The Liberty Structural Specification Language: A High-Level Modeling Language for Component Reuse

Manish Vachharajani Neil Vachharajani David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
Princeton, NJ 08544
{manishv, nvachhar, august}@princeton.edu

ABSTRACT

Rapid exploration of the design space with simulation models is essential for quality hardware systems research and development. Despite striking commonalities across hardware systems, designers routinely fail to achieve high levels of reuse across models constructed in existing general-purpose and domain-specific languages. This lack of reuse adversely impacts hardware system design by slowing the rate at which ideas are evaluated. This paper presents an examination of existing languages to reveal their fundamental limitations regarding reuse in hardware modeling. With this understanding, a solution is described in the context of the design and implementation of the Liberty Structural Specification Language (LSS), the input language for a publicly available high-level digital-hardware modeling tool called the Liberty Simulation Environment. LSS is the first language to enable low-overhead reuse by simultaneously supporting static inference based on hardware structure *and* flexibility via parameterizable structure. Through LSS, this paper also introduces a new type inference algorithm and a new programming language technique, called *use-based specialization*, which, in a manner analogous to type inference, customizes reusable components by statically inferring structural properties that otherwise would have had to have been specified manually.

Categories and Subject Descriptors

I.6.2 [**Simulation and Modeling**]: Simulation Languages;
I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms

Design, Languages

Keywords

Liberty Simulation Environment (LSE), Liberty Structural Specification (LSS), component reuse, simulator construction, structural modeling, type inference, use-based specialization

PLDI'04, June 9–11, 2004, Washington, DC, USA.

1. INTRODUCTION AND MOTIVATION

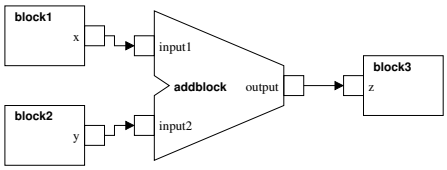
In digital hardware design, early design decisions significantly impact the overall quality of the system produced. Current analytical models fail to provide guidance due to the complexity of these systems. Producing and measuring candidate hardware prototypes to test these early design decisions is prohibitively expensive. As a result, designers turn to high-level (e.g. microarchitectural level) software simulation models for feedback in iteratively refining these critical early design decisions.

The quality of the resulting high-level design is directly related to the rate at which high-level design candidates can be explored. Just as reuse in software development significantly improves programmer efficiency, reuse in high-level design modeling reduces model specification time, dramatically increasing the exploration rate of design candidates. Reuse here is particularly attractive because high-level hardware design exploration is replete with opportunities to employ it; many behaviors such as arbitration and queuing, are extremely common in a wide range of hardware systems, and other common structures exist among designs within an exploration. Unfortunately, current modeling systems either do not support reuse or require significant effort to achieve reuse, negating any potential benefits. Consequently, model construction and modification times when using these systems is on the order of months to years, severely limiting the range of alternatives explored in the early design phases and negatively impacting final design quality [4, 15].

In the general purpose microprocessor research and design community, manually coding a simulator in a sequential programming language such as C or C++ is the most common method of high-level modeling[1]. This method does not allow component-based reuse because the notion of a hardware component does not map well to any type of modular block in C or C++ [17].

Concurrent-structural modeling tools present an alternative by modeling hardware components with concurrently executing software components that communicate through statically connected communication channels. Since this modeling paradigm matches the structural composition and concurrent processing of hardware components, component-based reuse becomes a possibility [17]. These systems fall into two categories, static concurrent-structural systems and structural object oriented programming (OOP) systems. Unfortunately, as will be described in this paper, the specification style in each of these systems forces a trade-off between ease of building reusable components and ease of using reusable components. In both cases, the difficulties in building and using reusable components discourage reuse in practice.

[1] In the 30th International Symposium on Computer Architecture at FCRC, at least 23 of 37 papers used this simulation technique.



(a) Structural Addition

```
...
/* Instantiate adder */
instance addblock:adder;

/* Connect ports */
block1.x -> addblock.input1;
block2.y -> addblock.input2;
addblock.output -> block3.z;
```

(b) Structural Code

```
component adder {
  ...
  void compute() {
    int x, y, z;
    x = receive(input1);
    y = receive(input2);
    z = add(x,y);
    send(output, z);
  }
}
```

(c) Behavioral Code

Figure 1: Structural specifications.

In this paper, we present a hybrid model specification style that allows easy construction *and* easy use of flexible, reusable components. This paper also presents the design and implementation of the Liberty Structural Specification Language (LSS), a high-level hardware modeling language that uses this hybrid specification style. This style opens the door to techniques that were possible, but of limited utility, in static structural systems and useful, but unimplementable, in structural-OOP systems. This paper presents the implementation of two such techniques used in LSS: structural type inference in the presence of both component overloading and parametric polymorphism and *use-based specialization*, a new technique to further reduce the overhead in using flexible components. These techniques required the development of a novel algorithm for type inference and unique evaluation semantics for use-based specialization.

The remainder of the paper is organized as follows. Section 2 provides an overview of high-level concurrent structural modeling. Section 3 identifies key attributes for fully supporting component-based reuse and surveys existing modeling systems to identify which attributes are missing. Sections 4, 5, and 6 describe the design of the LSS language, including implementation challenges and solutions to these challenges. Section 7 summarizes our experience with the LSS language, emphasizing the applicability of techniques described in this paper. Section 8 discusses related work, and Section 9 concludes the paper.

2. CONCURRENT-STRUCTURAL MODELS

Concurrent-structural systems, in general, are those systems in which computation is encapsulated in concurrently executing system components that communicate values to one another by sending and receiving data on a predefined communication network. Typically, each component defines a set of input and output ports and continuously computes its outputs given the current values of its inputs and internal state. Components whose input ports are connected to a particular output port receive the value sent on it and accordingly update their internal state and outputs. Synchronous digital hardware, where state update is synchronized to a global clock, is an example of a concurrent-structural system. Concurrent-structural modeling systems are the most natural way to model such hardware and are often used for this purpose. Figure 1a shows a structural model that adds two numbers together. Figure 1b shows a possible textual description of the structural model.

In *high-level* concurrent-structural models, the focus of this work, the primitive components' port I/O relation (e.g. the function that the `adder` component will compute) is specified using conventional function-invocation based code. Despite the use of function-invocation code *within* a particular component, communication *between* components occurs structurally. This distinction clearly separates functionality and communication, a hallmark of concurrent-structural systems. Continuing the earlier example, Figure 1c shows a possible description of the `adder` component's behavior.

Capability	Static Structural Theory	Static Structural Practice	Object Oriented Theory	Object Oriented Practice
Parameters	yes	yes	yes	yes
–Structural			yes	yes
–Algorithmic	yes	yes	yes	yes
Polymorphism	yes	yes	yes	yes
–Parametric	yes		yes	yes
–Overloading	yes	yes		
Static Analysis	yes	yes		
Instrumentation	yes		yes	

Table 1: Capabilities of existing methods and systems.

3. REUSE IN EXISTING METHODS

High-level concurrent-structural modeling languages should have certain capabilities to fully support component-based reuse. These languages should support:

Parameterizable Components - the ability to customize component properties with parameters. Example: a cache component whose replacement policy can be selected from an enumerated set of predefined policies.

- **Structural Customization -** the ability to customize hierarchical structure with parameters. This allows existing components to be reused hierarchically to create a *flexible* component. Example: customizing the mix of functional units and bypass connection specification in a structurally specified reusable CPU core.
- **Algorithmic Customization -** the ability to inherit and augment the behavior of an existing component with an algorithm. Example: customizing arbitration logic inside a bus arbiter component.

Polymorphism - the ability to support reuse across types.

- **Parametric Polymorphism -** the ability to create and use component models in a datatype independent fashion. Examples: queues, memories, and crossbar switches that support all types.
- **Component Overloading -** the ability for a component's implementation to be automatically selected from a family of implementations that support different datatypes. Note that function overloading, in which *argument* types select a function's implementation, differs from component overloading, where *port and connection* types select a component's implementation. Example: An ALU with an implementation family that operates on integer and floating point numbers.

Static Model Analysis - the ability to analyze the model for optimization and user convenience. Example: type inference to resolve polymorphic port types.

Instrumentation - the ability to insert probes into a model without

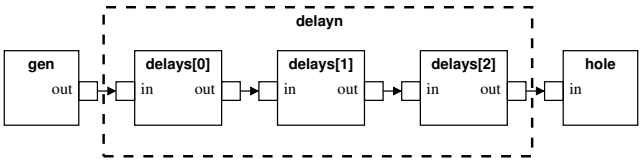


Figure 2: Block diagram of a 3-stage delay chain specification.

modifying the internals of any component. This allows models to be reused to satisfy different data collection needs. Examples: instrumentation for performance measurement, debugging, or visualization.

The following two subsections relate the above abilities to two existing modeling methodologies: static structural modeling and structural modeling in OOP. This analysis will identify desirable features and highlight potential pitfalls present in these methods. The insight gained will guide the design of the LSS language presented in the remainder of this paper. Table 1 can be used as a reference during the discussion.

3.1 Static Structural Modeling

Static structural modeling systems are concurrent-structural modeling systems which statically describe a model's overall structure. Models in these systems are often netlists of interconnected components augmented with component parameterizations, and typically these tools have drag-and-drop graphical user interfaces to construct models. An example of such a tool is Ptolemy II with the Vergil interface [7].

These systems support many of the features described above. Components typically export parameters so that they can be customized. Depending on the underlying language used to implement the components, inheritance may allow algorithmic customization. Some systems support polymorphism [7] and type inference to resolve the polymorphism [20]. Models could even be instrumented using aspect-oriented programming (AOP) [8] to weave instrumentation code into the structure of the described model.

Unfortunately, the fact that these specifications are static implies a fundamental limitation of static structural modeling systems. Consider the structure shown in Figure 2. In static structural systems, one would explicitly instantiate the three blocks within the dotted box in the figure. However, this chain of blocks could not be wrapped into a flexible hierarchical component where the length of the chain is a parameter since static structural systems provide no mechanism to iteratively connect the output of one block to the input of the next a parametric number of times. As a result, to permit flexibility, this simple hierarchical design would have to be discarded in favor of a more complex implementation of a primitive component. Implementing the primitive component for this simple example may not be difficult, but in more complex examples, such as controlling the mix of functional units in a processor model, implementation of a monolithic primitive component would be overwhelming. Note that some static structural modeling systems may provide idioms for common patterns, such as chained connections. However, the fundamental lack of general mechanisms to parametrically and programmatically control model structure still remains.

3.2 Structural Modeling in OOP

A promising concurrent-structural modeling approach, such as the one taken by SystemC [11], which allows flexible primitive *and* hierarchical components is to augment an existing OOP language with concurrency and a class library to support structural entities such as ports and connections. Objects take the place of components, and simulator structure is created at run-time by code that instantiates and connects these objects.

```
1  class delayn {
2    public InPort in;
3    public OutPort out;
4
5    Delay[] delays;
6    delayn(int n) {
7      int i;
8
9      in=new InPort();
10     out=new OutPort();
11
12     delays=new Delay[]();
13     in.connect(delays[0].in);
14     for(i=0;i<n-1;i++) {
15       delays[i].out.connect(delays[i+1].in);
16     }
17     delays[n-1].out.connect(out);
18   }
19 };
```

Figure 3: Structural OOP pseudo-code for an n-stage delay chain.

The basic features of object-oriented languages provide many of the capabilities described at the beginning of Section 3. Object behavior can be customized via instantiation parameters passed to class constructors. Algorithmic customization is supported via class inheritance. If the OOP language and the added structural entities support parametric polymorphism then type-neutral components can be modeled as well.

Since component instantiation and connection occur at run time, the OOP language's basic control flow primitives (i.e. loops, if statements, etc.) can be used to *algorithmically* build the structure of the system. This code can be encapsulated into an object and the internal structure can easily be customized by structural parameters thus producing *flexible* hierarchical components. For example, the n-cycle delay component (Figure 2) seen in the last section could be built by composing n single-cycle delay components as shown in the pseudo-code in Figure 3.

Unfortunately, run-time composition of structure provides component flexibility by precluding static analysis of model structure. This makes using these flexible components cumbersome. For example, any parametric polymorphism must be resolved via explicit type instantiation by the user since the constraints used in type inference are obtained from the model's structure which is unavailable at compile time. Ideally, connecting the output of a floating point register file to an overloaded ALU would automatically select the ALU implementation which handles floating point data. However, this component overloading is not possible since the user must codify the particular ALU implementation in the instantiation statement rather than the compiler automatically determining this based on connectivity. Additionally, parameters controlling the extent of all port arrays must be specified explicitly by the user rather than the compiler automatically inferring these extents from their connectivity. Finally, implementing instrumentation that is orthogonal to machine specification is at best cumbersome. Powerful techniques, such as aspect-oriented programming cannot be used since the desired join points (locations where instrumentation code should be inserted) are often parts of the model structure which is not known until run time.

4. THE LSS LANGUAGE

The modeling methodologies discussed in Section 3.1 and Section 3.2 possessed many of the capabilities necessary for reuse in

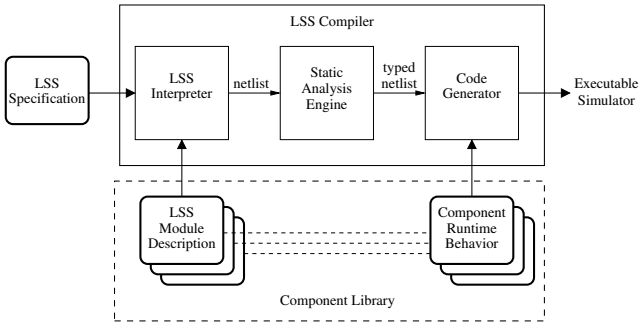


Figure 4: Overview of LSS based compilation.

a concurrent-structural modeling tool. Unfortunately, those systems forced a trade-off between the ability to algorithmically control structure and the ability to statically analyze it. The Liberty Structural Specification Language (LSS) is a language designed to specify structure of concurrent-structural systems that obviates this trade-off by combining the desirable attributes of both existing methodologies.

To allow algorithmic specification of structure, LSS possesses common imperative control flow constructs in addition to its structural modeling constructs. Using these constructs, a model's structure can be built using code similar to that in Figure 3. However, unlike modeling in OOP, the LSS code can be executed at *compile time* since the LSS code only describes the model's structure and *not* its run-time behavior. Since the LSS code is executed at compile time, the model's structure is still known statically and therefore can be used for analysis.

Figure 4 shows this LSS compilation process. In the first phase of compilation, the LSS specification undergoes interpreted execution to build a netlist that represents the static structure of the model. Once the netlist is generated, various static analyses are performed on the netlist. In the current implementation of the LSS compiler, these analyses include structure-based type inference, described in Section 5, and static concurrency scheduling [12]. Finally, after static analysis, the code generator combines the netlist with the leaf (i.e. not hierarchical) component behavior specifications, applies non-structural customizations to these behaviors, and emits an executable simulator binary. We refer to the language in which the leaf module behaviors are specified as the behavior specification language (BSL)[2].

This section describes the LSS language and its features that support all the desired capabilities listed in Section 3. Additionally, this section will describe enough about the language so that later sections can discuss the LSS type inference problem and the implementation of a novel technique, called use-based specialization, for parameter value inference. Note, since LSS was designed for algorithmic specification of structure, the design decisions for some features differ from those made in traditional programming languages. These design decisions and their implications will be discussed throughout this section.

4.1 LSS Modules

In LSS, components are created from component templates, analogous to classes in structural OOP modeling, called *modules*. The body of an LSS module specifies the component's parameterization interface, communication interface, and constructor. There are two types of modules in LSS. The first, *leaf modules*, are simple modules defined without composing behavior from other modules. The second, *hierarchical modules*, are more complex modules obtaining their behavior through the composition and customization of existing components. The next few sections will describe leaf and hierarchical modules and their parameterization.

[2]LSS is independent of the BSL, but a component written in LSS for a particular BSL is not necessarily compatible with another component written in LSS for another BSL. As a consequence of this independence, discussion of BSL details is beyond the scope of this paper.

```
1  module delay {
2    parameter initial_state = 0:int;
3    inport in:int;
4    outport out:int;
5
6    tar_file="corelib/delay.tar";
7
8    // BSL specific parameters here
9  };
```

Figure 5: An LSS module declaration for a leaf delay element.

```
1  instance d1:delay;
2  instance d2:delay;
3  ...
4  d1.initial_state = 1;
5  d1.out -> d2.in;
6  ...
```

Figure 6: A sample use of the `delay` module.

4.1.1 Leaf Modules

Figure 5 shows the declaration of a leaf module named `delay`. Line 2 in the figure declares a parameter named `initial_state` with type `int` and assigns the parameter a default value of `0`. Module parameters can be set by the user when instantiating a module to customize its behavior. These parameter values are forwarded to the BSL code so they can be used to customize module behavior.

Lines 3 and 4 define the communication interface of the module. These two lines define an input port named `in` and an output port named `out`. The computation which links the data received on port `in` to the data produced on port `out` is specified externally in leaf modules. The value in the internal parameter `tar_file` (shown on Line 6 of the figure) tells the code generator where to find the run-time behavior for instances of this module. Note that internal parameters, unlike module parameters, cannot be overridden by a user of the module.

Figure 6 shows an example of instantiating and parameterizing the `delay` module. Lines 1 and 2 each instantiate the `delay` module to create module instances named `d1` and `d2` respectively. Line 4 gives the `initial_state` parameter on instance `d1` the value `1`. Line 5 of Figure 6 connects the output of `d1` to the input of `d2`. Notice that the `initial_state` parameter on instance `d2` is not set. When such assignments are omitted, the parameter takes on its default value as defined in the module body (Line 2 of Figure 5).

Notice from the figures that parameters in LSS are referenced nominally and can be specified *after* the instantiation statement (e.g. `initial_state` is referenced on Line 4 of the figure) rather than in a positional argument list as part of the instantiation statement. These choices were made because flexible modules typically have many parameters. Nominal parameter references clarify models since parameter names describe the parameter's purpose better than position in an argument list. Similarly, flexible placement of parameter assignment allows groups of related parameter assignments

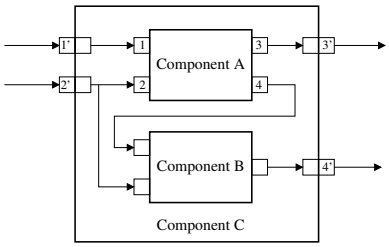


Figure 7: Hierarchical component composition.

```
1  module delayn {
2    parameter n:int;
3
4    inport in:'a;
5    outport out:'a;
6
7    var delays:instance ref[];
8    delays=new instance[n](delay,"delays");
9
10   var i:int;
11
12   in -> delays[0].in;
13   for(i=1;i<n;i++) {
14     delays[i-1].out -> delays[i].in;
15   }
16   delays[n-1].out -> out;
17 };
```

Figure 8: A LSS module declaration for an n-stage delay chain.

for different module instances to be co-located rather than scattered based on module instantiation location. Both features make using flexible components (i.e. those with many parameters) easier, encouraging their construction and use.

4.1.2 Hierarchical Modules

In addition to leaf modules, LSS supports the creation of complex modules by composing the behavior of existing modules into new *hierarchical* modules. Hierarchical modules, just like leaf modules, define a parameterization and communication interface by defining ports and parameters. However, unlike leaf modules, the behavior of the module is specified by instantiating modules and connecting these instances to the module's input and output ports (see Figure 7). Recall that allowing parameters to control the structure of these modules was a requirement outlined in Section 3. To allow this, LSS uses imperative control flow constructs to guide the sub-component instantiation, parameterization, and connection.

To understand how these features work together, consider the LSS code shown in Figure 8. (Note that this is the LSS version of the pseudo-code shown in Figure 3.) This code defines a module that models an arbitrary depth delay pipeline built using single-cycle delay modules. The module `delayn` declares a single parameter `n` (Line 2) that controls the number of stages in the pipeline. The parameter `n` is a structural parameter; anywhere after this declaration, the body of the module can read this parameter to guide how subinstances will be created, connected, or parameterized.

Line 7 and 8 create an array of instances of the `delay` module that will be named `delays` in the BSL. Notice that the length of the array (the value enclosed in brackets on Line 8 of the figure) is controlled by the parameter `n`.

Lines 12 through 16 connect the `delay` instances in a chain as shown for `n`=3 in Figure 2. Notice how the general purpose C-like for-loop causes the length of the connection chain to vary with the parameter `n`. Also, notice that while the C-like for-loop was able to

```
1  instance gen:source;
2  instance hole:sink;
3  instance delay3:delayn;
4
5  delay3.n=3;
6
7  gen.out -> delay3.in;
8  delay3.out -> hole.in;
```

Figure 9: LSS specification of a 3-stage delay pipeline.

capture a parameterized idiomatic connection pattern in this example, LSS supports constructs (including loops, conditionals, etc.) that permit general algorithmic specification of structure. This allows any non-idiomatic connection pattern to be created and parameterized.

Figure 9 shows how the `delayn` module can be used to create a 3-stage delay pipeline. The module is instantiated on Line 3, its `n` parameter is set on Line 5, and finally the instance is connected on Lines 7 and 8. A block diagram of this system is shown in Figure 2. The `gen` and `hole` instances just provide data to and consume data from the pipeline respectively.

4.2 Flexible Communication Interfaces

Module instances communicate via input and output ports defined by the modules from which they were instantiated. To facilitate scalable interfaces such as a register file with a customizable number of read ports, each port in LSS is actually a variable length array of *port instances*. Rather than connecting two ports together to have two instances communicate, one connects two port instances together. If no port instance number is specified, one is inferred by the interpreter. For each port in a module, the port's width (the number of connections made to the port) is available in the module body as a parameter. This parameter is automatically set by counting the number of connections actually made to the port. Modules can use the width parameter, like any other parameter, to customize the components created from it. This automatic customization is an instance of *use-based specialization* and will be discussed in more detail in Section 6.

LSS *also* supports leaving ports unconnected (ports with zero width). Using the width parameter, a module can detect whether or not a port is connected and customize its behavior accordingly. These *unconnected port semantics* allow modules to have rich communication interfaces without burdening a user with the responsibility of connecting all the ports. Without such a feature, it may be tempting to replicate simple functionality rather than reuse a complex module with many ports that are unnecessary for a given situation. These unconnected port semantics are especially useful when refining a model to a more precise model since the initial and refined model can reuse the same components; the initial model relies on unconnected port semantics, while the refined model connects the ports to achieve a specific desired behavior.

4.3 Customizing Component Computation

LSS provides two mechanisms to customize the computation of existing modules (algorithmic customization), increasing their reusability. The first mechanism is simply wrapping an existing module within another module and adding modules along computation paths to customize behavior. Figure 7 demonstrates how this can be done. Those computation paths which should be identical to the base component are connected directly to the wrapping component's input and output ports. In the figure, component C inherits the behavior of ports 1, 2, and 3 from component A. Component C overrides the behavior of component A's output port 4 since com-

ponent B has been placed between component A's output and component C's corresponding output. In this way, component C has extended component A.

The second way to customize computation is via *userpoint* parameters. These algorithmic parameters accept string values whose content is BSL code. This BSL code forms the body of a function on a module instance where the function signature is defined by the userpoint declaration in the declaring module's body. The declaration identifies a set of arguments whose values may be used in the userpoint BSL code and a return type for a return value that must be produced by the BSL code. The code in the userpoint is invoked by the module's behavioral specification to accomplish some computation or state-update task. Just like other parameters, userpoint parameters can have default values, thus allowing the module to define default behavior which can be overridden by the user.

A single userpoint parameter assignment on a module instance is the OOP equivalent of inheriting a class, overriding a virtual member function, and then instantiating the inherited class. This allows userpoints to dramatically reduce the overhead of one-off inheritance (i.e. inheriting a module and instantiating it once). Since one-off inheritance is common in structural modeling, this reduces specification overhead in LSS. More formal styles of inheritance can be achieved via userpoint assignment and module wrapping.

To allow userpoints to maintain state across invocations, LSS also supports the ability to add state by declaring runtime variables. Runtime variables are variables available during simulation rather than during model compilation. To allow initialization and *synchronous* update of this added state, all modules possess two system-defined userpoints, `init` and `end_of_timestep`, that are invoked at the beginning of simulation and the end of each clock cycle respectively. Once created, users can reference runtime variables in other userpoints to help customize computation.

4.4 Polymorphism

To support reuse across datatypes, LSS supports two types of polymorphism: parametric polymorphism and component overloading. Parametric polymorphism is supported through use of type variables. For example, any port in LSS, instead of having a basic type, such as `int`, may have a polymorphic type that contains type variables. Line 4 and 5 in Figure 8 state that the `in` and `out` port will have the type specified by the type variable `'a` (all type variables in LSS begin with a `'`). The type variable can be instantiated with any LSS type. The fact that both the `in` and `out` ports use the same type variable means that both ports must have the same basic type. While this example demonstrates parametric polymorphism on a hierarchical component, it can also be used on leaf components. In such cases, the BSL code for the leaf component is specialized based on the basic type given to all type variables.

Component overloading in LSS is achieved through the use of *disjunctive-types*. A disjunctive-type, denoted as `type1|type2` in LSS, specifies that the entity with this type may statically have type `type1` or `type2`, but not both simultaneously. Notice that this is *different* from union types which may store either type depending on the value assigned at runtime. Since modules may define many ports, implementing the full cross-product of allowable overloaded configurations may be extremely cumbersome. However, since the types are resolved statically, rather than implementing multiple *entire* behaviors for a given component, the BSL can specify type dependent code fragments and the code generator can customize this code using the statically resolved type information.

Since it is common to have many polymorphic components in a model (e.g. long chains of polymorphic data routing components and polymorphic state elements), *manually* resolving all the polymorphism can be very tedious. In practice, hundreds of explicit type instantiations are necessary to resolve the polymorphism [18]. To avoid this overhead, LSS *automatically* resolves polymorphism via type inference based on the structure of the model. For example, in the code shown in Figure 8, since the `delay` module requires type `int` on its ports and both the `in` and `out` ports of the `delayn` module are connected to instances of this module, the type variable `'a` will be resolved to have type `int`.

Since ports can have disjunctive types, the LSS type inference problem is non-trivial. Details regarding the LSS type inference problem can be found in Section 5.

4.5 Instrumentation

To allow a model to be reused for different data collection needs, LSS supports a mechanism to separate model specification from model instrumentation. As was possible in static structural modeling, LSS uses an aspect-oriented data collection scheme. Each module can declare that its instances emit certain *events* at runtime. These events behave like join points in aspect-oriented programming (AOP). Each time a certain state is reached or value computed, the instance will emit the corresponding event and user-defined *collectors* will fill these join points and collect information for statistics calculation and reporting. BSL code may be specified for the collector that processes the data sent with the event to accumulate statistics that can be reported during or at the end of simulation and used for visualization or model debugging.

In addition to declared events, LSS automatically adds code to emit an event whenever a value is sent on a port. Since many important hardware events are synchronized with communication, many useful statistics can be gathered using just these port firing events.

5. TYPE INFERENCE

As described earlier, LSS will attempt to assign basic types to all ports via type inference. This type inference greatly reduces the tedium in using polymorphic components since it frees the user from explicitly specifying each type.

The type inference problem for LSS can be formulated as trying to assign values to a set of type variables under a set of constraints. When defining a system's ports and connections, the user annotates each port and optionally annotates each connection with a type scheme. The legal types and type schemes in the system are specified by the following grammar:

Basic Types	τ	::=	$\texttt{int} \mid \ldots \mid \tau[n] \mid \texttt{struct}\{i_1 : \tau_1; \ldots i_n : \tau_n;\}$
Type Variables	α, β, γ	::=	$'i$
Type Schemes	τ^*	::=	$\alpha \mid (\tau_1^* \mid \ldots \mid \tau_n^*) \mid \texttt{int} \mid \ldots \mid \tau^*[n] \mid \texttt{struct}\{i_1 : \tau_1^*; \ldots i_n : \tau_n^*;\}$
Identifiers	i	::=	*any identifier*

If two ports in the system are connected, a constraint term that equates the corresponding type variables is added to the overall constraint. For each connection annotated with a type scheme, a pair of constraint terms that equate the connected ports' type variables to the annotated type scheme is added to the constraint. The form of a legal constraint in the type inference problem is given by the following grammar:

$$\text{Constraints} \quad \phi \quad ::= \quad \top \mid \tau_1^* = \tau_2^* \mid \phi_1 \wedge \phi_2$$

The constraint $\top$ represents the trivially true constraint, the constraint $\tau_1^* = \tau_2^*$ asserts the equality of two type schemes, and the constraint $\phi_1 \wedge \phi_2$ represents the conjunction of the constraints ϕ_1

and ϕ_2. The type inference engine must assign a basic type to all the type variables while satisfying the constraint.

The type system and constraints are very similar to those found in languages such as ML [9]. However, notice the $(\tau_1^* \mid \ldots \mid \tau_n^*)$ type scheme. Any entity annotated with this type scheme must statically have a *single* basic type which is accepted by one of the type schemes τ_1^*, τ_2^*, ..., or τ_n^*. This is different than a union type in which values that match any of the type schemes can be passed at run-time. This *disjunctive* type scheme comes from the need to allow overloading of port types as discussed in Section 4.4.

The presence of this disjunctive type prevents the typical unification algorithm from working for the LSS type system. The problem arises because it is not possible to assign a basic type to a type variable based solely on the disjunctive type constraint, even if the disjunctive type scheme has no unbound type variables. In fact, the LSS type inference problem is NP-complete [18].

The type system and inference problem presented here is very similar to the type system and inference problems in languages such as Haskell. However, the Haskell problem is undecidable in general [16]. There exist restricted versions of the type system that are decidable [10, 14]. Unfortunately, of these, the restrictions that yield acceptable computational complexity [10] are not desirable in a structural modeling environment since they forbid common port interface typings. For other restricted versions of the type system, we know of no heuristic algorithms that are appropriate for instances of the problem that arise when using LSS.

Thus, to perform type inference, LSS uses a modified version of the typical unification algorithm. This algorithm, upon encountering a constraint of the form $(\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi$, recursively applies itself to all constraint systems of the form $\tau^* = \tau_i^* \wedge \phi$. If at least one of those constraints has a solution, then that is also a solution to the original constraint.

Since this straight-forward extension of the unification algorithm is too slow in practice, a few heuristics are used make the algorithm practical. First, constraint terms are reordered so that non-disjunctive constraint terms are simplified first. This eliminates the need to re-solve them during the recursion used to handle disjunctive terms. Second, a heuristic is employed that intelligently solves certain disjunctive terms without recursion. Third, a divide-and-conquer heuristic that partitions disjoint constraint terms into separate simpler constraints is applied and these constraints are solved separately. Factoring the constraint in this way exponentially reduces the number of recursive calls. With these heuristics, type inference completes in several seconds for all cases we have observed, and we expect comparable times for all practical models. Without these heuristics, type inference times exceeded 12 hours for most models. A detailed description and analysis of the algorithm and heuristics is beyond the scope of this paper but can be found in the references [18].

6. USE-BASED SPECIALIZATION

Just as explicitly specifying types is tedious and often unnecessary, explicit parameter value specification is often tedious and redundant. In a manner analogous to type inference, the LSS compiler infers certain parameter values from the way in which a component is used. We refer to the process of components customizing themselves according to these inferred parameter values as *use-based specialization*.

6.1 Use-Based Specialization Design

The simplest example of use-based specialization involves port widths. Consider the `delayn` module presented in Figure 8. The `delay` module used as the building block of the `delayn` module supports multiple connections to its `in` and `out` ports. However, any such connections made to the `in` and `out` ports of the `delayn` module will do nothing since only the first port instance is connected internally. To alter the behavior of `delayn` we could change the code so that it defines a `width` parameter that makes multiple connections between the chain of delay elements. Figure 10 shows what the code would look like.

```
1   module delayn {
2     parameter n:int;
3     parameter width = 1:int;
4
5     inport in:'a;
6     outport out:'a;
7
8     var delays:instance ref[];
9     delays=new instance[n](delay,"delays");
10    var i:int;
11
12    /* The LSS_connect_bus(x,y,z) built-in does:
13     *
14     *   for(i=0; i<z; i++) { x[i]->y[i]; }
15     */
16    LSS_connect_bus(in,delays[0].in,width);
17    in -> delays[0].in;
18    for(i=1;i<n;i++) {
19      LSS_connect_bus(delays[i-1],delays[i].in,width);
20   }
21   LSS_connect_bus(delays[n-1],out,width);
22  };
```

Figure 10: Modified `delayn` module that supports multiple port connections.

```
1  instance gen:source;
2  instance hole:sink;
3  instance delay3:delayn;
4
5  delay3.n=3;
6  delay3.width=5;
7
8  LSS_connect_bus(gen.out, delay3.in, 5);
9  LSS_connect_bus(delay3.out, hole.in, 5);
```

Figure 11: Use of the modified `delayn` module.

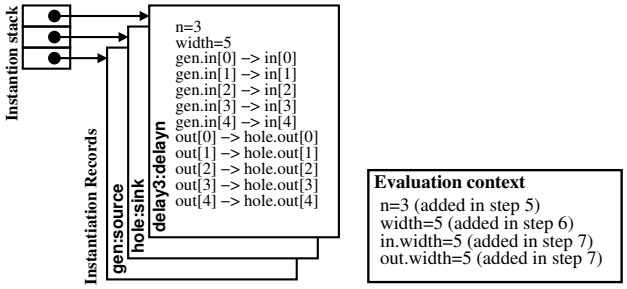
To use the module, one would instantiate the new `delayn`, set the width parameter, and then make the corresponding number of connections to the instance. The instantiation code is shown in Figure 11. While setting this single `width` parameter does not seem overwhelming, with many ports this type of construct can quickly clutter code and easily lead to errors, especially if this parameter must be kept consistent with the connectivity of each port. Use-based specialization can be used to avoid this scenario. In LSS, an implicit parameter named `width` is defined on each port. Rather than the user explicitly setting this parameter, its value is inferred by counting the number of connections made to the port.

With use-based specialization, the code in Figure 11 would be identical, except that Line 6 could be omitted. Figure 10 would be modified by omitting Line 3, replacing occurrences of `width` with `in.width`, and adding a check to ensure that `in.width` is the same as `out.width`.

While the above example is extremely simple, use-based specialization can be extremely powerful. Consider, for example, a branch prediction module which also supports branch target buffer (BTB) functionality. Since it is clear that requesting branch target information requires a BTB, using use-based specialization, the branch prediction module can infer whether BTB behavior is necessary by checking to see if a `branch_target` port is connected.

```
1  module ... {
2    inport in:'a;
3    outport out:'a;
4    ...
5    if(out.width < in.width) {
6      parameter arbitration_policy:
7             userpoint( /* args */ =>
8                        /* ret */);
9      instance arb:arbiter;
10     arb.policy = arbitration_policy;
11     ...
12   } else {
13     ...
14   }
15   ...
16 };
```

Figure 12: Use-based specialization exporting additional parameters



(a) Instantiation stack. (b) Context for delay3.

Figure 13: LSS interpreter state.

Figure 12 is another example of use-based specialization. Here, the module infers whether an internal arbiter is necessary by comparing the width of input ports to those of output ports. If the arbiter is necessary (i.e. the input port is wider than the output port), the module can export a userpoint parameter so that the arbitration policy can be parametrically specified (Lines 5-12). Without use-based specialization, the parameter `arbitration_policy` would have to exist and, since it has no default value, would also have to be set even when no arbitration is necessary. A default arbitration policy can be added. However, in the case where arbitration is required, having the module quietly select a policy is undesirable since there are many reasonable default policies. Use-based specialization eliminates this trade-off by providing the best of both alternatives; the user is required to specify the policy when it is necessary and is not required to specify the policy otherwise.

It should be noted, as the last example demonstrates, that use-based specialization allows a module's parameterization and communication interface to be affected by the connectivity and parameterization of other ports and parameters on the module. This creates a dilemma for straight-forward evaluation of LSS. A module cannot be used until it is instantiated, but the constructor cannot be called until code following the instantiation line has executed. To resolve this difficulty, LSS has novel evaluation semantics that are described in the next section.

6.2 Use-Based Specialization Implementation

Use-based specialization requires that the module body has access to values that are defined by the usage of the module instance (e.g. the number of connections to a port and values of all explicitly specified parameters). However, use-based specialization also allows the module's interface (i.e. the module's ports and parameters) to depend on the same values. Thus, use-based specialization requires deferring module body evaluation until after the module is instantiated and used, however, conventional evaluation requires that the module body be evaluated before it can be used so that its interface is known. To remedy this circularity, LSS uses the novel evaluation semantics described in this section.

Clearly, evaluation of the module body cannot occur as soon as an instance is created since the module body depends on the values of the module's parameters. Thus, rather than invoking the module body when creating a new instance, the name of the newly created instance and the module from which it was instantiated are pushed onto an instantiation stack.

Code continues to execute from the current module body, and whenever an assignment to a subfield of a sub-instance (e.g. Line 5 in Figure 9), is encountered, the assignment is recorded as a potential parameter assignment. Similarly, whenever any connection is made to a subfield of a sub-instance, the connection is recorded as a potential port connection.

When the current module body finishes evaluation, the instance at the top of the instantiation stack is popped off and its module body executed. Whenever the module body declares a parameter, the previously recorded potential parameter assignments are consulted to see if the parameter has a user specified value. If so, the type of the value is checked against the parameter's type and if the types match, the parameter is assigned that value. If no assignments were recorded, the parameter will get its value from default parameter assignments inside the module body, if they exist. Similarly, when a port is declared, the recorded list of connections is consulted to see if any attempts to connect to this port have been made. If so, the port is connected, and its implicit width attribute is set. After evaluation of the module completes, the potential subfield assignment and potential connection records are checked to make sure no non-existent parameters or ports on this instance were referenced. Additionally, all the parameters are checked to ensure that they have some value.

The following example will illustrate the execution of the code shown in Figure 11.

1. Line 1. The interpreter records that an instance of the `source` module named `gen` was created by pushing it onto the instantiation stack.
2. Line 2-3. The same is done for the `hole` and `delay3` modules.
3. Line 5-6. The interpreter records the assignments to potential parameters `n` and `width`.
4. Line 8-9. The interpreter records the connections made.

At this point, the top-level code has finished and so the module at the top of the instantiation stack is popped and its constructor evaluated. Figure 13a shows the instantiation stack in the LSS interpreter at this point in the execution. In this case, the next set of code to run is that for the `delayn` module shown in Figure 10.

5. Line 2. Check to see that `n` has an integer value in the record. If so, add it to the evaluation context based on the value in the record. The evaluation context is shown in Figure 13b.
6. Line 3. Do the same for the `width` parameter.
7. Line 5-6. Record the fact that connections to the `in` and `out` port are valid, compute the port widths, and add the *portname*.width parameters to the context.

8. Line 12-21. Execute the code, recording the instantiations, assignments, and connections as before.

Once this code is finished, the interpreter ensures that no connections were made to non-existent ports and no illegal parameter assignments were made. In this example, the code is correct so evaluation continues. The next instance on the stack, in this case one of the `delay` modules pushed onto the stack during the evaluation of the instance `delay3`, is popped off the stack and evaluated.

This behavior can be specified more formally by describing the execution semantics for the LSS language as an evolution of program states. Execution semantics expressed in this way are typically called *small-step semantics* [5]. The complete small-step semantics has numerous state transition rules that closely resemble the semantics for common imperative programming languages. Therefore, in this section, only the state transitions that relate to the implementation of use-based specialization will be described. This section uses the notation and terminology that is used by Harper [5], unless defined differently below.

The state of an LSS program during execution will be represented by a 7-tuple, (M, I_s, L, A, B, e, S). M is the netlist of the design as it is known at the current point in program evaluation. I_s is the stack of instances that need to be processed. L is the evaluation context and maps symbols to values. A is the recorded list of potential parameter assignments and port connections obtained from the parent instance (the instance in the hierarchy above the one currently being processed). B is a context that records potential parameter assignments and port connections for children of the current instance. e is the current expression being evaluated, and S is the current list of statements being evaluated.

The program starts in the initial state $(\cdot, \cdot, L_0, \cdot, \cdot, \cdot, S_0)$, where L_0 defines built-in functions and S_0 is the statement list at the top-level of the LSS specification.

The state transition function for LSS program states are expressed using propositional logic. In the transition rules below, items that appear above the horizontal bar represent the hypothesis of a logical statement. The items that appear below the bar represent the conclusions of the statement. The notation $q_0 \rightarrow q_1$ is used to denote that q_0 can transition to state q_1. The conclusions of all the transition rules identify all the legal transitions. Since any given state of an LSS program satisfies the hypothesis for at most one transition rule, the transition rules define the state transition function.

The interesting rules for LSS evaluation are used when a new instance is created. The rule for this statement is shown below. Note that the portion of the rules related to actually augmenting M is not shown but the extension is straightforward.

$$\frac{\begin{array}{c} c \text{ current instance name} \quad n \notin \mathsf{dom}(L) \quad m \in \mathsf{dom}(L) \\ S' = \mathsf{body}(m) \quad i = (c.n, S') \end{array}}{\begin{array}{l} (M, I_s, L, A, B, \cdot, \texttt{instance}\ n : m; S) \rightarrow \\ \qquad (M', i \rhd I_s, \{n \mapsto (c.n, S')\} \cup L, A, B, \cdot, S) \end{array}}$$

This rule pushes the constructor for instance i onto the stack of constructors I_s that must be evaluated and continues evaluating the statements in the current statement list. (Note that $i \rhd I_s$ denotes a stack with the element i at the top and the stack I_s below it.) Notice that this differs from standard evaluation which would have immediately begun processing S'.

When the current instance is finished (i.e. no statements are left in the current statement list), the following rule begins evaluating the next instance constructor.

$$\frac{\begin{array}{c} A = \emptyset \quad c \text{ current instance name} \quad i = (c.n, S') \\ A^* = \mathsf{extract}(c.n, B) \quad A' = \mathsf{strip}(A^*) \end{array}}{(M, i \rhd I_s, L, A, B, \cdot, \cdot) \rightarrow (M, I_s, L_0, A', B \setminus A^*, \cdot, S')}$$

The function $\mathsf{extract}(c.n, B)$ extracts from B all parameter assignments and connections for the instance named $c.n$. The function $\mathsf{strip}(A^*)$ strips the $c.$ prefix from all the symbol names, $c.n$, in the context A^*. The state for the next instance to be processed is established by extracting the recorded potential assignments for the about-to-be-processed instance and making this set of assignments the new A context. The $A = \emptyset$ hypothesis ensures that no assignments to undefined parameters can occur. The mechanism for this is explained below.

The remaining small-step inference rules are very similar to other imperative programming languages. The most complex rules not shown are the ones for parameter and port declarations. The parameter rule removes from A any assignments to the parameter being defined and updates L and M appropriately. The port declaration rule is similar. Since the records are removed from A, if $A \neq \emptyset$ after a module finishes evaluation, then an assignment or connection was made to an undeclared parameter or port.

7. EXPERIENCE WITH LSS

LSS is the front-end language to the Liberty Simulation Environment (LSE). This section gives some background that provides additional insight into the design of LSS, describes experience with LSE, and discusses the reuse provided by LSS.

LSE originally used a static structural specification language instead of LSS. Thus, the system resembled those described in Section 3.1. In this system, a few models were created, the largest being a cycle-accurate clone of a popular simulator hand-coded in C, SimpleScalar [1]. Development of larger, more ambitious models was hindered by the lack of flexible hierarchical components. In developing microarchitectural models, common instance and connection patterns emerged, but in each case the structure or width necessary varied slightly. For the reasons described in Section 3.1, these patterns could not be encapsulated into reusable hierarchical components. This forced cut-and-paste reuse rather than the more formal reuse patterns described in this paper. As model complexity grew, managing code riddled with cut-and-pasted fragments became overwhelming. An alternate solution was clearly necessary.

In response to this need, the LSS language was designed and implemented. After a direct conversion of the non-LSS version of the SimpleScalar model to the LSS-based model, there was a 35% reduction in line count. Furthermore, since the creation of LSS, we have been able to build larger more aggressive models for research and instruction. Table 3 lists several such models.

LSS allows the creation of flexible models and components in practice. For example, the IA-64-based chip multiprocessor (CMP) model (Model E), was constructed by instantiating two IA-64 cores (Model D) and connecting them to a shared cache hierarchy. Model E was sufficiently flexible to allow exploration of a novel communication structure between processor cores, to study the effect of the number of functional units and their mix, to evaluate the effect of static and dynamic instruction scheduling, and to measure the effect of various memory subsystems. To facilitate this study, each processor core in the model exported parameters ranging from simple parameters which controlled the number of instructions that should be fetched per cycle to complex parameters which controlled whether the processor can issue instructions out-of-order. Other researchers have also used LSS and LSE to create flexible models and libraries of flexible components. For example, researchers at Rice University have used LSS to model programmable network interface architectures [19].

LSS's features greatly simplified construction of these models. Table 2 summarizes data that quantifies these benefits. Overall, use-based specialization was able to infer 3904 port widths across

Model Name	Instances	Hierarchical Modules	Leaf Modules	Instances per Module	Instances from Library	Modules from Library	Explicit Type Instantiations w/o Type Infer.	Explicit Type Instantiations w/ Type Infer.	Inferred Port Widths	Connections
A	277	46 (10)	18	4.33 (8.61)	73%	13	115	8	816	919
B	281	46 (11)	18	4.39 (8.48)	73%	13	116	8	823	929
C	62	1	18	3.37	73%	10	38	30	147	304
D	192	4	25	6.62	86%	22	147	71	611	3975
E	329	4	26	10.97	89%	22	162	71	984	4528
F	183	18 (3)	19	4.95 (8.32)	82%	18	101	38	523	1395
Total	1324	69 (19)	39	12.26 (22.83)	80%	22	679	226	3904	12050

Model descriptions are in Table 3. Values in parenthesis discount trivial hierarchical modules used simply to wrap a collection of components.

Table 2: Quantity of Component-based Reuse

Model Name	Model Description
A	A Tomasulo Style machine for the DLX instruction set.
B	Same as A, but with a single issue window.
C	A model equivalent to the SimpleScalar simulator [1].
D	An out-of-order processor core for IA-64.
E	Two of the cores from D sharing a cache hierarchy.
F	A validated Itanium 2 processor model.

Table 3: Several models developed with LSS.

the models, obviating the need to keep these parameter values consistent with the 12050 connections in the models. As is obvious from these numbers, manually specifying all the connections would be impractical; algorithmic specification of structure was vital in developing this model. Further, notice that type inference reduced the total number of required type instantiations from 679 to 226, a 66% reduction. The aspect-oriented instrumentation features of LSS also proved invaluable allowing easy migration between data collection probes for experimentation and debugging probes to correct inaccuracies in the models.

Overall, the quality of the LSE system and degree of component reuse has improved dramatically with the addition of LSS. From Table 2 one can see that 80% of the 1324 component instances in these models came from a library of only 22 components. Over all the models, each module was used approximately 12 times. If trivial hierarchical modules used simply to wrap a collection of components are discounted each module is used about 22 times across the models. This reuse translates into reduced specification time. For example, a single student, in only 7 weeks, was able to specify *and* validate, to within a few percent of hardware CPI (cycles per instruction), the Itanium 2 model (Model F).

8. RELATED WORK

Section 3 described the two major classes of concurrent-structural systems, OOP-based concurrent structural systems and static structural systems, and explained the strength and weakness of each. In this section, we classify several well known concurrent-structural modeling systems and explore their strengths and weaknesses.

SystemC. SystemC [11] is an OOP-based concurrent-structural modeling system built as a library for the C++ programming language. Its limitations are described in Section 3.2. In terms of simulation speed, *reusable* components in LSE with LSS are at least as fast as *custom* components written in SystemC [12].

Ptolemy II. Ptolemy II [7], when used without the Vergil interface, is an OOP-based concurrent-structural modeling system for systems with heterogeneous models of concurrency. Ptolemy allows users to define system structure directly in Java. However, when used in this way it suffers from the shortcomings of structural-OOP systems described in Section 3.2. At no point during execution does structure need to be immutable limiting static analysis capabilities described in Section 3. Using LSS with Ptolemy II as a BSL, however, would address these shortcomings and bring powerful reuse to heterogeneous system exploration.

Ptolemy II with Vergil. Ptolemy II, when used with the Vergil interface, is a static structural modeling system that supports structure based analyses such as parameterization of non-structural specification (but not of structural specification). Ptolemy cannot be used with and without Vergil simultaneously.

VHDL. VHDL [6] is a tool commonly used for RTL-level modeling and synthesis of hardware systems rather than high-level modeling. Despite its low-level target, VHDL does have a flexible type system, limited support for parameter-based structure, and syntax for behavioral (as opposed to RTL) specification. But, VHDL does not support many of the features outlined in Section 3, making component reuse difficult. For example, VHDL does not support polymorphic types, thus forcing reimplementation of common components like arbiters based on the datatypes involved. Furthermore, VHDL has no inheritance mechanism to allow the customization and extension of components. Finally, VHDL does not support use-based specialization. It may be possible, however, to add these capabilities to VHDL by using it as a BSL for LSS.

Balboa. Balboa [2] is a structural modeling system designed to allow components developed in various otherwise incompatible C++ modeling environments to be composed. Balboa separates interface definition and structure from component behavior by using an interface definition language (IDL), with behavior is specified in C++. They call this approach *split-level programming*. The composition language in Balboa is a program, and thus split-level programming is closely related to the compilation process with LSS.

Unlike LSS, however, the Balboa structure specification language is evaluated at simulator run-time and thus suffers from problems similar to structural-OOP systems. However, Balboa does use run-time component connection to provide component overloading. The IDL program is run to determine the structure of the model and the resulting structure is used to infer the particular version of an overloaded component that should be instantiated and the correct component is connected. Unfortunately, this approach cannot resolve parametric polymorphism via structure since the resolved types direct compilation of behavioral code at compile-time.

Note that the techniques Balboa uses to reconcile differences between components from different simulation systems can be used in conjunction with LSS to gain the benefits of both LSS and Balboa. Also note that Balboa's type inference problem [3] is closely related to LSS's type inference problem. However, the LSS algorithm is more closely related to the basic unification algorithms used in languages such as ML [9].

Polylith. Outside the hardware specification domain, structural interface definition languages (IDL) such as the MIL language used in the Polylith software bus system [13] also employ concurrent-

structural modeling. This approach is a natural fit since the goal of these tools is to separate the specification of functionality from the specification of interface and communication. Similar to static structural systems, Polylith's MIL structure specification language uses a declarative syntax to specify component interconnectivity. Most of the features related to reuse involve resolving differences in datatypes and communication semantics in systems written in different languages or running on different platforms in a distributed computing environment. Since these systems specify the behavior of a program by structurally composing components, the ideas presented in this paper can be used to add reuse, such as the ability to reuse components to structurally build new easy-to-use reusable components. Other IDL systems, similar to Polylith, exist, but a comparison to these systems is beyond the scope of this paper.

Asim. Asim [4] is a pseudo-structural processor modeling tool developed at Digital Equipment Corporation (and now worked on at Intel). Of all the related work, its high-level architectural modeling goals are perhaps the most in line with LSE's. Asim uses structural composition between components, but uses functional composition for communication within a clock cycle. Thus, the system is not *purely* concurrent-structural, making component based reuse difficult [17]. Since the system is not a pure concurrent-structural system, the techniques presented in this paper may not be directly applicable to Asim. Asim, however, does support some component reuse through inheritance. Asim details are difficult to evaluate because the tool is not openly available.

9. CONCLUSION

This paper presents several programming language techniques targeted at improving hardware systems development by increasing the rate at which ideas can be evaluated. These techniques, described in the context of the design and implementation of the Liberty Structural Specification language (LSS), increase designer productivity by encouraging wide-scale component-based reuse. While prior structural modeling approaches forced a trade-off between easy reuse and easy construction of reusable components, LSS provides the best of both through a hybrid solution. The LSS language provides imperative programming constructs to allow for programmatic control of structure in flexible hierarchical components, but is evaluated at compile time thus *also* allowing static analysis of the model structure. These static analyses allow the many properties of flexible components to be inferred, reducing specification overhead. This paper presents the static type inference algorithm used in LSS to resolve both parametric polymorphism and component overloading. It also presents a new programming language technique, called *use-based specialization*, which, in a manner analogous to type inference, customizes reusable components by statically inferring component parameter values that would otherwise have had to be specified manually. Experience with LSS indicates that these features significantly improve the reusability of components, dramatically improving productivity in hardware modeling.

Acknowledgments

We thank Jason Blome, Azmat Hussain, Sharad Malik, Vijay Pai, David Penry, Ram Rangan, Paul Willmann, and the entire Liberty Research Group for their support throughout the development of LSE and LSS. This work has been supported by the National Science Foundation (NGS-0305617) and Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation or Intel Corporation.

10. 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] DOUCET, F., OTSUKA, M., SHUKLA, S., AND GUPTA, R. An environment for dynamic component composition for efficient co-design. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2002).

[3] DOUCET, F., SHUKLA, S., AND GUPTA, R. Typing abstractions and management in a component framework. In *Proceedings of Asia and South Pacific Design Automation Conference* (2003).

[4] 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.

[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. Disciplining 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] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[10] ODERSKY, M., WADLER, P., AND WEHR, M. A second look at overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1995), pp. 135–146.

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

[12] PENRY, D., AND AUGUST, D. I. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference* (June 2003).

[13] PURTILO, J. M. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 1 (January 1994), 154–174.

[14] SEIDL, H. Haskell overloading is dexptime-complete. *Information Processing Letters 52*, 2 (1994), 57–60.

[15] SKADRON, K., MARTONOSI, M., AUGUST, D. I., HILL, M. D., LILJA, D. J., AND PAI, V. S. Challenges in computer architecture evaluation. *IEEE Computer* (August 2003), 30–36.

[16] SMITH, G. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming 23*, 2-3 (1994), 197–226.

[17] VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture* (November 2002), pp. 271–282.

[18] VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., MALIK, S., AND AUGUST, D. I. The Liberty Simulation Environment: A deliberate approach to high-level system modeling. Tech. Rep. Liberty-04-02, Liberty Research Group, Princeton University, January 2004.

[19] WILLMANN, P., BROGIOLI, M., AND PAI, V. Spinach: A Liberty-based simulator for programmable network interface architectures. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (June 2004).

[20] XIONG, Y., AND LEE, E. A. An extensible type system for component-based design. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (March 2000), pp. 20–37.