System-on-Chip Specification and Modeling Using C++:

Challenges and Opportunities

The recent push toward C++-based modeling of IC and system designs—either through libraries like SystemC, Cynlib, or OCAPI or through abstractions like SpecC—has renewed the long-running debate on the design community's appropriate choice of a modeling language. The focus of this panel was on the technical issues facing development and adoption of C++-based modeling approaches, the new capabilities the user community can realistically expect, and the difficulties that remain.

IEEE Design & Test thanks roundtable participants Rahul Bhatt (Intel), Giovanni De Micheli (Stanford Univ.), Daniel D. Gajski (Univ. of California, Irvine), Christopher K. Lennard (Cadence), Stan Liao (Synopsys), John Sanguinetti (CynApps), and Patrick Schaumont (IMEC). *D&T* gratefully acknowledges the help of Rajesh K. Gupta (Univ. of California, Irvine), our moderator; and Kaushik Roy (Purdue Univ.), our Roundtables Editor, who organized the event. Special thanks go to the IEEE Computer Society Design Automation Technical Council and the IEEE Circuits and Systems Society CANDE Technical Committee, for sponsoring the roundtable.

D&T: What makes C++-based design different from existing techniques?

Sanguinetti: As designs get more complex, system architects model in C to iteratively refine from an architecture model to a hardware implementation. C++ is the best, perhaps only, language that we have between the starting and, ultimately, the ending point. C++ offers a unified environment for architects, verification engineers, and implementation engineers.

Schaumont: We're using C++ because, first, C++ gives us object-oriented design technology, which is more powerful than whatever has been done till now. The second reason has to do with reusability: C++ supports better reusable design description. Finally, C++ gives us a single environment, which means we can model at multiple levels of abstraction.

Bhatt: C++ lets us handle legacy concisely. The object-oriented concept doesn't really matter. When I need to get a job done, the embedded aspect of C++ is very important. C++ also simplifies hiring. I don't need special vendor-provided training, so it's easier for a person to become productive in my environment. Third, and most important, I don't need a vendor-provided compiler—I can use my own—to compile the very custom language solution. This simplification when using C++ lets us do all kinds of other things more easily than if we were to create a language by ourselves.

Liao: In addition to the gradual refinement and the multiple levels of abstraction, one of the many advantages is in the area of test benches, which the system designer can write or has written in C or C++. If we need to rewrite our models in an HDL, we would have to do a lot more work to adapt the test benches into our simulation environment. So by using C++, we can very easily reuse the test benches.

Lennard: Object orientation allows us to

Sanguinetti: Object orientation is the key; C++ is just the environment to define objects that can introduce different abstraction levels.

implement true system refinement because we're able to instantiate the way that objects communicate with one another through classes and methods. This lets us develop rapid-substitution methods for the reuse of IP components and models. The key is that the language of design implementation is not critical; what's critical is how we build an environment to enable reuse based on object orientation. I agree that C++ makes it easy to hire and train people, yes. People understand what C++ syntax is. But the semantics that underlie the execution of any C++-based system language depend on how we're going to structure our entire environment around C++. So the fact that we are using C++ doesn't give us an answer; it has to do with how we architect the entire object environment. That's much more critical than the language itself.

De Micheli: Designers have been using C and C++ for hardware design for many years. There is nothing new about that. What's new is the possibility of having synthesis and verification tools that apply directly to the C/C++ descriptions. Research and development emphasis in EDA should focus on this direction.

D&T: What abstraction levels of design can be done with C++? Are we talking about a system-level view of things or more of a gate level design?

Sanguinetti: Object orientation is the key; C++ is just the environment to define objects that can introduce different abstraction levels. The idea of building complex systems in a top-down, iterative refinement method using layers of abstraction is at least 30 years old and has been done in software for a long time. What we've started to do recently is create formal layers of abstraction in C++. Over the past year or so, a couple commercial or quasi-commercial offerings have created a hardware abstraction layer in C++—essentially, an RTL layer. But because C++ has this capability of creating objects that can build on other objects—building layers of abstraction by extension—there's really no limit to the level of abstraction we can represent.

Schaumont: I agree; whatever is an executable level, we can do in C++. We start from a performance model in which all architecture artifacts are abstracted into simulation time. Next, we incrementally refine this model. We go down over all models of computation that we want to work with, like data flow, RT, and even down to continuous-time modeling if we want to cope with analog aspects.

Gajski: I'll have to disagree. When I ask companies for design examples, they give me C, never C++, examples. The reason is that every processor in the world has a C compiler, so everybody would like to do C. However, C cannot represent hardware concepts like parallelism. Therefore, C++ is the quick choice, since we can extend it by adding classes. But let's not deceive ourselves that adding classes is just another C++ extension. We have to learn the meaning of every class and what it's going to do in our design. Therefore, we're creating a new language—we would like to call it C++, but C++ with classes for hardware is essentially a new language.

Bhatt: Learning is definitely involved. Especially when we bring in a new library, we have to make sure we have the training, and it doesn't solve our primary problem—we still don't have parallelism in our designs by any means. As a result of using C++, there's a fair amount of complexity, partly due to performance, when we're running it for complex machines, and it's been hard to achieve some of the research that we've done. I've been working with object orientation for a long time, and it still doesn't solve the whole problem.

Liao: My answer to the question of what abstract levels of design can be done in C++ is, at whatever level we wish. We can have mixed levels of design, different parts of design at different levels. And we can also model our environment—even do analog design in C++.

Gajski: A language that models everything from requirements to transistors would be too complicated, too big, and too difficult to implement, maintain, and train. Below the RTL level, we need electrical engineering expertise: timing, currents voltages, and so on. So I'd say that C++ abstractions should be above the RTL level, and that we shouldn't go below the RTL level since we already have Verilog, VHDL, and extensions to VHDL. Therefore, we do not need another language.

De Micheli: To some extent, the C/C++ representation would represent a sign-off level between system designers and component designers. It should not be a hard boundary; we want some flexibility because different design styles, design methodologies, and different applications may be of interest.

D&T: We're saying we want more high-level abstraction but not a language that does everything. What are we looking for with C++? Is there a need for object orientation in modeling when we talk about IC system design?

Sanguinetti: There's a need for object orientation in hardware design, certainly. We don't usually think of it, but Verilog and VHDL are both object-oriented languages; they're just pretty weak ones. C++ is a much richer object-oriented language, but one of the reasons we find C++ attractive is because it matches so well to the way we think about hardware.

D&T: What does C++ buy us? It does buy ease in developing tools; what other things does it buy?

Lennard: The real issue is how we architect the entire design-refinement flow with a class hierarchy, from a functional specification down to a hardware instantiation. At the hardware level, we can get rid of the object hierarchy by flattening it and representing it in a VHDL or Verilog-like manner. But at higher levels of abstraction, what should the class hierarchies represent? Allowing arbitrary class hierarchies would make C++ very difficult to reuse.

D&T [Gupta]: We're saying we want more high-level abstraction but not a language that does everything. What are we looking for in C++?

Schaumont: With respect to reuse, the hardware design community has been focused for many years on the structural form of reuse. Object orientation allows exactly the next step beyond the structural view. Software engineering communities have already figured out how to work with combinations of objects. They talk about design patterns—which is actually about documenting the way we do or reuse things. We can use exactly the same design patterns for documenting the way we design a system, which makes it much more reusable. One improvement that we can get from object-oriented design is reuse at the behavioral level.

Lennard: In terms of reuse, consider platform-based design techniques. The platform will fix how system components communicate with one another, but then there's the issue of how we predict communication performance. In theory, the platform-based design concept lets us percolate statistics, in terms of delay and congestion estimates, up to the top levels of abstraction to get a view of how the platform structure is going to perform before we use it. But, to date, there are no clear strategies on how to manage these performance statistics in an object hierarchy.

Gajski: The object-oriented approach provides encapsulation for a concept or a component.

Liao:
Perhaps one day the hardware community won't care so much about performance, but will face the reality of time to market….

However, if every company or designer puts slightly different information into that encapsulation, the whole thing isn't worth much. Therefore, we must first agree on what information is included in these classes.

Liao: The hardware community has focused, understandably, on implementation details to extract the last bit of performance out of a design. The object-orientation features we've used so far are structure and instantiation and not much else, not abstraction. Perhaps one day the hardware community won't care so much about performance, but will face the realities of time to market and be compelled to apply software techniques. Then they could use more abstraction. For example, we might model and deal with a protocol at an abstract level, and there might be different implementations with different trade-offs for that protocol. And we can just plug in the different implementations to evaluate the trade-off. Someday, that might be done.

D&T: Do you think that designers will choose to give up performance in favor of gains in design productivity?

Bhatt: I think we'll want to compromise. We're already at the point where we can compromise if the complexity that we've reached right now results from a specification that's so general that a certain implementation might be different. We're finding a lot of complexity and issues, like time to market. I think we're not compromising on complexity but on time to market. We must recognize, regardless of how much we'd like object orientation, that with extremely large, complex systems, if we don't have a predefined means of communication or framework within which all these C++ libraries are supposed to operate, we'll have serious problems. One little tweak will break the whole definition. And then we can't reuse the class library, and we'll have to rewrite some portion; every tool does. All of us, in our tools, provide a capability to go outside the framework and then come back. So the framework is as important. If we look at the changes that have occurred in terms of MFC [Microsoft Foundation Classes], it's a very good example of how MFC has gone to COM and to ActiveX instead of just relying on class reuse.

Sanguinetti: The single greatest source of intellectual property for reusability is algorithms written in C. There are far more algorithms available for almost anything than what's available in C++ classes or Verilog or VHDL. When we talk about design reuse, I think we get a little parochial in the EDA world. We just don't do much design reuse in any real sense.

D&T: Even with Verilog and VHDL, high-level synthesis is not yet widely used. Isn't C++ going to make synthesis even more difficult? Is it a step backward or does it actually put us a step forward in synthesis?

De Micheli: Looking at synthesis from the C/C++ perspective, we find new challenges. C/C++ hardware models have a higher abstraction level. For example, we find different data types in C/C++ that don't exist in VHDL, Verilog HDL, or lower-level languages. The market currently offers only partial synthesis tools, making the C/C++ approach less attractive to the designer than it should. Designers want C/C++ with a full synthesis capability. Some research tools have shown that it is possible to cope with the advanced features of the languages, such as resolving pointers and handling dynamic memory allocation in hardware. But we still need to see these ideas percolate into commercial tools. Clearly, the expressive power of a hardware modeling language relates to the hardware synthesis capability.

Lennard: But how far do we need synthesis to go; are we trying to synthesize protocol tasks, for example? Those could be "synthesized" into software, but then we're changing the conventional meaning of synthesis. Are we only talking about synthesis into hardware? If we're talking about synthesis into hardware, then we have to restrict designers from writing objects that use recursion. So how critical is the concept of recursion, for example, at becoming part of synthesis capabilities? Is synthesis going to be something we push for in the software domain, as well as in hardware?

Schaumont:
In the past 10 years of high-level synthesis, people have concluded that, basically, high-level synthesis doesn't work at the system level.

De Micheli: One basic issue is migration between software and hardware models. Designers don't want to rewrite models (from C to Verilog/VHDL), because this is an error-prone, time-consuming task. We cannot compile Verilog/VHDL into software, and so using these languages prevents easy migration from hardware to software.

Liao: As an implementer of synthesis tools, I struggled with the trade-off between how much I could synthesize and the quality of results—automatic synthesis will only go as far as the designer is willing to accept the quality of results. The larger the subset of C++ we need to support in synthesis, the worse the quality of results we can expect. There's a limit to C++ synthesis in terms of practicality; beyond that we'll probably look for domain-specific solutions that will improve productivity. Using C++ solely is probably not the best path for EDA providers.

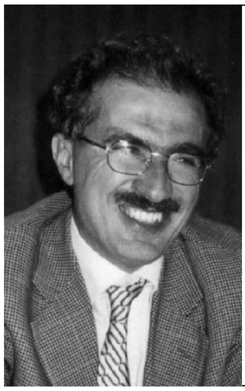
Gajski: C is a software model of computation that executes everything sequentially on a single processor. C and C++ do not deal well with the concept of parallelism. However, every hardware processor is parallel and pipelined. Therefore, we've got something that's sequential in software that we have to turn into something very parallel, with new concepts that software people can't even comprehend, like pipelines. I don't believe that we can really synthesize from C; we have to introduce hardware concepts by extending C or adding appropriate classes to C++.

Bhatt: It's hard to make a system that can't be synthesized yet still be sellable. I'm not saying we can't impose constraints, however, whether it's on C++ or libraries. We would do ourselves an injustice if we limited ourselves by not permitting certain kinds of abstractions for the capability of nonsynthesizable systems. I think the best bet would be to achieve synthesis with some restrictions that we propose.

De Micheli: I agree; we need C++ classes in order to define semantics and synthesis. A complete suite of classes is something we'll achieve over time. I don't think that C++ design will be a stable solution for electronic system design if we don't define precisely the semantics, synthesizeability, and verifiability of the whole language. If we limit ourselves to subsets of the language, we will find ourselves with several dialects. This will defeat the purpose of finding a unifying language.

Schaumont: In the past 10 years of high-level synthesis, people have concluded that, basically, high-level synthesis doesn't work at the system level. This is why we need platform-based design. Synthesis, however, works very well for local solutions. Software designers have already figured out how to deal with this. Large software systems are not synthesized; they are constructed as assemblies of objects and components. This also holds for C++-based design: We glue a system together with fixed communication artifacts but will use automatic compilation locally within a component.

Lennard: There'll always be the case where we have things that are not necessarily synthesizable. But that does not reduce the value of the language. Engineers nowadays spend about 70



De Micheli:
If you are a user, you love standards. If you are a developer, you hate them.

to 80 percent of their time in verification. So here's a question: If I'm going to build an abstract model, where am I going to put its expense? The model's development cost must be absorbed into one of two areas: either into the design cost—in other words, that model should be synthesizable or usable in the software implementation—or into the development cost of the test bench, which is not a synthesizable component. So if we build a fully executable model and we can show that that executable model is "golden," we can associate the development of the abstract models that aren't synthesizable into the cost of verification development. That's where we'll see that we don't have to synthesize everything.

Bhatt: That's exactly why I said we shouldn't compromise all other abilities to the abstraction simply for synthesis. I've seen enough difficulties result from golden reference models. But as a community, we should develop usage models of things we can abstract. This is where "golden reference" is very important. Yes, synthesis is very important for certain things, but let's not shortchange our level of abstraction ability.

D&T: Let's talk about standardization. How much standardization is needed for tool developers, how much for designers?

Sanguinetti: Standardization is clearly important. There's a lot to be gained by looking at different ways of doing things before we make a decision that this is what's going to be the "standard." The good thing about the situation we're in with C++ is that because we're essentially talking about class libraries that implement layers of abstraction, we can agree on the semantics of the abstraction layer. And competing implementations for doing that, if we have them, won't be all that far apart. We can agree on the semantics fairly easily. Eventually, we'll have standards for several different layers.

Schaumont: I think we don't need standardization to build a good tool. We need standardization because we want to exchange models. The whole issue here is actually not a tool issue; it's an IC issue.

Gajski: Simulation is a weaker concept than synthesis and verification. As long as it's syntactically correct and produces correct results, anybody can write a simulation model and do simulation. To synthesize and verify, we have to understand what that model means. Therefore, synthesis and verification imposes a much stronger set of requirements on the language, on the tools, and everything else. The only way to solve this problem is to have some fixed levels of abstraction with well-defined semantics, so that everybody knows what we mean when we write $a + b$. Syntax is not enough. Look at VHDL: It's a simulation language and it's not synthesizable. And what we did after 10 years of messing around: We have a synthesizable VHDL subset. To avoid this mess again, let's define a synthesizable language and not expand it. Then we'll have one language that's synthesizable, verifiable, and simulatable.

Lennard: Some things in standardization I've found quite frustrating, such as when one little group of people disagrees with another: rather than building to an agreement, they go off and start their own little initiatives. Then the problem is, what's a standard? A standard isn't a standard unless everyone agrees. At least to start talking of standardization, we need to agree on taxonomy, terminology, and semantics.

De Micheli: If you are a user, you love standards. If you are a developer, you hate them. Why? Because if you represent any research group or any EDA developer, freedom from standards gives you a competitive advantage. Do we currently have a standard for synthesis from

C/C++? I'm afraid that at this stage of development, it is still too early to talk about standards.

D&T: We've said synthesis is hard, and so is verification, compared to cobbling together a model that we can execute that gives us the right results at the right time. Surely we can relate to that. Really, the path to synthesis is not clear. Now supposing we're trying to do a standardization. What should our user community's mind-set be?

Bhatt: From the viewpoint of what layers of abstraction we require for standardization, we can't determine that now, but I don't think we should give up for lack of trying. We could make an attempt—say yes, we know this isn't what the final answer will be, but we should try something. I think we could agree on a very coarse level of standardization that will enable people to stay within their own frameworks and deliver together, which is just a simple clocking mechanism. We should agree on one clocking mechanism. Between the parallel world and the serial world, there's only one difference, and that's the clocking. I think we should be able to agree on that, and then go forward.

Gajski: From a historical perspective, standardization always takes flexibility and fun out of the design. But, it moves this industry forward to the next generation of methodology and tools.

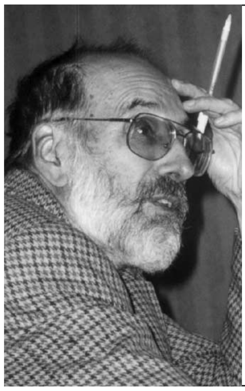
Sanguinetti: Standardization should happen from the bottom up. We can probably agree on some simple things—for example, clocking—at the lowest abstraction level. Then we can move up to the next level. As we go higher, we have less experience to draw on, so it will take longer to do something that's universally accepted as useful. Although we have enough experience to agree on some things now, I think that the standardization process is going to take a long time, because this effort is pretty new and there just isn't that much real use out there. It's often commented that the effective standards are de facto standards first, and then they're ratified. We're not in that situation yet.

Bhatt:
Between the parallel world and the serial world, there's only one difference, and that's the clocking.

Lennard: The need for communication between objects is the primary driver that makes standards effective. For example, competitors in the cell-phone market will agree that to grow the market, they all need their products to communicate with each other. So these companies agree that there has to be a standard. The second major reason for standards is an impacted market, one where everybody already possesses a fairly fixed market share. EDA is a primary example of such a market. In these markets, companies will often try to maintain market share through holding of proprietary formats. But this limits the customers, like Intel or Nokia, who will then try to drive the EDA companies, like Synopsys and Cadence, to standardize on formats so that they have access to a broader selection of tools and services. Market forces such as these are going to determine any form of standardization in the C++ space.

D&T: Does anybody have a closing remark on anything we've discussed?

Sanguinetti: C++ is clearly an advance from where we are today with Verilog and VHDL. We see how to produce good environments that are layered, that give us plenty of head room for environments having a hierarchy of levels of abstraction. We can provide the environment that allows system design from an executable spec through iterative refinement to include all of the major parts of systems design and implementation from specification to verification to implementation. This is coming, and standardization will follow. We're living in a relatively immature world right now, but we see the technology that will give us real capability to support this rich environment.



Gajski:
Each company cannot have its own semantics, modeling style, guidelines, layers of abstraction, and so on.

Schaumont: We should not forget what C++ will actually give us. In the world of designers, 95% have a software background and 5% have a hardware background. In the long term, it's far easier to bring hardware designers to work and think like software designers than vice versa. Platforms, also the ones in the SoC sense, are a good step in the right direction. A platform is nothing more than a service abstraction layer, onto which a designer can implement a service with more ease. A platform in C++ is nothing more than a set of objects that presents an easy-to-use design model.

Gajski: Each company cannot have its own semantics, modeling style, guidelines, layers of abstraction, and so on. We've got to work on some standards. The easiest way to do that is to define a minimal set of orthogonal concepts that will support software and hardware in simulation, synthesis, and verification. We should start with semantics, define no more than two or three levels of abstraction, then proceed with syntax. Syntax isn't so important because if two things are semantically the same, then translation from one syntax to another is a piece of cake. This is the only way we can move forward.

Bhatt: Commitment to standardization is the only way we'll succeed as a community. A second point is that there are so many auxiliary capabilities that I can think of that people can spawn off in terms of businesses that are going to be required on top of this, it's just phenomenal. We're fortunate we got to hear from the EDA side of things. Specifically looking at HLM, please do not forget there's a wide variety of uses for this model. It's not necessarily to make a chip. It's sometimes to look at the variety of implementations, to look at a "golden reference" model, to look at things that we don't even possess. We still want to come out at a time to market at the same time our competition does. So there are many different reasons why just being able to synthesize is not important.

About the participants

Rahul Bhatt is a senior engineer at Intel Corporation, Santa Clara, California.

Giovanni De Micheli is a professor of electrical engineering and, by courtesy, of computer science at Stanford University, Stanford, California.

Daniel D. Gajski is a professor of computer science and director of the Center for Embedded Computer Systems at the University of California, Irvine.

Christopher K. Lennard is an architect in the Emerging Businesses business unit, Cadence Design Systems, in San Jose, California, and chair of SLD DWG of VSIA.

Stan Liao is a principal engineer, Advanced Technology Group, at Synopsys Inc., Mountain View, California.

John Sanguinetti is the founder of CynApps in Santa Clara, California.

Patrick Schaumont is a senior research engineer, Design Technology for Integrated Information and Communication Systems Division, IMEC, Leuven, Belgium.

Rajesh K. Gupta, our moderator, is an associate professor of information and computer science in the Center for Embedded Computer Systems at the University of California, Irvine.

Kaushik Roy is a professor of electrical and computer engineering at Purdue University, West Lafayette, Indiana.

Lennard: Most important, we must be careful how we define the architectures in object-oriented design.

Liao: C++ is going to be a viable language for some system design but won't by itself be sufficient. Just as the software world is moving toward CASE with even higher levels of abstraction—for example, UML—so people won't just write C++ code. They'll use other tools to integrate different parts of the system. Domain-specific tools such as those for designing DSP and control-dataflow systems will be needed and might be used to automatically generate C++ code. But C++ itself is going to be used as the platform for system-level design.

Lennard: We need to work out how system design is handed off to the software design flow. If we think we're going to change the way that software design is done, we're really barking up the wrong tree. We need standards to aid communication as these system languages develop. Also, verification and its link to system-level languages is critical, because system-level languages will be used for verification before they are used for synthesis. Object orientation is critical to reuse; and C++ will probably be used, but that's not fundamental. Most important, we must be careful how we define the architectures in object-oriented design. Finally, the concept of platforms will become critical, because platforms can offer a set of services and can show, at high levels of abstraction, what those services mean in terms of performance.

D&T: Thank you all very much for your time.