# A Component Composition and Customization Language for the Liberty Simulation Environment

## ABSTRACT

To build efficient hardware, designers must be able to *rapidly* construct simulators for a wide range of candidate designs. Fortunately, different hardware designs utilize many similar hardware blocks, and thus a good library of reusable components can greatly ease the task of constructing a simulator. Unfortunately, existing simulator construction methodologies do not facilitate construction of such libraries.

The Liberty Simulation Environment (LSE) addresses this gap in simulator construction technology. LSE consists of a user-extensible library of reusable components, the Liberty Structural Specification (LSS) language, and a tool that generates an executable simulator from an LSS program and the component library.

This paper discusses the design and implementation of the LSS language. The LSS programming language is unique in that it is designed not to specify the execution behavior of a simulator, but rather, to specify how components from the library should be composed and customized to form new components or to build a complete system model. LSS allows the designer to specify an arbitrarily deep hierarchical description of components in an intuitive fashion. Furthermore, the designer can use component parameters and system connectivity in one level of the hierarchy to determine the structure of lower levels in the hierarchy. To allow this *use-based specialization*, the LSS language uses novel evaluation semantics, an algorithmic specification of structure, and several programming language techniques including type inference and aspect oriented programming.

## 1. INTRODUCTION

In order to design an efficient hardware system, designers must balance many considerations when making their design decisions. Often, there are no analytical models with which to evaluate and optimize a design given the constraints on the system. As a result, designers are forced to simulate candidate designs in order to discover their characteristics and address any unforeseen shortcomings.

Often, the limiting factor preventing the exploration of many candidate designs is the time taken to build simulators. The process of mapping a structural design which is inherently concurrent into a sequential simulator is tedious and error prone [13]. It would be useful to the designer if the structure of the simulator closely resembled the actual hardware being modeled. Further, once a simulator has been written, it is desirable to leverage this code in other simulators. Since different hardware systems are often very similar in structure and functionality, this reusability could be achieved by using customizable components from a extensible library of components. Unfortunately, most existing simulation methodologies preclude the creation of libraries of reusable components.

The most common simulator construction technique, hand coding the simulator in a traditional sequential programming language, suffers from the difficulty of mapping a concurrent design into sequential code and from an inability to reuse existing code in the construction. Simulators written in this style typically model the structural hardware components and their communication using functions and function invocation, respectively. Unfortunately, the hardware concurrency that must be modeled by the software forces code for a component to be partitioned so that it may be appropriately interleaved with code from other components. The specific way in which the code must be partitioned is often dependent on the *connectivity* of the system rather than the *behavior* of the component [13]. Thus, producing components that work in varying connection schemes is extremely difficult.

To address the shortcomings of these sequential simulators, researchers have proposed domain specific simulation systems [3, 11, 15]. Unfortunately, these systems are restricted to modeling hardware in that domain only (e.g. simple pipelined microprocessors) and thus do not provide a generic solution.

Other simulation systems use structural composition and concurrency to avoid the pitfalls of hand-written sequential simulators. Unfortunately, these systems are not well suited for creating libraries of reusable components and are not capable of specifying systems of arbitrary granularity. Low level hardware description languages [6, 12] model systems at too fine a granularity to explore system-level design decisions. While it is possible to build coarser grained components, these components are not sufficiently parameterizable to create an effective module library.

Higher level concurrent-structural simulation systems [2, 7, 9] exist, but they suffer from inappropriate concurrent semantics or lack a stringent contract governing inter-component communication. As a result, components developed independently cannot always communicate with one another, precluding a useful library of components. For some systems, the concurrent semantics do not allow arbitrarily timed communication and thus force a particular decomposition into components, precluding reuse. In other systems, designers are free to specify the interaction of components, especially control interaction, in an ad-hoc fashion with custom communication interfaces that are optimized for the interaction of a given set of components. This specialization makes it very hard to have two components inter-operate if they were not explicitly designed to interact in the first place [13]. Thus, components described in these systems are inappropriate for component libraries.

We developed the Liberty Simulation Environment (LSE) to address these shortcomings of existing systems. LSE allows for the construction of hardware system simulators by specifying the interconnection of components instantiated from a user-extensible library of reusable components. In this paper, we present the Liberty Structural Specification (LSS) language, a language developed to allow concise and clear descriptions of hardware systems and specifications of powerful and highly parameterizable components. By allowing dynamic component behavior to be specified separately, the LSS language is able to leverage novel evaluation semantics making the system more parameterizable. In addition, the LSS language utilizes programming language techniques such as type inference and aspect-oriented programming to increase specification ease and power.

The rest of this paper is about the design and implementation of the Liberty Structural Specification language. Section 2 provides an overview of LSE. Section 3 will discuss the necessary characteristics of a system description language. Section 4 describes the LSS language and how the key features address the concerns raised in Section 3. Section 5 explains the evaluation semantics needed to make the syntax behave as expected. Section 6 presents some experience using the language. Section 7 describes related work. Finally, this paper concludes in Section 8.

## 2. OVERVIEW OF LSE

The Liberty Simulation Environment automatically constructs an executable simulator from a hardware system description written in the LSS language. These LSS system descriptions may utilize components from a user extensible library in order to simplify description. The simulator construction flow is shown in Figure 1. The compiler which transforms LSS descriptions into simulators is comprised of two components. The first piece, the LSS interpreter, emits a system's static structure and runtime configuration parameters by interpreting the LSS system description and each of its components' descriptions. Then, the emitted system structure is combined with each component's runtime behavior by the codegenerator to produce an executable simulator.

Hardware components in LSE are instantiations of predefined or user-defined hardware component abstractions called *modules*. When the module is instantiated with the parameters that it defines, a customized *module instance* is produced for use in the system. A designer will describe the system's communication paths by specifying connections between the *ports* on module instances. Thus, a system is described in the LSS language through the customization and instantiation of modules and the specification of port-to-port connections.

In addition to declaring the ports and parameters that an instance has, a module will also specify the runtime behavior of the instance that is built from it. An LSS module can specify its behavior structurally by instantiating other modules and internally connecting the ports of the resulting sub-instances to its own ports. Modules defined in this way are called *hierarchical modules*. Alternatively, a module can specify its runtime behavior by including a behavioral specification written in a behavioral specification language (BSL). This BSL is more suited toward specification of computation rather than structure. Modules defined in this style are called *leaf modules*. Currently, LSE uses a stylized version of C for the BSL, however, LSS is independent of this language.

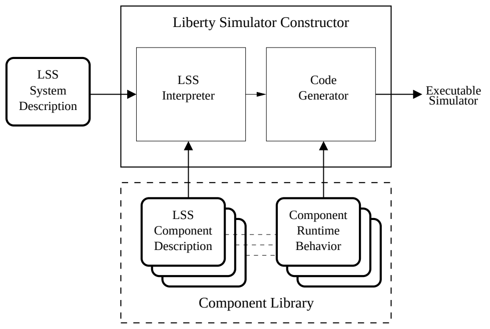


**Figure 1: Overview of the LSE system.**

The output of the LSS interpreter is used by the code generator to weave together the instance behaviors specified in the BSL with a scheduler to correctly order the computation specified by the system's connectivity. Topics related to concurrent execution semantics and building an efficient simulator from the behavior descriptions are beyond the scope of this paper, but they are discussed in references [10, 13].

## 3. LANGUAGE REQUIREMENTS

To manage the complexity of system design and allow for the specification of highly customizable components, the system specification language must be sufficiently powerful to allow concise specifications of designs. This language should facilitate the design process by allowing the designer to describe hardware structurally, to define components which comprise the system, and to specify what data should be collected from the simulation. Furthermore, the specification should allow the system to be analyzed by both the designer and automated tools.

Specifying the static structure of a hardware system may seem trivial, however, the size and complexity of the system makes the obvious brute force techniques extremely cumbersome to use. While not ideal, it may be manageable to enumerate all the components in a given system. However, the massive number of connections precludes complete enumeration of the entire system structure. Luckily, much of a system's structure is regular and can be concisely specified using algorithmic or idiomatic descriptions. A language could, for example, provide idioms for connecting a bus, describing a butterfly network, or repeating some subset of a machine's structure. In order to effectively specify a system's structure, the specification language *must* include a convenient syntax for concisely specifying regular structure.

In addition to specifying the system structure, the system design language must also be able to specify the interfaces and behavior of the components which make up the system. To allow for component reuse (via the definition of truly generic components) the

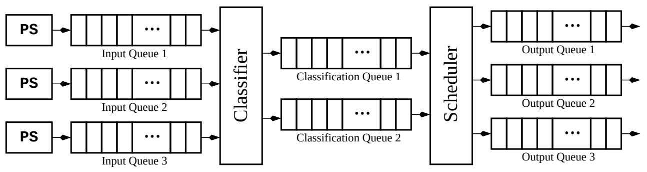


**Figure 2: A block diagram of a hardware router.**

description language must allow module definitions to export parameters which control *both* a module's runtime behavior and its interface to other components. For example, the parametrization should allow for polymorphic types on communication paths to allow the definition of type-neutral components such as generic data routers and also allow unanticipated behavior modifications such as a novel replacement policy on a cache memory.

Finally, the design language needs also to be able to specify what data will be collected from the simulation. This specification, while tightly coupled to the behavior *and* structure of the machine, ought to be specified independently since, for even a fixed hardware model, the set of statistics collected from a system simulation may vary from one simulation to another.

In the next section, we provide an overview of the Liberty Structural Specification and describe how it achieves each of these goals.

## 4. OVERVIEW OF LSS

In this section, we will introduce the LSS language by walking through a small example of system specification. We will then discuss how to build reusable components by composing existing modules or by referencing externally specified behavior. Finally, we will discuss how to instrument a system specification to collect statistics, drive visualization, or debug designs.

### 4.1 System Specification

Consider the high level view of a router shown in Figure 2. Three packet sources (marked `PS` in the figure) feed the routers input queues. The router then classifies the packets coming out of the input queues and places them into the appropriate classification queues. The packet scheduler pulls packets from the classification queues, determines their destination, and sends the packets into the appropriate output queue which ultimately feeds one of the routers output ports.

Figure 3 shows the LSS specification for this router. The first thing to notice from the specification is that LSS is an imperative programming language that describes system structure rather than a format for enumerating instances and connections. For programmers familiar with the C or JAVA programming languages the syntax should look familiar, although unique syntactic elements exist. The description begins by instantiating the modules which make up the system (lines 2-19). The instances are declared using the `instance` keyword, specifying the instance name and the module name from which it should be instantiated (lines 2-5, 7, and 13). An instance can be customized by setting its parameters using simple assignment statements. For example, we set the classification policy (line 9) of the classifier, the routing map of the crossbar (line 15), and the queue size of the input queues (line 25). The system communication paths can be specified by connecting together instances' ports using the `->` operator (lines 31-32).

```
1  /* Router */
2  instance ps : packet_source[3];
3  instance input_q : mqueue[3];
4  instance class_q : mqueue[2];
5  instance output_q : mqueue[3];
6
7  instance classifier : demux;
8  /* Define class decision behavior */
9  classifier.decide =
10   <<< if(get_priority(data) > 10) return Class1;
11       else return Class0; >>>;
12
13 instance scheduler : crossbar;
14 /* Define packet scheduler behavior */
15 scheduler.map =
16   <<< /* The 2nd high priority input overrides lower
17          priority input. */
18       in_to_out[get_out_port(data[0])] = 0;
19       in_to_out[get_out_port(data[1])] = 1; >>>;
20
21 var i : int;
22
23 /* Connect packet sources to input queues */
24 for(i = 0; i < 3; i++) {
25   input_q[i].size = 4;
26   /* Drop packet if queue is full */
27   input_q[i].in.control =
28     <<< return SIM_port_ack |
29                SIM_port_status_data(istatus) |
30                SIM_port_status_enable(ostatus); >>>;
31   ps[i].out -> input_q[i].in;
32   input_q[i].out -> classifier.in[i];
33 }
34
35 ...
36 /* Repeat the above for the other connections */
```

**Figure 3: LSS description of the hardware router.**

In addition to these declarations and statements, an LSS program can define variables (line 21) and use standard C-style control flow statements (lines 24-33) and expressions. These constructs, when used together, allow for an *algorithmic* specification of structure, rather than a declarative one. This approach is far more powerful than supplying idioms for common connection schemes, since *any* regular pattern, rather than a limited set, may be concisely expressed. Moreover, specifying the structure using a programming language allows for the structure to be parametrically controlled. As the example in the Section 4.2 will show, a module parameter, for instance, can influence the network topology in a communication fabric. To further increase the specification clarity and reduce redundancy, algorithms for common connection or instantiation patterns can be packaged into functions or other modules.

Note that while the syntax for declaring an instance looks strikingly similar to declaring a variable, the two operations are *significantly* different. Instance declarations create components of the system being described, while variable declarations create local storage for use in algorithms specifying what instances should be created, how they should be parameterized, or how they should be connected. Instance declarations, parameter declarations, parameter assignment, and connect statements have *side-effects*. In fact, these side-effects are the *only* outputs an LSS program has.

### 4.2 Module Declaration

After seeing how to instantiate and connect together module instances in the LSS language, the next logical step is to see how to define modules. As was mentioned earlier, there are two ways to define modules in LSS. The first, defining modules hierarchically, uses the syntax discussed above with only a few more statements and declarations. Specifying leaf modules, on the other hand, uses only a limited subset of the features discussed. When defining modules it is possible to put them into packages. These packages provide a mechanism to organize module libraries and they also provide definitions for *events* which module instances can emit at run-time (see Section 4.5).

### *4.2.1 Hierarchical Module Declarations*

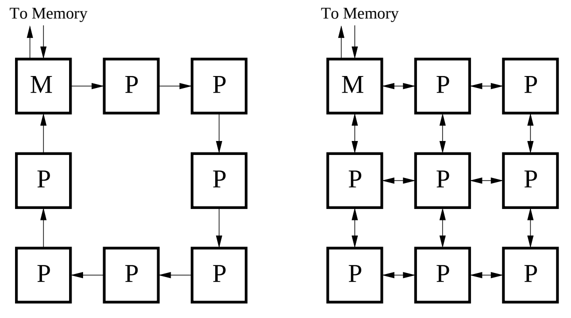
Consider the task of specifying a system-on-a-chip. As part of the system, one needs to specify an array of processing elements, and the interconnection network between the processors. Figure 4 shows two possible design alternatives. Suppose that we wanted to build an LSS module which had a parameter to select the number of processors in the network and other parameters to specify the network topology. Figure 5 shows a possible definition for such a module.

Notice that the specification of this module is similar to the system specification illustrated earlier. To create a hierarchical module, LSS statements are simply wrapped in a `module` block. The `MultiProcessor` module specifies its input/output behavior by connecting its external ports (declared on lines 7-8) to internal instances (lines 11-12).

In addition to the system features illustrated earlier, this example also demonstrates parameter declarations (lines 2, 3, and 24) and port declarations (lines 7-8). Notice the use of the parameter to control the number of processors instantiated (line 4). Within the module block, parameters behave like variables, except all assignments (such as that on line 3) give the parameters *default* values and are thus not unconditional assignments. Any value already specified during instantiation overrides these default values. To promote specification ease, assigning to a parameter multiple times is permitted (e.g. to allow conditional default values). However, once a parameter value is read, its value is fixed; any subsequent attempt to write to the parameter would produce an error. This behavior ensures that all users of a parameter see a consistent value. Variables are intended to behave as normal variables in an imperative programming language.

Take special note that the `topology` parameter is controlling the connectivity and thus the functionality of this module (lines 14 and 23). Enumerating all the connections in such a system would be extremely tedious and it would be impossible to have such a design have a parameterizable number of processing elements as this example does. While idiom based connectivity systems could potentially describe these two topologies even in the presence of a variable number of processors, it is unlikely that they would easily be able to handle the irregularity caused by the `memctrl` instance (the node marked "M" in the Figure 4). The algorithmic specification provides an elegant mechanism to specify the connectivity even in the presence of such an irregularity.

Finally, notice that the `topology` parameter is guarding the declaration of two additional parameters: `height` and `width` (line 24). These parameters only have meaning when using the grid topology and thus their declaration is appropriately predicated. In much the same way, it is also possible for a module declaration to guard the declaration of ports with control flow statements predicated on parameters. Since parameters and ports can conditionally exist based on the values of other parameters, it may seem as though it is impossible to evaluate the language. However, fairly simple execution semantics exist that allow evaluation, and such semantics prove quite natural for structural specification. The details of the execution semantics are discussed in Section 5.



(a) Ring topology (b) Grid topology

**Figure 4: Multiprocessor network.**

The `MultiProcessor` module defined above behaves just like any other module. This means that it is possible to define another hierarchical module that uses `MultiProcessor` as one of its sub-instances. This process can continue recursively as far as the user needs.

### *4.2.2 Leaf Module Declarations*

Just like hierarchical modules, leaf module declarations are specified within a `module` block. The LSS portion of a module specification for a basic storage element is shown in Figure 6. A module indicates that it is a leaf module by assigning a value to the `tar_file` parameter (line 2). This parameter identifies the behavioral code that will implement this module's functionality. In addition to the `tar_file` parameter, leaf modules and leaf modules' ports have several other parameters which declare scheduling properties of the module instances (lines 3-6 and 19-22).

This example also demonstrates the type polymorphism available on ports. The ports declared on lines 16 and 17 use a type variable,`'type`, instead of explicitly specifying types for the ports. This allows the ports to have any type, but the `src` and `dest` ports must have the same type, since both ports reference the same type variable. The specification of types through unconstrained type variables means that port types in LSE are polymorphic. In the example, the storage element is indifferent to the type of the value it stores; without polymorphic port types, a separate module would have to be created for each type. However, polymorphism allows a single, generic module to specify the behavior for *all types*.

In general, the symbol to the right of the colon in a port declaration specifies a constraint on the valid types for that port. Type variables have module statement scope and impose the constraint that ports with the same type variable have the same type. More complex constraints, such as set membership are also possible. Type inference is used to identify actual port types and is discussed in more detail in Section 5.2.

## 4.3 Parameters

While scanning through the code in Figures 3 and 5, one may have noticed that many of the parameter assignments (e.g. Figure 3 lines 9-11) look strange. These statements are assigning to code-

```
1  module MultiProcessor {
2    parameter topology : enum{ring, grid};
3    parameter numprocs=4 : int;
4    instance procs : Processor[numprocs];
5    instance memctrl : MemoryController;
6
7    inport responseIn : int32;
8    outport requestOut : SIM_addr_t;
9
10
11   memctrl.requestOut -> requestOut;
12   responseIn -> memctrl.responseIn;
13
14   if(topology == ring) {
15     var i : int;
16
17     for(i = 0; i < numprocs - 1; i++) {
18       procs[i].dataOut -> procs[i+1].dataIn;
19     }
20     memctrl.dataOut -> procs[0].dataIn;
21     procs[numProcs - 1].dataOut -> memctrl.dataIn;
22   }
23   else if(topology == grid) {
24     parameter height, width : int;
25
26     assert(height * width == (numprocs-1));
27
28     memctrl.dataOut -> procs[0].dataIn;
29     procs[0].dataOut -> memctrl.dataIn
30     memctrl.dataOut -> procs[x.width-1].dataIn;
31     procs[x.width-1].dataOut -> memctrl.dataIn;
32
33     var x,y : int;
34     for(x = 0; x < width; x++) {
35       for(y = 0; y < width; y++) {
36         if(x != 0 || y != 0) {
37           if(x - 1 > 0) {
38             procs[y*width+x-1].dataOut ->
39               procs[y*width+x-2].dataIn;
40             procs[y*width+x-2].dataOut ->
41               procs[y*width+x-1].dataIn;
42           }
43           /* Repeat the above code for x+1,
44              y-1, and y+1 */
45         }
46       }
47     }
48   }
49 };
```

**Figure 5: LSS description of a multiprocessor network.**

```
1  module flop {
2    tar_file="flop";
3    phase=FALSE;
4    phase_start=TRUE;
5    reactive=FALSE;
6    phase_end=TRUE;
7
8    parameter unblock_at_receive=TRUE:bool;
9    parameter initial_state : userpoint(
10       <<<int porti, SIM_dynid_t *init_id,
11          SIM_port_ctype(dest) *init_value>>> =>
12       <<<gboolean>>>);
13
14    initial_state = <<< return FALSE; >>>;
15
16    inport src:'type;
17    outport dest:'type;
18
19    src.handler=FALSE;
20    dest.handler=TRUE;
21    src.independent=TRUE;
22    dest.independent=FALSE;
23
24  };
```

**Figure 6: LSS specification for a basic storage element.**

typed parameters. The values, enclosed within $<<<$ and $>>>$, are pieces of algorithms, written in the BSL (stylized C), that are used by the code-generator to customize run-time instance behavior. Modules may declare these code-typed parameters, including the signature of the arguments and return value of the code, using the syntax shown on lines 9-12 of Figure 6. The code-value to the left of the `=>` operator indicates the list of arguments available to the code, and the value to the right of the operator indicates the code's return value. Since all the values enclosed within $<<<$ and $>>>$ are only used during code generation, the syntax and semantics of code values are determined by the BSL. During code generation, the code inside the code-typed parameters is type checked to verify that it is compatible with the signature. The code values are essentially treated like string constants by the LSS interpreter, however, they do have access to all the variables and parameters presently in-scope, including other code-typed parameters. These code-typed parameters increase the reusability of modules by providing a very flexible mechanism to modify the runtime behavior of a component.

In addition to code-typed parameters, LSS supports a strongly-typed and slightly augmented (e.g. support for specific width integers) version of the C type system (excluding pointers). As the example in Figure 5 showed, parameters with more conventional types like `int` and `enum` can be used to customize the structure of hierarchical modules. However, since the LSS type system closely parallels that of the BSL, these conventionally typed parameters may also be used by leaf modules to customize BSL specified behavior. For example, the `size` parameter of the leaf module `mqueue` (see Figure 3) controls the maximum number of elements that can be stored in the queue. Notice, however, that LSS is BSL independent provided that there is a map from LSS types to BSL types, a fairly loose restriction.

## 4.4 Ports and Connections

The example shown in Figures 4 and 5 illustrates another important feature of LSS, its management of multiple connections to a port. When using the grid topology, the `MultiProcessor` module makes multiple connections to each processor's dataOut and dataIn port, one connection for each neighboring processor. In most structural systems, multiple connections to a single port specifies fan out or fan in. However, in LSE all connections are point-to-point to allow for a uniform, customizable specification of control semantics [13].

Since all connections in LSE are point-to-point, multiple connections to a single port could be considered an error. However, LSS uses multiple connections to a port to provide an easy mechanism to scale the number of inputs or outputs a module has or provide a simple way to connect buses of wires.

Each port in LSE is actually a variable length array of ports, where the size of the array is specified by the *port-name*.`width` parameter on the the port. Each connection to a port is actually a connection to one of the ports in the array; each connection behaves as a separate communication channel. The user can specify to which index of a port a connection be made by indexing the port like an array. For example, `A.p[3] -> B.p[2]`. However, since connections to an instance's ports may be distributed throughout a system specification, it may be difficult to keep track of the first unused index on a port. Similarly, it may be burdensome to explicitly assign the `width` parameter for each port. To relieve this burden, LSS supports an alternate syntax for connectivity. As shown on lines 28-31 of Figure 5, the user can omit the port index when specifying the end point of a connection. LSS will automatically assign a unique port index to each such end point. Furthermore, the LSS interpreter will automatically set the `width` parameter for any port, for which it is not explicitly specified, to the number of connections made to the port. When the parameter *is* explicitly set, it is used as an assertion to verify that the number of actual connections matches the number specified in the parameter. Finally, to avoid confusion, the LSS interpreter forbids simultaneously using a port in connection statements with explicitly specified indices and connection statements with inferred indices.

Entire port arrays and individual ports within these arrays may be unconnected. LSS and BSL descriptions assign semantics to these unconnected ports to allow designs to be specified *iteratively* and to allow for simple use of generic modules. Accordingly, for each port the module has a parameter, *port-name*.`connected`, to indicate whether a port is connected. If not explicitly set, these parameters are automatically set to `TRUE` or `FALSE` if the port is connected or unconnected, respectively. If explicitly set, the parameter asserts that a port should be connected or unconnected and produces an error if the assertion is not true. These semantics give the module writer an easy way to restrict the usage of ports, without excessively verbose error checking code.

## 4.5 Instrumentation

In order for simulation to be useful, a user must be able to collect data from a simulation run. However, different users want different data from a system. Furthermore, different systems will have different characteristics worth examining. In traditional simulation systems, the simulator usually has hard-coded instrumentation. This does not allow for a modular simulator specification.

To separate behavior from data gathering and statistics computation, data collection in LSE occurs through events and data collectors. Each module instance has a set of events that the BSL behaviors can emit during simulation. Each event has a collection of data that is emitted along with the event. For example, on a processor branch predictor module there may be an event called `PREDICTION`. The prediction event would occur every time a prediction is made, and it would have the prediction and an instruction identifier attached as data. This data could be used with other events to compute the branch mispredict rate in a processor simulation. In LSS, events are declared within packages and modules within each package state which of these events they can emit.

Statistics computation and other uses of event data are cross-cutting concerns in a program. In order to debug a design or visualize the flow of data through a piece of hardware, events from throughout the system need to be caught and processed. For example, in order to visualize data flow, every event corresponding to a data transfer on the ports needs to be caught by the visualization tool and then used for display. Probing each event individually, of course, would prove to be very tedious since there is one of these events for each port on each module instance. Further, any changes to system structure would require the specification to be re-instrumented. On the other hand, some data collection requires catching very specific events. For example, to compute the branch misprediction rate in a microprocessor, the prediction event on the branch predictor module and the branch-completed event on the branch unit need to be caught.

Since data collection is a cross-cutting concern, LSS has an aspect-

```
1  package branch_predictor {
2    event PREDICTION:<<<struct { SIM_dynid_t id;
3                                 int predicted_dir; };>>>;
4    ...
5    module bpred {
6      inport predict:SIM_addr_t;
7      outport predicted_dir:int;
8
9      ...
10     emits PREDICTION;
11     ...
12   };
13 };
```

**Figure 7: Example event declaration and specification of emission.**

```
data_collectors {
  !*:processor1.!* = <<< /* Visualization code */ >>>;
  PREDICTION:processor2.bpredict =
                    <<< /* misprediction calculation code */ >>>;
};
```

**Figure 8: Example data collector specification.**

oriented [8] mechanism to specify code that will run when the corresponding event occurs.. Figure 7 shows the LSS code for a branch predictor package and parts of a branch predictor module definition relevant to defining the `PREDICTION` event. Lines 2-3 define the event and declare that it has two data items. The first, `id`, is a dynamic identifier which will identify for which branch instruction the prediction is made; the second, `predicted_dir` is the direction predicted for the branch, either taken or not. Line 10 states that module `bpred` emits the `PREDICTION` event. In addition to the user defined events, there is also a set of system wide events that are associated with data transfers. In particular, there is an event that is emitted any time data is sent along a port.

To specify what code should be associated with an *event instance* (the combination of the event name and the generating instance or instances), a user defines a data collector at the top level in a section named `data_collectors`. Each event instance behaves like a join point and the data collector like an aspect in an aspect-oriented programming language. The user identifies the event and module instances of interest and assigns a code-value to the event instance. Rather than requiring individual instances and events to be enumerated, the user may specify regular expressions (with `.` replaced by `!` as the wildcard character since `.` occurs frequently as a normal character delimiting hierarchy), and any matching event instance gets the code. Figure 8 shows several examples of the syntax for this specification. As the example shows, regular expression syntax satisfies both the data collection requirements outlined above. Thus, visualization and debugging tools can easily catch many events, while specific statistics computations can easily catch only specific events.

## 5. LSS EVALUATION SEMANTICS

This section describes the evaluation semantics of the LSS language. While the language shares many properties in common with other imperative object-oriented programming languages, an important distinction exists in the way in which the module abstractions are instantiated to create module instances. Typically, in an imperative object-oriented programming language, a program explicitly instantiates a class by using an operator of the language such as `new`. As arguments to the operator, one normally specifies all the parameters necessary for instantiating the abstraction.

These standard evaluation semantics preclude the use of several features that are desirable in a structural specification language. For example, standard evaluation semantics would prevent the constructor of a module from using the number of connections to its ports unless this connection count is explicitly passed as a parameter. However, as shown in the grid processor example in Figure 5, the user may not always care about or know the exact number of connections made to a port. It would be very inconvenient to force the user to change the connection count in the instantiation statement every time they added or removed a connection.

Standard instantiation schemes require all of a constructor's arguments to be specified before an abstraction can be instantiated. However, it is preferable to have these parameters disjointly specified throughout the code on a single level of the hierarchy. Furthermore, the existence or absence of certain parameters should be able to depend on the value that *will be* assigned to another parameter. Thus, the LSS evaluation semantics must allow for delayed abstraction instantiation. Since this allows instances to adapt themselves based on how they are used, we refer to this style of instantiation as *use-based specialization*.

Use-based specialization has the benefit that a component can implicitly inform the compiler of valid parameter combinations and connection patterns and have the compiler enforce the consistency. The module does this by predicating the declaration of parameters and ports on the values of other parameters. If the port usage and parameter settings are incongruent, the compiler will flag an error that an undeclared port or parameter was used.

## 5.1 Instantiation

To implement use-based specialization, the system must first formally define the evaluation semantics. For this discussion, we will identify all code with an instance in the hierarchy. Code on the top-level (not explicitly in a module instance) will be considered part of the `top-level` instance. Evaluation proceeds as a pre-order depth-first traverse of the instance hierarchy tree. Obviously, this tree is not fixed at the beginning of evaluation. Instead, each node in the tree, when evaluated, adds the children it defines to the instance tree. Once the evaluation of a particular node completes, the node's children are instantiated and evaluated. Since instantiation is delayed until after the current node evaluates, *all* the parameters needed by the child, including its connectivity, are available.

During evaluation of a particular instance, all statements which set sub-instance parameters or connect to sub-instance ports are logged. These logged values and connections are used when evaluating the sub-instances to obtain parameter values or determine port connectivity. Parameter types are checked versus assigned types to ensure that they are consistent. Just before evaluation of a sub-instance completes, the log is checked for assignments to undeclared parameters or connections to undeclared ports. As an additional precaution, during instance evaluation all sub-instance parameter assignments are checked for consistency. It is illegal, for example, to assign to a parameter $p$ a value of type $\tau_1$ and subsequently assign to it a value of type $\tau_2$ even if the parameter will be declared with type $\tau_2$. This ensures that all executed code has a *consistent* view of what the type of a parameter will be. This view is then verified when the sub-instance is evaluated.

## 5.2 Type Inference

```
module A {
  module B {
    parameter p:bool;

    module C1 {
      inport src:int;
      outport dest:int;
      ...
    };
    module C2 {
      inport src:int;
      outport dest:int;
      ...
   };
    ...
    inport src:*;
    outport dest:*;

    if(p) {
      instance c:C1;
    } else {
      instance c:C2;
    }
    src -> c.src;
    c.dest -> dest;
  };
  ...
};
```

**Figure 9: 3 levels of hierarchy.**

While parameter types are checked for consistency using the technique described in section 5.1, port types are handled differently. If ports were handled like parameters, all types in the system would be resolved from the outermost to the innermost levels of the instance hierarchy. While this may not seem restrictive, the following example will demonstrate why more flexibility is desirable.

Consider the LSS in Figure 9. Here we see that if port types were resolved like parameter types, it would be necessary for module `A` to specify the types of module `B`'s ports, even though setting the parameter `p` would be sufficient to determine the ports' types. To avoid this pitfall, LSS supports type inference on ports which uses information from connections to sub-instances as well as external connections to the module itself to infer a port's type. The algorithm to allow this is described below.

As was mentioned earlier, types given in port declarations are actually constraints on the set of types possible on that port. These constraints take two possible forms: equality constraints and set membership constraints. Port declarations like the one on line 7 of Figure 5 or line 16 of Figure 6 are equality constraints where the former asserts equality to the `int_32` type while the latter asserts equality to the type variable `'type`. Set membership constraints can be specified during port declaration by replacing a type name or type variable name with a * character or a comma separated list of types enclosed in brackets. The * character represents the universal set of types, while the comma separated list is the set of all types in the list. Connection statements also add constraints to the system. An equality constraint between the two connected ports is added since connected ports must share the same type. The connection itself may specify additional constraints using the `->:`*`constraint`* operator instead of the conventional `->` operator. The `->` is actually shorthand for `->:*`. Constraints are added asserting that both ports in a connection are equal to the type or members of the set specified with the connection.

After all evaluation completes, the LSS interpreter uses these constraints to infer the type of each port. The types are determined by solving the system of constraints described by the system where, in addition to the explicit type variables, one type variable per port is introduced. Obviously contradicting constraints are problematic, but under constrained systems are also illegal. This is because the underlying BSL behaviors are type abstractions which take, as arguments during instantiation, the type of *all* ports on a given instance. The one exception concerns unconnected ports. Since the data on the port is not used anywhere in the system, systems in which unconnected ports have insufficient constraints to infer its type are permitted. The type inference engine will introduce arbitrary artificial constraints on these ports to force them to a single type.

## 6. EXPERIENCE WITH LSS

It is truly difficult to evaluate the overall quality of a programming language since much of its value relates to subjective metrics like clarity or ease of specification. However, extensive use of a programming language allows one to gain significant insight into the value of various language features. In this section, we will present some anecdotal evidence that shows that the LSS language is an effective language for structural hardware system specification. We will present our experience with various systems that we have modeled in LSE using the Liberty Structural Specification language and contrast this with similar experiences describing systems in a predecessor description language. We will also discuss the design of components we have built and describe the reusability that LSS facilitated.

As a computer architecture research group, we have modeled several microprocessor cores. As a proof-of-concept for LSE, we modeled a hardware system behaviorally identical to the one modeled by SimpleScalar [1], a popular architecture simulator hand-written in C [13]. Our system model was initially specified in an XML syntax that predated LSS. Specifications done in this XML format were static enumerations of all the instances and connections in the system. Additionally, this earlier system lacked the ability to hierarchically compose modules (all modules were leaf modules), and the system lacked type inference.

Despite these specification language shortcomings, LSE still provided large gains in specification ease compared to existing simulation systems. However, the weaknesses of specifying a system in XML were immediately apparent. Enumerating the system's connectivity required cut-and-paste of large sections of XML and minor conceptual changes required significant changes to the description. The overall system structure was lost in the tangle of connections. To ease some of the tedium, we introduced connection idioms into the XML syntax for bus connectivity. However, due to minor irregularities, these idioms often could not be used, and the specification complexity remained fairly high.

This experience provided the impetus for the development of the Liberty Structural Specification language. As a case study we converted the SimpleScalar machine configuration from the XML syntax to LSS. The LSS description proved to have a 35% reduction in the number of lines of code, but more importantly, it was considerably more clear. The algorithmic specification of structure allowed easy specification of regular connectivity even in the presence of slight irregularity. Type inference also made the specification clearer. Since the type of connections is often understood (only one type makes sense), avoiding specification of inferable types allowed the machine's actual structure to come through.

In addition to the SimpleScalar machine model, our group modeled an in-order superscalar processor. The experiences obtained while building this model were similar to those obtained above. Outside of our research group, the modeling language has been used to model and study power consumption in interconnection networks [14]. In fact these network simulators were constructed from many of the same modules used for the processor architecture models. Only modules related to domain specific system control needed to be added to the library. The modules' use in these two disparate modeling domains is a testament to LSS's ability to specify flexible modules.

For component specification, the LSS language proved to be incredibly useful. For example, specifying a generic memory cache module required the creation of a few leaf modules to independently describe the cache array, the tag comparator, and the cache controller. Once defined, these components could then be connected in a variety of ways to describe different cache styles. Without the ability to describe a new module as the composition of existing modules, changing the cache configuration for varying contexts would prove exceedingly difficult. However, in the LSS language, typical memory organizations could be wrapped into a single cache module which could be parametrically controlled. In one such instance, the cache has a parameter which toggles between simulation of a perfect cache, which only requires a single sub-instance, and a traditional cache, which required the composition of the three principle modules with several auxiliary ones to appropriately route data. Additionally, when the non-perfect cache is selected, additional parameters controlling its size, replacement policy, and other behaviors were exposed. Thus the use-based specialization proved particularly powerful for this component's design.

## 7. RELATED WORK

In the introduction, we compared LSE to existing simulation approaches and briefly argued why LSE is needed. In this section, we examine how LSS compares to existing structural specification systems, concentrating on structural composition and component construction. We also compare the evaluation semantics used by LSS with existing schemes such as lazy evaluation.

As discussed earlier, other concurrent-structural specification systems have been proposed for hardware modeling and programming in general [2, 4]. In addition to the shortcomings of these systems for high-level hardware modeling described in the introduction, these systems lack flexibility in the way in which users can customize and construct new components from existing components. Furthermore, many of them do not allow parameters to control how components are instantiated and connected.

In SystemC [9] it is possible to build new components from existing ones using features of the C++ language, but this process can be extremely cumbersome. The language mixes structure and behavior specification and consequently the structural specification is encumbered with the execution semantics of C++ which proves inappropriate. Furthermore, the system does not explicitly support polymorphism, but instead relies on C++'s heavyweight template system. C++ templates add considerable syntactic overhead for specification and use of polymorphic components, however they offer only limited polymorphism which must be resolved without the benefit of type inference.

Other systems, such as hardware description languages [6, 12], do not allow any component customization and only limited forms of parameter based instantiation. Components are written for a specific purpose and used only for that purpose. This is acceptable for hardware synthesis, since the designer is likely to want detailed control over exactly how a component is specified (and thus synthesized), but it is not acceptable for high-level modeling where reusable components are highly desirable.

Finally, systems such as Ptolemy [7], allow for some flexibility and reusability through component parameters and a polymorphic type system for the data passed between component ports. However the system structure is specified with a graphical user interface or through MoML (an XML schema), thus precluding algorithmic specification of structure. Instead the system is restricted to explicit enumeration of connections and instances. Thus, although parameters may affect leaf component behavior, system structure is independent of parameterization.

The evaluation semantics of LSS resemble lazy type evaluation in an eager programming language [5]. With lazy evaluation, evaluation of expressions resulting in lazy types is deferred until that evaluation is necessary. The LSS evaluation semantics similarly defer module instantiation, however, this instantiation is deferred until all uses of the instance have been specified. Knowledge of the uses guides the module evaluation during instantiation allowing use-based specialization.

Since LSS has many of the execution semantics of a programming language, it is possible to create new flexible and reusable components from existing components as well as concisely specify complex but regular connection patterns. LSS allows the number of connections to a port as well as the module's parameters to control what modules a hierarchical module instantiates and how it interconnects them.

## 8. CONCLUSION

The Liberty Simulation Environment is the first simulation system to separate the description of structure and the specification of runtime behavior into two distinct descriptions using a specialized programming language for each. LSE employs the Liberty Structural Specification which allows for the algorithmic specification of the structure of a system. Through the use of simple control flow statements common to most imperative programming languages, many regular patterns present in the hardware can be clearly and concisely specified even in the presence of slight irregularities. The separation of structural specification and behavioral specification allows the syntax and semantics of the LSS language to be tailored to ideally match the goals of structural specification.

The LSS language uses novel evaluation semantics to allow for use-based specialization. The evaluation semantics allow component customization parameters to be specified in a distributed fashion, a natural paradigm for hardware specification. Additionally port and parameter existence can be predicated on other parameter values providing powerful compiler support for assuring appropriate parameterization of highly flexible reusable components.

LSS also leverages powerful programming language techniques such as type inference and aspect-oriented programming to further facilitate component reusability and modular system specification. Using polymorphic port types and type inference allows the specification of type-neutral reusable modules with type resolution occurring automatically without user intervention. Aspect-oriented data collection, separates simulator instrumentation from both behavior and structure specification. Additionally, data collection can easily be specified for large subsets or specific pieces of the machine using pattern matching to facilitate overall system visualization or individual component performance analysis.

Overall, LSS provides designers with a powerful tool for efficiently and accurately describing hardware in an elegant manner. An LSS description closely resembles the hardware that is being modeled and therefore the description is intuitive and easily modified. Further, the nature of LSS language was designed specifically with code reuse and design efficiency as a goal and this allows for the specification of component libraries that maybe used to facilitate description.

## 9. REFERENCES

[1] BURGER, D., AND AUSTIN, T. M. The SimpleScalar tool set version 2.0. Tech. Rep. 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[2] EMER, J., AHUJA, P., BORCH, E., KLAUSER, A., LUK, C.-K., MANNE, S., MUKHERJEE, S. S., PATIL, H., WALLACE, S., BINKERT, N., ESPASA, R., AND JUAN, T. Asim: A performance model framework. *IEEE Computer 0018-9162* (February 2002), 68–76.

[3] GRUN, P., HALAMBI, A., KHARE, A., GANESH, V., DUTT, N., AND NICOLAU, A. EXPRESSION: An ADL for system level design exploration. Tech. Rep. TR 98-29, University Of California, Irvine, 1998.

[4] HALAMBI, A., GRUN, P., GANESH, V., KHARE, A., DUTT, N., AND NICOLAU, A. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test (DATE)* (March 1999).

[5] HARPER, R. *Programming Languages: Theory and Practice*. Draft, 2002.

[6] IEEE. VHDL: IEEE Standard 1076. http://www.ieee.org.

[7] JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. Discplining heterogeneity – the Ptolemy approach. In *ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001)* (June 2001).

[8] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *Proceedings of the 11th European Conference for Object-Oriented Programming* (1997), pp. 220–242.

[9] OPEN SYSTEMC INITIATIVE (OSCI). *Functional Specification for SystemC 2.0*, 2001. http://www.systemc.org.

[10] PENRY, D. A., AND AUGUST, D. I. Optimizations for a simulator construction system supporting reusable components. Tech. Rep. Liberty-02-03, Liberty Research Group, Princeton University, September 2002.

[11] SISKA, C. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. of the 11th International Symposium on System Synthesis (ISSS)* (Dec. 1998).

[12] THOMAS, D. E., AND MOORBY, P. R. *The Verilog hardware description language*. Kluwer Academic Publishers, Norwell, MA, 1998.

[13] VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symp. on Microarchitecture* (November 2002).

[14] WANG, H.-S., ZHU, X.-P., PEH, L.-S., AND MALIK, S. Orion: A power-performance simulator for interconnection networks. In *Proceedings of 35th Annual International Symposium on Microarchitecture* (November 2002).

[15] Živojnović, V., Pees, S., and Meyr, H. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing* (San Francisco, CA, October 1996).