function are known as the formal parameters of a function. This is in contrast to the actual parameters, or the parameters which are passed to a function when it is invoked. A connection sourced by an input port of a compound represents a use of the formal parameter representing that port. If the destination of such a connection is an input port of a model contained in a compound, the formal parameter is used as the actual parameter in the call to the function representing the model.

Connections between models contained in a compound represent intermediate results, for which a temporary variable must be generated. Temporary variables are named according to the name of the port which sources the connection. This is only a naming convention; any temporary name could have been used. An integer is added to the end of the parameter name to guarantee uniqueness. When a function generates an output which is to be used as an input to another function, the temporary variable is used as an output parameter for the function call representing the source of the connection, and as an input parameter to the function call representing the destination of the connection. For example, in the case of Figure 3, the AcquirePosition model sources a single connection to the DoCorrections model. This connection will cause the generation of a

temporary variable, named after the output port of the AcquirePosition model, which is named Position. The parameter name will use the port name with a unique integer attached to the end of the name as the port name, giving "Position_1," for example. Position_1 will be used as an output parameter in the call to the function representing AcquirePosition, and as an input parameter to the function representing DoCorrections.

After all function calls have been generated for a file representing a particular compound, the formal output parameters of that compound must be updated properly. The updating of formal output parameters represents input connections to the output ports of the compound model. In order to simplify the algorithm which generates code for a compound, temporary variables are generated for every output parameter written to during execution of the function. After the last function has executed, the temporary variables which represent the connections to the output ports of the parent compound are assigned to the formal output parameters. For example, the code generator will insert a call to the function representing the DoCorrection model in Figure 3. This call requires one output parameter, so the code generator uses a temporary variable, named after the output port of the DoCorrection model. When the code generator discerns that no more functions remain to be called, it will update the output parameters of the function being generated by assigning the temporary variable generated in the call to the DoCorrection function to the formal output parameter Out.

Generating a Complete Function

MatSim uses the algorithms described to generate a complete Matlab function for each compound in the system. For a given compound, the function is named after the compound model itself, because there is no scriptname attribute for a compound. Just as with primitive models, the letter "M" is attached to the front of the name for consistency. For each compound, MatSim generates a separate file, named the same as the function. The first line of the file contains, as per Matlab syntax, the function prototype, generated from the port interface of the compound for which the file is being generated. Next, Matsim executes the topological enumeration to determine the proper order in which to call the functions which represent the models contained in the compound. Function calls are written according to the port interface of the model. Parameters are passed in the proper order as dictated by the port numbering and model connections. Temporary variables are generated to hold the output parameters of function calls, and are used as input parameters to subsequent function calls. After all function calls have been written, the output parameters are updated by assigning the appropriate temporary variables to the formal output parameters of the function.

For each function MatSim generates, it writes a global instance count variable inside the function. In most functions, this instance counter plays no role. However, for models involved in a feedback connection, this global variable plays an important role, which will be demonstrated later. Figure 6 shows the function generated by MatSim representing the GenCorrection model shown in Figure 3. The function name and file name correspond to the name of the model, with the "M" added to the front: MGenCorrection. There is a single output parameter, Status, corresponding to the single output port of the GenCorrection model, and a single input parameter, Correction, corresponding to the input port of the model. The next line of text corresponds to the declaration of the global instance count variable. This is followed by the calls to the two

functions representing the models contained in GenCorrection. The AcquirePosition model is selected first in the topological enumeration algorithm, as discussed above. The scriptname attribute of the AcquirePosition primitive model has been set to acqPos, so MatSim generates a call to a function named MacqPos. MatSim passes no input parameters to the MacqPos function because the AcquirePosition model contains no input ports. The data generated from the invocation of MacqPos is stored in a temporary variable named Pos_4, named after the output port of the AcquirePosition model. MatSim next selects the DoCorrection in the topological enumeration, and generates a call to the function represented by that model. The scriptname attribute of DoCorrection is set to doCorrection, as was shown in Figure 1, so MatSim generates a call to the function MdoCorrection. The actual parameters of MdoCorrection are passed according to the port interface of the DoCorrection model. The first port of the DoCorrection model is connected to the Correction port of the GenCorrection model. This connection is represented as the use of the formal input parameter Correction as an actual parameter in the call to MdoCorrection. The second input port of DoCorrection is connected to the single output port of the AcquirePosition model, resulting in the use of the temporary variable Pos_4 generated in the call to MacqPos as the second actual input parameter in the call to MdoCorrection. The single output of the MdoCorrection function is stored in a temporary variable called Out_7. Because there are no more models to be processed, MatSim now generates code to update the formal output parameters of the function before terminating. The statement Status = Out_7; represents the connection from the output port of the DoCorrection model to the Status output port of GenCorrection. The last line of the function updates the instance counter, flagging that the function has been invoked at least once.

```
function [Status] = MGenCorrection(Correction)

global MGENCORRECTION_INSTANCE_CNT;
[Pos_4] = MacqPos;
[Out_7] = MdoCorrection(Correction, Pos_4);
Status = Out_7;
MGENCORRECTION_INSTANCE_CNT=1;
```

Figure 6. Function generated by MatSim representing the GenCorrection compound shown in Figure 3

A Complete Functional Simulation

MatSim iterates over the hierarchy of models and generates a function for each compound in the hierarchy. When generating function calls, if a template is encountered, a call is generated instead to the alternate within the template which was selected through design-space exploration. After all functions have been generated, the user may execute the functional simulation in the Matlab environment simply by calling the function representing the root model in the structural hierarchy. Executing this function will invoke in turn each of the other generated functions, tracing through the hierarchy until all user-provided functions have been called as well. By providing a function to

represent the behavior of each primitive, the designer can invoke MatSim to generate a complete functional simulation of a modeled system.

Code Generation Issues

Some issues of model semantics needed to be resolved in order to represent a real modeled system in the Matlab language. While the generated functional simulation as has been described to this point does accurately represent the system models, it does not represent the intended execution semantics of system components. Also, MatSim depends on the ability to model data dependencies as a DAG. When a feedback connection is used in a model, the directed graph representation is no longer acyclic, and MatSim cannot determine a proper static function schedule. These two issues required some minor, but important, changes to the code generation and modeling environment.

Accurately Representing Execution Semantics

Software components are implemented based on a dataflow kernel, masking interprocess and interprocessor communication details. The kernel is responsible for periodically re-invoking each component. Components are written to receive control from the kernel, perform their task, and quickly and responsibly return control to the kernel. With the code generation discussed so far, there is no concept of a scheduler to ensure all components are repeatedly invoked. If, for example, the GenCorrection model from Figure 3 represented the root model of a system, executing it once in the Matlab environment would invoke the MacqPos function once, and the MdoCorrection function once. This does not accurately represent the execution semantics of a modeled system.

The kernel repeatedly invokes each component. What is needed in the generated system is to repeatedly invoke each component. MatSim resolves this issue by generating a loop around the code representing the root or top-level model in the hierarchy. When the designer invokes this component, instead of simply executing each of its function calls once, it will repeatedly execute the scheduled function calls, properly mirroring the execution semantics of the final system. As a means of allowing a simulation to terminate gracefully, the loop executes conditionally based on a global variable called TERMINATE_SIMULATION, which the user can set in one of the user-provided component to cause the simulation to terminate.

Representing Feedback Connections

Feedback connections present a problem when attempting to represent the contents of a compound model as a Directed Acyclic Graph. Figure 7 shows a root compound model named SimpleControl. The connection from the output of GenCorrection to the input of Comparison represents a feedback connection. While attempting to perform a topological enumeration on the graph representing the data dependencies between the models contained in SimpleControl, MatSim encounters the graph represented in Figure 8. MatSim can successfully determine that ReadSensorA and

ReadSensorB can be properly scheduled. However, this graph shows that Comp is dependent on GenCor, and GenCor is dependent on Comp. MatSim cannot schedule one function before the other because it will violate the presumed data dependencies.

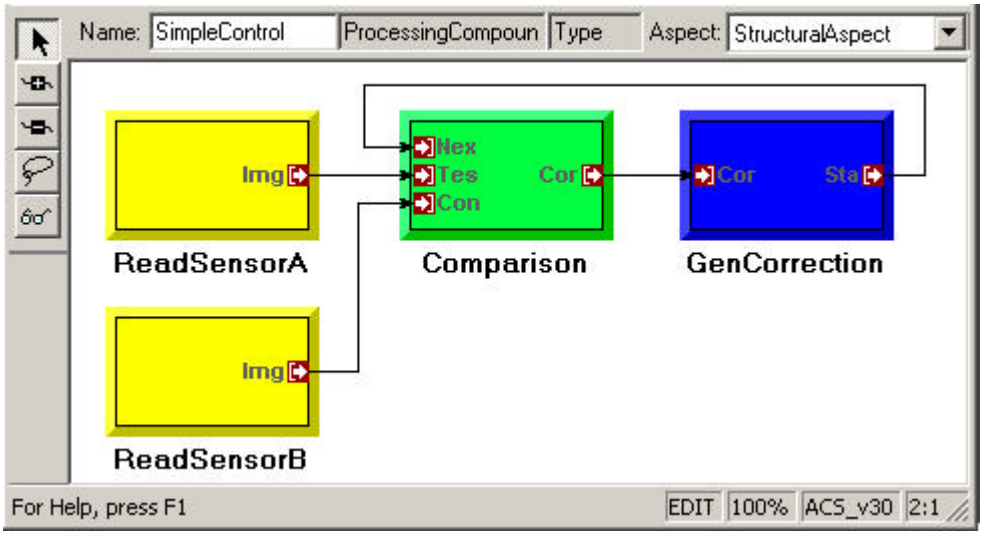


Figure 7. Compound SimpleControl, with a feedback connection from GenCorrection to Comparison

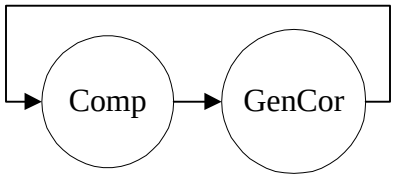


Figure 8. An unschedulable DAG, caused by the feedback connection

This directed graph, however, does not accurately represent the actual data dependency which feedback represents. In most applications, the component that receives data from the feedback connection does not receive data from the connection in the initial invocation of the function. That component simply generates an initial output and feeds the result forward in the network. Not until the component which generates the data to be fed back has had a chance to execute will the initial feedback data be generated. So the missing dimension which is not shown in the directed graph is time. While Comp does depend on GenCor, on the ith invocation of Comp, Comp depends on the data which was generated in the i-1st invocation of GenCor. It cannot be expected that GenCor produce an output before it can execute, so Comp is constructed "knowing" an initial state for the feedback input.

In order to resolve the feedback connection in such a way that the proper results are computed in the functional simulation, the user must implement Comp to "know" the initial state of the feedback connection. Further, the modeler must denote in the model which function "knows" to execute without receiving the initial data. This information is critical, because MatSim has no means of knowing if Comp is the component which should ignore the cyclic input, or DoCor. The modeling language has been augmented with an atom to allow the modeler to make such a distinction. Figure 9 shows an updated SimpleControl model, with an initializer atom and connection in place. The initializer atom allows the user to specify where to "break" a cycle in a directed data dependency graph containing a cycle. With this updated information, MatSim can now determine that Comparison has been written to ignore the initial input from the feedback connection, and can therefore be scheduled before the function representing GenCorrection.

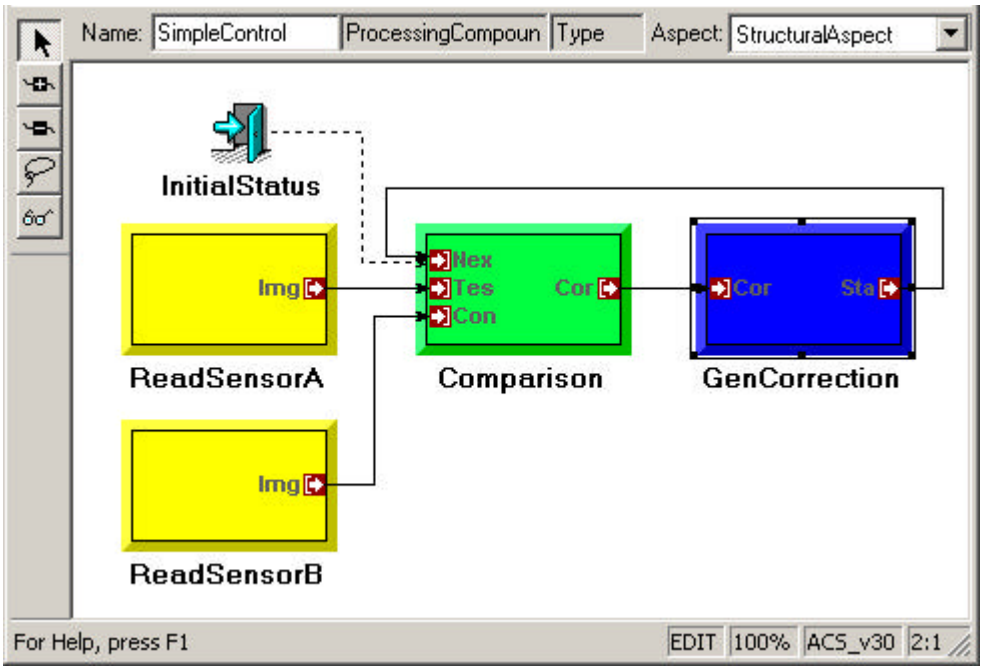


Figure 9. SimpleControl compound with Initializer atom and connection

MatSim must make a provision for passing a parameter which is involved in a feedback connection. When MatSim generates the call to the function representing Comparison, the input parameter representing the first input will not exist. The variable will be generated as an output parameter in the call to the function representing GenCorrection. This is not a problem, because Comparison has been written to ignore the input parameter on the first invocation. However, Matlab syntax dictates that some variable be passed as the actual parameter to the function on every invocation. MatSim therefore creates a variable to pass to the function. The variable must be assigned some value before Matlab will allow it to be cleanly passed as a parameter, so on the initial invocation of the function, the variable is initialized to the empty matrix. After the call to the Comparison function returns, the call to the GenCorrection function is executed, generating the feedback parameter which is to be passed to the Comparison function on

the next invocation. All temporary variables are created as local variables and are destroyed when a function returns to its caller. In the case of this feedback parameter that is generated in invocation i, and then used on invocation i+1, the scope of the parameter must extend beyond the local scope. MatSim therefore declares the temporary variable as global before it is used in the call to the first function, so on the next invocation of the function the first feedback function will receive as an actual parameter the proper value generated by the second feedback function in the previous invocation. Figure 10 shows the code MatSim generated from the SimpleControl model. The function inherits the name just as before, and the instance counter is declared, as before. The next line declares global the variable that will act as the loop condition controller, and initializes it to the empty matrix. The next line begins the loop which will repeatedly execute the functions representing the model hierarchy. The functions representing the ReadSensor components are called, followed by the statement declaring a variable called Status_3 as global. This variable is the parameter to be used to store the feedback connection results. The next statements will initialize the Status_3 variable to the empty matrix when the instance counter dictates that the function has not been executed before. Next, the function representing the Comparison model is called, using Status_3 as an input parameter. This function will ignore the value of the first parameter during its first invocation. The next function generates the Status_3 variable to be used as input to the Mcomp function on the next iteration. Because this function represents a root model, it was not technically necessary to declare Status_3 as a global variable, because the variable will not leave scope before it is needed again. However, in general, the generated function will not contain a loop, and allowing the execution of the code as is has a relatively mild impact on performance, and produces the correct results. Therefore MatSim simply generates code representing the common case.

```
function MSimpleControl()

global MSIMPLECONTROL_INSTANCE_CNT;
global TERMINATE_SIMULATION;
TERMINATE_SIMULATION = [];
while isempty(TERMINATE_SIMULATION),
   [Img_0] = Mread_sensA;
   [Img_1] = Mread_sensB;
   global Status_3;
   if isempty(MSIMPLECONTROL_INSTANCE_CNT)
      Status_3 = [];
   end
   [Correction_11] = Mcomp(Status_3, Img_0, Img_1);
   [Status_3] = MGenCorrection(Correction_11);
   MSIMPLECONTROL_INSTANCE_CNT = 1;
end
```

Figure 10. Code generated by MatSim representing the SimpleControl model displayed in Figure 9

Bit-Width Simulation

When a designer explores different algorithms, it is often desired to perform an analysis of the effects of fixed-point limitations on the correctness of an algorithm. This type of trade-off analysis is most appropriate in a simulation setting, allowing a designer to determine the optimal bit-width required for a particular application, without needing to implement and test each solution.

MatSim provides a limited support to the designer to perform bit-width tradeoff analysis. When a modeler represents a system, he/she includes in the attributes of each data port the datapath width to be used for that communication path. MatSim ensures that the widths specified in the source and destination ports of each connection are consistent. Matlab performs all mathematics in double-precision floating point format, and MatSim cannot change that. However, between computations, MatSim can round parameters to the equivalent widths specified in the ports. This is what is done when the user specifies to include a fixed-point arithmetic simulation during code generation. The rounding is performed by a function called roundfix, which takes a vector and a bit-width as inputs, and outputs the vector with each element rounded to the precision specified. Figure 11 shows the function generated by MatSim representing the GenCorrection model from Figure 3 with bit-width arithmetic simulation code included. The first statement after the function declaration is now a call to roundfix to round the formal input parameter Correction. After the call to MacqPos, the output Pos_4 is rounded as well. Each call to roundfix contains the parameter 16, representing the bit-widths specified in the model on each port. In this case, each port happened to be set to 16 bits.

```
function [Status] = MGenCorrection(Correction)

[ Correction ] = roundfix(Correction, 16);
global MGENCORRECTION_INSTANCE_CNT;
[Pos_4] = MacqPos;
[ Pos_4 ] = roundfix(Pos_4, 16);
[Out_7] = MdoCorrection(Correction, Pos_4);
[ Out_7 ] = roundfix(Out_7, 16);
Status = Out_7;
MGENCORRECTION_INSTANCE_CNT=1;
```

Figure 11. Code generated for GenCorrection model with fixed-point simulation code included

By including the fixed-point simulation in the functional simulation, the designer is allowed a somewhat closer view of what to really expect during component execution on a fixed-point architecture. If the user provides components which accurately represent fixed-point arithmetic during functional simulation, a better fixed-point simulation will result.

Functional Simulation Conclusions

MatSim provides the designer with the capability to generate a Matlab representation of system algorithm models. This representation can then be executed, along with user-provided simulation components representing system primitives, to verify the models and experiment with algorithms. MatSim allows a developer to apply model-integrated design techniques at the very earliest stages of a system design, during algorithm development and concept design.

VIRTUAL PROTOTYPING

A virtual system prototype is a simulation-based representation of a system that can demonstrate core design behaviors. A virtual prototype in the context of the ACS toolset is distinct from the functional simulation discussed in Chapter III. A functional simulation is used to experiment and test models. The functional simulation does not simulate the runtime middleware, and has no concept of system resources. A virtual prototype not only simulates the functionality of the system, but also can interface to the system hardware, allowing the exchange of data between the simulation environment and actual system components at runtime. The virtual prototype executes on a layer of simulated middleware, just as system software components execute. A virtual prototype provides a more accurate representation of the system, and provides a unique platform on which to perform component and system testing, as well as system integration.

Extending the Modeling Environment and Synthesis Tools

Modeling Environment Extensions

Supporting the generation of a virtual prototype from system models requires a few extensions to the modeling environment. The extensions allow the user to explicitly represent simulation components, and to interface those components to other components in the network. The resource modeling capabilities were extended to allow the modeler to represent the Matlab processing environment as a processing element in the system. Figure 12 depicts a resource model for a heterogeneous network, with the Matlab environment modeled as a processing resource in the network. The modeling language requires that there be only one connection to the Matlab model, and that connection must connect to the host. A network is allowed to contain only one Matlab model.

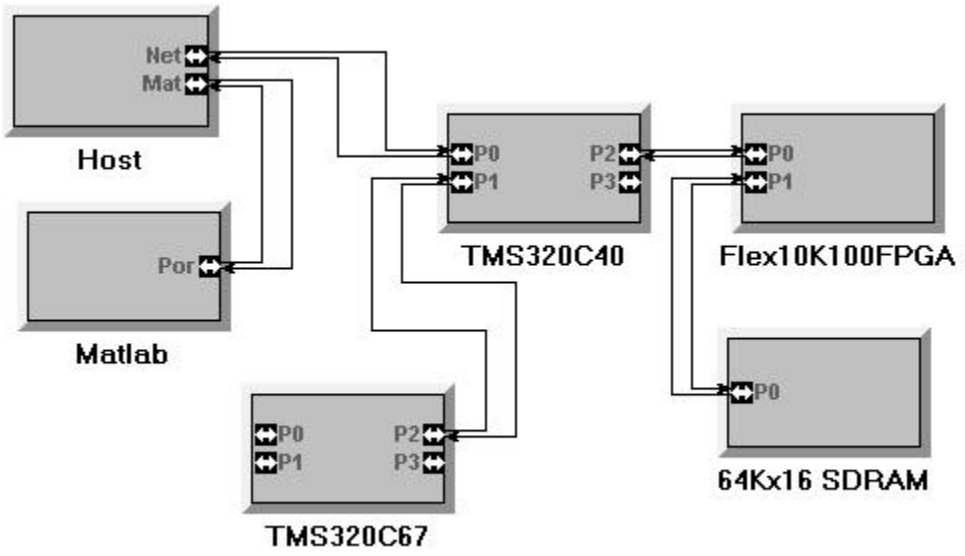


Figure 12. Resource Model showing the Matlab environment interfaced to the host

To facilitate the exchange of data between the Matlab environment and the network, a new communication protocol was established. Figure 13 shows some of the attributes of the port of the Matlab environment model, showing the Matlab Protocol being selected as the communication protocol. The port on the host processor which connects to this port must also have Matlab Protocol selected.

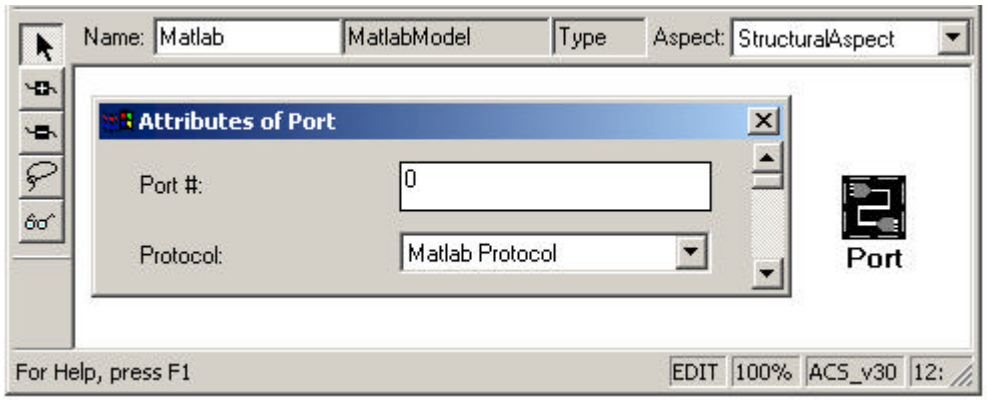


Figure 13. Port Attributes showing Matlab Protocol as the selected communication protocol

The modeling language has been extended to allow the designer to explicitly create simulation components to be integrated into a system. Setting the resource category of a ProcessingPrimitive model to Matlab creates a simulation component. By selecting Matlab for the resource category of a particular component, the designer dictates that a particular component will be implemented in the Matlab language and

should be executed in the Matlab environment. All primitives whose resource categories have been set to Matlab will be mapped to the Matlab resource during synthesis.

Extending Codflow: Prototype Synthesis

The codflow interpreter was extended to allow the synthesis of a virtual prototype from the system models. Because the Matlab environment was modeled as a kind of processor, similar to a DSP, codflow could treat it as just another node in the network when generating middleware initializations and configurations. However, a few modifications were needed to perform some configurations of the simulated middleware in the Matlab environment. These modifications included the generation of a list of simulation components which are to be executed in the Matlab environment, and the generation of an initialization file stipulating the number of streams and processes needed in the Matlab environment. Specification of how the streams in the execution graph to execute in the Matlab middleware is generated in the form of commands in the runtime command file. It was not required to modify codflow to perform this command generation for the Matlab node because to codflow, the Matlab node appeared as a typical processor node.

An Execution Environment for Running Matlab Processes

An execution or runtime environment supports the execution system components on a node in the resource network. Each node in the network provides a stand-alone execution environment, which can communicate with other nodes in the network. The execution environment allows the runtime system to be easily configured from the model interpreter. It also provides services to system components, abstracting the details of inter-process and inter-processor communications. The runtime environment simplifies the details of building components. In order to allow Matlab components to accurately represent their implementation counterparts, a Matlab execution environment was constructed to simulate the execution environment of a processor node in the network. The Matlab execution environment allows simulation components to be constructed following the same semantics as other software components. The execution environment also supports and abstracts communications with the rest of the network, allowing Matlab components to exchange data with other components in the network. The implementation of the Matlab execution environment follows closely from the concepts implemented in the execution environment run on a processor.

The Processor Execution Environment

Each processor in the network runs a small dataflow kernel which implements the concepts of the execution environment **Error! Reference source not found.**. The kernel, based on a previous kernel implementation **Error! Reference source not found.**, supports deterministic dynamic memory management, stream-based inter-process communication, inter-processor communication, and process scheduling and

management. Simple services are made available to components through an API. Figure 14 depicts the organization of the runtime kernel of a network processor. The processes on top represent user-provided components, which were represented as primitives in the system models. The API layer services calls made by the processes. Through servicing these calls, the API layer interacts with the stream, memory, and process management facilities of the kernel. When it is determined that data needs to be exchanged with another node in the network, the inter-node communication protocols are invoked, which transfer data via the communication hardware. Each layer abstracts details about the lower layers and provides services to the upper layers.

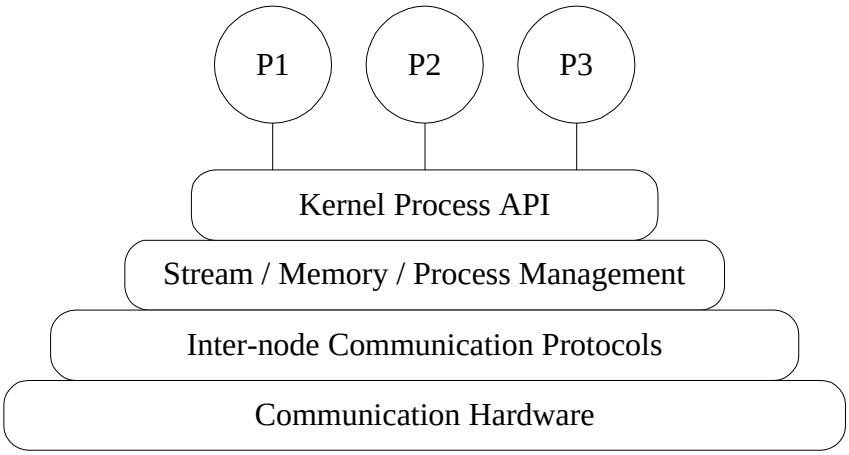


Figure 14. Organization of runtime kernel executing on a processor in the network

The models are used to configure the kernel on each node in the network. A minimal amount of configuration information is compiled into the runtime environment. Instead of compiling this configuration information into the runtime environment, the network nodes are configured to receive command messages from the host containing configuration information. Through these command messages a kernel is instructed to, for example, install a stream, to connect a stream to a particular port of a source process, to activate a process, or to unhalt the node. Only when a node is unhalted can it begin to execute processes. All commands are generated by the codflow interpreter from the models.

Once the execution of system processes has begun, the kernel cycles through all the processes which it has been instructed to execute. A process is simply a subroutine that the kernel calls. The subroutine is required to check the status of any input streams it is to read from before attempting to perform computations, to ensure the presence of data to operate on. It must also check that there is sufficient room in its output streams to store the results of the computations. In this manner, the dataflow formalism is adhered to, in that the flow of data across the processing graph dictates the control flow of the graph. After the kernel invokes each function, it invokes the inter-processor communication facilities to attempt to send any pending messages, and to receive into the kernel data structures any messages which the hardware may have received. Next, the

kernel checks the command message stream and appropriately dispatches any messages sent from the host.

Kernel state is maintained through persistent data structures. Processes can gain access to the contents of these data structures through the API. When a process fills a buffer to be passed to another process, it passes that buffer to the kernel through the API. This buffer is held in a stream send queue data structure, waiting to be sent by the communication hardware. The stream management layer will enqueue a send request with the communication protocol layer. When the protocol, or interface, layer services the send request, it will retrieve the buffer to be sent from the stream and will invoke the communication hardware to send the buffer. How the protocol layer and communication hardware send the buffer depends on the protocol that is used and the capabilities of the communication hardware. The communication hardware on the receiving node will receive the buffer and will wait to be serviced by the remote protocol layer. When the remote protocol layer services the receive request it will pass the received buffer to the remote stream management layer, which stores the buffer into the stream data structures. When the process that is designated as the destination for the stream is invoked, it will retrieve the buffer from the kernel stream through the API, and can then use the data. In this fashion data can be exchanged between processes on different nodes in the network. When the source process and destination process reside on the same node in the network, the stream management layer simply forwards all sent buffers to the receiving stream, instead of passing the buffers through communication hardware. These streams are referred to as "local" streams.

Interface functions implement the communication protocols used by the kernel. There are two types of interface functions, a send and a receive. A send function on one node matches with a receive function on a remote node. A receive function expects data in the order and format that its counterpart send function emits. Each node may have several communication ports or channels. Each channel is assigned a particular set of interface functions. The interpreter coordinates the assignment of interface functions, ensuring that two connected nodes "speak the same language" by having valid pairs of interface functions across each channel.

Matlab Execution Environment

In order to execute processes in the Matlab computational environment, and be able to exchange data with the processing network, a simulated execution environment was developed for Matlab. This execution environment consists of many of the same concepts as the execution environment for the other processor nodes in the system. Figure 15 depicts how the Matlab execution environment, or kernel layer, is organized. Processes execute in the Matlab environment, with services provided by the kernel layer through an API, just as before with network processes. The API exchanges data and invokes services from the Stream / Process Management layer. There is no need for memory management in the Matlab kernel layer because all memory allocation is performed by the Matlab computational environment implicitly. The kernel on one of the nodes in the network would at this point invoke services from the interface functions to perform message passing between nodes. In the case of the Matlab kernel layer, when the stream and process management facilities are invoked, they store data and state

updates to a set of data structures which maintain the kernel layer state in a persistent fashion. The mechanics of how data is exchanged with other nodes in the network will be discussed in the next section. The process management section of the kernel invokes one process every time it is invoked, and each process must query the status of its input and output streams via the API before it performs computation. Each process is invoked in a round-robin fashion, with dataflow dictating which process will actually perform its computations.

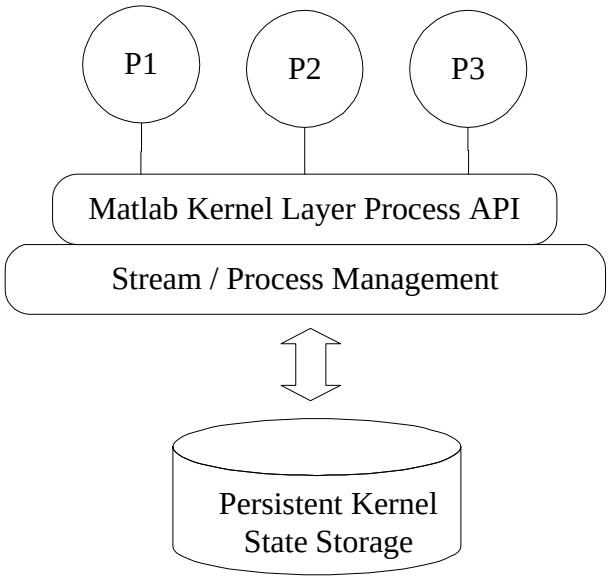


Figure 15. Organization of Matlab kernel layer

The kernel layer stores its state in a set of global data structures. It maintains a table to track all kernel layer streams and a process table to manage processes allocated to the Matlab execution environment. These data structures are configured through the receipt and dispatch of host messages on system startup. However, as with the other nodes in the network, the interpreter provides a small set of "bare bones" setup code, to provide for the initialization of communications with the host node in the form of an initialization function.

Communicating With the Host

The kernel layer, in order to perform its basic function, must exchange data with the host node in the network. Matlab provides an interface, called the Matlab engine **Error! Reference source not found.**, whereby a standalone program can manipulate and use the Matlab computational environment. The engine allows a standalone C program, for example, to create Matlab variables and place them into the Matlab workspace, to invoke native Matlab functions, and to retrieve the results of the functions to use in other computations in the C program. The engine gives to a C program the full functionality of the Matlab command line interface.

Through the Matlab engine interface, the system kernel executing on the host PC can exchange data with the Matlab kernel layer. The Matlab protocol was implemented as a set of interface functions to enable this exchange. The send function in the host kernel passes data to a receive function in the Matlab kernel layer. The Matlab receive function is written in native Matlab code. There is also a native Matlab send function, which has a corresponding receive function on the host kernel. These interface functions mask the details of buffer exchanges from their respective kernels.

The Matlab kernel layer does not execute as a stand-alone process on the host PC. It executes as a slave to the host kernel. The host kernel must periodically invoke the Matlab kernel layer, allowing it to send and receive data, and to execute processes. The Matlab kernel layer has been designed to perform its tasks quickly and efficiently, and to responsibly return control to the host kernel promptly after being invoked. The details of how the Matlab kernel layer is invoked is abstracted by the host interface functions from the rest of the host and network. Outside of the interface functions, it appears as though the Matlab kernel layer is executing concurrently with the other nodes in the network. The kernel layer is invoked after every buffer send and buffer receive, in order to allow Matlab processes to perform their computations and send and receive data. The kernel layer is invoked through the Matlab engine by calling a Matlab function designated as the kernel layer entry point. This entry point invokes the process scheduling and stream management facilities of the kernel layer, and then returns control to the host kernel.

The interface functions are responsible for exchanging data with the Matlab environment. Data is always exchanged between processes in the form of messages. A message consists of a header and a body. A message header contains information as to the source node, stream and channel, as well as the destination node, stream, and channel of the message. The stream and channel management facilities of a kernel use this information to route messages to their proper destinations. A message in the Matlab execution environment is represented as two vectors. Each field of a message header has a corresponding index into a header vector. A message body in the host kernel is represented as simply an array of numbers. This is consistent with the Matlab representation: a vector of numbers. When the host send function prepares to send a message to the Matlab kernel layer, it allocates two vectors from the Matlab workspace, one for the header and another for the body. The data from the host header is then copied, field-by-field, into the header vector. The message body is then copied into the body vector.

The Matlab computational environment supports only the double-precision floating-point data type, so all fields are cast to doubles then they are copied. When a message body is copied, the interface function must make an assumption about the current data format of the host message body. Regardless of the explicit type declared in the message body data structure, a component can store in a message whatever data in whatever format is desired, as long as the source component is consistent with the receiving component. However, the interface functions between the host and Matlab must make an assumption about the format a message body is in, because it must convert the data into double precision format. By convention, the interface functions assume that all message bodies are currently stored in single-precision floating-point format, and the responsibility for ensuring that this is the case is placed on the developer of the components. Because there is no characterization of the type of information being passed

in a buffer, the interface functions are left with no other choice but to assume a format, otherwise, it has no means of knowing how to perform the conversion to/from double.

Once the send function copies the header and body into Matlab vector variables, the Matlab kernel layer receive function is invoked. The host send function through a Matlab engine call, invokes a Matlab m-function representing the receive function, passing the header and body vectors as parameters. This function simply verifies that the message passed is consistent, and copies the message into the kernel layer global stream table for later reference through the API. The Matlab send function then returns control back to the host send function. The send function then frees the vectors which were allocated from the Matlab workspace. Next, the send function invokes through an engine call, the Matlab kernel layer entry point, passing control momentarily to the kernel layer to allow process execution. When the call returns, the send function returns control to the host kernel.

The host receive function is very similar to the send function, only performing the actions in the reverse order. The receive function first invokes the Matlab kernel layer send function through an engine call. If during processing, a Matlab process has enqueued a message to be sent to the host, a flag is set in the state storage. When the kernel layer send function is invoked, it queries the flag and retrieves the message to be sent. The message is returned as two parameters, a header vector and a body vector. When the send function returns control to the host receive function, the receive function retrieves pointers to the two output parameters of the kernel layer send function, and determines if a message was in fact sent. When a message is sent, the receive function allocates a message buffer from the host memory management, and copies, field-by-field, the header vector and body vector into their appropriate locations. Again, the host must assume that the format that the message body is supposed to be in is single precision floating point. When the message is successfully copied, it is passed to the stream management layer of the host kernel, for later access by system processes. After storing the copied message, the receive function invokes the entry point of the kernel layer through an engine call, and then returns.

These interface functions allow the Matlab execution environment to exchange data at runtime with processes running on the network. When a process executing on a DSP in the network sources a stream connected to a process mapped to the Matlab execution environment, the interpreter facilitates the exchange of data via the insertion of forwarding components on the host and any other nodes in the path to the DSP executing the source process. A forwarding component simply forwards a message from one stream to another, allowing a message to propagate across a node. This communication framework allows data to be exchanged between Matlab processes and any other process in the network.

Figure 16 depicts the full kernel layer software and how it communicates with the host kernel. Processes written in the Matlab language execute, interacting with the kernel layer through the API. The API allows access to the stream and process management facilities of the kernel, which are responsible for updating the kernel layer data structures. The kernel layer is periodically given control by the host kernel, where it invokes one process and then performs communications housekeeping, checking whether the communications layer has updated any streams with new information. The actual communications functions are not invoked by the kernel layer itself, but rather by the

host communications layer. The host send function translates message data structures to the Matlab kernel layer representations of those data structures, as vectors, and then invokes the Matlab receive, which updates the kernel layer state storage. The host receive function invokes the Matlab send function, which retrieves a message needing to be sent to the host. The receive function then retrieves the Matlab message, and translates it to the host kernel message format, and passes it along to the host kernel stream management facilities, and processing continues.

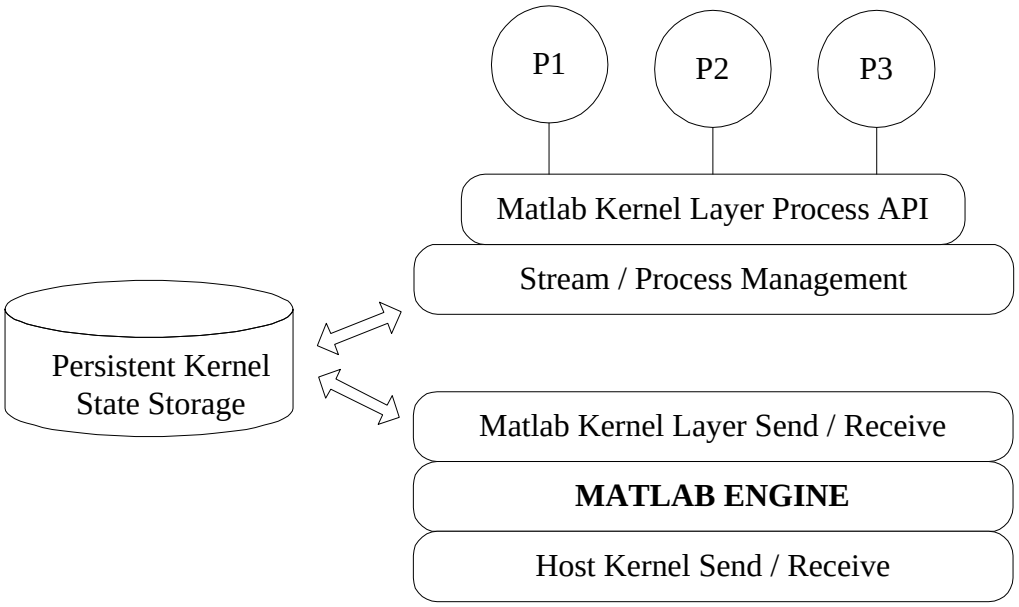


Figure 16. The Matlab kernel layer and interface to the host kernel

Virtual Prototyping

With the extensions to the runtime environment and modeling tools, a developer can now construct a virtual prototype of a system. A system can be modeled as a set of simulation components, which can be implemented using the Matlab language and kernel layer API. A functional system can be synthesized from the models with the codflow interpreter, and can be loaded onto the resource network. At first, this resource network could consist of a host PC with the Matlab environment. After compiling and loading the code, the developer may test the system to verify its behavior. After the behavior has been verified, the resulting system represents a virtual system prototype, exhibiting the functionality of the target system, but implemented using a simulation language. The virtual prototype will obviously not meet the performance constraints of the target system, but will demonstrate the core behaviors of the target.

After the virtual prototype has been constructed, it can be used during the component implementation design phase. The modeling tools automate the selection of implementation alternatives from the models. When a system is modeled, alternative implementations for each component can be explicitly included in the models. As one

alternative implementation for a component, the user provides a Matlab-based simulation implementation. Another implementation would be the target implementation. When constructing the virtual prototype, the target implementations for system components need not be constructed, nor even modeled. However, the user should make use of template models to allow alternative implementations to be modeled later. Through the design-space exploration utility, the designer may select the simulation implementation for each component, and then use codflow to generate the simulation implementation. After verification of the virtual prototype, the user can use the prototype as a testing framework for testing component implementations. When a component is implemented, the tools can be used to select the simulation implementations of all system components except for that particular component, whose target implementation is included in the final design. After this design is synthesized and loaded, the user can verify that the component's target implementation exhibits the same behavior as the simulation implementation. The virtual prototype provides an ideal framework because each simulation component has at this point already been verified, and the Matlab components can be used to manipulate the inputs and display the outputs of the component under test. Each component is tested in the context of the system, and the designer is saved the effort of building a testbench framework for each component.

The virtual prototype also provides an excellent framework to perform system integration. As components are implemented and tested, they may be integrated into the prototype system, replacing their simulation-based counterparts. As more components are included in this system, integration issues may be uncovered. Previously, integration issues could not be thoroughly examined until most or all of the components had been implemented and could be included into the system. Because the virtual prototype system allows components to be integrated seamlessly, integration issues can be examined much earlier in the design phase. Systems can be synthesized consisting of half simulation-based components, and half target implementation components. Through testing these systems, the designer can examine how components interact in a more controlled environment.

Matlab components can be used as a debugging tool. Because network components can exchange data with Matlab components at runtime, the designer can insert a Matlab component in the data path between two components to visualize the messages being exchanged. Matlab components can also be used to modify the inputs to a network component, allowing a greater versatility in testing.

The user is provided with a powerful tool to perform verification, debugging, component testing and system integration through virtual prototyping. By providing an interface between the Matlab computational environment and the processing network, a developer is allowed to utilize the power of Matlab at runtime, intermixed with implemented components.