Interoperability as a Design Issue in C++ Based Modeling Environments

Frederic Doucet
Center for Embedded Computer Systems
University of California
Irvine, CA 92697-3425
doucet@ics.uci.edu

Rajesh Gupta
Center for Embedded Computer Systems
University of California
Irvine, CA 92697-3425
gupta@uci.edu

Masato Otsuka
FUJITSU Ltd
Kawasaki, Japan
otsuka.masato@jp.fujitsu.com

Patrick Schaumont
Departement of Electrical Engineering
University of California
Los Angeles, CA 90095-1594
schaum@ee.ucla.edu

Sandeep Shukla
Center for Embedded Computer Systems
University of California
Irvine, CA 92697-3425
shshukla@ics.uci.edu

ABSTRACT

The increasing heterogeneity and complexity of VLSI systems has made the use of C++ popular for building simulation and synthesis models at higher levels of abstraction. Currently, there are several different embodiments of C++ based environments, mostly in the form of hardware modeling libraries built on top of C++. However, the *semantic gap* between hardware modeling concepts, and the software programming language constructs, poses several issues which require critical examination. In this paper, we address the issue of *interoperability* between models built using different C++ based modeling libraries, or even modeling "styles" including home-grown C++ models. Model interoperability is the ability to use C++ based descriptions across different C++ based modeling environments. Two important aspects of interoperability are model *composability*, and model *reusability*. In this paper we focus on model reusability, analyzing various dimensions of the reusability of C++ based models, in an integration environment for building SOC models. We show how an inheritance based composition may be used to make two distinct C++ based class libraries interoperate. We also outline the implementation of a dynamic composition environment, which allows automatic *run-time delegation* based composition, to achieve interoperability. These strategies allow system integrators to focus on design composition, rather than software programming details inherent in the current inheritance based solutions.

1. INTRODUCTION

System architects and verification engineers build C++ models of hardware systems, and testbenches, for architectural exploration, fast prototyping, hardware/software co-design, etc. C++ provides them with increased functionality/abstraction over other languages. Often these C++ based models need to express hardware concepts such as concurrency, structural hierarchy, and data types. The C++ syntax to express these hardware concepts vary from one environment to the other. In the transition of C++ from a software programming language to a language for high level modeling of micro-electronic systems various artifacts are introduced into the language, often in the form of library elements to express such hardware concepts. This is one of the major advantage of an extensible language like C++, but this also gives rise to the issue of semantic mismatch of the hardware concepts, and software programming artifacts. For instance, consider a hardware module with a 7-bit output, and another module with 7-bit input. From a hardware semantics perspective, they should be composable along their interfaces (provided the timing requirements are satisfied). However, in C++, 7 bits might be represented differently in the different modeling libraries. (For example, bit vector, or a parameterized class for a collection of n-bits.) To compose these two models, additional programming, or other modifications of the source code may be needed. The problem in this example is actually an easier one to solve than other problems germane in the semantic gap, such as differing models of concurrency in two modeling libraries, interfacing along combinational paths, differing functional accuracy in the models, differing timing accuracy in the models, and so on. Some of these problems might dictate imposition of methodological and notational requirements to high-level modeling styles using C++. We will not concern ourselves with those in this paper. For a treatment of methodological issues, the reader is referred to [5] [6].

The *interoperability* in the context of C++ models, means the ability to use C++ descriptions across different C++ based modeling environments. These may include interfacing of standard library based hardware models to legacy C++ code. Two major aspects of interoperability, are model *composability*, and model *reusability*. We want to construct models of a system, using components modeled in different libraries or different versions of the same library. However, we want the models to compose without having to change the models themselves. One can argue that composability, and reusability are the two sides of the same coin. Com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.
Copyright 2001 ACM 1-58113-418-5/01/0010 ...$5.00.

posability can be thought of as the problem of composing existing component models with maximal reuse of the components. Consequently, reusability is the main focus of our work. We often use the terms 'composability', and 'reusability' interchangeably in this paper, not to mean, that they are the same, but with the understanding that our objective is to achieve composability of C++ models, with maximum reuse of existing modules. Reusability is a major requirement due to the increased expectation of model integration in C++ (including *home grown* C++ models), and also due to extensive flexibility, and therefore diversity in modeling using C++ based libraries.

When building larger models by assembling smaller component models, the major technical challenges for achieving composability are: (1) Differing models of computation in the different models; (2) Differing levels of abstraction in the different models; (3) Different data types to represent the interfaces; (4) C++ being too flexible a language, the same hardware concepts can be programmed in completely different ways in the different models; (5) Differences in the notions and implementation of concurrency between hardware and software. Clearly not all these problems are related to the semantic gap problem. In fact, one major source of problems is differing models of computation in the models, which has been previously addressed extensively in the context of Ptolemy [14, 6, 7]. In this paper, we focus on the problems related to the semantic gap, because interoperability across differing models of computation, is not specific to C++ based modeling, and is present in most system modeling languages. This problem is important since software engineering and programming issues are becoming the problem of the hardware engineer when he/she uses C++ for modeling hardware systems. This also poses a difficult human resources challenge which is beyond the scope of this paper. However, ideally, the solutions to the problem of reusability should be such, that the hardware designers can focus on the hardware semantics, and design issues, rather than struggling with programming issues, and hacks in order to make a composed system work.

2. C++ BASED DESIGN ENVIRONMENTS

A number of C++ based micro-electronic high level modeling approaches have been proposed and discussed extensively in [10, 9, 12, 8, 13, 19, 20, 18]. C++ as a general syntax is often too complicated for a hardware designer. As a result, some environments are so designed that the models are expressed in a reduced syntax (like e.g. Cyn++ from Cynapps [19]) or a graphical one (like VCC from Cadence [18]). A preprocessor then translates the restricted syntax to C++, which is then linked with a modeling library. Consequently, a C++ design environment can have different embodiments that make it seem different from the traditional software development compile-link cycle. We want to stress however that this difference is mainly dependent on where one sees the interaction with the end-user. The core of the design environment can be a C++ modeling library even though the user writes a specialized language. Therefore, the interoperability problem extends into those environments as well, but rather from a tool API standardization perspective. Some examples are given in Table 1. This list is far from exhaustive and only targets to show that C++ is beginning to be used in a wide variety of environments.

Name	Origin	Embodiment
SystemC	OSCI	C++ library
Cynlib	Cynapps	C++ library
OCAPI	IMEC	C++ library
Cruise	Conexant	In-house C++ system simulation environment
Cyn++	Cynapps	Macro package on top of Cynlib
VCC	Cadence	Graphical notation using underlying C++ objects
SpecC	UC Irvine	Dedicated language compiled into C++ based simulation
ART/Library	Frontier Design	C++ Library (focus on datatypes)
Testbuilder	Cadence	C++ Library (focus on testbenches)

Table 1: C++ Based Design Environments

2.1 Modeling Dimensions

The complexity of the design modeling (and consequently the complexity of the C++ programs) quickly becomes clear by considering a system level design modeling taxonomy. The VSI SLD Taxonomy [15] is a good guideline to this. It distinguishes four orthogonal design model characteristics.

1. *Temporal detail*: which expresses the degree of precision of the ordering of the modeled events. This includes partial-ordered event accurate models, token-cycle accurate models, instruction-cycle accurate models, clock-cycle accurate models, clock-phase accurate models, and so on.

2. *Data value detail*: which expresses the representation or format of data values specified in a model. Data values could be enumerated values, word-level values, bit-true representations, etc.

3. *Functional detail*: which expresses the level of detail in the functionality of a model, ranging from mathematical formulae to detailed intermediate operations (gate-level or instruction level).

4. *Structural detail*: which expresses the level of detail in the structure of a model, ranging from single-block code to multiple levels.

While these are theoretically orthogonal design axes, in practice they are never described or explored separately. Instead, designers often use well-defined combinations of modeling precision to express a *model of computation*. For example, DSP hardware might be expressed using the data-flow model of computation, which expresses partial ordering (temporal level), actor-primitive code (functional level), flow-graphs (structural level). An executable simulation always fixes a "working point" on each of the four axes. Thus, if we want to simulate the dataflow computational model (that fixes temporal detail, functional detail and structural detail), we will also have to make a decision on the data value detail of the simulation by choosing the data type of the tokens.

2.2 Mapping of Modeling Dimensions

When a design model is constructed in C++, a supporting design model library is often used. This library contains a number of classes that allow expression of different modeling dimensions.

2.2.1 Temporal Detail

The standard C++ language is executed sequentially, with no notion of time. However, concurrency and parallelism are fundamental in hardware models. A modeling environment can explicitly

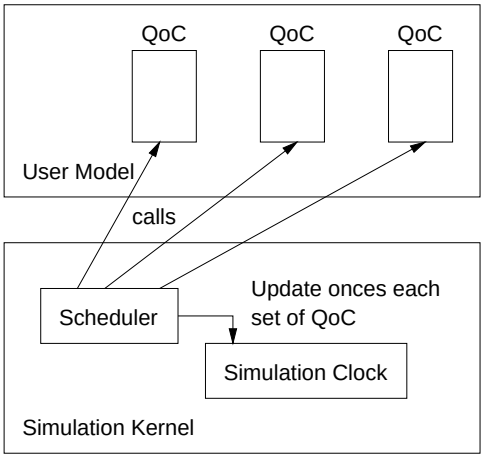

Figure 1: Generic Model of Concurrency in C++ Modeling Libraries

reference the time or else make it implicit through simulation cycles. Therefore, the introduction of concurrency and time characteristics always require specific classes and modeling support. The approaches used in C++ models are very often a derived form of Figure 1. The modeling library provides a *scheduler* that manages the simulation clock. The scheduler also decides when computation in the user model should be executed. The user model itself consists of several pieces of behavior called *Quantum of Computation*. A QoC is all activity in a user model that can happen in one step, without interaction with the simulation model environment. Concurrency is simulated in this model by calling several different QoC before updating the simulation clock. Thus, to the observer of the model, it looks like time has not advanced while several different QoC are executing: their activity seems concurrent. Cycle simulation is a frequently implemented time model in current C++ libraries and is often extended with the delta-cycle concept from HDL simulation. It works as follows. The QoC in the user model describe one cycle of processing. The QoC communicate through state variables or direct connections. Each state variable has a present- and a next-value. The overall simulation works in a two-phase approach. During the evaluate phase (phase 1), the execution of a QoC reads the present-value of the state variables and writes the next-value. During the update phase, next-value in state variables is copied to the present-value, and the simulation time is advanced. Additionally, the evaluate-phase might be driven by a value-change sensing mechanism (upon QoC inputs) to support relaxation-types of simulation (called the *delta cycle* in HDL simulation). This style of temporal modeling is reasonably simple, but yet many different approaches can be used in implementing it in C++. The actual implementation of a scheduler, of the simulation clock and the QoC can be done in a variety of ways: native execution or interpreter based, call-back mechanisms, multiple threads, etc.

While a detailed discussion of each of those approaches is beyond the scope here, it should be clear that there are can be as many implementations of the evaluate/update mechanism as there are C++ design environments. For example, different libraries can use different thread packages [4], and synchronization mechanisms. By itself, the synchronization interoperability can be very complex if it introduces many ordering protocols. Before two C++ environments can be interconnected, it must be clear how they implement the temporal axis in a model. The simulation detail (level of abstraction) should match, but in addition the implementation approach (API) must also match.

2.2.2 Data Value Detail

One of the big advantages in using C++ is the ability to introduce new data types (and classes). Most C++ design environments offer a wide spectrum of data representation and detail. Typically included are various types of scalar values with specific quantization characteristics. However, this kind of flexibility adversely affects interoperability. A common base of information representation is needed in order to transfer information from one environment to the other. There are three approaches in modeling data values.

1. Direct use of *built-in types*. C++ has a wide range of built-in types. Those values might be applicable for the model. In addition, the C++ built-in data types are interoperable by themselves.

2. We can use a *class model* of the actual representation of data values. For example, a bitvector can be modeled as an array of bits. This array can be stored as a private member of a class.

3. Sometimes an *abstracted representation of data values, with operational behavior* is used. For example, the operations on a bitvector (of limited word-length) might perfectly be done in terms of the built-in type "int" if we are only interested in the aggregate data value of the bitvector. Similarly, fixed point DSP calculations can perfectly be done using floating point C++ built-in types. In both cases, we need to take quantization effects (overflow and rounding) into account in the functional description.

The interoperability of (scalar) data values has been more successful than the other axes of modeling dimensions. A VSIA datatype standard [16] is under development as a reference for data values. Data types for which C++ libraries are available include 2-value and 4-value bits and bitvectors, arbitrary sized signed and unsigned integers, and fixed point values.

The interoperability of different C++ design environments implies that different data types can be cast across environments without loss of precision. If the target environment has no direct support for a particular datatype (for example, fixed point), then two situations can occur. First, there can be a data type in the target environment that has sufficient range. In that case a "converter stub" is needed to translate data values between the two environments. Second, the target environment can lack a type that has sufficient range. In that case, the target environment must be extended with an "emulator" that supports this data type. But, the interoperability problems can occur when this is to be specified, and checked. There is two approaches that can be used: (1) static type checks, and replacements in the compilation phase (2) dynamic conversions at runtime through stubs. Type systems have been used to infer types and provide some type looseness in programming languages such as ML or Ptolemy. As we discuss later in this paper, our solution is based on a run-time type inference and *delayed type instantiation*[2].

2.2.3 Functional Detail

The third axis of detail is the functional one. It expresses the precision of the model functionality. Each executable description implements the behavior of a model of computation.

The C++ Semantics: The C++ language uses a sequential consistency for correctness and composition. The language is designed for the implementation of software programs. The usage of classes

makes C++ very convenient to capture real world structures as object in the software environment. The C++ language has no formal semantics, and is very flexible and extensible. In that sense, the usage of the syntactic constructs can be different from the original semantic intent. For example, the usage of exceptions is recommended for handling software error condition, but it has been used to model interrupt behavior in hardware systems. When different C++ environments are matched to each other, they will need similar styles of software design at their matching points.

Syntactic variations with C++: Modeling flexibility inherent in C++ often creates additional programming overhead for the IC designer. Consider, for instance, modeling of state machines, which of course, have a well understood semantic model. A common technique is to use a nested switch with a scalar state variable, or in more complex cases sparse arrays of actions and transitions for each state, with actions as function pointers. In object oriented modeling states are often treated as subclasses with common interface defined by a super class. Events are modeled as methods in the interface, and a separate "evaluate-class" is used to send events for processing to the current state object. Therefore, a state transition changes the current state object. Here inheritance is used for state addition to the model (by subclassing), and for adding events (by adding new functions to the subclass, or overwriting virtual methods of the super class). In order to execute the model correctly, the state machine needs to be integrated with other objects in the environment. This introduces *context dependencies* from the FSM to other software components, reducing the reusability. In other words, execution of the FSM model relies on the existence of other objects which are not a part of the FSM model. As a result, to use the FSM Model in different environments, one has to program the environment objects in the new environment again.

2.2.4 Structural Detail

The final modeling axis relates to structural detail. All modeling environments have the concept of a "block" that encapsulates other blocks or leaf behavior. In addition, the blocks recognize port and/or port lists that provide handles to express communication. The structural detail in a model allows relating it to an implementation view. Structural interoperability means that one environment is able to encapsulate a block from another environment.

However, structure can also include *design patterns* [3], which rely on C++ programming techniques such as polymorphism and component substitutability [11]. Organizing a C++ model at this level of abstraction provides a higher abstraction and flexibility than the entity-port-entity connection models. An open problem in this approach is how should these patterns be used in C++ hardware system modeling, and their impact on interoperability since the discrete event semantics rely on port-signal-port-process semantics.

3. INTEROPERABILITY STRATEGIES

Interoperability is often achieved, by creating *wrappers* around the existing C++ descriptions that allows communication of data values between different modules and co-ordination between them. However, wrappers can be implemented in many ways. The first and most common strategy is by using inheritance. In this approach, the wrapper is programmed by manually inserting code to align various design axes inside the inherited class. The component and the wrapper have a *common self* in this implementation [11]; i.e. the wrapper and the component are the same object. As a result, the the interoperability issues related to typing are resolved at compile time, and the wrapper and component have strong dependencies.

An alternative is to use wrappers that, if needed, *delegates* to the design component. In this case, the component is not modified, and the wrapper and the component are two distinct objects. Modules from different libraries can be imported as is, and dynamically placed in wrappers at runtime. Let us review both of these strategies in more details.

3.1 Interoperability with Inheritance

An example of inheritance based composition is shown in Figure 2. It shows how two models drawn from different C++ based libraries (in this case, Ocapi and Cynlib), can be composed in a single executable model. In Figure 2, the system shows a dataflow example where the modules are a mix of Cynlib and Ocapi-1 blocks. The system consists of four actors (active modules), three of which will be under the control of the Cynlib simulator, and one under control of the OCAPI simulator. For simplicity, all communications and data types are also taken from the Ocapi domain.

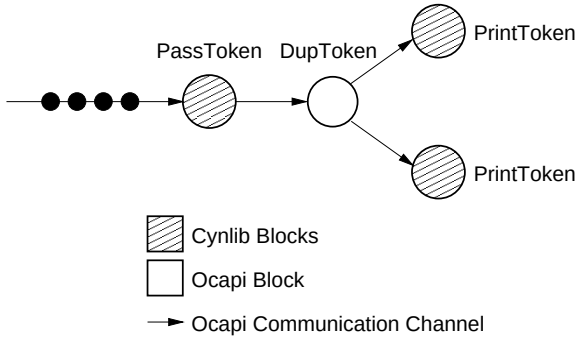

Figure 2: Example for Interoperability

As seen in the main function below, there is a main scheduling loop that calls the Cynlib and Ocapi schedulers in alternating fashion, until all dataflow tokens are consumed out of the system.

```
void main() {
  dfbfix Q1("queue1");   /* OCAPI data type */
  dfbfix Q2("queue2");
  dfbfix Q3("queue3");
  dfbfix Q4("queue4");
  PassToken2  P1(Q1, Q2);     // Cynlib block
  DupToken    P2(Q2, Q3, Q4); // Ocapi block
  PrintToken  P3(Q3);         // Cynlib block
  PrintToken  P4(Q4);         // Cynlib block
  sysgen ocapisystem("ocapisystem");  /* OCAPI */
  clk ocapiclk;  /* OCAPI clock */
  ocapisystem << P2;
  Q1.put(0.1);
  Q1.put(0.2);
  Q1.put(0.3);
  Q1.put(0.4);
  while ( Q1.getSize() > 0 ||
          Q2.getSize() > 0 ||
          Q3.getSize() > 0 ||
          Q4.getSize() > 0) {
    ocapisystem.run(ocapiclk);
    /* simulates one ocapi clock */
    CynTick();
    /* simulates one cynlib clock */
  }
  CynFinish(); /* clean up for cynlib */
}
```

In this example, PassToken, and PrintToken modules are derived from the `CynModule` class from Cynlib, and suitable code insertion has made it capable of accepting OCAPI datatypes such as `dfbfix`, while the DupToken inherits from the `base` OCAPI class for processes. Due to lack of space, we do not show the code, and necessary programming intervention into the modules from both domains, but is is available online [13]. Interoperability is achieved by making the two modeling environments match on the four dimensions by programming new classes via inheritance from the respective reactive class interfaces in OCAPI and Cynlib, and implementing call backs explicitly for the two simulation kernels as follows

1. On the temporal axis, the (untimed, partially ordered) dataflow model of computation was used. This is required to express the cycle simulation semantics of Cynlib in terms of a polling firing-rule check (DF_actor object). Note that because queues are used instead of signals, no delta events were exchanged between the two simulators. The simulation kernels cannot exchange delta events at this time because the event format is not interoperable, and the eval/update calls are not accessible to the programmer.
2. On the data value axis, interoperability was implicit since we used only the OCAPI-1 data type dfix for all calculations.
3. On the functional axis, we organized the processing of a token in a method, which was executed as native code. The method itself was slaved to a system scheduler (either Cynlib or Ocapi) that implemented dataflow semantics.
4. On the structural axis, the same approach was used as on the functional axis (since we did not consider synthesis of this model, only simulation).

While inheritance can quickly achieve interoperability in some cases, it is not a recommended approach for many reasons. For one, it may actually hinder reusability in the long term. Figure

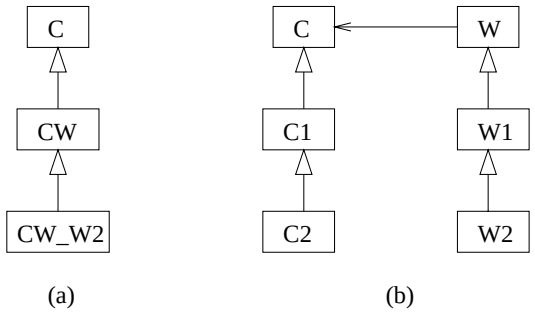

Figure 3: Wrapper implementation strategies: (a) by inheritance (b) by composition

3(a) illustrates the UML class diagram of the typical problem of inheritance based composition (the *common-self* [11] problem). If a designer wants to reuse a component of class `C`, the class can be specialized by inheritance to a subclass `CW` to implement the wrapper functionality. Let us suppose that the behavior of the original class `C` is modified in the class `CW` by adding more functionality or by re-implementing a virtual function. If the class `CW` is to be reused in a different context, then it can be also inherited into a class `CW_W2` that implements more wrapper code to interoperate in the new integration context. The problem in this scenario is that all the three classes have a common self, and the original component has to be modified in every reuse context, via inheritance. Also, substituting an object of class `C` with a `CW` (which is legal in C++) may introduce subtle side effects. For instance, the wrapper code may modify the state of the component in ways that may not be obvious to the designer. If in a reuse environment, the original component being reused is unaltered, then it keeps its identity distinct from the identity of the wrapper that contains the code for the interoperability. Figure 3(b) shows the UML diagram of how a wrapper hierarchy can be built for composition (the open arrow indicates an association). In this case, the wrappers are separate from the component object hierarchy, and the interoperability interface remains separated in the wrappers, and any call to functionality of the original component is delegated from the wrapper to the component.

3.2 Interoperability with Dynamic Composition

To implement design interoperability and reuse with dynamic composition, wrappers can be individually implemented and used for each component. However, this is very difficult to perform on a large scale. A strategy to automate the interoperability is to use a wrapper generator. In our experiments, we built the BALBOA system [2] for dynamic composition. This environment is designed to relieve the system integrator from manually generating wrappers to align data-type detail axis, as well as the other axes of interoperability. This enables the system integrator to work with hardware semantics of components in their mind, and to compose components from a library, which makes sense from a hardware design stand point.

Our wrapper generator in BALBOA is a compiler, that parses component descriptions described in a special interface definition language (called BALBOA IDL or BIDL), and automatically builds smart customized wrappers. The use of BIDL is optional for the system designer since the description are automatically generated by the library builder. In addition to wrapper generation, the system also provides appropriate type determination in the context in which an object is used. Our dynamic typing mechanism, implements a delayed typing useful to compose objects since in some cases, in order to compose hardware and software objects, the designer may not know the exact object types used by the C++ libraries to implement the model. Our environment, upon getting a request for composition, enables the designer to figure out such details without having to go through the source code, and can rely on the tool to instantiate the types correctly for interoperability of the components. Note that it can also fail to instantiate the composition request due to incompatibilities in the designers choices.

BALBOA provides an interpretive command shell to instantiate and compose objects. The wrappers are visible from the user command shell. This visibility enable designers to manipulate the components by writing composition scripts. A composition language, called *Component Integration Language* is used to control the wrapper layer. In the current implementation, the interpreter is built using an extended OTcl interpreter. We call the wrappers *Split Level Interfaces*. This is because they are the link between the compiled C++ domain and the interpreted composition domain. Figure 4 shows the runtime view of the object interactions in the environment. The split level interfaces are in gray and execute composition commands. The simulation control flow is in the compiled hierarchy, from the discrete event simulator (DES) to the components C1, C2 and C3. Whenever possible, usually the interpreter is not involved in the simulation loop for efficiency reasons.

The split level interface allows *introspection*, or *reflection* [1], of the attributes of the component models from the component library layer to the command interpreter layer. As a result, the designer

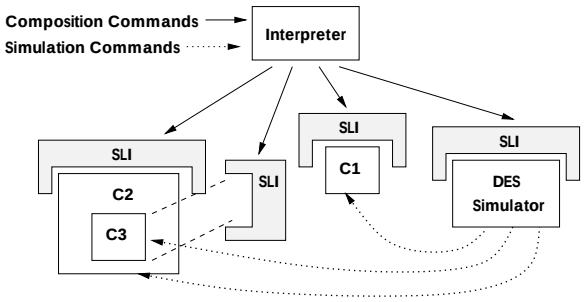

Figure 4: Layered composition with Split Level Interface wrappers (in gray) around design and tool components

or the tool can query a component for its characteristics (such as structural information, type, connectivity, attributes, methods), allowing dynamic composition decisions to be made. In BIDL, the component's exportable interface is described, along with certain other meta-level information about the component. For example, abstraction levels can be specified (behavioral, structural). The split level interface inheritance hierarchy, such as in Figure 3(b), is specified in this language. The BIDL compiler generates the C++ code for the split level interfaces, and the code is compiled into a library and loaded in the environment. The implementation of such a "smart" composition environment is quite challenging. However, this requires little (if any) changes in the existing C++ library environments.

Another possible approach to achieve interoperability is through interfacing the different simulators at the simulation run-time. The distinction from the approach just described, and this approach, is that in the first case, the interoperability is designed into the composed system at the environment run time, and in the latter case, it is achieved at simulation run-time. In the currently available C++ simulation libraries, there is no programming interface to the simulation kernels to make two distinct simulation kernels exchange delta cycle events, or synchronize the increment of the simulation clock. However, we can implement, or change existing simulation kernels to provide such APIs. If the purpose of system integration is to build a complete system simulation, this could be done by building a simulator by composing single or multiple simulation kernels through these APIs.

4. DISCUSSION AND FUTURE WORK

System level modeling and integration of components, using C++ models of micro-electronic systems can be decomposed in four different dimensions. Doing so, the problem of interoperability is divided in smaller chunks, such that each might be easier to conquer. In particular, the VSIA standards on data types [16] already have addressed the problem in the data value details axis. In this paper, we elaborate on the interoperability problem, in terms of composability, and reusability of existing models in an integration environment. We have presented the various solution strategies, and how they are implemented in C++. However, the extendibility of C++, and the sequential model of execution inherent in C++, lead to simulation and structural semantic gaps when composing models together. This is because of the possible different implementations of a component, and the use of inheritance to build wrappers around components that implement the function needed for the interoperability. We presented a solution for model interoperability using composition based techniques. We then extended our solution to create a dynamic composition environment that relies on composition and delegation. We also discussed the automatic wrapper generation, configured by a specific language to capture hardware semantics. Further experiments for matching different modeling dimensions and specifically their mapping to C++ are under way as a part of our ongoing research on C++ based hardware/system modeling issues [17].

5. ACKNOWLEDGMENTS

We gratefully acknowledge the support for this research from DARPA/ITO under contract DABT63-98-C-004, the National Science Foundation (NSF), the Semiconductor Research Corporation (SRC) and the Fond pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR).

6. REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[2] F. Doucet, M. Otsuka, R. Gupta, and S. Shukla. Efficient system level co-design environment for split-level programming. Technical Report TR-01-34, CECS, Univ. of California, Irvine, 2001.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] P. Garg, S. Shukla, and R. Gupta. Efficient usage of concurrency models in an object-oriented co-design framework. In *DATE*, 2001.

[5] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs*. Kluwer Academic Publishers, 1998.

[6] E. A. Lee. What's ahead for embedded software. *IEEE Computer*, September 2000.

[7] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD*, December 1998.

[8] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity in modeling hardware in the scenic synthesis and simulation environment. In *DAC*, 1997.

[9] G. D. Michelli. Hardware synthesis from C/C++ models. In *DATE*, 1999.

[10] L. Semeria and A. Ghosh. Methodology for hardware/ software co-verification in C/C++. In *HLDVT*, 1999.

[11] C. Szyperski. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[12] C. Weiler, U. Kebschull, and W. Rosenstiel. C++ base classes for specification, simulation and partitioning of a hardware/software system. In *CS Workshop on VLSI*, 1995.

[13] IEEE/DATC C++ Modeling Standardization Effort home page: http://www.ics.uci.edu/∼rgupta/datc/.

[14] Ptolemy 2 project, UC Berkeley, home page: http://ptolemy.eecs.berkeley.edu/.

[15] VSI system level design taxonomy, v1.0, vsi alliance, http://www.vsi.org/library/specs/summary.htm#sldtax.

[16] VSIA data type standard, under development.

[17] BALBOA Project home page: http://www.ics.uci.edu/ balboa.

[18] Cadence VCC home page: http://www.cadence.com/eda_solutions/hcd_l3_index.html.

[19] CynApps home page: http://www.cynapps.com.

[20] SystemC home page: http://www.systemc.org.