A Survey:
System-on-a-Chip Design and Verification

Ali Habibi and Sofiène Tahar
Electrical & Computer Engineering Department, Concordia University
Montreal, Quebec, Canada
Email: {habibi, tahar}@ece.concordia.ca

Technical Report

January 2003

Abstract. *In this technical report, we survey the state-of-the-art of the design and verification techniques and methodologies the System on-a-Chip (SoC). The advancement in the hardware area made it possible the integration of a complete yet complex system on a single chip. Over 10 million gates, integrated together and running a real time optimized software red crossed classical design techniques. Traditional Regiter Transfer level (RTL) will serve as an assembler language for the new design languages or so called system level languages. A challenge facing the SoC designers is to decide which system level language we have to use and how the verification task will be accomplished. This report presents the main proposals in defining a system level language and discusses the eventual verification techniques that can be used.*

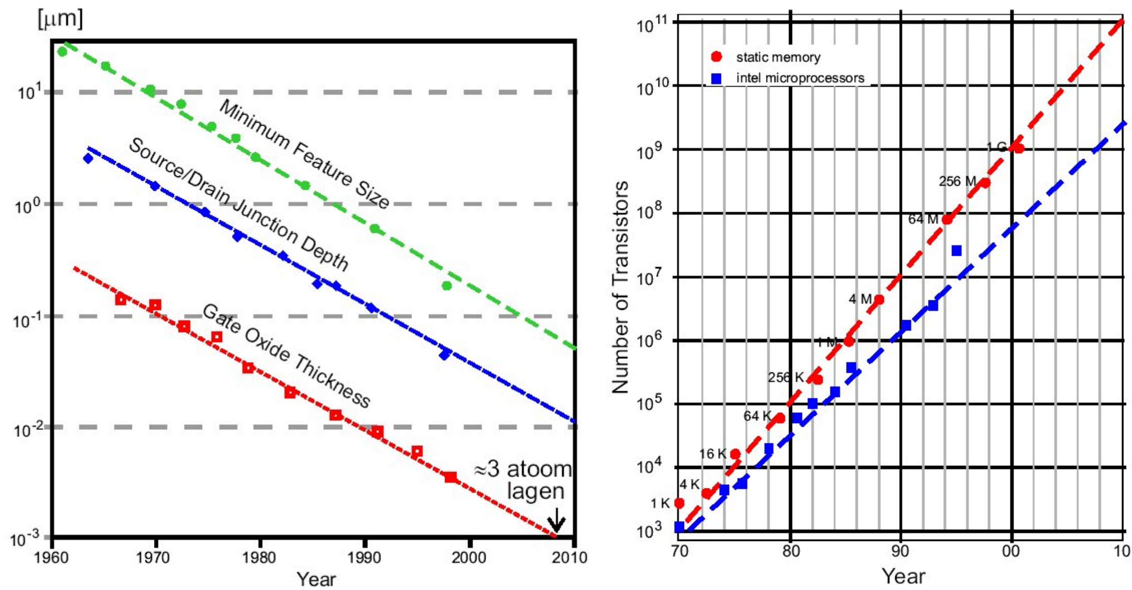
1. Introduction

A decade ago, the EDA industry went progressively from gate level to register-transfer-level abstraction. This is one of the basic reasons why this process gained a great increase in the productivity. Nowadays, an important effort is being spent in order to develop a system level languages and to define new design and verification methodologies and verification at this abstraction level.

The reason for all this activity is simple. Register-transfer level (RTL) hardware design is too low as an abstraction level to start designing multimillion-gate systems (as shown in Figure 1). What is needed is a way to describe an entire system, including embedded software and analog functions, and formalize a set of constraints and requirements - all far beyond the capabilities of existing HDLs. VHDL and Verilog both will become the assembly languages of hardware design. Designers and verifiers will write RTL code for things that are performance-critical, nevertheless, for everything else, they will stop at a higher level.

By looking at the specifications of embedded systems, particularly for communications, portable and multimedia equipment, we can realize an important and rapid growth in complexity making it a requirement System-on-a-Chip (SOC) solutions that generally integrate diverse hardware and software. Time-to-market and cost also need to be reduced more than ever before and backed up by an effective marketing-driven strategy that can

meet today's highly competitive and demanding circumstances. To achieve all this, the product development process must assure the product specification phase is integrated smoothly with the product design phase, allowing the customer's demands, marketing goals and designer expertise, to be evaluated and analyzed at significantly less cost in time and resources, and to be rapidly incorporated into the final product.



(a) The complexity growth. **(b)** The scaling of the dimensions.

Figure 1. Hardware Design Evolution [43].

System level languages proposals can be classified into three main classes. First, reusing classical hardware languages such as extending Verilog to SystemVerilog [30]. Second, readapting software languages and methodologies (C/C++ [44], Java [5], UML [16], etc.). Third, creating new languages specified for system level design (Rosetta [2] for example).

The verification of SoC is a more serious bottelneck in the design cycle. In fact, defining SoC design language and methodology is a matter of time, however, the verification is a very open and ambigous question. Functional verification is consuming an inordinate amount of the design cycle time. Estimates vary, but most analysts and engineers agree that as much as 70 percent of the design cycle is consumed by functional verification. In addition, the quality of these verification efforts has become more important than ever because the latest silicon processes are now accompanied by higher re-spin costs. No doubt, classical random simulation is no more able to handle actual designs. Going further in complexity and considering hardware/software systems will be out of the range of the nowadays used simulation based techniques [32].

For intance, the main trends in defining new verification methodologies are considering a hybrid combination of formal, semi-formal and simulation techniques. The first step in

this direction was the definition of a standard for assertion languages. Such languages, which let engineers check properties during simulation and formal verification, have become crucial for complex chip design. For instance, the Sugar language [4] from IBM was selected as a standard for this effect. A lot is expected from combining an assertion language such as Sugar with both smart test generation and coverage analysis. This kind of hybrid techniques can offer a partial answer to the question: “Is verification task complete?”

This report is divided into three main sections. In Section 2, we introduce the concept of a System Level Language. We presented the UML language as a case study to illustrate the definition of such a class of languages in the software engineering domain. In Section 3, we consider the main proposals regarding the definition of a hardware system level language. In Section 4, we classify the main promising verification techniques for SoC verification. Finally, Section 5 concludes the report.

2. System-Level Design

2.1. Problem Statement

Modern electronic systems grow in complexity encouraging combinations of different types of components: microprocessors, DSPs, memories, etc. On the other side, and in the same time, designers and programmers are asked to meet demands for shorter development time and lower development cost. Designers agreed that there is no way to meet this difficult target expect using higher abstraction levels in the design process, so-called *System Level Design* [45]. System level design, requires system level tools that simultaneously handle both hardware and software, for modeling, partitioning, verification and synthesis of the complete system.

The software evolution story is replayed again in the hardware world. Migrating to the system level of abstraction introduces a higher order of complexity. To reach the next level of complexity, EDA vendors and analysts are telling designers that another leap is necessary - from RTL to system-level design. Such a leap implies that an increasing amount of hardware design will be done using C/C++, Java, SuperLog, or other high-level languages, while RTL VHDL and Verilog will be relegated to smaller blocks of timing-critical logic.

Nevertheless, a fact is the level of highest abstraction will be the one that survives, and that is clearly the software domain. Considering the economic reality, this transition will not be abrupt, but it will occur more as an evolution than a revolution. The most likely transition will be along the lines that software followed as it evolved from a strict use of hand-coded assembler in the fifties to extensive use of compilers in the sixties.

The most realistic scenario will start by migrating the non-critical portions of time-to-market-driven designs to higher levels. Then, progressively over time, more sophisticated compiler and synthesis technology augmented by increasing hardware functionality will extend the reach of automatic techniques until only extremely critical portions of highly performance-driven designs will be implemented at the register transfer level.

Eventually, the software and hardware design will end by getting into a single flow. Optimistically, in a few years, the difference, if it will exist, will be a matter of compiler options ("-software" or "-hardware"). However, to get there, we need at first to define a system level language.

2.2. Proposed Solutions

The usage of a system level language is a direct result of the way the SoC design process works [31]. Following the software evolution, the most likely methodology would be to push toward successive refinement, an extremely successful methodology in software development. A possible solution is to define what are the basic requirements for a system level language; intuitively, we would first spend a look to the way software design is actually performed.

No doubts, a short-term solution will come from languages that "push up" from or extend current HDL-based methodologies. We can not omit that languages like SuperLog [46] hold great promise for the next three to four years basically because SuperLog does not require a radical shift in methodologies and allows groups to retain legacy code. This approach can be seen as a refinement of the existent languages. Some people define SuperLog as “Verilog done right”.

The medium-range solutions will likely be C++-based languages that have hardware design capabilities, we mention here SystemC [49] and Cynlib [10] as two promising languages. The revolution can also come from the Java side.

Long-term language solutions will likely come from new languages developed just specifically to work at the system level such as Rosetta [2] which promise to make true system-level design a reality.

2.3. Case Study: UML as a Software System Level Languages

Software developers faced the problem of large yet complex projects long time before their hardware counterparts. Dedicated system level languages have been introduced since the last 80s. However, the most and widely used solution is the Unified Modeling Language (UML) [8]. No one can deny that fact is that UML is becoming a *lingua franca* for system level modeling. Nevertheless, an open question is whether UML is rich enough to represent hardware adequately.

As defined, UML is a vendor and tool-dependent notation for specifying, visualizing, constructing and documenting software systems. Originally UML was developed by Rational Software Corp. and its partners and now under the auspices of the Object Management Group (OMG) [8]. The first yet principle objective of UML was to let software developers draw diagrams that can be, at least in theory, universally understood. At the time when UML was born, it was quite early to think about the support of hardware systems. However, the SoC facto made no choice for UML designers to rethink about their language if they want to keep it as a "default" system level language.

Most hardware companies are keeping an eye on UML. Trying, to cut the road on the software developers, Cadence as one of the leading EDA companies, for example, is looking at UML as a possible source of input into the Cadence VCC hardware-software co-design system. Moreover, Cadence has joined the OMG as an auditing member and is tracking OMG's real-time analysis and design working group. On the other side, the embedded software companies are also tracking the UML guest. For instance, Project Technology has developed a prototype UML-to-VHDL compiler [16].

In an optimistic way, people supporting UML consider that this system level language is rich enough to represent any system. They go further and say that designing a SoC in UML will be as simple as taking requirements for the entire system and breaking it down into big subsystems, then, the last step will be a simple breaking into hardware, software and mechanical components.

The People in the EDA busniss look at the problem in a more realistic way. They consider that UML needs much more work before it can fully support real-time embedded hardware [21]. A fact is UML can not represent an "architectural" view with such attributes as hierarchy, structure, connectivity or bus width. No doubt this limitation will make it hard to do any real estimation about the use of UML.

Omitting the technical constraints, a fundamental problem facing UML is the question of receptivity. Will hardware engineers move up from block diagrams and schematics to the higher-level abstraction of UML? The first step was done by software people to upgrade UML to support hardware by adding a causal model offering an automatic conversion of requirements into test vectors. However, without a strong collaboration between both hardware and software plyers there is no way for UML to become a default system level language for SoC.

3. System-on-a-Chip System Level Language

3.1. Requirements

Any language proposing to support system-on-a-chip design must address two important design characteristics. Namely, the integration of information from multiple heterogeneous sources and the ability to work at high levels of abstraction with incomplete information [31]. This can be defined as:

- Support for integrating domain specific specifications
- Composition of models of computation that describe systems and components
- Representation and integration of constraint information
- Support for more abstract modeling
- Moving up from implementation technology
- Support for predictive analysis and verification
- Within specific domains
- Across multiple domains

- Formal semantics is a must

- Consistent and continuous design process and exploration of design space from specification design to implementation design

- Design reuse, which means not only of component IP but also specification IP and design case IP.

3.2. Extending Existent HDLs: SystemVerilog

A radically revised Verilog [47] language took shape at the International HDL Conference as presenters unveiled a language reaching toward much higher levels of abstraction. SystemVerilog blends Verilog, C/C++ and Co-Design Automation's SuperLog language to bring unprecedented capabilities to RTL chip designers.

As shown in Figure 2, key additions to SystemVerilog, which is a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language [30] to aid in the creation and verification of abstract architectural level models, include interfaces that allow module connections at a high level of abstraction; C-language constructs such as globals; and an assertion construct that allows property checking. SystemVerilog includes the synthesizable subset of SuperLog, and its assertion capability will likely be derived from the Design Assertion Subset that Co-Design and Real Intent Corp. recently donated to the Accellera standards body.

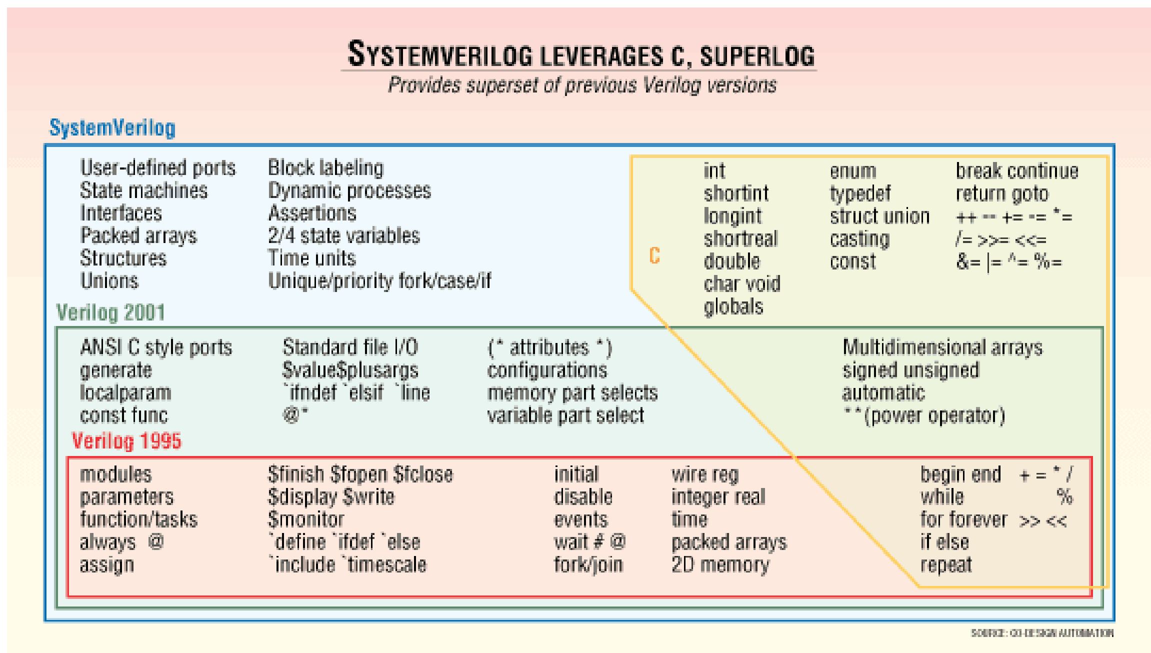


Figure 2. SystemVerilog [27].

With all these enhancements, SystemVerilog may remove some of the impetus from C-language design, at least for register-transfer-level chip designers. The basic intent is to

give Verilog a new level of modeling abstraction, and to extend its capability to verify large designs.

Interfaces are one of the most significant introductions in SystemVerilog. The IEEE 1364-2001 Verilog spec, Verilog-2001, connects one module to another through module ports, but this is tedious. The new "interface" construct makes it possible to begin a design without first establishing all the module interconnections. Usually, interfaces allow the grouping of connections between modules, and can also contain logic. In other terms, it is possible to have procedures and other functionality within the interface itself. Also important is the introduction of global declarations and statements. For now, only module and primitive names can be global in Verilog. SystemVerilog will allow global variables and functions via a new "$root" construct.

Abstract data types are another addition. SystemVerilog borrows the C-language "char" and "int" data types, allowing C/C++ code to be directly used in Verilog models and verification routines. Both are two-state signed variables. Other new constructs include "bit," a two-state unsigned data type, and "logic," a four-state unsigned data type that claims more versatility than the existing "reg" and "net" data types. Other new features include signed and unsigned modifiers, user-defined types, enumerated types, structures and unions, a "jump" statement and parameterizable data types within modules. SystemVerilog also removes any restrictions on module ports, allowing users to pass arrays and structures through ports.

The slogan of the people supporting Verilog is: "The right solution is to extend what works". However, this may face a harsh critisim from people supporting C++ based solutions and who advance that Verilog or SystemVerilog or whatever is the Verilog based name will not offer a short simulation time. The debate is quite hard and the last word will not be said soon!

3.3. Extending Software Languages

Some optimistic software designers are supporting the establishment of C/C++ [25] or even Java [29] based languages for future SoC designs. They think that, over time, improvements in automatic methods and increases in hardware functionality will extend the pieces handled automatically to where whole designs can be implemented by "cc -silicon". In the near term, however, tools are not going to be good enough and manual refinement will be required. This means that the version of the language that will be used must allow for the expression of hardware at lower levels, not just the algorithmic level.

Mainly reusing existent languages is performed either by extending the language itself (by adding new keywords and constructors) or by defining new libraries allowing the description of hardware systems. The most relevant approaches consider either C/C++ or Java. In following, we will put more light on the proposals based on these two languages.

3.3.1. C/C++ Based Proposals

Over the past decade, several different projects have undertaken the task of extending C to support hardware [25], including SpecC [13] at the University of California, Irvin, HardwareC [12], Handel-C [25] at Oxford University (now moved to Embedded Solutions Ltd.), SystemC++ [37] at C Level Design Inc., SystemC [48] at Synopsys Inc., and Cynlib [10] at CynApps.

These variety of projects fall roughly into two complementary categories. One category, exemplified by SpecC, has focused on adding keywords to the basic C language, supporting hardware description at a high level as a basis for synthesis [13]. The other category, exemplified by SystemC, exploits the extensibility of C++ to provide a basic set of hardware primitives that can be easily extended into higher level support [44]. These two complementary approaches span all levels of hardware description (algorithmic, modular, cycle-accurate, and RTL levels).

People supporting this approach argue that the best way to do system-level design is with a general-purpose language and that C++ is clearly the best choice. Because it is extendible and notions such as concurrency can be easily represented in class libraries. Besides, the object-oriented nature of C++ corresponds perfectly to the HDL hierarchy [51].

The art of applying SoCs to real systems is dominated by software. So it is easy to see why C/C++-based design languages are rapidly gaining popularity. Many have mistakenly believed that such languages, like SystemC, aim to replace hardware-description languages, such as Verilog, for pure hardware design at the register transfer level. But that holds very little advantage, and the vast majority of companies backing SystemC, including CoWare, are not trying to replace the Verilog designer.

3.3.1.1. SpecC

The SpecC language is defined as extension of the ANSI-C programming language [15]. This language is a formal notation intended for the specification and design of digital embedded systems including hardware and software. Built on top of the ANSI-C programming language, the SpecC language supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling and timing [14].

Originally, the SpecC language was developed at University of California, Irvine, with sponsorship from Toshiba, Hitachi and other companies [13]. Based on C, it allows the same semantics and syntax to be used to represent specifications in a conceptual system, hardware, software, and, most importantly, intermediate specification and information during hardware/software co-design stages.

SpecC targets the specification and design of SoCs or embedded systems including software and hardware whether using fixed platforms, integrating systems from different IPs, or synthesizing system blocks from programming or hardware description languages [24]. The SpecC methodology leads designers from an executable specification to an RTL implementation through a well-defined sequence of steps. Each model is described and

guidelines are given for generating these models from executable specifications. SpecC is intended to be used for specification analysis, specification capturing and system co-design as described in Figure 3.

To defend SpecC against C++ proposals and mainly SystemC, when this latter was first introduced in 1999, SpecC supporters claim that SystemC is primarily aimed at simulation, however, SpecC was developed with synthesis and verification in mind. They also considered that SystemC targets RTL design, but SpecC is a system-level design language intended for specification and architectural modeling. Nevertheless, after the release of the version 2.0 of SystemC, all these argument were broken. In fact, SystemC is nowadays supporting all System Level Design requirements.

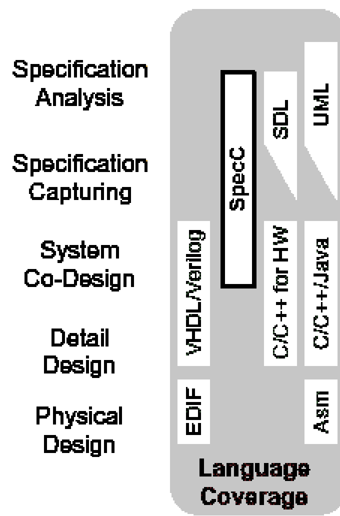


Figure 3. SpecC in the Design Flow[14].

3.3.1.2. HardwareC

Under the same class of C-based languages we find also a language called HardwareC [34]. This is a language that uniformly incorporates both functionality and design constraints. A HardwareC description is synthesized and optimized by the Hercules and Hebe system [11], where tradeoffs are made in producing an implementation satisfying the timing and resource constraints that the user has imposed on the design. The resulting implementation is in terms of an interconnection of logic and registers described in a format called the Structural Logic Intermediate Format [33].

HardwareC attempts to satisfy the requirements stated above. As its name suggests, it is based on the syntax of the C programming language. The language has its own hardware semantics, and differs from the C programming language in many respects. In particular, numerous enhancements are made to increase the expressive power of the language, as well as to facilitate hardware description [34]. The major features of HardwareC are the following:

- Both procedural and declarative semantics - Designs can be described in HardwareC either as a sequence of operations and/or as a structural interconnection of components.
- Processes and interprocess communication - HardwareC models hardware as a set of concurrent processes that interact with each other through either port passing or message passing mechanisms. This permits the modeling of concurrency at the functional level.
- Unbound and bound calls - An unbound call invokes the functionality corresponding to a called model. In addition, HardwareC supports bound call that allows the designer to constrain the implementation by explicitly specifying the particular instance of the called model used to implement the call, i.e. bind the call to an instance.
- Template models - HardwareC allows a single description to be used for a group of similar behaviors through the use of template models. A template model is a model that takes in addition to its formal parameters one or more integer parameters, e.g. an adder template that describes all adders of any given size.
- Varying degrees of parallelism - For imperative semantic models, HardwareC offers the designer the ability to adjust the degree of parallelism in a given design through the use of sequential ([ ]), data-parallel (f g), and parallel (< >) groupings of operations.
- Constraint specification - Timing constraints are supported through tagging of statements, where lower and upper bounds are imposed on the time separation between the tags. Resource constraints limit the number and binding of operations to resources in the final implementation.

As any real system a SoC is no more than a number of entities and objects interacting together. This is the reason for the limitation of the C-based proposals. By reference to software design, languages that can be used at the system level have to be preferably object-oriented. This is the reason why nowadays more mature proposals are based on C++. In this report we will discuss mainly two C++ based proposals: Cynlib and SystemC.

3.3.1.3. Cynlib

Cynlib provides the vocabulary for hardware modeling in C++. It is a set of C++ classes which implement many of the features found in the Verilog and VHDL hardware description languages. It is considered as a "Verilog-dialect" of C++, but it is more correct to say that it is a class library that implements many of the Verilog semantic features. The purpose of this library is to create a C++ environment in which both hardware and testing environment can be modeled and simulated [10].

Cynlib supports the development of hardware in a C/C++ environment. To do this, Cynlib extends the capabilities of C/C++ by supplying the following features, Concurrent Execution Model, Cycle-Based, Modules, Ports, Threads, Deferred Assignments, Two-state data values, Multiple clocks and Debugging support.

By using Cynlib to model at the highest levels of abstraction and throughout the design process, any problems (bugs) with the algorithm or implementation will get fixed as early as possible. For example, it is much easier to isolate, debug and fix a protocol error at the algorithmic level than it is at the register-transfer level. In the worst case, fixing the RTL could be as time-consuming as adding or removing states from the finite state machine and reworking the data path. Of course, this is assuming that the bug can be easily identified as a protocol error in the first place.

Another advantage of design with Cynlib and Cynthesizer [22] is that designers can use the object-oriented features of C++ to produce a more natural and understandable representation of the design. The most obvious application is the use of C++ classes as high-level data structures. Using high-level data structures where appropriate provides a less error-prone approach to design. Cynthesizer's ability to transform object-oriented features of C++ (such as classes) to Verilog or VHDL allows designers to maintain the usage of these features throughout the design process.

Finally, adopting C++ as a design language promotes a unified flow between hardware and software designers. Most system and algorithm developers have been working in C++ for years. By using Cynlib C++ to describe hardware, we get into a situation where the algorithm, the software, the hardware and the testbench are all written in the same established language. The complete Forte chain is illustrated in Figure 4.

.

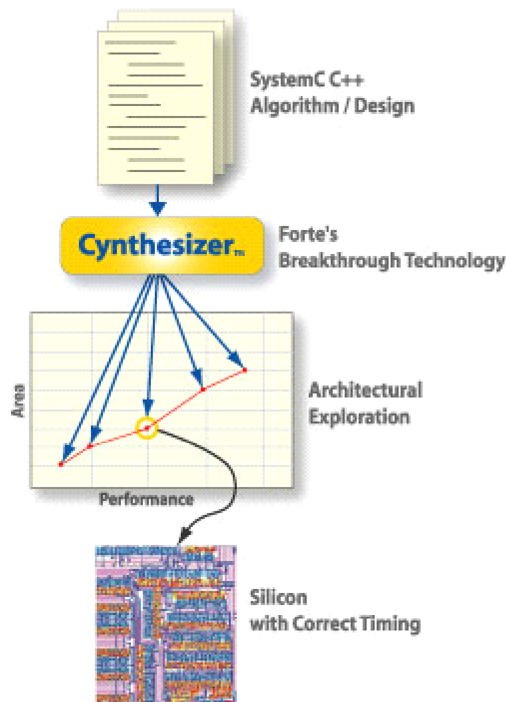


Figure 4. Cynapp Proposal [22].

3.3.1.4. SystemC

SystemC [49] is also among a group of design languages and extensions being proposed to raise the abstraction level for hardware verification. It is expacted to make a stronger effect in the area of architecture, the co-design and integration of hardware and software [41]. The reason lies partly in the deliberate choice to extend a software language with nonproprietary extensions to describe hardware and system-level modeling. However, SystemC is merely a language, so a greater part of the reason for any success in this area is the availability of supporting tools and methodologies proved to cut design cycles at such leading companies as Fujitsu, Matsushita, Sony and STMicroelectronics.

SystemC comes to fill a gap between traditional HDLs and software-development methods based on C/C++. Developed and managed by leading EDA and electronics companies, SystemC comprises C++ class libraries and a simulation kernel used for creating behavioral- and register-transfer-level designs. Combined with commercial synthesis tools, SystemC can provide the common development environment needed to support software engineers working in C/C++ and hardware engineers working in HDLs such as Verilog or VHDL.

Designers have started to create virtual platforms of their SoC designs in C or SystemC for good reasons. First, they can provide an early simulation platform for the system integration that executes much faster than HDL-based co-verification. Second, they can verify the architecture at an early stage of the design process, testing different hardware-software partitions and evaluating the effects of architectural decisions before implementation. After all, by the time they are coding in RTL, the hardware architecture is established and there is little room for flexibility. Third, they can start work on derivative or next-generation designs by reusing the platform before the first-generation design is complete [48].

Simulating in C or SystemC avoids the need to use the Verilog Program Language Interface (PLI) [47] or VHDL Foreign Language Interface (FLI), both of which are used to hook C-based models or testbenches to HDL simulations. By converting to C, a single executable code can be used hence there is no need for a PLI or FLI, which becomes real bottlenecks in large simulations.

Virtual platforms can, and should, be used at various abstraction levels, starting with a very simple functional description and then refining towards implementation. As each part (that includes hardware or software) of the design is refined, it can be tested in the context of the entire system. Often, parts of the design that did not interface with the new block being tested can be run at higher levels of abstraction to speed simulations.

The main question: will SystemC primarily be used by the small number of system designers already working in C/C++ or by the much larger number of RTL chip designers using VHDL or Verilog? The latter group has so far shown a lot of skepticism about C-language design and has warmed more to SystemVerilog and then SuperLog as a way of moving up in abstraction.

The next question: why wouldl system designers benefit from system? There are already SystemC tools that help automate the transition from C to HDLs. The best argument,

probably, is that SystemC models of intellectual-property (IP) blocks can be exchanged with other companies, while proprietary C models generally cannot.

The point that may often be missed is that C/C++ language design is already in widespread use today. It is being used by many major electronics manufacturers, particularly in Europe, for system-level modeling and silicon intellectual property (IP) design. In this sense, C language design is an old methodology, not a new one.

Most of these companies, however, use their own proprietary C/C++ class libraries, making IP exchange very difficult. That's where SystemC comes in. One standardized library will allow companies to mix and match C language IP from different sources. SystemC is really about IP modeling, not forcing RTL designers into a new methodology.

C language models can also help speed RTL simulation. It has always been possible to bring in C language models through the Verilog programming language interface (PLI), but some vendors are looking at more efficient ways to create this link. C/C++ language approaches to testbench generation, such as Cadence Design Systems' TestBuilder library, may also appeal to some designers.

New releases of Cadence Design Systems Inc.'s Signal Processing Worksystem (SPW) software and Axys Design Automation Inc.'s MaxSim Developer Suite increase support for SystemC. Designers are now allowed to add custom blocks using SystemC using SPW. The tight connection SPW 4.8 has with SystemC 2.0 - through a Block Wizard - accelerates reuse of models written in an industry-standard modeling language, as well as models customers have written in SPW. Cadence will also announce that it is donating its Testbuilder class libraries to the Open SystemC Initiative to encourage testbench reuse.

Mentor Graphics Corp. has also added a C language interface to its Seamless hardware/ software co-verification environment that lets designers use mixed C/C++ and HDL descriptions for hardware. The interface, called C-Bridge, will be included with Seamless version 4.3, which is to be released in late March 2003. HDL simulation users today can bring in C code through the Verilog programming language interface (PLI) or VHDL foreign language interface (FLI), but C-Bridge supports faster performance and a higher level of abstraction. The interface supports abstract reads and writes for bus connections, but can still be cycle-accurate. C-Bridge provides an applications programming interface (API). To use it, designers adapt their C language models to read and write through the API. They instantiate the models in Seamless, where they're dynamically loaded into instruction-set simulation (ISS). However, C-Bridge provides its own source-level debugging environment rather than using the ISS debugger. C-Bridge imposes no limitations on C language code, and can be used with SystemC version 2.0 models.

SystemC in one of the most important players as a future SoC design and verification language. The big interest given by the major companies in the SoC design and verification fields to SystemC is quite a good proof for that. The debate is being concentrated on either going for SystemVerilog (and then SuperLog) or for SystemC. For instance, both approaches are coexisting together; they eventually could play together in the future.

3.3.2. Java-Based Proposals

While there has been some discussion about the potential of Java as a system-level language or high-level hardware description language (HDL), LavaLogic [5] may be the first commercial EDA provider to bring that option into contemporary design systems. The company's Java-to-RTL compiler is an "architectural synthesis" tool that turns Java into synthesizable HDL code. LavaLogic is offering a tool that will take high-level Java descriptions down to gate-level net-lists, starting with FPGAs [29]. The architecture of the Lavalogic solution is given in Figure 5.

According to Java advocates, Java appears to be the purest language to solve the productivity problems currently at hand. They also claim that the language can express high-level concepts with far less code than today's HDLs, yet offer more support for concurrency than C or C++.

Specifically, LavaLogic advances that the code required for a functional description in Java is typically one-tenth that required for Verilog. Java simulation models are 100 to 1,000 times faster. This is what the company advised however no rigorous proof was done until now regarding this point.

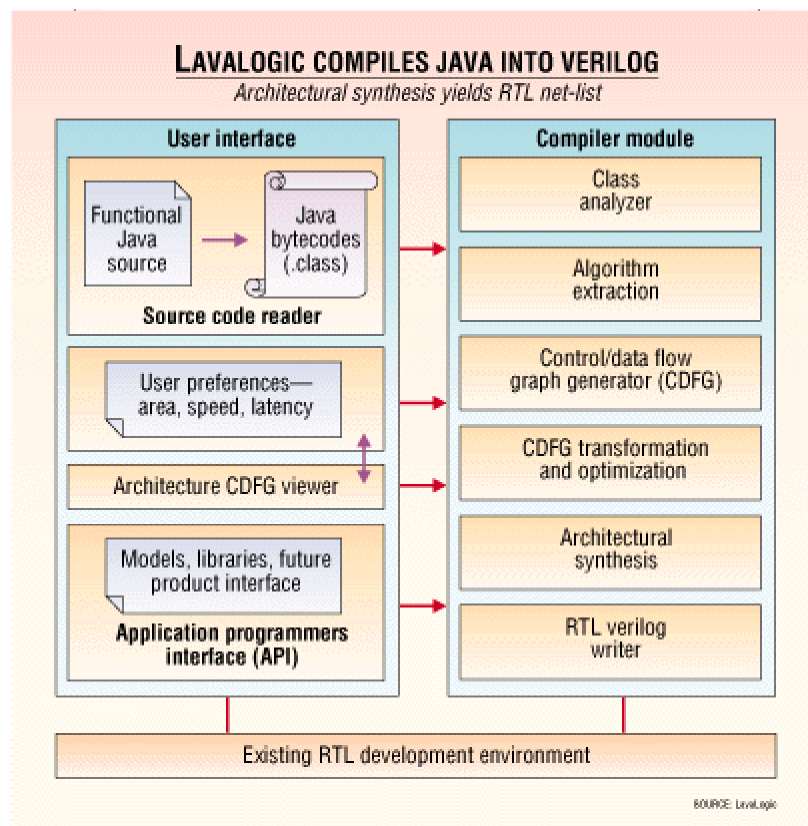


Figure 5. Lavalogic Java Based Proposal [5].

The main point all Java advocates stressed in comparing their approach to those C/C++ is the concurrency. In fact, a classical problem with C and C++ is their inherent inability to express concurrency. In Java, in contrast, concurrency can be explicitly invoked with threads. Nevertheless, this unique criteria for comparison is not enough to balance the choice C/C++ to Java. Java can be classified as the "next best" choice after C++, since it does not support templates or operator overloading, resulting in a need for numerous procedure calls.

3.4. Developing a new Language

3.4.1. SuperLog and SystemVerilog

SuperLog [46] is a Verilog superset that includes constructs from the C programming language Because of its Verilog compatibility, it has earned good reviews from chip designers in such forums as the E-Mail Synopsys User's Group, where some designers have expressed considerable skepticism about C language hardware design. SuperLog holds a great promise for the next three to four years. It is the case mainly because SuperLog does not require a radical shift in methodologies and allows groups to retain legacy code.

SuperLog combines the simplicity of Verilog and the power of C, and augments the mix with a wealth of verification and system features [20]. Realizing that there is no point in having a language without being able to do something with it, ex Co-Design (bought recently by Synopsys) also offers a suite of simulation tools making it ready for industrial applications.

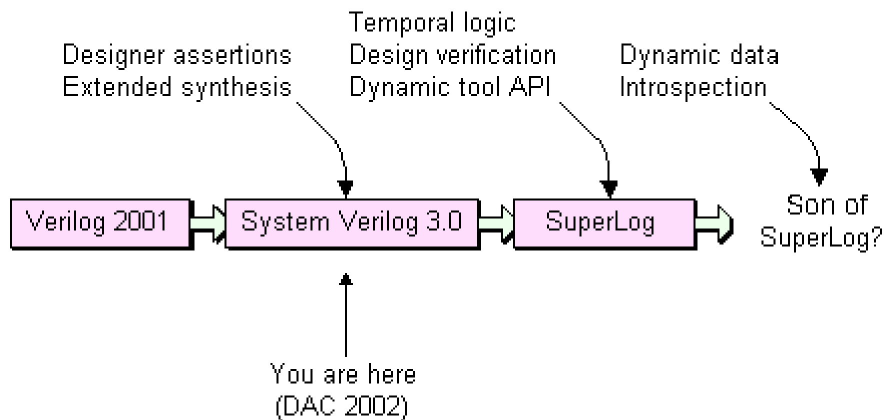


Figure 6. From Verilog to SupeLog [36].

For now we have Verilog 2001 at the lower end of the sophistication spectrum ("jolly nice but lacking a lot of features") and SuperLog at the other end ("incredibly powerful, but as yet falling well short of industry-wide support"). This is where Accellera reenters the picture. By working with Co-design and others to come up with a common meeting ground, which is SystemVerilog 3.0, Figure 6.

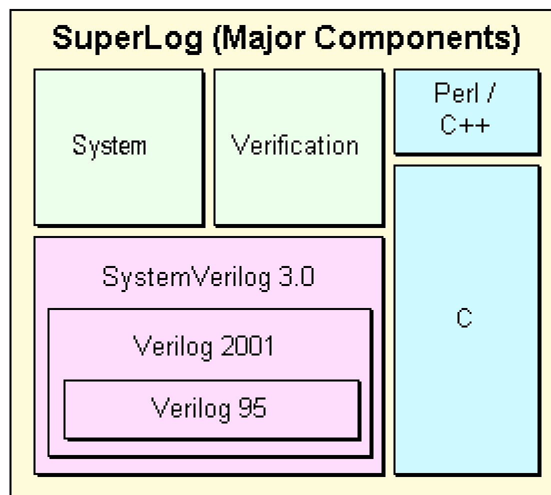


Figure 7. SuperLog Description [36].

The great advantage for SystemVerilog 3.0 from most people's perspective is that it includes things like assertions and extended synthesis capabilities, and it is an Accellera standard, so it will quickly gain widespread adoption. The advantage for Co-Design is that it's a step along the way to transmogrifying the Verilog everyone is using into SuperLog.

Actually Figure 6 is somehow simplifief, because there could be a number of intermediate steps before SystemVerilog 3.0 evolves into SuperLog. Figure 7 displays how Co-Design expects SuperLog to be.

SuperLog, as described in Figure 7, retains most of the features of Verilog and VHDL, including support for hierarchy, events, timing, concurrency and multivalued logic. It also borrows useful features familiar from software-programming languages such as C and Java, including support for dynamic processes, recursion, arrays and pointers. It includes support for communicating processes with interfaces, protocol definition, state machines and queuing [9].

In a certain way as defined, SuperLog is a smart idea to deal with system level design. In fact, SuperLog utilizes the power of C with the simplicity of Verilog to provide the right balance for productive design.

3.4.2. Rosetta

Now in the final stages of development by the Accellera standards organization, Rosetta represents a unique approach to system-level design. It provides one language that lets users describe behavior and constraints for nearly all engineering domains, including analog, digital, software, microfluidics and mechanical [2]. The declarative Rosetta language preceedes, but does not replace, implementation languages like Verilog, VHDL and C.

Launched in 1997 with Defense Advanced Research Projects Agency (Darpa) funding, Edaptive started working on system-level design before Rosetta was developed. The concept was to develop innovative products for designers of large and complex computing systems, especially mixed technology systems.

Rosetta is not designed to take the place of other languages, it tries to coordinate them. Rosetta is introduced to fill one "hole" in present capability --- the definition, capture, and verification of constraints and requirements on high level systems and their components. It provides language support for defining and combining models from multiple domains using multiple-domain semantics. In addition, it supports modeling and analysis at levels of abstraction much higher than those of current RTL-based languages. Its semantics are formally defined and it is highly extensible to support the adaptation to emerging systems.

Rosetta allows modeling at high levels of abstraction, modeling of heterogeneous systems and support for model composition. By allowing the user to define and compose models, it allows flexibility and heterogeneity in design modeling and supports meaningful integration of models representing all aspects of system design. Furthermore, Rosetta's formal semantics provide an unambiguous means for defining and composing models. Finally, the default Rosetta syntax provides a user-friendly mechanism designed to be comfortable for today's existing HDL user base [1].

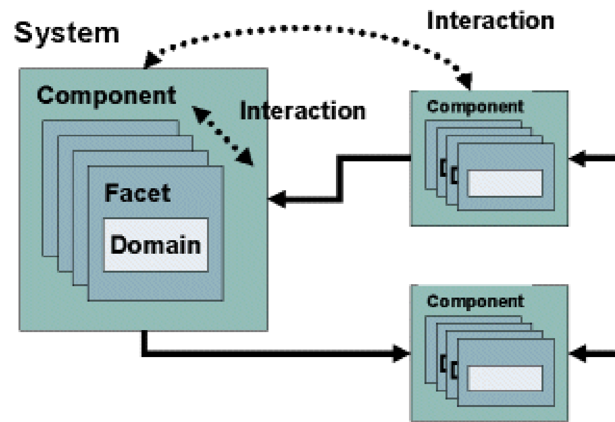


Figure 8. Facet Modeling Concept in Rosetta [52].

The Rosetta design methodology is based on the facet-modeling concept (Figure 8). A facet is a model of a component or system that provides information specific to a domain of interest. To support heterogeneity in designs, each facet may use a different domain model to provide domain-specific vocabulary and semantics. Facets are written to define various system aspects and are then assembled to provide complete models of components, component aggregations and systems.

The definition of facets is achieved by directly defining model properties or by combining previously defined facets. The former technique allows users to choose a specification

domain and specify properties that must hold in that domain. The latter technique allows users to select several models and compose a system model that consists of elements from each facet domain. The abstract semantics of Rosetta are based on this specification and combination of models representing information from various design domains. The syntax of Rosetta facets is designed to be familiar to engineers using existing hardware description languages. What makes the facet model different than traditional HDL approaches is the identification of specific domains.

A primary difficulty in addressing the system-on-chip problem is combining information from multiple domains within a single design activity. The domain concept allows this by enabling each model to refer to a base set of concepts that supports definition of information in one particular design area. For example, in a trigger specification the continuous domain provides the concepts of time, instantaneous change, ordering and state. Such concepts are important to continuous-time modeling, but may not be important when modeling power consumption or area.

The Rosetta semantic model is important because it reflects how system engineers currently tackle their problems. The syntax makes the semantics approachable. But the real contribution of Rosetta is the semantics of designing and composing component models. Through the use of domains, users are provided with design vocabulary specific to their domain of expertise rather than forcing the domain expert to work in a language that is too general or otherwise unsuitable for his or her needs.

Heterogeneity in system design not only emerges when defining multiple facets of the same system, but also when describing systems structurally by combining components. VHDL provides a structural definition capability that is mimicked in the Rosetta semantics using a concept called relabeling. When one facet is included in another, an instance of that facet is created by relabeling or renaming the facet. Facet parameters are used as ports and channel data types used to model communication between components.

Like single components, Rosetta systems may include component models from multiple design domains. Thus, Rosetta provides a mechanism for defining and understanding systems comprised of components from analog, digital, mechanical and optical domains in the same semantic framework. Furthermore, users may define new specification domains to extend Rosetta to address new domains and new modeling techniques for existing ones. Such a capability is extremely important for the future of any potential system-design language.

Unfortunately, it is not sufficient to simply model domain-specific information in isolation. Cross-domain interaction is the root cause of many systems failures and difficult design problems. System-level design requires understanding the collective behavior of interconnected components from different design domains, not simply the component-level behaviors. Further, interaction also occurs between different models at the component level. Rosetta provides methodologies for explicitly modeling and evaluating such interactions by describing how definitions from individual domains affect each other.

Finally, no one can deny that Rosetta, as defined, is too close to VHDL. On the other hand a question is why to complicate the design procedure by bridging between multiple functional languages? The single approach can solve most system-on-chip design problems and complicating the problem too much is not the way industry is supposed to work!

4. SoC Verification

As system-on-chip designs become a driving force in electronics systems, current verification techniques are falling behind at an increasing rate. A verification methodology that integrates separate but key technologies is needed to keep up with the explosive complexity of SoC designs [32].

A SoC verification methodology must address many more issues than were prevalent even a couple years ago, in particular the integration of purchased and in-house IPs into new designs, the coupling of embedded software into the design, and the verification flow from core to system. Several key concepts are important to understand, including the transition from core to system verification, the re-use and integration of multiple sources of cores, and the support needed to optimize core re-use.

There are two major problem areas with SoC verification today that keep the verification as a true bottleneck: IP core verification, and the System Level Verification (SLV).

The verification of today's IP cores tends to be inward focused. For example, the verification of a PCI IP core would test the bus modes and address space. This verification is useful, and helps provide information on the core functionality to the IP integrator. However, all this verification does not help a great deal at the system level, when the PCI core is connected to the rest of the system. How should the software team write drivers involving this IP core? What if the core needs modification? Will there be any architectural issues that arise when it's too late to change the core? All these make IP use, and reuse, challenging.

One effective technique for SLV is to use pre-defined verification IP. These can take two forms:

- Verrification Components (VCs). These are generally stand-alone verification building blocks that are based on industry standards, and that can be plugged into a verification environment and provide "instant-on" validation. They ideally incorporate monitors and protocol checkers, bring-up tests, efficiency routines, and sometimes a standard conformance suite, and are designed for use at the block level up to the system level with no modification. An example is a 1-Gigabit Ethernet verification component.
- Verification Platforms (VPs). These are generally an integration of multiple verification components, specific data generators, score-boarding mechanisms, and a high-level verification suite that are targeted at a specific domain application. An example is a Storage Area Networking verification platform, which includes Fiber Channel, Gigabit Ethernet, and PCI VCs, a packet generator and scoreboard, and iSCSI support.

4.1. Transactional Based Verification

Functional testing of system-on-chip software is difficult, especially when mature hardware does not exist yet [18]. A similar situation holds true for SoC-based devices whose behavior is a function of many factors interacting in complex ways. Testing each feature or subsystem separately is not enough to ensure correct operation, and testing for all possible factor combinations is infeasible [7].

Testing SoC-based devices, with a focus on software functionality, is inherently difficult. Testing SoC-based devices under construction only makes things worse, adding hardware and software immaturity issues, typically compounded by limited availability of test devices.

SoC and multi-chip system developers can no longer rely on traditional simulation tools, testbenches, and methodologies, but must augment these approaches to enhance verification productivity and to achieve quality and time-to-market goals [23]. This has created a serious need for a change in verification methodology. The industry must raise the verification bar to the next level of abstraction - transactions - to ensure that designers and verification engineers have the tools and methodologies that give them a higher degree of confidence in their designs [35].

A transaction is a single conceptual transfer of high-level data or control. It is defined by its begin time, end time, and all the relevant information associated with the transaction. This information is stored with the transaction as its attributes.

For example, the relevant information (or attributes) of a Read transaction includes address and data. Transactions can be as simple as a memory Read/Write or as complex as the transfer of an entire structured data packet through a communication channel. Transaction-Based Verification (TBV) enables the use of transactions at each phase of the verification cycle. The transaction level is the level at which the intended functionality of the design is specified and, therefore, the level at which the design can be verified in a most effective way. Raising the level of abstraction from signals to transactions facilitates the creation of tests, the debugging process, and the measurement of functional coverage.

While the design operates at the signal level with ones and zeros, a transaction-based methodology allows a hardware designer to create tests in a more intuitive way. System architects do not start out thinking about the relationship between the enable signal and the address bus. They start their design by thinking about what kind of data flows through the system and how and where it is stored. TBV is a natural extension of this high-level design process.

The typical verification process is comprised of three phases: test creation, design debugging, and functional coverage analysis. Each of these phases must be abstracted to transactions to take full advantage of the benefits of transaction-level verification.

Some companies have adopted transaction-based test development to verify their designs. In the Verilog world, tasks are used to implement transactions. In the VHDL world, procedures are used to implement transactions.

Although this is acceptable for basic tests, it has too many limitations when creating complex data structures, complex test scenarios, and dynamic test creation. High-level verification (HLV) languages, such as TestBuilder (C++), Vera, E, and others were developed to address these complicated issues.

A significant amount of time is spent debugging verification tests, which is an area that can greatly benefit from the use of transactions. The benefits come from the quick identification of transactions in the waveform tool by design and verification engineers without wading through low-level signal details.

Today users have to write massive testbenches to shake out easy bugs. Using assertions allows users to shake out bugs early in the process, so verification engineers can focus on corner-case bugs-the areas where they think they are really going to have problems with a design. Assertion-based technology has thus far been slow to catch on, but that it will become a critical element as designs-and, thus, functional verification-become more complex.

To this end, Cadence has folded into its Verification Cockpit a package that adds advanced features to its NC-Sim simulation offerings, along with support for Sugar 2.0 and static-checking capabilities. The new assertion capability lets users capture specifications, requirements and assumptions as assertions, then verify the assertions statically using the tool's new static-check capabilities.

4.2. Code Coverage Analysis

Coverage has become a key technology in the pursuit of efficient and accurate verification of large designs [32]. The easiest form of coverage to introduce into a verification methodology is register transfer level (RTL) coverage. Tools are available that, given an existing RTL design and a set of vectors, will provide valuable information. Apart from learning to use the tool and spending some time understanding the reports, RTL coverage does not fundamentally change the existing methodology and therefore can be added to any design project [19].

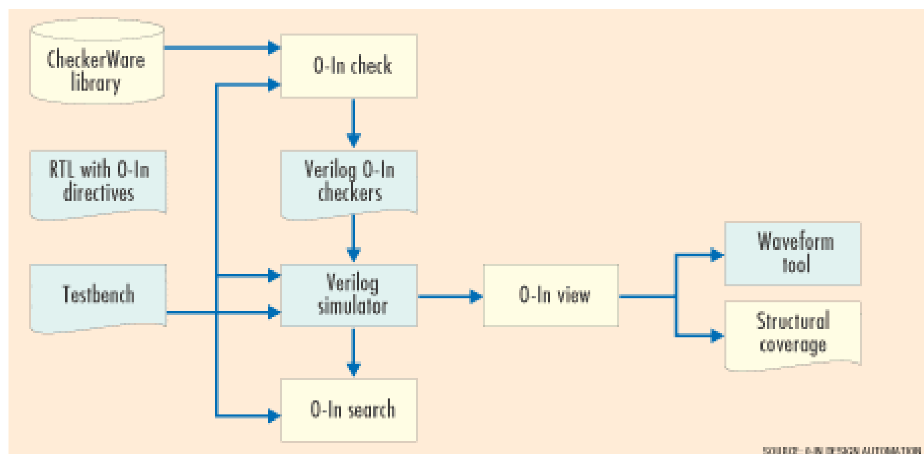


Figure 9. Semi-Formal approach: Hybrid Model Checking and Simulation Solution.

The basic idea behind RTL coverage is that, if the designer wrote a line of RTL, it was meant to be useful for something. While the tool does not know what for (a limitation that we will discuss later), it can correctly identify problems when a line was not used for anything at all.

RTL coverage users should heed one major warning: it is tempting, whenever the tool finds one case that was not covered, to add some vectors to cover it. While doing so will quickly increase the coverage results to whatever management asks for, it is the wrong approach from a methodology standpoint.

Often overlooked, the first limitation of RTL coverage tools is that they do not know anything about what the design is supposed to do. Therefore, the tools can only report problems in the RTL code that has been written. There is no way for them to detect that some code is missing. This simple fact means that RTL coverage will never find more than half of any design's bugs. Some bugs are due to incorrectly written RTL, while some bugs are due to RTL that is simply not there.

The second limitation of RTL coverage is the lack of a built-in formal engine. An expression coverage tool would see no problem in reporting that certain combinations of the expression inputs were not covered, even though by construction, they are mutually exclusive and therefore formally unreachable [39]. This "spurious" reporting adds to the confusion and limits the effectiveness of coverage. Constant propagation and effective dead code detection would also benefit from such an analysis.

Functional coverage analysis is used to determine if verification testing has exercised all of the required design functionality. Functional coverage exploration can be as simple as determining if a specific transaction occurred, such as a Read or Write to the RAM, or as complicated as a sequence of transactions, such as back-to-back Reads of the same address in the RAM.

Functional coverage is being a hot topic these days because companies need more than ever needed before to be able to determine when verification is complete. Functional coverage refers to a metric that defines this completeness. Many companies are stating functional coverage will be an important component of their next-generation verification methodologies.

Verification metrics are a set of numbers used to determine how much design functionality the verification suite has exercised. Functional metrics and structural metrics are the two major classifications in verification metrics.

Functional metrics refers to metrics that create reports on the measurement of functionality exercised in the design. This class has the best correlation to functionality defects. For example, a defect is found if required functionality, such as the execution of a multiply instruction in a processor, is exercised and it fails. Functional metrics is difficult to implement because the verification engineer must extract the list of functionality to be searched for, and devise a means of proving that the functionality was exercised. This work is done in an ad-hoc manner today.

Structural metrics refers to metrics that create reports on structural issues, such as which lines of code or which states in a state machine have been exercised. The types of metrics that fall into this class are code coverage (statement, branch, path, expression, toggle coverage) and state machine coverage. This class has a weaker correlation to functionality defects. For example, if an entire regression suite of tests is executed, and findings show that lines of code used to implement the multiply instruction of a processor were not executed, further investigation is required to determine if a defect was found. Structural metrics, however, are easier to implement because it requires no additional effort by the verification engineer except execution of a code coverage tool.

Nowadays, structural metrics, such as code coverage, are getting more interest. A fact is, these metrics are relatevely easy to implement and interpret. However, increasing design complexity has made the use of structural metrics alone inadequate; therefore, adding functional metrics for future designs is becoming a must.

Coming from the software side, code coverage metrics provide a structural coverage view of design verification. Structure refers to components of the design such as lines of code, variables, and states. Metrics tell a verification engineer which structure (line, variable, state) of the design was exercised.

Functional coverage metrics provides a functional view of verification completeness, which is more desirable than a structural view. Historically, functional coverage metrics have been harder to use because it requires the verification engineer to decide which functionality should be checked and then determine if that functionality was exercised. Schedules usually do not include time to create the design knowledge and measurement methods needed for this coverage. For this reason, many engineers perform manual functional coverage. This involves manually reviewing the results of a test to determine the functionality exercised. Manual review can mean looking at signals in a waveform viewer. This time-consuming and sometimes error-prone process usually occurs only a small number of times, sometimes only once. Generally, manual functional coverage is not done when a design changes due to bug fixes or feature creep (additional features added during the development cycle). It is assumed that despite a slight change in functionality, overall coverage remains the same.

Functional coverage is the good methodology to test designs exhaustly but never being able to answer the question is verification complete? Traditionally, this metric has not been used because of the historic difficulty in implementing it. The two most complex issues are defining tests that execute the desired functionality and collecting the functional coverage data to determine the metric. Advances in directed randomization technology address the issue of creating tests. The development of tools to capture and analyze transaction-based information addresses the collecting, analysis and determination of the metric. The combination of these two advancements will reduce the complexity of implementing functional coverage metrics.

The Accellera organization was considering four languages for the job of beign a standard for system and properties specification: Temporal e from Verisity, CBV from Motorola, Sugar from IBM and ForSpec from Intel. Sugar was selected and therefore will serve as a

standarad for model checkers in the future. A details description about Sugar is given in the coming paragraphs.

4.3. Formal Verification

Two main formal verfication techniques are used in the EDA industry. The first is the equivalence checking which refers mainly to a comparison of an RTL version of the design to its gate-level equivalent, or the comparison of two gate-level netlists. The other technique is model checking [42] which concerned with properties verification mainly at the RTL level.

Model checkers are the most adequate formal technique to be used at the system level design. Classicaly, model checkers accept the RTL for a design block and a set of properties associated with that block (for instance, "Signals A and B must never be low at the same time". The model checker then exhaustively analyzes the design to check if these properties are true in every possible case.

With this technique there are no corner cases, because the model checker examines 100% of the state space without having to simulate anything. However, this does mean that model checking is typically used for small portions of the design only, because the state space increases exponentially with complex properties and onr quickly runs into a "state space explosion.

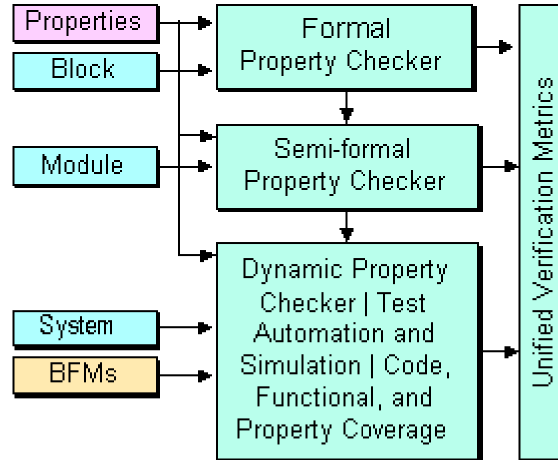


Figure 10. Using Model Checking for SoC Verfication.

For instance, there is no “new” model checkers adapted for system level design of the SoC domain. The main reason for that is the classical poblem of state explosion related to this kind of tools. Nevertheless, what is interesting about these techniques is the definition of hiereachical verification allowing the use of the checkers for small design portions as described in Figure 10.

The most relevant progress for the future use of model checkers for SoC verification is the selection of the Sugar [27] language as a new standard for formal properties and systems

specification. In the coming two paragraphs, we will put more light on Sugar as a standard and also on one of its relevant concurent OpenVera [28].

4.4. Properties Specification Languages

Accellera's VFV was originally looking at four proposals - Intel's ForSpec, IBM's Sugar, Verisity's "e" language, and Motorola's CBV. But none of them met the committee's original requirements. Each company came back with proposed revisions, and the choice was narrowed to Sugar and CBV. The final vote was a 10 to 4 in favor of Sugar. Synopsys and Intel, which remain on the VFV committee, voted in favor of CBV [26].

The Accellera organization has selected Sugar, IBM's formal specification language, as a standard that it would drive assertion-based verfication. The choice puts Sugar against OpenVera 2.0 (an open-source language from Synopsys Inc. and Intel Corp.)

Both languages let designers write properties and check assertions in simulation or formal verification, and both seek to end the proliferation of proprietary assertion languages. Yet the two proposals are dividing the EDA industry into two camps: a large group of vendors, including Cadence Design Systems Inc., supporting Accellera's decision, while a smaller group has signed on to support OpenVera 2.0.

Assertions allow designers to check properties, such as sequences of events that should occur in a given order. ForSpec provides syntax that lets designers describe complex sequences and clocking styles, and employ asynchronous resets.

Basically, assertions are created to describe design specifications. They can describe undesirable behaviors or specific behaviors that are required to complete a test plan. They can then be checked in either dynamic simulation or formal verification.

4.4.1. Sugar

Sugar [26] was developed at IBM's Haifa Research Laboratory, and has been used for more than eight years inside IBM. According to IBM, the Sugar language is described as "a declarative formal property specification language combining rigorous formal semantics with an easy to use style." Sugar did not require an advanced degree in formal verification, yet had solid mathematical underpinnings.

Sugar is a language for the formal specification of hardware. It can be used to describe properties that are required to hold of the design under verification. It is also possible to use Sugar for the specification of the system [17]. A Sugar specification can be used as input to a formal verification tool, such as a model checker or a theorem prover, which can automatically determine whether a design obeys its Sugar specification. A Sugar specification can also be used to automatically generate simulation checkers, which can then be used to check the design "informally" using simulation.

The structure of Sugar consists of four layers [4]:

- The Boolean layer is comprised of boolean expressions. For instance, A is a boolean expression, having the value 1 when signal A is high, and 0 when signal A is low. Sugar interprets a high signal as 1, and a low signal as 0, independent of whether the signal is active-high or active-low.
- The temporal layer consists of temporal properties which describe the relationships between boolean expressions over time. For instance, always (req -> next ack) is a temporal property expressing the fact that whenever (always) signal req is asserted, then (->) at the next cycle (next), signal ack is asserted.
- The verification layer consists of directives which describe how the temporal properties should be used by the verification tool. For instance, “assert always (req -> next ack)” is a verification directive that tells the tool to verify that the property always (req -> next ack) olds. Other verication directives include an instruction to assume, rather than verify, that a particular temporal property holds, or to specify coverage criteria for a simulation tool. The verification layer also provides a means to group Sugar statements into verification units.
- The modeling layer provides a means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables. The modeling layer is also used to give names to properties and other entities from the temporal layer. Sugar comes in three flavors, corresponding to the hardware description languages Verilog and VHDL, and to the language EDL [17], the environment description language of IBM's RuleBase model checker. The flavor determines the syntax of the boolean and modeling layers. In all three flavors, the temporal layer and the verification layer are identical.

4.4.2. OpenVera-Forespec Proposal

OpenVera-Forespec proposal was advanced by Synopsys Inc. and Intel Corp as a candidate for an industry-standard assertion language. The proposal was denoted as OpenVera 2.0 [38], which includes Intel's ForSpec property language [4]. This quite strong proposal was rejected because it was very formal and too complicated for engineers to use it!

OpenVera 2.0 is based on OpenVera 1.0 [40] and ForSpec. OpenVera 1.0 is an open-source verification language managed by Synopsys which already had an assertion capability for simulation. With the addition of ForSpec, OpenVera 2.0 supports formal verification tools as well. The whole concept in this combination was to have one language that supports both simulation and formal verification. ForSpec is a formal-property language used primarily by systems architects.

But the main additions to the previous version of Vera are new constructs and operators that support formal verification tools. In addition to formal operators, ForSpec brings directives such as "assume," "restrict," "model" and "assert" that are aimed at formal verification tools. An "assume-guarantee" construct lets assertions be used as module-level properties become monitors at higher levels of hierarchy [6].

The OpenVera Assertion (OVA) [6] language was developed to describe accurately and concisely temporal behaviors that span multiple cycles and modules of the device under

test. HDLs, such as Verilog and VHDL, were designed to model hardware behavior on single cycle transitions with procedural features. Such an execution model is not sufficient to efficiently specify multi-cycle temporal functions. With OVA, users can easily and intuitively express input/output behavior, bus protocols and other complex relationships of the design. Assertions captured with OVA are typically 3-5 times more concise than HDLs, enabling significant improvement in time spent writing and debugging assertions.

In contrast to four levels structure of Sugar, the OpenVera 2.0 language is broken down into five main sections or levels. The first level is called Context and is used to help and define the scope of the assertions, and the sampling time used. The next level, Directive, is used to specify what properties are to be monitored or checked. The third level consists of Boolean Expressions. The fourth level contains Event Expressions that describe temporal sequences. The last level is Formula Expressions, which are useful to describe how temporal sequences must occur in relation to one another.

Sugar and Open Vera 2.0 have two common layers: the Boolean and Temporal layers. This forms the core of the assertion language. The difference lies in the other layers. Open Vera 2.0 focuess on the detailed structure of the properties allowing the user to interact deeply with the assertions which inquires a certain degree of knowledge in formal verfication. Sugar went for two additional layers serving for the “formatting” of the properties and simplifying the interaction with the user.

In summary, whatever language used, Open Vera 2.0 or Sugar, assertions provide an evolutionary mechanism to greatly improve the productivity and confidence in the verification of complex SoCs.

4.5. Hybrid Verification Techniques

Simulation is no more able to meet the verification challenges faced by IC design teams, further innovations are needed. Today's high-performance simulation farms must be complemented with advanced verification technologies [50]. Leading design teams are complementing simulation with innovative tool technologies such as advanced testbench automation, assertion driven verification, coverage analysis and formal techniques.

However, HDL simulation is still at the heart of verification and these advanced technologies need to work and interact very efficiently with HDL simulation. Synergistic integration of all these technologies in a unified platform to create higher automation is called *hybrid verification*. Hybrid verification is supposed to be enough smart to enable higher productivity, performance and quality, resulting in higher confidence in verification, as well as reduction of overall time spent in verification. A structure of a possible proposal is given in Figure 11.

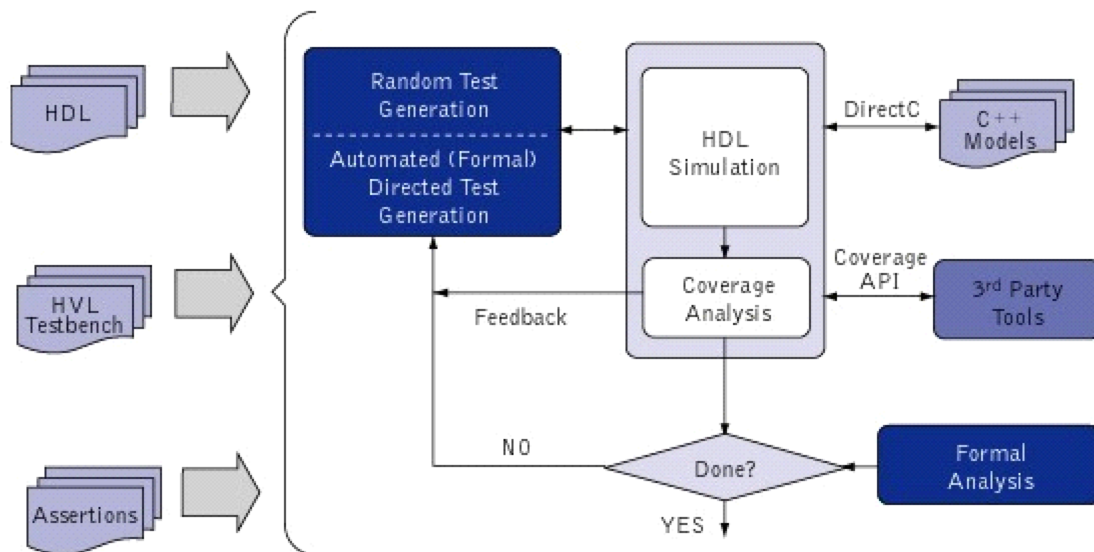


Figure 11. Hybrid Verification.

Higher abstraction Hardware Verification Languages (HVLs) have already been proven to speed the development of the verification environment and go above and beyond HDLs or homegrown tools to improve verification productivity [42]. HVLs help to create high quality tests faster.

Creating testbenches to verify designs requires dynamic and high-level data structures, such as C/C++ classes and other inter-process communication constructs such as "events", "semaphores" and "mailboxes." The OpenVera HVL is an intuitive, easy-to-learn language that combines the familiarity and strengths of Verilog, C++ and Java, with additional constructs targeted at functional verification, making it ideal for developing testbenches, assertions and properties [53].

Assertions are higher abstraction mechanisms that concisely capture design specification. They drive dynamic simulation and formal analysis to pinpoint bugs faster. They are a useful way to represent design specifications that are readable and reusable by multiple tools in the verification flow. HDL simulation tools use assertions to dynamically run "checkers" and "monitors" during simulation. Dynamic simulation of assertions allows for detecting bugs, whether simulating with random or directed tests. Functional coverage tools analyze simulation activity and provide information on coverage of functional test plans by reporting on coverage of assertions. Semi-formal tools use assertions to automatically generate tests to increase verification coverage. Assertions are also used as properties that formal analysis engines can use to exhaustively analyze and prove or disprove, greatly enhancing verification confidence.

Many design teams today integrate the higher-level data structures provided by C/C++ in a Verilog verification environment using the Verilog Programming Language Interface (PLI), but are confronted with the complexity of writing and maintaining PLI routines. VCS DirectC, for example, provides the ability to easily mix Verilog and C/C++ con-

structs without the need of a PLI. Users can use C/C++ to define high-level algorithms and then call these C/C++ functions directly in Verilog code.

With concurrent modules of DirectC, existing C models can be brought and mixed with Verilog constructs of "timing", "concurrency" and "port interfaces". With direct data exchange between C/C++ and Verilog, DirectC provides the best possible performance. DirectC has been successfully used in a wide range of designs such as microprocessor, image compression and networking applications.

Synopsys has donated the DirectC/C-API to Accellera for standardization in SystemVerilog. A standard C-API provides ease-of-use, higher performance and tighter integration of C/C++ within the Verilog HDL. DirectC will extend SystemVerilog, and make it easy to use higher abstraction C/C++ models within Verilog. It allows much faster simulation of design models at a higher level of abstraction, while staying within the existing RTL verification and implementation methodology. Similar to the standardization of the Value Change Dump (VCD) file format created by third-party debug and analysis companies a standard Coverage API will accelerate the development of third-party tools and applications that will use coverage information to enhance verification quality and productivity.

5. Conclusions

SoC design and verfication is an open project initiated a debate between hardware and software communities. The real issues for embedded chip design are increasingly embedded software issues. Three main trends are proposed to design of SoC. The software based methodologies see that basing everything from the HDL approach is starting from the wrong point. On the other side, the hardware designers are looking into upgrading Verilog to support system level design requirements. They think that the best is to go for what it works. The third group is supporting redefining new language redesigned from scartch for system level design. Nevertheless, this last approach is facing very hard criticism because most of the designers see that the idea of a grand unified language is fundamentally flawed. The best choice will be to have multiple languages targeting specific areas. These languages will be easier to learn than a grand language and they will also be easier for tools to parse.

Whatever the syntax looks like C, Verilog, VHDL or UML, most options share a common goal: to transit functional design from the minutiae of Boolean logic, wiring and assembly code up to the level of the designer's issues. Thus, functionality becomes modeled as an architectural interaction of behaviors, protocols and channels. This is great, except that most languages handle only digital functionality and cannot comprehend trade-offs of power, timing, packaging or cost.

There are also shortcomings with today's methodology in the face of the growing use of SoCs. Individual designs that were once an entire system are now blocks in an SoC. The associated verification strategy with many of these blocks was not designed to scale up to a higher integration level. If the methodology does not permit easy integration of block-level verification code into a system-level SoC environment, then the project verification will become a major bottleneck to the entire system.

The successful SoC verification methodology must be able to integrate multiple in-house or third-party IP cores, effectively migrate testing from the block to the system-level to maximally leverage testing, integrate software such as drivers and diagnostics, debug, and provide for the adoption of HDL acceleration and formal verification.

It is important to understand that SoC verification does not imply a homogeneous environment. There are lots of tools and methods out there. IP comes from many sources, internal and external. A solid verification methodology must be able to incorporate verification code from a number of sources, tools, languages, and methods. A successful methodology is a collection of tools integrated in an open and documented platform. The most effective way to improve the quality of results, shorten the development time, and decrease costs is with a careful, consistent verification methodology used throughout the project!

The ideal scenario is to achieve comprehensive validation without redundant effort. Coverage metrics helps approximate this ideal by acting as heuristic measures that quantify verification completeness, and identifying inadequately exercised design aspects and guiding input stimulus generation. A fact is, simulation is no more single player in the verification of hardware designs. The hybrid approach, mainly based on property checkers and smart test generation, is being set a default for SoC verification.

References

[1] P. Alexander and D. Barton, "*A Tutorial Introduction to Rosetta*," Hardware Description Languages Conference (HDLCon'01), San Jose, CA., March 2001.

[2] P. Alexander, R. Kamath and D. Barton, "*System Specification in Rosetta*", presented at the IEEE Engineering of Computer Based Systems Symposium, Nashville, Edinburgh, UK, April 2000.

[3] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar, "*The ForSpec temporal logic: A new temporal property-specification language*", Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), LNCS, Springer-Verlag, 2002.

[4] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. "*The temporal logic sugar*". Computer-Aided Verification (CAV'00), volume 2102 of Lecture Notes in Computer Science, pages 363-367, Springer-Verlag, 2001.

[5] P. Bellows and B. Hutchings, "*JHDL - An HDL for Reconfigurable Systems*", IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Pres",Los Alamitos, CA, pp. 175-184, 1998.

[6] J. Bergeron and D. Simmons, "Exploiting the Power of Vera: Creating Useful Class Libraries" Technical Report, Qualis Design Corporation, 2000.

[7] P. Bergmann and M.A. Horowitz. "*Improving coverage analysis and test generation for large designs*". IEEE Int. Conf. for CAD, pp. 580-584, 1999.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.

[9] Co-design Automation, "*SuperLog Tutorial: A Practical Approach to System Verification and Hardware Design*". HDL Conference, Santa Clara, CA, USA, February 2001.

[10] CynApps, Inc. "*Cynlib: A C++ Library for Hardware Description Reference Manual*". Santa Clara, CA, USA, 1999.

[11] G. De Micheli and D. C. Ku, "*HERCULES - A System for High-Level Synthesis*", The 25th Design Automation Conference, Anaheim, CA, June 1988.

[12] G. De Micheli, "*Hardware Synthesis from C/C++ models*", Proceedings of the conference on Design, automation and test in Europe, Munich, Germany, pp. 80, 1999.

[13] R. Domer, "System-level Modeling and Design with the SpecC Language", Ph.D. Thesis University of Dortmund, 2000.

[14] R. Domer, D. D. Gajski, A. Gerstlauer, S. Zhao and J. Zhu, "SpecC: Specification Language and Methodology", Kluwer Academic Publishers, 2000.

[15] R. Domer, J. Zhu and D. Gajski, "The SpecC Language Reference Manual", Technical Report, Department of Information and Computer Science, University of California, Irvine, 1998.

[16] D B. P. Douglass, "Real-Time UML", Addison-Wesley, 2000.

[17] C. Eisner amd D. Fisman, "*Sugar 2.0 tutorial explains language basics*", EE Design, May 2002.

[18] A. Evens, A. Siburt, G. Vrchoknik, T. Brown, M. Dufresne, G. Hall, T. Ho and Y. Liu, "*Functional Verification of Large ASICS*", ACM/IEEE Design Automation Conference, 1998.

[19] F. Fallah, P. Ashar, and S. Devadas, "*Simulation vector generation from hdl descriptions for observability enhanced-statement coverage*", Design Automation Conference, pp. 666-671, 1999.

[20] P. L. Flake and Simon J. Davidmann, "*SuperLog, a unified design language for system-on-chip*", Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 583 -586, 2000.

[21] J. Floch, "Using UML for architectural design of SDL systems", SINTEF Telecom and Informatics, 2000.

[22] Forte Design Systems, "Cynlib Language Reference Manual Release 1.4.0".

[23] P. Frey, R. Radhakrishnan, H. Carter, P. Alexander and P. Wilsey, "A *Formal Specification and Verification Framework for Time Warp Based Parallel Simulation*", IEEE Transactions on Software Engineering, January 2002.

[24] A. Gerstlauer, R. Domer J. Peng and D. D. Gajski, "System Design: A Practical Guide with SpecC", Kluwer Academic Publishers, 2001.

[25] A. Ghosh, J. Kunkel and S. Liao, "*Hardware Synthesis from C/C++*", Design, Automation and Test in Europe, pp.382-383, March 99.

[26] R. Goering, "*Accellera picks IBM's formal property language as standard*", EE Times, April 2002.

[27] R. Goering, "*Next-generation Verilog rises to higher abstraction levels*", EE Times, March 2002.

[28] F. I. Haque, K A. Kahn, J. Michelson, "*The Art of Verification with Vera*", Verification Central, 2001.

[29] B. Hutchings and B. Nelson, "*Using General-Purpose Programming Languages for FPGA Design*", Design Automation Conference, pp. 561-566, June 2000.

[30] IEEE Std. 1364-2001, IEEE Standard for Verilog Hardware Description Language 2001. ISBN 0-7381-2827-9. IEEE Product No. SH94921-TBR.

[31] A.A. Jerraya, M. Romdhani, Ph. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J.M. Daveau,and N.-E. Zergainoh, "Multi-language Specification for System Design and Codesign", System Level Synthesis, Kluwer Academic Publishers, 1999.

[32] M. Kantrowitz and L. Noack, "*I'm Dome Simulating: Now What? Verification Coverage Analysis and Correctness Checking of the DECchip21164 Alpha Microprocessor*". ACM/IEEE Design Automation Conference, 1996.

[33] D. C. Ku, G. De Micheli, "*High-level Synthesis and Optimization Strategies in Hercules/Hebe*", EuroASIC Conference, Paris, France, May 1990.

[34] D.C. Ku and G. DeMicheli. "Hardware C - a language for hardware design.Technical Report", Computer Systems Lab, Stanford University, August 1988.

[35] M. Kudlugi, S. Hassoun, C. Selvidge and D. Pryor, "*A Transaction-Based Unified Simulation/ Emulation Architecture for Functional Verification*", Design Automation Conference, pp. 623-628, 2001.

[36] C. Maxfield, "*A plethora of languages*", EE Design, August 2001.

[37] M. Meixner, J. Becker, Th. Hollstein, M. Glesner, "*Object Oriented Specification Approach for Synthesis of Hardware-/Software Systems*", GI/ITG/GMM Workshop Methoden und Beschreibungsprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 1999.

[38] M. Mohtashemi and Azita Mofidian, "*V*erification Intellectual Property (IP) Modeling Architecture", Guide to Structured Development Using OpenVera, 2002.

[39] D. Moundanos, J. A. Abraham, and Y. V. Hoskote. "*Abstraction techniques for validation coverage analysis and test generation*", IEEE Trans. Computers, vol. 47, no. 1, pp. 2-14, January 1998.

[40] "OpenVera 1.0", Language Reference Manual, March 2001.

[41] P. R. Panda, "*SystemC A modeling platform supporting multiple design abstractions*", International Symposium on Systems Synthesis, pp. 75-80, 2001.

[42] P. C. Pixley, "*Integrating model checking into the semiconductor design flow*", Computer Design's Electronic Systems journal, pp. 67-74, March 1999.

[43] R. Otten, P. Stravers, "*Challenges in Physical Chip Design*", International Conference on Computer Aided Design (ICCAD), pp. 84-91, 2000.

[44] R. Roth and D. Ramanathan. "*A High-Level Hardware Design Methodology Using C++*", 4th High Level Design Validation and Test Workshop, San Diego, pp 73-80, 1999.

[45] S. E. Schultz, "*The New System-Level Design Language*", Integrated System Design, July 1998.

[46] SuperLog website: http://www.SuperLog.org.

[47] S. Sutherland, "The Verilog PLI Handbook", Kluwer Academic Publishers, 1999.

[48] S. Swan, "An Introduction to System Level Modeling in SystemC 2.0. Open SystemC Initiative (OSCI)", Cadence Design Systems, Inc. May 2001.

[49] SystemC website: http://www.SystemC.org.

[50] D. Van Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. B. Brown. "*High-level design verification of microprocessors via error modeling*", ACM Trans. Design Automation of Electronic Systems, 3(4):581-599, October 1998.

[51] S. Vernalde, P. Schaumont, I. Bolsens, "*An Object Oriented Programming Approach for Hardware Design*", IEEE Computer Society Workshop on VLSI'99, Orlando, April 1999.

[52] E. E. Villarreal and Don S. Batory, "*Rosetta: A Generator of Data Language Compilers*", Symposium on Software Reusability, Boston, Massachusetts, pp. 146-156, 1997.

[53] J. Yuan, K. Schultz, C. Pixley, H. Miller, and A. Aziz, "*Automatic Vector Generation Using Constraints and Biasing*", Journal of Electronic Testing: Theory and Applications, pp. 107-120, 2000.