A Comparison of Reuse in Object-oriented Programming and Structural Modeling Systems

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

Modeling systems in which users construct models by specifying the interconnect between concurrently executing components are a natural fit for hardware modeling. These *concurrent-structural modeling systems* allow the specification of the model to parallel that of the hardware making specifications clear, easy to understand, and straight-forward to build and modify.

Unfortunately, many concurrent-structural modeling systems provide few mechanisms to create flexible-reusable components. Others have built these modeling systems on object-oriented programming (OOP) languages in the hope that the OOP features will allow the creation of useful flexible reusable components for structural modeling. Unfortunately, simply adopting OOP features is insufficient for this task. By drawing an analogy between OOP programs and concurrent-structural models, it is possible to understand why. Furthermore, this analogy highlights what would be needed in a concurrent-structural system to gain the same benefits seen in OOP systems. This report compares the features necessary to obtain reuse in concurrent-structural modeling systems by drawing this analogy between object-oriented programming constructs and concurrent-structural modeling constructs.

1. INTRODUCTION

In the digital hardware design process, decisions made very early in the design cycle can have a large influence on final system performance. The cost of designing and manufacturing a hardware prototype for complex digital hardware to test its effectiveness is prohibitively high. Thus, high-level (i.e. microarchitectural level) models that can be compiled to simulators must be used to study the performance and other properties of a design to make these critical design decisions.

The more designs that can be explored, the better the chance of finding the optimal high-level design. Unfortunately, building a high-level model can take many months; model development time is a key bottleneck preventing wide range design space exploration early in the design cycle [**?**]. One reason for this is that pieces of a model are not easily usable in models for subsequent designs, tying models to a specific design. Higher levels of reuse among designs would greatly reduce the development time, especially for widely differing designs.

This lack of reuse is often caused by poor modeling methodologies. For example, in the general purpose microprocessor research and design community, hand coding a simulator in a sequential 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++ [5].

Special purpose modeling tools are one solution to this problem. These tools model hardware components with concurrently executing software components that communicate through statically connected communication channels. Since this modeling paradigm matches the structural composition of concurrent hardware components, component based reuse becomes far easier [5].

Unfortunately, even in these *concurrent-structural* modeling systems, components are rarely reused. Components in many of these modeling tools are inflexible and can only be reused when designs share identical. Thus, designs with even marginally different behavior often require reimplementation of components.

To gain the flexibility necessary for component reuse, a common solution is to build the high-level concurrent-structural modeling tool around an object-oriented programming (OOP) language such as C++ [2] or to introduce object oriented concepts to a structural modeling system such as VHDL [4]. The goal is to make the OOP features enable reusable components.

Unfortunately, object-oriented programming (OOP) techniques were developed for use in general purpose software systems. The coding paradigm for these systems is significantly different than for hardware modeling, thus the OOP techniques do not necessarily permit reuse in the hardware modeling domain. Naïvely grafting structural-modeling techniques to object-oriented languages or vice-versa often makes using preexisting reusable components unnecessarily cumbersome, creating a barrier for reuse that promotes

[1]In the 30th International Symposium on Computer Architecture, out of 37 papers (including some which used no simulation), 23 used this simulation technique

reimplementation that promotes reimplementation.

This report provides a mapping of concepts in OOP languages to analogous concepts in concurrent-structural systems to determine what features will enable component reuse. It also explains how common hardware reuse patterns are facilitated by these features in concurrent structural models.

The remainder of this report is organized as follows. Section 2 gives an overview of concurrent-structural modeling concepts. Section 3 compares object-oriented programming and structural modeling. Section 4 concludes.

2. CONCURRENT-STRUCTURAL MODELING

Concurrent-structural modeling systems 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 interface. Typically, each component defines a set of input and output ports. A sending component sends a value on an output port and this value is received by all instances that have an input port connected to this output port. The important characteristic of this style of composition is the separation between functionality and communication. A component defines a computational relation between its inputs and its outputs, while the communication between components is orthogonally specified by the interconnectivity of the components.

In contrast, traditional programming languages encapsulate computation inside of objects and functions. Two functions communicate through function invocation; the calling function collects arguments, calls the other function, and then processes the return value. In this style of composition, the communication pattern among functions is intermingled with the function computation. While this style of composition is very powerful, it is inappropriate for hardware modeling[5].

In this report we are concerned with *high-level* concurrent-structural modeling of *hardware*. That means that within a component, the use of the traditional function invocation paradigm is not only acceptable, but preferred over low-level (i.e. gate-level) specifications.

3. COMPARISONS TO OBJECT ORIENTED PROGRAMMING

The object-oriented programming (OOP) paradigm eases the development of reusable software components by providing the programmer encapsulation, polymorphism, and hierarchy. These techniques make producing a single reusable component easier, allow easy customization of existing components, and allow the creation of components which are more powerful since they are more generic. In this section, the concurrent-structural analogues of OOP techniquess will be identified. The techniques important for reuse identified in this section can serve as the foundation for the design of a structural specification language that supports component-based reuse.

3.1 Objects and Components

The foundation of the analogy between object-oriented programming and concurrent-structural modeling is the structural analogues of the object, member variables, and member functions. It is obvious that the analogue of objects are components. Components are the building block in the concurrent-structural domain; objects are the building block in OOP systems. Since member variables maintain an object's state, the obvious analogue to an object's member variables is a component's internal state. Identifying the structural analogue for an object's member functions is not as straightforward.

An object's member functions can be roughly split into three categories. First, there is typically a function or set of functions, called constructors, responsible for initializing an object. Second, there is a set of pure functions which return the object's internal state or some value which results from computation on input arguments and the object's internal state. Finally, there are functions which are responsible for updating the object's internal state. A function does not necessarily belong to only one of the three categories; hybrid member functions do certainly exist. In the next few sections, a structural analogue for each of these types of functions will be identified.

3.1.1 Constructors

An object is typically instantiated from a *class* and then initialized through the invocation of the object's constructor. In addition to initializing state, the constructor typically customizes the instance using arguments passed to it. For example, a class that represents a window in a GUI environment may be instantiated with parameters that will define the specific window appearance, modality, title, etc. This customizability of the class enables its implementation to be reusable in a variety of situations rather than being specific only to one purpose.

In order to obtain any reuse in a structural modeling system, components, just like objects, will need to be instantiated from component abstractions. Just like classes, these abstractions will need to be flexible so that component instances may be customized. Clearly, the degree of parameterization possible will directly affect the amount of reuse a component can get. The complexity of defining and using this flexibility, however, will further influence the amount of reuse seen in practice [3].

While the notions of object instantiation and component instantiation are similar, there are differences. During the program execution, many objects are dynamically created and destroyed. In hardware systems, and thus concurrent-structural modeling the set of components that exist during simulation is fixed. Thus, a component's constructor can be run once during model compilation rather than being run once for each simulation. This in turn means that the language in which a component's constructor is written and called can be *different* than the language in which its behavior is implemented. This allows for the creation of *domain-specific* languages to handle object instantiation, customization, and composition.

3.1.2 Pure Functions

A pure member function, one that does not side-effect internal object state, represents a computation path that takes input arguments and internal state and computes the function's return value. Objects typically contain many member functions and a client may call none, some, or all of the functions. If certain object behavior is not needed, there is no obligation to invoke a particular member function. Conversely, if a computation is needed multiple times, a member function may be invoked multiple times even with different input arguments.

Unlike an object, a hardware component does not perform computation on demand. Instead the component is constantly computing all of its outputs concurrently. The component reacts to value changes on its input ports and responds by altering values on its output ports. It is logical to compare an object's pure functions with the computation paths that exist inside of a hardware component model. In this comparison, function arguments are replaced with values on input ports and return values with values on output ports.

Notice that there is a loss in flexibility when consuming a components computation in the concurrent-structural domain. First, all the computation paths of the component are constantly enabled and thus, there is a responsibility to provide the component with all its inputs during each clock cycle. Additionally, a particular computation cannot be invoked twice during the same cycle since the values on the input ports must be fixed for the whole cycle.

To obtain reusability similar to that obtained in object-oriented programming, a concurrent-structural modeling system must remove these limitations on flexibility. The modeling system needs to introduce concepts analoguous to not invoking a function and to invoking a function multiple times with distinct input arguments.

3.1.3 State-Updating Functions

Many member functions are not pure functions like those described in the previous section. Instead, these functions alter the internal object state and thus the sequence in which they are invoked relative to all other function invocations is significant.

Conversely, in a hardware component, due to the system's concurrency, there typically does not exist a well defined order between the computations occurring in a single cycle. To avoid race conditions, all state updates are typically synchronized to the rising or falling edge of a clock signal. Thus, we can compare state-updating functions to those computation paths whose outputs are sampled at the rising or falling edge of a clock. This behavior can be encapsulated in a state element component (e.g. a component which represents a buffer or a queue) or can be internal to a more complex component (e.g. a branch predictor or cache memory).

3.2 Polymorphism

Member functions in object oriented languages, and functions in general in certain sequential programming languages, may be polymorphic. The degree of polymorphism allowed varies, but there are three general flavors: parametric polymorphism, function overloading, and subtyping. The next few sections will discuss the applicability of each of these types of polymorphism to the concurrent structural domain.

3.2.1 Parametric Polymorphism

Parametric polymorphism allows the definition of a class or function to be abstract with respect to some set of types. The class or function abstractions (sometimes called *templates* or *generics*) can be instantiated with a set of concrete types to make a concrete class or function. This polymorphism is particularly useful when defining abstract data type objects such as lists, sets, hash maps, etc. The behavior of these objects is type neutral and thus allowing the type stored in the object to be specified parametrically greatly increases their reusability.

Similarly, in structural modeling, certain components' behaviors are type neutral, and thus parametric polymorphism can greatly increase the reuse these objects will experience. For example, queues, memories, and routing components are typically type neutral. Forcing reimplementation of the components for various data types is a significant, unnecessary burden that can be avoided by a structural modeling language that supports parametric polymorphism on component state, computation, and interfaces.

While increasing the reusability of components, parametric polymorphism may also cause programs to become cluttered with type instantiations. To avoid this problem, many programming languages employ *type inference* to automatically infer the types with which functions must be instantiated. This convenience, which allows parametric polymorphism to be used with no additional overhead, is particularly important in strucutral modeling due to the potentially large number of parametrically polymorphic components.

3.2.2 Function Overloading

A second type of polymorphism used in object-oriented programming languages is function overloading. With parametric polymorphism, a single function definition could be used with arguments of varying types. While this type of polymorphism is useful in many cases, it is often necessary to tailor the computation of a function based on the types of its arguments. Function overloading allows a single function to be defined with multiple signatures and bodies, thus allowing custom behavior based on the types of arguments.

In the structural domain, the natural extension of overloading is to allow a component to define various type signatures for its ports, and to implement different behaviors based on the types actually used. Just as in the object-oriented case, this can improve reusability by increasing the components applicability.

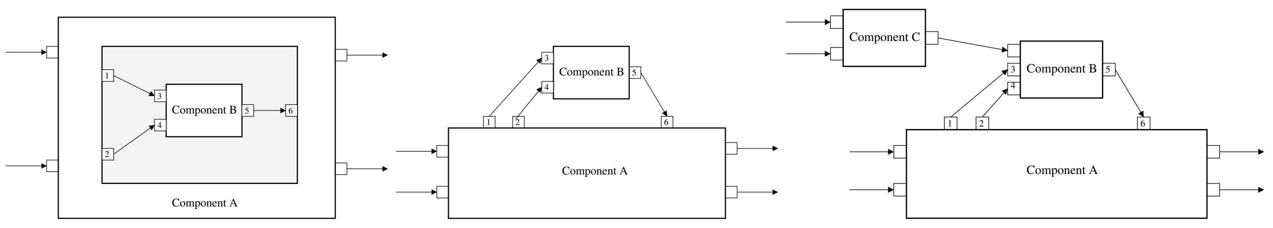
3.2.3 Subtyping

The third type of polymorphism available in object-oriented programming languages is subtyping. Subtyping allows a value of one type to be used where a value of another type is required, provided that first type is a subtype of the latter. Thus in function invocation, rather than supplying arguments of the types specified in the function signature, one could alternatively provide data whose types are subtypes of the arguments' types.

The language's subtype relation defines which types are subtypes of which other types. While some portion of the subtype relation is fixed by the programming language, the relation can be extended via subclassing. Subclassing will be discussed in section 3.3.2, however its relation to subtyping will be discussed here.

In structural modeling, it is natural to allow subtyping when making connections between components' ports. Note, however, that when leveraging subtyping in this way, the augmented subtype relation is irrelevant. Components are not being sent along connections between components, and thus, only the language defined subtyping relation is relevant.

In object-oriented programming, the use of subtyping is not restricted to computation function invocation, but can also be used when invoking an object's constructor. When an object is passed to another object's constructor, typically, the receiving object stores the passed object in a member variable, and the member functions act as clients of the passed object. The subtype relation guarantees that the passed object will conform to a specific interface, however, since the actual type of the argument can vary, the behavior the passed object provides is polymorphic.



(a) Subtyping in Structural Composition

(b) Subtyping in Structural Composition (redrawn)

(c) Modified Interface

Figure 1: Using Structural Composition in Place of Subtyping

Subtyping could be used in this way in concurrent-structural modeling, however, alternative techniques exist to achieve more general reusability. The effect of passing one component to another component's constructor is shown in Figure 1(a). The passed component (component B in the figure) is used to define the receiving component's (component A in the figure) behavior. In object-oriented programming, it is necessary to pass objects to other objects because computation must be actively requested from the object which will perform it. In concurrent-structural modeling, computation is not demand driven but is constantly occurring. Thus, provided that a client exposes the correct interface to drive a computation and receive its results, it is unnecessary to pass a component into the constructor of another component. Figure 1(b) illustrates this style of composition which is a simple transformation of the original figure. This style of composition is *more* powerful than the subtyping analogue since the interface that a computation provider exposes need not be fixed. Figure 1(c) illustrates an augmented interface where the component providing computation requires more inputs than the client can produce. The two components, however, can still be used together because an auxiliary component can supply the additional inputs.

3.3 Extending and Composing Objects

Object-oriented languages also provide features for easily being able to define complex objects by composing or augmenting the behavior of existing objects. These features significantly improve the reuse of an object by easily allowing it to be extended beyond its initial applications. The next two sections will relate these object-oriented constructs to the concurrent-structural domain.

3.3.1 Object Composition

Object composition is the simplest way to build complex objects. A class is defined that contains other objects and implements its behavior by invoking the functions on the objects which it contains. Just as before, these complex components are parameterizable via arguments to the object's constructor. These arguments can guide the instantiation and parameterization of the contained objects and can additionally influence how the object will orchestrate the contained objects' functions to define its own behavior.

Analogously, in the concurrent-structural modeling domain, components may be composed to form hierarchical components. In this style of component definition (shown in Figure 2), rather than providing behavioral functions which describe the various computation paths of the component, the component's behavior is implemented by instantiating other components and connecting them to each other as well as this component's input and output ports. Thus, the interconnectivity of the contained components determines the hierarchical component's behavior.

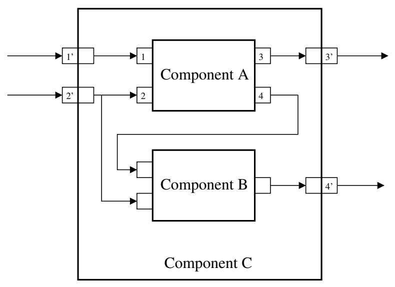


Figure 2: Component Composition

Just as in the object-oriented case, it is important to retain the ability to parameterize hierarchical components. To allow these components to be truly flexible, a structural modeling system must not only allow component parameters to be passed to contained components, but also allow parameters to guide the instantiation and connectivity of contained components.

3.3.2 Inheritance

The second mechanism provided by object oriented programming languages to define complex objects is inheritance. This technique allows an existing component to have its state and behavior augmented without reimplementation of the entire component. Inheritance dramatically increases a class's reusability since its behavior can be customized in ways that were unpredictable to the class's original author.

A similar mechanism should exist in structural modeling to extend the reusability of model components. Components should be able to expose the equivalent of virtual functions to allow computation to be overridden. Further, it should be possible to inherit another component and augment its state and interface definitions.

One way this extension can be achieved is through hierarchical composition. Rather than directly extending a component, the component can be wrapped inside of a hierarchical component. Figure 2 demonstrates how this can be done. Those computation paths which should be inherited from 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. The component's interface can be augmented by adding ports, its behavior modified by adding components along any computation path, and its state augmented by adding additional components. The figure shows component C overriding the behavior of component A's output port 4 by inserting component B between component A's output and component C's corresponding output.

While this implementation of inheritance is extremely powerful, it is not always sufficient or desirable to use. First, forcing all inheritance to occur structurally can cause an explosion in the number of components and connections in the system. Figure 2 required six connections just to override the behavior of only one output port. A mechanism is necessary to allow a component to be inherited and its behavior and state augmented using a convenient imperative or functional syntax rather than structural composition.

Additionally, performing inheritance through wrapping only allows overriding computation which is exposed via the component's ports. Internal computation and state update functions, however, can not be overridden or augmented. Thus, for complete inheritance, it is necessary that a component be able to export an interface allowing users to override internal and intermediate calculations.

In practice, one-off inheritance is very common in hardware modeling. For example, it is extremely convenient to have a single component handle arbitration with inheritance used to customize the arbitration policy. Since arbitration policies are often very specific to a particular situation, the inherited component will probably only be instantiated once and *not* be useful in other designs. A similar situation occurs when writing event handlers to process Java AWT events. In Java, it is common to use anonymous inner classes which inherit from event adapters to reduce the syntactic overhead of full-fledged inheritance and to prevent namespace clutter. A structural modeling language needs a similar mechanism to eliminate any overhead associated with inheritance.

3.4 Aspect-Oriented Programming

Despite the success of the object-oriented programming in producing reusable code, certain kinds of reuse patterns are still hard to achieve. Specifically, the inheritance mechanism, while allowing a user to augment the behavior of one object, does little to help the user augment system-wide behavior or implement behavior which crosses object boundaries. For example adding logging behavior to an object-oriented system would typically involve modifying every object in the system.

Another programming methodology, aspect-oriented programming [1], alleviates this burden by allowing code designed for a specific purpose (e.g. logging) to be grouped together and have the code be automatically weaved together with the base program in order to implement the complete desired behavior.

In hardware modeling, gathering statistics about a simulation is a cross-cutting concern. It involves almost all the components in the system and depending on the statistics desired, the code necessary to gather the data can vary dramatically. Thus, incorporating code to gather statistics is very similar to inserting code into objects to handle logging. For a component to be reusable and a user to have control over the statistics gathered. An aspect-oriented approach to data collection is necessary and *should* be supported by a hardware modeling language.

4. CONCLUSION

Object-Oriented Programming (OOP) languages are designed to facilitate component based reuse in traditional programming languages. Our analysis shows that simply grafting a structural modeling language on top of an OOP language (or vice-versa) does not allow the same level of component based reuse in hardware modeling systems as it does in traditional software systems.

In this report, we examined OOP features and mapped them to the concurrent-structural hardware modeling system domain to identify the language techniques needed to enable component based reuse in hardware modeling systems. The techniques described only need to concern themselves with instantiation, interconnection, parameterization, computation customization, and composition. Thus, they can be used in a wide range of different modeling systems.

5. REFERENCES

[1] 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.

[2] Open SystemC Initiative (OSCI). *Functional Specification for SystemC 2.0*, 2001. http://www.systemc.org.

[3] Putzke-Roming, W., Radetzki, M., and Nebel, W. Objective VHDL: Hardware reuse by means of object oriented modeling. http://eis.informatik.uni-oldneburg.de/research/request.html.

[4] Swamy, S., Molin, A., and Convot, B. OO-VHDL Object-Oriented Extensions to VHDL. *IEEE Computer* (October 1995).

[5] 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.