Modeling Modern Micro-architectures using CASL *

Edward K. Walters II, J. Eliot B. Moss, Trek Palmer, Timothy Richards, and Charles C. Weems
Department of Computer Science, University of Massachusetts Amherst

Abstract

We overview CASL, the CoGenT Architecture Specification Language, a mixed behavioral-structure architecture description language designed to facilitate fast prototyping and tool generation for computer architectures with deep pipelines and complicated timing. We show how CASL can describe pipelines, dynamic information contexts, and contention using the DLX/MIPS architecture as an example.

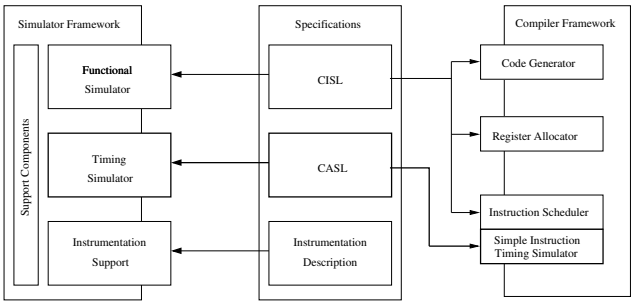

Figure 1. The CoGenT Project

1 Introduction

Computer architecture has now embraced multiple instruction streams for the masses. Multi-core processors and related strategies lead to much more complex systems—implying the need for more complex and capable simulators to explore and evaluate designs. Further, the speed of evolution of architecture requires much more rapid development of simulators and matching tools (such as compiler back-ends). The field needs a framework that can model these complex systems and produces, automatically, tools that are efficient.

One of the most effective ways to describe and generate an architectural simulator is to use of an architectural description language (ADL), a domain-specific language designed to describe a target architecture's behavior, timing, and structure. Ideally, an ADL provides abstractions and primitives that make descriptions more intuitive and compact. It also removes the need to create tools using general-purpose languages such as C++, which is more prone to inaccuracies and errors because of its lower level of abstraction.

Current ADLs fall into three categories [12]: behavioral, structural, and mixed. Our primary contribution here is a detailed description of the design and novel features of the new mixed ADL CASL, the CoGenT Architecture Specification Language. CASL aims to provide intuitive and powerful means to describe current and future architectures at a component level of abstraction, without resorting to a general-purpose language to implement novel components.

CASL includes the following:

1. A *component-oriented behavior and structure* language, with primitives and control structures designed to capture common micro-architectural idioms.

2. The concept of *strands*, dynamic execution contexts that traverse the structural model, e.g., instructions traversing the pipeline, or requests to distributed memory.

3. A rich *timing* description facility, including the ability to describe contention, pipelining, and complex timing without needing to resort to explicit signals as in hardware description languages such as VHDL [4] or Verilog [16].

We proceed as follows: Section 2 provides an overview of the CoGenT project and the role of CASL within it; Section 3 describes the DLX architecture and explains how CASL allows the specification author to describe pipelined structures, dynamic information flow, and contention; Section 4 describes related work, and Section 5 concludes.

2 Overview of CoGenT and CASL

CoGenT stands for Co-Generation of Tools, particularly compilers and simulators. Compiler and simulator tools for systems research are difficult to develop and coordinate since each tool is complex in its own right, and both are dependent on aspects of the target architecture. The CoGenT project (Figure 1) addresses this problem by provid-

*This material is based upon work supported by the National Science Foundation under grant number CNS-0615074. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.
1-4244-0910-1/07/$20.00 ©2007 IEEE.

ing two sets of components: multiple coordinated specifications that describe the target instruction set architecture and micro-architecture; and tools that process these specifications and produce compiler and simulator components such as code generators, instruction schedulers, and simulators at varying degrees of detail. Generating system tools from the same descriptions ensures that they will always be consistent, and since tool generation is fast and automatic, CoGenT allows designers to explore more design space in less time and reduces programmer-introduced errors.

Prior systems tend to use either a simple language, making descriptions unwieldy, or an ADL augmented with a complete general purpose language, rendering analysis difficult. Instead, we provide multiple coordinated ADLs, including: CISL, the CoGenT Instruction Specification Language [7, 8], which describes instruction formats and behavioral semantics; and CASL, the CoGenT Architecture Specification Language, which describes micro-architectural structure, behavior, and timing. We focus here on CASL. Being a mixed ADL, CASL contains three kinds of constructs: *structural* elements in the form of a component graph, where components contain an interface and an implementation; *behavioral* elements similar to C or Java functions (called *actions* in CASL) that describe the behavior of the component; and *timing* elements that describe the timing relationships between components and their actions. We do not cover all CASL syntax here—interested readers are directed to the CASL manual [17]. A more thorough treatment of many of the issues covered in this paper can also be can be found in [3]. Rather, we describe certain advanced features of CASL, motivated by the DLX architecture as an example.

3 Modeling a Processor Using CASL

We will now use the MIPS/DLX architecture (DLX for short) to introduce parts of the design of CASL. Even an in-order architecture such as the DLX presents a number of interesting modeling issues, particularly related to hazards, pipeline structures, and contention.

3.1 Overview of the DLX architecture

DLX [10] is a general purpose processor adapted for educational purposes from a MIPS [6] micro-architecture. DLX has a single five-stage in-order integer pipeline with a three-ported (two read, one write) register file, a forwarding network, and simple branch prediction. The stages include instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and register write-back/commit (WB). Latches connect the stages to provide storage between cycles.

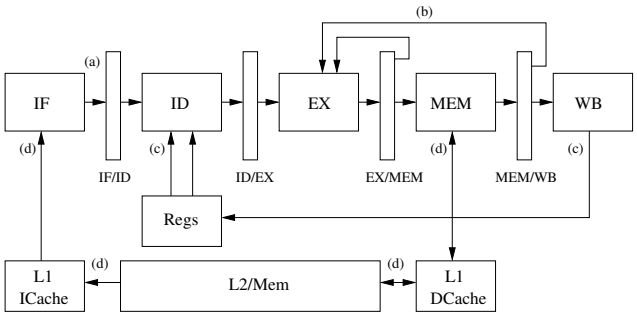

Figure 2. DLX and Memory Structure

We offer an abstract view of this pipeline in Figure 2. In addition to the pipeline and register file, we include a simple memory hierarchy consisting of separate instruction and data L1 caches and a unified L2/Memory system. We also identify four different means of communication between the different components (stages, latches, and memory elements): conventional inter-stage communication *(a)*, forwarding logic *(b)*, communication with the register file *(c)*, and communication with the memory hierarchy *(d)*.

We use this example to illustrate how CASL supports describing *pipeline* component structures, *information flow* between components, and *contention*, in a succinct and intuitive manner.

3.2 Pipeline Support for the DLX

While CASL's features support simple forms of communication between components, it also provides support for more complex parallel communication. One common form is producer-consumer parallelism, i.e, pipelining, *(a)* in Figure 2. CASL implements pipeline support using two types of elements: an extensible library of *components* that ease describing pipeline connectivity, capacity, and functionality; and a novel language construct called a *strand* that describes the control and routing of a related group of information traversing the pipeline.

The CASL components that support pipelining are *stages*, *buffers*, and *connectors*. Unlike conventional CASL components, which have arbitrary interfaces that contain simple (non-component) types, pipeline components reference other components in their connection lists, handle strands, and must implement certain pre-defined interfaces.

A *stage* component gives the behavioral specification for one or more stages of a pipeline. For example, the fetch stage for a simple MIPS-style pipeline would contain behavioral code for fetching instruction words from the i-cache. The only pre-defined action a stage must implement is the `visit()` action, which performs the stage's work.

Buffer components track strands as they move from stage to stage. Each CASL strand, at any point in time, is held in

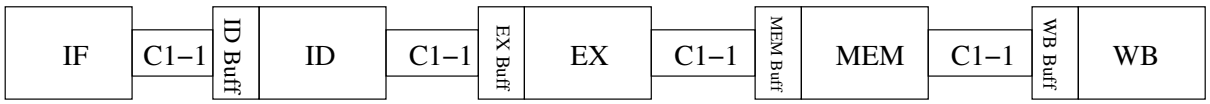

Figure 3. DLX Pipeline Structure Using CASL Components

exactly one buffer. This is not a limitation on the power of strands because strands can also produce and consume other strands, so CASL can simulate strand splitting for applications such as predicated speculative execution. The simplest implementation of a buffer is a latch, which can hold one strand. Buffers can also possess more involved behavior such as issue logic. Buffers must provide actions for querying their available capacity and sending or receiving strands from other pipeline elements.

Connection components provide detailed routing and connection behavior between buffers and stages. Connections query the capacity of buffers and push or pull strands through the stage servicing the buffer. Connections also can express many-to-one and one-to-many connectivity. For example, the issue stage in most multi-issue processors involves routing from a single issue stage to multiple execute stages' reservation stations. In this case one uses a one-to-many connector customized to the issue constraints of the architecture.

Building a pipeline in CASL involves integrating these three types of components into a structure that corresponds closely to the modeled pipeline. We have found buffer-stage-connector chains to be most useful for this. Connectors drive the pipelined system: they pull strands out of buffers, send them through stages, and dispatch them to the buffers of the next stage.

Figure 3 shows an example pipeline using this pattern for the DLX. We omit the forwarding and storage (e.g., register) elements for clarity. This example has the five stages mentioned previously, a buffer attached to the beginning of the latter four stages, and a series of one-to-one (C1-1) connectors joining the stages and buffers. Each stage except for IF is preceded by a corresponding buffer that stores a current or pending strand. The issue stage is a special case because it *generates* strands, which do not exist in a buffer until they are sent to the next stage; likewise, WB is special in that strands die when they complete that stage.

While one can describe pipelines purely structurally (connecting up the ports), CASL supports special, more concise, notation, adapted from CASL timing annotations. CASL has three operators for describing concurrency: sequential: `a ; b` (for "`a` runs before `b`"); parallel: `a || b` (for "`a` runs in parallel with `b`"); and pipelined: `b1:s1 -{c1-1}-> b2:s2`, which indicates that connector `c1-1` joins stage `s1` with pre-attached buffer `b1` to stage `s2` with pre-attached buffer `b2`. Using these operators, in conjunction with a variety of connectors, we can describe arbitrary pipelines.

```
component DLX {
 implementation:
  Fetch IF;
  Decode ID;
  Execute EX;
  Mem MEM;
  WriteBack WB;
  OneToOneConnector C1, C2, C3, C4;
  Buffers B_ID, B_EX, B_MEM, B_WB;

  action clock
   << IF -{C1}-> B_ID:ID -{C2}-> B_EX:EX
         -{C3}-> B_MEM:MEM -> B_WB:WB >>
  { C1.pull(); }
}
```

Figure 4. Pipelines in CASL

CASL code for the DLX pipeline appears in Figure 4. One feature of CASL's timing description facilities is that one can attach timing annotations to actions. This example declares the five stages, four buffers, and four connectors for the pipeline, and connects them with pipeline notation. Here the only code the components must execute is the `visit` action, which models a clock tick by telling the connector `C1-1` between IF and ID to pull (i.e., generate) a strand from IF and start the execution process. The rest of execution is driven by the strands produced by IF. While we have used generic buffers and connections for this example, most examples require the use of custom buffer or connector behaviors to implement the semantics of the particular pipeline.

CASL can accommodate multiple issue by using multiple buffer visits, dynamic issue using connection routing, pipelines with loops (one can use a component reference an arbitrary number of times inside a pipeline), and arbitrary issue conditions such as alignment constraints, specific issue slots for specific instruction types, and complicated windowing schemes.

3.3 Dynamic Information Flow—Strands

An obvious issue for the pipeline of Figure 3 is how to represent the instructions that flow through the pipeline. Correct handling of instructions requires two elements: *control* for the instructions (routing through the pipeline network) and *decoding*, which places information such as register indices and computed values under the control of the instruction. A *strand* is a structure that holds this information.

Re-examining Figure 2, there are multiple candidates for strands. A clear choice is modeling each instruction as it

passes through the pipeline. Other candidates include memory operations, especially if the particular processor is being used in a shared-memory multiprocessor, and the forwarding network. Each strand contains information particular to a dynamic element of the architecture, which we call an *info-set*. For example, the info-set of the main pipeline includes instruction information, such as opcode, registers used, and intermediate results. Strands make it easier to reason about and model a computer architecture by describing the characteristics of the information passing through it. This balance between structural behavior and reactive (i.e., dynamically triggered) behavior was an important consideration in the design of CASL.

A strand explicitly describes an info-set processed by an architecture, such as an instruction, memory request, or independent and/or asynchronous action in the system. Unlike the static structural elements of a CASL description, strands can be created dynamically, destroyed, and transferred from component to component in CASL behavioral code. Strands are similar to Java classes in that they contain data and operations that can be accessed by other elements (i.e., other CASL components), but they are more restricted in their use. For example, a strand can be in only one "location" (buffer, storage element, etc.) at a time—copying out the strand means that the previous value is invalid. In addition, all strand data and operations are public by default, since the primary aim is interaction with static components.

The main advantage of strands is that they address, using one construct, two major concerns of modeling dynamic timing in an architecture: they encapsulate control information, exploited by structural elements to route and process the info-sets; and each strand models a particular resource request when determining structural hazards. This gives CASL the best of both worlds: we specify static structural elements as black boxes in a component graph, and separately describe the dynamic elements that traverse this graph, assisting the structure in routing and processing the elements.

3.3.1 Properties and Examples

Figure 5(a) shows an example strand definition for a DLX instruction. This strand holds two read register indices and one write index, along with raw representation of the instruction word. The decoder (Figure 5(b)) creates a strand with `new` and populates the register indices for use down the pipeline. We omit decoding logic, since decoders are not typically written using CASL—they are generated from CISL (our ISA language) descriptions. We idealize the issue logic for brevity. Figure 5 (c) shows termination of a strand using `delete`.

Each strand must implement the `path` action, which specifies the strand's routing and actions as it traverses the

```
strand DLXInst {
  reg_index_t read_reg1;
  reg_index_t read_reg2;
  reg_index_t write_reg;
  word_t inst_word;
  ...
  action path() {  ... }
}
```

(a)

```
action fetch {
  ...
  word_t raw_inst = ICache.fetch(PC);
  DLXInst inst
    = new DLXInst(raw_inst);
  // fills in regs
  decoder.decode(inst);
  ...
}
```

(b)

```
action writeback {
  ...
  DLXInst inst
   = MEM_WB_Buffer.next();
  delete(inst);
  ...
}
```

(c)

Figure 5. CASL Strand Operations

```
strand DLXInst
{  ...
  action path
    { Fetch.visit();
      Decode.visit();
      Execute.visit();
      Memory.visit();
      Writeback.visit(); }
  abstract action execute;
}
```

Figure 6. CASL Strand Paths

structural components. For example, an instruction strand traversing a pipeline has a `path` action designating what actions it requires in each stage. An integer instruction requires execution on an integer unit, in addition to stages such as memory and writeback. Describing a strand's requirements in this manner has the following advantages: it allows the strand to choose the actions it needs from a component as it passes through, facilitating the simulation of heterogeneous strands that traverse the same path; and it provides a natural way of choosing among multiple functional units with the same operation set (e.g., two integer pipes), since the path action denotes only the type of components the strand interacts with, not the specific instances.

We provide an example of a path function for a basic DLX instruction in Figure 6. This fills out the path action in the previous figure. Here, `DLXInst.path` contains the pertinent actions for the each stage that the strand traverses. In this instance we assume that every stage has a `visit` action implementing the functionality required by `DLXInst`.

Paths can also contain triggers for ("calls on") other actions in the current strand or any superclass of the strand (all CASL components and strands can use method overriding). This allows a strand to choose its path at run time. CASL supports this by using a limited form of dynamic subclassing similar to roles [1].

Dynamic paths work as follows: the description writer creates a superclass with the general path and the actions that will be dynamically sub-classed. These actions can ei-

ther have no implementation, or some default functionality. The actions must be included in the path description to be available for subclassing. Once the strand is designed to be subclassed, it can be transformed into a given subclass using casting syntax (e.g., `inst = (DLXIntegerInst) inst;`). The dynamically subclassed strand subsequently uses the subclass's actions, and possesses the subclass's data fields.

Dynamic subclassing typically occurs in response to additional information inferred about the strand. For example, an instruction that has been decoded is an excellent candidate for dynamic subclassing, since now more information is available within the strand that can be used for specialized processing or routing. This is how one would deal with a pipeline with multiple execute paths, e.g., fixed- and floating-point.

One final point to note about strands is how they interact with the pipeline components of CASL. Buffers store the strands and determine factors such as ordering and capacity. The combination of buffers and strands is CASL's preferred method of modeling structural hazards. Connectors are responsible for movement through the system by querying buffers and allowing eligible strands to proceed to the correct destination buffer. The strands then traverse the stage using the action mentioned in their path.

The actions within a path are all calls to stage actions. When a strand is inside the code for a particular stage, it is as if the strand called the stage with an implicit context called `this`. In this way the strand data (and actions) can be referred to within the stage code without forcing the stage to identify a specific strand. However, all strands that deal with a specific stage must conform to the same type (or supertype).

3.4 Contention in the DLX

Any element in a storage unit is a candidate for side-effects during a clock cycle. Most architectures ensure that these side-effects possess some form of sequential consistency. In a simulator, non-deterministic behavior is usually undesirable because it makes results more difficult to reproduce. While there are circumstances where conflicts and contention are desirable, we do not want it to occur when it is unwanted. CASL addresses these concerns by providing language support for eliminating undesired contention. The architect may define an order for the actions of a particular component. When multiple actions are triggered during a particular clock cycle, this order arbitrates between the actions in a straightforward and predictable manner.

Using contention in combination with strands, we can model the other forms of communication in Figure 2 beyond pipelining. Forwarding *(b)* can modeled in two ways: either standard inter-component messages (actions) are sent between pipeline stages, or strands are spawned from the memory and writeback stages whose plan encompasses an execute stage action. Either way, the contention primitives in CASL serve to arbitrate between the actions to ensure that the semantics of forwarding are correct.

For example, given three actions `visit`, `forward_mem`, and `forward_wb` implemented by the `execute` stage component, one can set up arbitration between them that dictates that forwarding actions take place before `visit` if all three actions occur during the same time interval (e.g., clock tick). The syntax for this arbitration is `(forward_mem || forward_wb); visit`, where `||` and `;` have parallel and sequential semantics as they do for pipelines. The two forwarding actions do not depend on each other, so they can occur in parallel.

This approach has the benefit of enforcing correct semantics without requiring an explicit execution model such as executing each pipeline stage in reverse order during each clock tick. Since CASL does not assume a default execution model, this approach also frees the specification author to use purely declarative descriptions of concurrency protocols.

We can also model register communication *(c)* and unified L2 cache contention *(d)*. In DLX, the register file can be accessed by two strands (instructions) each cycle: read in the `decode` stage and write in the `writeback` stage. To model this activity correctly without resorting to phase-level timing (that is an alternative), one only needs to impose a write-before-read ordering for the `read` and `write` actions of the register file component. These actions will then be referenced as part of the respective stage code, and the correct ordering results. Imposing the order L1 I-cache access before L1 D-cache access functions similarly for L2 cache accesses.

We have shown how CASL provides a number of features that facilitate a simple, declarative style of representing pipelines and contention that removes much of the conceptual difficulty in describing such structures.

4 Related Work

While there are many structural and behavioral ADLs, it is only recently that there has been substantial research in mixed ADLs. Current mixed ADLs include EXPRESSION [2], LISA [11], Facile [15], MADL [13], ADL [9], Pipe [5], and Liberty [18]. CASL is most closely related to Liberty and Facile.

Comparing with Liberty, CASL's structure and parameterization design is similar. However, CASL differs in the following respects: it does not treat components as strict black-boxes for inter-component analysis—its behavioral language is much more integral; it provides explicit sup-

port for pipelines and timing annotations; it handles control using strands instead of explicit control ports; and its descriptions support generating compiler-related components.

Facile provides what could be considered a precursor to strands in both its inference of dynamic simulation code from a description and its pipeline memoization techniques [14]. However, CASL's techniques apply to any dynamic element that appears in the architecture, not just pipelined instructions.

Further advantages of CASL include: better and more comprehensive tool support; expressing details of modern architectures, such as complex pipelines and timing; and separate ISA specifications (syntax and semantics) in CISL [8], keeping CASL specifications smaller and cleaner, and simplifying generation of functional simulators and compiler code generators.

5 Conclusions

We examined how our ADL CASL facilitates describing pipelines and contention. We also examined strands, a CASL control abstraction that leverages features from object-oriented languages such as dynamic subclassing. These features provide compact means for describing information flow through pipelines, at the same time enabling specialization of instructions to improve simulation performance. Strands also serve as a convenient representation of control information, and can be used in groups to model structural hazards.

Future work on CASL includes completing a static analysis engine attached to the compiler front-end that takes advantage of CASL's extensive behavioral and timing features. We also intend to generate compiler and simulator components from CASL in conjunction with the other languages that comprise the CoGenT project.

References

[1] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

[2] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM Press.

[3] E. K. W. II, J. E. B. Moss, T. Palmer, T. Richards, and C. C. Weems. CASL: A rapid-prototyping language for modern micro-architectures. Technical Report 04-07, University of Massachusetts Amherst, Department of Computer Science, Jan. 2007. Submitted for Publication.

[4] R. Lipsett, C. F. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

[5] C. W. Milner and J. Davidson. Quick piping: a fast, high-level model for describing processor pipelines. In *Proceedings of the Joint Conference on Languages Compilers and Tools for Embedded Systems (LCTES)*, pages 175–184, 2002.

[6] MIPS Technologies Inc., http://www.mips.org. *MIPS Technologies*, 2007.

[7] J. E. B. Moss, T. Palmer, T. Richards, E. K. W. II, and C. C. Weems. CMDL: A class-based machine description language for co-generation of compilers and simulators. In *Proceedings of the 2004 International Parallel and Distributed Processing Symposium Workshop on Next Generation Software*, page 8 pp., Santa Fe, NM, Apr. 2004. IEEE, IEEE Computer Society.

[8] J. E. B. Moss, T. Palmer, T. D. Richards, E. K. W. II, and C. C. Weems. CISL: A class-based machine description language for co-generation of compilers and simulators. *International Journal of Parallel Programming*, 33(2-3):231–246, 2005.

[9] S. Onder and R. Gupta. Automatic generation of microarchitecture simulators. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, page 80, Washington, DC, USA, 1998. IEEE Computer Society.

[10] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[11] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA—machine description language for cycle-accurate models of programmable DSP architectures. In *Design Automation Conference*, pages 933–938, 1999.

[12] W. Qin and S. Malik. *Architecture Description Languages for Retargetable Compilation*. CRC Press, 2002.

[13] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 47–56, New York, NY, USA, 2004. ACM Press.

[14] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, New York, NY, USA, 1998. ACM Press.

[15] E. C. Schnarr, M. D. Hill, and J. R. Larus. Facile: A language and compiler for high-performance processor simulators. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–331, June 2001.

[16] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1995.

[17] University of Massachusetts Amherst, http://osl-www.cs.umass.edu/cogent/documentation.htm. *CASL Manual 1.0*, 2007.

[18] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty structural specification language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.