Environments for VLSI and Software Engineering

Randy Katz

University of California—Berkeley

Walt Scacchi

University of Southern California

P. Subrahmanyam

University of Utah—Salt Lake City

A description of environments for VLSI and software development is provided. Many similarities exist because of the common difficulties of building complex systems in either software or hardware. These similarities include support for the complete system life cycle as well as the differences, e.g., the overwhelming need for simulation aids in the VLSI design domain. Development environments of the future will have to be able to support both disciplines because distinctions between the two continue to disappear, and because large systems need to incorporate both hardware and software components in their designs.

1. OVERVIEW

This report elaborates on the discussions of the Design Environments Working Group that took place at the *Workshop on VLSI and Software Engineering,* in Rye, N.Y., in October, 1982. Our purpose is to address how environments for hardware/VLSI design and software development are similiar and different, survey the available development environments, and speculate on what will be available in the near and long-term futures. While our working group was not able to come to a consensus on what constitutes the ultimate development environment of the future, we were able to identify many of the most desirable features. We believe that these can be attained with extrapolations of available technology. Our most significant observation is that future environments for *system* development must encompass both hardware and software design. This is because the two design approaches share much in common, and because future systems will require elements of hardware and software design, as distinctions between the disciplines continue to blur.

The report is organized as follows. In the next section, we describe the guiding principles behind environments for hardware and software development. Since engineering support for the complete system life cycle is a crucial aspect of development environments, Section 3 defines the life cycles for both hardware and software development. Section 4 describes the basic functions of a development environment. In Section 5, we discuss what is similar and what is different in the current environments for VLSI and software. Finally, in Section 6, we describe the state-of-the-art in design systems for VLSI and software development, and look to what should be available in the near term future. We also speculate on what could be available in the more distant future, tempering the view with some of the difficulties likely to be encountered in developing future design systems. Section 7 contains our summary and conclusions.

2. GENERAL PRINCIPLES

The following "list of principles" represents our view of the most important features of a development environment for either hardware or software design. Any future design system must give careful attention to the provision of these features.

Easy to use user interface. The users of a design system are designers, not design system implementors,

Address correspondence to Walt Scacchi, Computer Science Department, University of Southern California, Los Angeles, CA 90089.

The Journal of Systems and Software 4, 13–26 (1984)

0164-1212/84/$3.00

and should not have to learn the intricacies of a complex system to get their design tasks completed. The interface should use menus extensively, helping to lead designers through the system with little effort. It should be able to operate in both expert and novice user mode, so as not to hinder the experienced user. Finally, graphical display of information, especially design data, presents it in a more meaningful way, and should be used.

Integrated support for the entire system life cycle. The environment should support a system development project through its entire life cycle, from inception through completion. Most development systems only provide help for a piece of the overall problem. Support should be included for project management (e.g., budgeting and scheduling), as well as design, implementation, maintenance, and validation of a system being constructed.

Layered, portable, evolvable development environment. No single development system will be suitable for all computing environments, application domains, and application team organizations. Therefore, it will be important that the environment be structured to facilitate the incorporation of alternative implementations of components of the development environment. For example, a simpler user interface package would be used with dumb terminals than if sophisticated engineering workstations were available. It should be possible to personalize the environment to a given application domain. Configurable language-directed text editors are examples of tools that provide such a capability. In addition, the development environment itself will evolve over time through incorporation of new facilities and components, and provision must be made to make this evolution as effortless as possible. Clean interfaces between components will be critical.

Environment does not restrict methodology. Development environments should aid the design process without constraining it. Several methodologies should be supported by the environment, since different methodologies may be appropriate for different components of the overall system. The design system must provide adequate performance, so as not to restrict designers from rapidly iterating between the stages of the life cycle.

3. ENGINEERING THE SYSTEM LIFE CYCLE

The "life cycle" of a system development project denotes the set of formal activities that occur in producing a system. These activities span from the conception of a system through its routine use and evolution. Ideally, we would like to support these activities with automated tools and a system engineering methodology for their use [7]. While comprehensive life cycle support is the goal, we do not yet understand what is required to provide this to project managers and system engineers alike. Nonetheless, we can describe what our current understanding is and where there is need for further investigation.

The development and evolution of a VLSI or software system follow a similar, but distinct course. These activities (or "stages") of the life cycle for VLSI or software system include:

Initiation and Acquisition: making a commitment to acquire or develop a new computing system

Requirements Analysis: determining why the system is needed and what resources must be available to support its development and use

Selection and Partition: determining whether the system components are available and choosing between alternative divisions of hardware and software components to be built

System Specification: describing what computational functions the desired system is to perform

Architectural Design: organizing and dividing system functions across computational modules and people who will build them

Detailed Design: designing algorithms and procedural units that realize the computational modules

Implementation: coding and integrating designed components in preparation for installation in another environment

Testing: verifying that the implemented system fulfills system specifications while validating its performance

Documentation: providing a written record of what the system does and how to use it

Use: operating the developed system under a variety of idiosyncratic circumstances according to the discretion of its users together with the mistakes they make in using it

Evaluation: evaluating the operation, performance, and applicability of the system in light of changing circumstances

Evolution: enhancing, tuning, repairing, and converting the installed system to maintain its useful operation

However, the apparent sequential order of these activities does not reflect the order we have actually observed [22, 34, 35].

The real work of system development entails *articulating* a variety of system descriptions, *transforming*

these into an operational system, and *evaluating* them and the developed system to establish a basis for its improvement. This work is by necessity incremental, iterative, and ongoing. However, we are not yet at a point where system engineering environments are available to thoroughly support this work. Thus, much of this work must either be done manually or else shoehorned into some existing tool assembly during system development.

Although these system life cycle activities are familiar to most system engineers and project managers, the organizational arrangements that affect the ease by which these activities can be completed are less familiar. It is not that system developers are not aware of these conditions; instead, it is more a matter of the extent to which these social conditions are explicitly accounted for when organizing and performing hardware or software system development [23, 35]. Nonetheless, the grouping of system life cycle activities listed above does provide a simple conceptualization of the history, organization, and performance of a system development effort that we can further examine.

3.1 Initiation and Acquisition

A software or VLSI system is initiated when participants propose and make the decision to acquire it. Acquisition denotes a binding commitment of organizational resources needed to perform system development. The proposed system somehow meets local "needs" that existing systems do not. The needs substantiate the decision to acquire the system. However, on closer examination, the possible range of conflicts within identified needs is large. For example, the need to acquire a new computer-aided design system for VLSI development may depend upon (a) overcoming organizational contingencies such as frequent failures or delays with the existing circuit design facilities, (b) the perceived ease with which design activities (or designers) can be better controlled, (c) the apparent technical benefits arising from standardized circuit patterns generated by the new system, (d) whether users are convinced that the new system will make their work more satisfying or entertaining, (e) possessing a "state of the art" VLSI design work station that will help attract or retain talented engineering staff, and (f) reducing the lead time necessary for complex circuit development. The point here is not whether all of these needs can be met; instead, whose agenda are they on, how are they prioritized, who determines the priorities, and whose interests are met when some need is fulfilled. In any case, this organizational phase of a system's life cycle puts the development effort into motion and shapes its direction.

3.2 Requirements Analysis

Participants are concerned with two kinds of system requirements, nonoperational and operational. Nonoperational requirements indicate the *package* of computing resources that the new system assumes must be in place to ensure its proper operation.[1] These requirements may indicate that certain development tasks or production processes be structured to be compatible (i.e., made efficient) with the new system. On the other hand, operational requirements for a system are expressed in terms of its performance characteristics (e.g., response time), standard interfaces, engineering quality practices, testability, reusability, user-orientation, and so forth. Finally, requirements for the system to be cost effective, produced within resource constraints, delivered on schedule, and easy to use and manage have both operational and nonoperational implications. Taken together, none of these requirements specify what the system's operations are. Instead, they outline preferences of participants to achieve a certain kind of engineering discipline through the life cycle of the system under development. Subsequently, these requirements form the criteria for evaluating the success of the system development effort. As such, what we need is some sort of "system requirements planning" facility that can take these preferences and system requirements as input. Managers, users, and engineers could then use the facility to produce plans coordinating their system development activities. Further, such an automated tool must operate with incremental updates to its inputs as well as be interfaced to system development databases. However, such a tool is not yet available.

3.3 Selection and Partition

Once a binding decision is made to acquire a new system, which system will do the job? Should the system be developed with in-house staff or should it be purchased elsewhere? Going with in-house staff facilitates the cultivation of local product and production knowledge useful in system maintenance. But if the system to be developed represents an unfamiliar or unproven technology, uncertainty over project completion within resource constraints may point to a lack of incentives for an in-house effort. Going with off-the-shelf compo-

[1] A computing package consists of not only hardware and software systems, but also organizational facilities to operate and maintain these systems, organizational units to prepare data and analysis, skilled staff, money, time, management attention, application specific knowhow, staff commitment to modern engineering practices, and policies and procedures for ensuring the orderly production of additional applications [21, 35].

nents, on the other hand, requires figuring out whether they will do the job in the target system. Subsequently, what criteria can be used to filter information for readily available components: (1) developer reputation, (2) prior experience of similar users, (3) performance characteristics, (4) quality of available documentation, or (5) ease of fit into local computing arrangements? In any case, uncertainty over what to consider in selecting system components is present. Thus, it is very likely that the system selection will be influenced either by the mobilization of participants favoring one component set, or by the participants whose input is trusted by those making the decision.

How to partition a system into hardware and software components is a related problem. Partitioning a system is more art than science. The final result depends on the order in which partitioning decisions are made. When no ordering prevails, the resulting partitioning tends to be ad hoc and circumstantial. Alternatives to this can be found through experimentation via system simulation, subsystem prototyping, or through the prior experience of system developers on similar projects. However, experimentation to evaluate partitioning trade-offs can be very costly. In short, we lack a notation and calculus for expressing the trade-offs between hardware and software system components as well as the automated tools that support such analyses.

3.4 System Specification

What is the system to do? What are the objects of computation and what operations are applied to them? How can these specifications be represented so that either their internal or external consistency, completeness, and correctness can be checked? Clearly, use of system specification languages and analysis tools helps. If the application domain for the system being specified is bounded and well understood, then a system generator may be available that produces a working implementation of a system from its specification. Examples in software include the so-called "application generators" popular in business settings, whereas in VLSI we have aids such as PLA generators. However, system generators that transform more general system descriptions into modular implementations are still subjects of advanced research. Subsequently, evaluating or transforming the specifications of either VLSI or software systems is an error-prone, manual activity. But it is not clear that more automation will eliminate difficulties in system specification.

Problems found in specifications may be due to oversights in their preparation or conflicts between participants over how they believe the system should function. Although a system specification language or methodology may serve as a medium of communication among participants, these aids do not resolve conflicts that might exist between participants; instead, they may make them more apparent. Therefore, who decides how to resolve a specification conflict? Who has a visible stake in achieving a particular outcome? How will specification responsibilities be divided among participants? Each of these questions points to tacit or explicit negotiations between project participants that occur in the course of developing system specifications. Further, the outcome of these negotiations will shape how stable the specifications will be and how frequently they must be reiterated.

3.5 Design

Designing a system entails deriving its configuration and detailing the computational procedures and objects from the available specifications. Developing the system's architectural design means articulating an arrangement of system modules (e.g., a "floor plan") that progressively transform the objects of computation into the desired results through local computational units. This articulation includes (1) choosing a system design technique, (2) developing and rationalizing alternative configurations, (3) employing a standardized notation for describing system architecture and module interfaces, (4) determining the order of module development (i.e., top-down, bottom-up, hardest-first, easiest to test, user interfaces first, etc.), (5) mapping system configuration onto staff to divide the labor, (6) performing system design, and (7) renegotiating any of these if local circumstances do them in.

On the other hand, developing a system's detailed design means articulating a symbolic description of the computational procedures organized in the architectural design. This stage of design requires interactive access to user knowledge of the procedures being codified into the system. This knowledge is usually dispersed across many participants with varying degrees of familiarity and commitment to the precision of articulation required for computational codification. Since this knowledge is difficult to access, gather, evaluate, codify, and stabilize, system designs will be plagued with errors of omission or misarticulation. As these problems emerge, system designs and possibly the software development artifacts preceding them will be redefined and reiterated.

3.6 Implementation

System implementation involves coding the design into an operational form. Choice of programming language for software, or layout format for VLSI comes into play

here. Techniques for verifying that the implementation systematically realizes the system's design, specifications, and requirements must be considered. Additionally, the choice may be constrained by the kinds of automated optimizations and performance validations sought. However, implementation also includes introducing early versions of the system to users for hands-on evaluation.

Much hand-holding between system engineers and users can take place to smooth the introduction. If users believe that the system is being imposed on them without their earlier participation, then a variety of counter-implementation actions may appear marking their resistance to the system's introduction. Therefore, to ensure the system's integration into an existing computing environment, participants will engage in a series of negotiations to (1) establish sustained service for the new system, (2) get enhancements to the delivered system to improve its fit, (3) eliminate major system bugs, and (4) train new users. But participation of users with the system's developers earlier in the life cycle may obviate the need for these negotiations.

3.7 Testing

In testing a software or VLSI system, we seek to verify that the implementation is a consistent, complete, and correct realization of the system's designs, specifications, and requirements. Formal testing of systems is still very costly and not widely practiced. Ideally, this verification could be controlled through selectively generating test data that allows system developers to evaluate the system's static (control-flow), symbolic, and dynamic (data-flow) behavior. However, most system testing is heuristic and generally performed through system simulation or hands-on operation.

The division of labor in testing a system usually leads system developers to perform isolated tests on system components and users to discover additional problems as the delivered system supports more routine usage. Then, when difficulties ("bugs") appear, a collective effort begins to try to locate the source of the problem. This effort usually entails a partial reconstruction of what transpired and how to make it appear again for further study and correction. Well-organized system development documentation helps, but if it's not available, people who might know about how the system works in its current form must be found and engaged. This situation grows worse if the attribution of responsibility for the bug or its adverse effects is unclear. Thus, if the reconstruction is marked by uncertainty and frequent negotiations, participants may subsequently choose to work around the system anomaly leaving it for other staff to rediscover, reconstruct (again), and attempt to rectify.

3.8 Documentation

Documentation represents the most tangible product of system development activities. However, its utility has a very short cycle unless effort is directed to continually update it. We usually hear more about (and experience) the inadequacy of available system documentation than of its superlative comprehensiveness.

Standards and incentives for good documentation are few. Users need one kind, developers another, and maintainers possibly a third. If the system development process has been volatile or behind schedule, then documentation may be put off.

Documentation work is labor intensive and revealing of personal communication skills. System evolution continually makes obsolete available documentation unless countervailing support is provided. Further, uncorrected system aberrations may not be documented since they may be used as evidence indicating either a lack of interest or lack of technical competency by certain participants. In short, in order to assure high quality of the most visible—and in the long term, the most important products of system life cycling—development and use of system documentation must be planned, organized, staffed, controlled, coordinated and scheduled in much the same way as the system must be. This suggests that life cycle engineering tools must also support the development and evolution of system documentation in a manner similar to the development of the system itself.

3.9 Use

How systems get used is not well understood from an engineering viewpoint. We do not know the extent to which well-engineered systems are easier to use. But we do know that system use is shaped by (1) the discretion a participant has over when and for what he/she can use the system, (2) how easy it is to learn how to use an unfamiliar system, (3) what kinds of mistakes or errors are likely to be encountered in using the system, and (4) how easy it is to integrate the system into the existing computing environment and established work routines [21, 22, 23]. Each of these arrangements is articulated only after a period of hands-on use of the system. These conditions cannot be thoroughly predicted during initial system development. However, as new systems are cycled through various user groups, the demands placed on the system (i.e., its requirements) change and subsequently so must the system. The system also changes because of staff turnover, thereby requiring new staff to (re)negotiate the arrangements which shape their use of the system. Thus, circumstantial conditions in the work setting play a large role in deter-

mining the pace at which a system is consumed and evolved.

3.10 Evaluation and Evolution

Local participants regularly evaluate how well-delivered systems work and how useful they are. As their experience with a system grows, so will a system's apparent inadequacy. System developers or users will seek enhancements, adaptations, repairs, or conversions as this occurs. System maintenance is really just ongoing, incremental system development. It is done to improve system performance, to keep the system usable, or to reduce the cost of its use. It can also be supported with the same automated tools used during initial system development. However, systems developed without contemporary engineering aids will be more difficult (and costly) to maintain.

The care and attention to detail by which maintenance work gets done shapes long-term system usability. However, many conditions counter an ideal practice of system maintenance: (1) system users often have more requests for enhancements than can be realized within the local computing environment, (2) poor quality of development documentation complicates the ease of figuring out where to make system alterations, (3) maintenance work often competes with new system development for staff attention, (4) multiple system versions appear when maintenance activities are not coordinated or when unwanted alterations are resisted by users, (5) turnover of system development staff fragments local system knowhow, and (6) bureaucratic mechanisms such as "change control boards" create a new source of resistance that must be engaged (or bypassed) in order to keep the system well integrated as a computing package. As maintenance activities lag, users may begin to take on maintenance work in order to keep the system usable. Otherwise, they work around the system through ad hoc extensions. Subsequently, as this arrangement becomes too demanding for users or as new technological alternatives appear, participants may let the system sink in order to establish the "need" to acquire a new system. This of course marks the termination of one system life cycle and the initiation of another.

3.11 Summary

Overall, there is a high degree of concurrency across the activities occurring during a VLSI or software system's life cycle. Each life cycle activity requires a different kind of description of the system being developed. A complete and consistent description is the desired product of each activity, although these descriptions will often be incrementally constructed and revised. Developing these descriptions using a machine-processable language or formal notation appears to be a key to providing automated life cycle support tools. However, three higher-order problems remain:

articulating the various system descriptions,

establishing a mapping or set of transformations between successive system descriptions, and

evaluating the completeness, consistency, and correctness of the system descriptions vis-a-vis one another.

These activities are by necessity incremental, iterative, and ongoing reflecting many system sublife cycles. But as we improve our ability to engineer this cycling, a growing array of resources must be provided, new tools will be necessary, new forms or subdivisions of work will emerge, and a more complex web of organizational and technological arrangements will appear which must be managed [23, 34, 35].

4. COMPONENTS OF A DEVELOPMENT ENVIRONMENT

A development environment for software–hardware systems should ideally consist of a well-integrated set of tools that support all the activities occurring during the system life cycle. We delineate here some of the more important components of such a development environment, focusing only on those parts of the overall life cycle that are subsequent to the decision to (re)develop a system. Thus, we do not discuss any managerial tools that might be used in reaching such decisions: these might include, e.g., tools which support project scheduling, resource accounting, etc.

In essence, the development of an integrated software–hardware system involves:

the acquisition of a specification of the problem from the end user(s) of the system

the actual design and implementation of the system

the subsequent evolution of the design and implementation as a result of changes in the requirements and/or specifications (until a decision to supplant the system by an entirely new one is eventually made).

The system development process typically involves reasoning at several different levels of abstraction. For instance, these include:

representation independent or "abstract" problem specifications (wherein the only objects and operations present are those of direct interest in the problem domain),

high-level model dependent "programs" that embody a recipe for the solution of the problem, and

any other levels of abstraction that facilitate the subsequent implementation in terms of some combination of (1) software executing on one or more general purpose host computers, and (2) special purpose hardware. Two examples of such levels of abstraction are (i) architectural descriptions of systems, and (ii) electrical descriptions of circuits.

It is desirable to have design paradigms wherein the choice of a final implementation medium (hardware or software) is not fixed a priori. This mandates that a uniform perspective be adopted both in the specification of the problem to be solved and, in as far as is possible, in the development of an implementation. Ideally, a problem specification must be representation independent, and as close to the user's conceptualization of the problem as possible. However, there is sometimes a tradeoff involved between the use of completely representation independent (axiomatic) specifications (e.g., [18], [12], [31]), and the use of conventional high-level languages as "specification" languages that yield specifications that are more or less representation dependent. While there are promising axiomatic specification techniques being investigated, the average (current day) system designer is not yet adept at writing formal specifications; in fact, most tend to be put off by the mere presence of formalism, although there are notable exceptions. A second, and more technical, issue is that it is sometimes not easy to specify a desired behavior axiomatically because of inadequate understanding about the problem domain. In such cases, it may not always be feasible to invest the extra effort required to acquire such an understanding due to project deadlines. Consequently, an alternative specification technique that is viable in such circumstances is useful as a practical alternative: this can generally take the form of an understandable "specification" in a high-level language.

4.1 Design Paradigms and Support Tools

Given that a problem specification is available, it is necessary to follow some design paradigm to obtain an implementation that is consistent with this specification. The tools needed to support this design process will, in general, depend on the particular design methodology adopted. We will here attempt to convey a flavor of some of the concepts involved by considering the tools that are needed in the context of a design paradigm discussed in [38, 39].

Both software and hardware design may be viewed as the process of representing the abstractions of objects and operations relevant to a given problem domain using primitives that are already available [31].

When synthesizing programs, the primitives used are those provided by the lower-level abstractions: these may either be chosen from an available library, or be explicitly supplied.

When synthesizing hardware implementations, the "primitives" available during conventional (i.e., printed circuit board) logic design are in the form of off-the-shelf chips; this is quite analogous to synthesizing programs. On the other hand, considerable flexibility in the number and nature of the primitives is available when doing special purpose VLSI design: this potentially enables the structure of the problem to be more directly mirrored in silicon.[2]

In the event that a truly abstract specification of a system is not viable for some reason, a high-level language program may be viewed as comprising a "specification" of the system, e.g., GIST [16] and Ada [13].

Depending upon the availability of axiomatic or high-level problem statements, there are two approaches to system design that one may adopt:

One can attempt to synthesize special purpose (software/VLSI) systems proceeding from abstract, representation independent, specifications of problems. This paradigm can potentially mirror the structure intrinsic to a problem directly in an implementation structure in silicon, and thus provide a way of tailoring machine architectures to specific problems. As a consequence of the modularization inherent in such specifications at different levels, parts of the system may either be in software or hardware.

Alternatively, a (more or less representation dependent) specification of the problem in the form of a program can be transformed into a hardware implementation and/or more efficient software implementation.

We now outline some of the prototypical components that need to be present in a development environment to support the design paradigm alluded to above. Briefly, these include tools that help interface with the end user(s), tools that aid in the development and eval-

[2] In the solution of a specific problem using specific technology, however, the number of primitives is usually quite small; e.g., pass transistors, boolean logic gates, flip-flops, multiplexors. This is partly necessary to reduce the complexity of the design process at this stage: such primitives are at a very low level, their correct implementation depends critically upon the technological "design rules," and usually require substantial effort to be "debugged." It is somewhat fortuitous that a small number of carefully chosen primitives suffices for a large class of problems.

uation of a system, and tools which support the evolution of a system after its initial design. We believe that in the long run it is important to base such tools on a reasonably cohesive theory of the various aspects of the design process, rather than to merely have an ad hoc collection of tools. Research therefore needs to be directed into gaining fundamental insights in such areas, rather than emphasize attempts to build fancy "front-ends" for otherwise ill-designed development environments [26].

4.2 Interacting with the User

To facilitate user interaction at the different levels of abstraction involved in the design process, it is convenient to use different forms of syntactic sugaring (even multiple external representations): e.g., both formal and informal textual forms for high-level specifications, graph representations for networks, and more detailed (color) graphic layouts for viewing lower level circuit structures.

It is possible to build knowledge based "expert" systems that have "natural-language" interactions with a user and aid in the acquisition of the initial problem specifications. Since the problem domain typically determines the most convenient way(s) of interfacing with a user, such user "front ends" should be designed so as to be extensible. We envision the sophistication of such front ends to improve with progress in related areas such as artificial intelligence, and to eventually include more-or-less natural language dialogue, speech recognition and visual sensors.

Backend interfacing: rapid prototyping. In order to facilitate interfacing with the other design development tools that form the "core" of an environment, it is useful to minimize the number of different forms that the input to this backend can assume. We believe that such an interface should allow for both axiomatic specifications and high-level language programs.

To facilitate rapid prototyping, it is desirable that the initial specifications be "interpretable" or "executable" to some degree. In particular, in order to ensure that the specifications do in fact embody the user's requirements, it is important to provide for some dialogue with the user that enables a clarification of the system's understanding of the problem. A tool that is able to accept various forms of user queries and answer them, e.g., using formal manipulation systems, a simple database retrieval mechanism, etc. is therefore useful in this context.

In general, it is convenient to have a means of prototyping systems at various levels of abstraction, so as to be able to exploit the design details as they are fleshed out. For example, one potential advantage of prototyping at a lower level is that more accurate information about various performance metrics can be obtained.

4.3 Design Development Tools

The paradigm of development followed can be viewed, to a greater or lesser extent, as a transformation of the problem statement expressed at a higher level of abstraction into a form that elaborates at a lower level of abstraction the representation of some of the constructs used.

Linguistic primitives. In order to support specification and reasoning at different levels in the design spectrum, it is important to have appropriate linguistic primitives, and languages that embody these. Examples of the kinds of primitives that are useful include: primitives that enable axiomatic specifications; primitives for parameterization of the design (yielding what are sometimes known as "macro languages"); primitives for describing the composition of design components ("interconnection primitives"); primitives to describe component interfaces, etc. It is important to be able to specify environmental characteristics of systems and performance characteristics [40]. These primitives may either be embodied in a "broad spectrum" language, or be viewed as a set of distinct languages.

Several common processes such as editing, pattern matching and replacement, and parsing are used at almost all levels in the design hierarchy and can benefit from language tailored tools such as editors, pattern matchers etc. [39, 46]. Further, tools that aid in symbol manipulation are useful in computing complexity measures of designs at various stages of development [41]. A machine-readable documentation of the history of a development, which incorporates the reasons for adopting strategic decisions, will aid in the incremental redesign of the systems in responses to evolving specifications.

It has been observed that the global strategy that guides a design depends to a large extent on (1) the performance requirements desired of an acceptable implementation and (2) the characteristics of the global environment that the resulting system is intended to function in. The tools needed to support the acquisition of information needed for the transformations and to support the mechanisms of the transformation process itself are therefore important ingredients of a development environment.

Evaluation/verification. Once a system has been designed, it is usually desired to evaluate its performance along one or more dimensions. Examples of relevant characteristics include, for instance,

memory and time utilization (in software implementations);

paging frequency (for virtual memory systems);

communication traffic density (in distributed systems); etc.

chip area, response time, throughput, reliability (for silicon implementations).

Tools to measure performance along these and other dimensions, and support the validation of designs are an important part of an overall development environment.

Simulation and testing. It is useful to be able to ensure at some intermediate stages in the development of a system, that a design is consistent with the specifications and has approximately the right performance characteristics. Although there are a few isolated instances where formal verification is being used in practice, the prevalent modes of improving the probability of a design being correct are still testing and simulation.

In particular, "simulation" is fast becoming a very important tool in the existing VLSI design cycle, whereas it is not as widely used nor as applicable in the milieu of software systems. On the other hand, "software testing" can be thought of as the (software) counterpart of (hardware) simulation. Just as in the case of software it is well known that testing can only help in the detection of bugs and can never demonstrate their absence, simulation of a hardware circuit usually helps in reducing the number of logical and electrical errors in a design, and cannot guarantee their absence.

In software, simulation is typically used only for systems that have characteristics that cannot be deterministically modeled very well, e.g., operating systems. In addition, it is used for software modeling of systems wherein the various interactions in the problem domains are not well understood, e.g., war games.

A circuit may malfunction because of errors in its logical design, or due to unforeseen electrical characteristics of the circuit elements used, or due to fabrication faults during reproduction. "Simulation" in the sense it is used in the hardware domain is used to reduce the probability of the first two happening, whereas "hardware testing" is used to detect erroneous circuit function after fabrication. Simulators in the hardware domain need to model both the electrical characteristics of circuits as well as their logical behavior. It is recognized by now that it is not practical to simulate large circuits at the device level using a simulator like, say, SPICE, because of the computational explosion that occurs. It is therefore important to have hybrid or multilevel (hierarchical) simulators in the development environment that allow simulation at the functional, logical, switch, and electrical levels, and that allow for a smooth transition between the various levels.

Test/simulation data generation: testing environments. Testing plays an important role both in the software system development life cycle and in the development of hardware systems. However, as pointed out above, there is a substantially differing set of requirements in software and hardware testing, mainly because of different interpretations of the terms. In the case of software, duplication is almost error-free, and testing relates primarily to testing the design of the logic of programs and their robustness. In hardware design, duplication (i.e., fabrication) is far from error free, and therefore testing plays a major role in both the logical design of the system, and in testing for the correct fabrication of ostensibly correctly designed components.

A set of support tools for test data generation, both for software and hardware and modules, is needed. In addition, it is important to have test beds or "drivers" that allow a software or hardware module to be "plugged in" and tested. In particular, apparatus for physically testing the functionality of fabricated chips is needed, as well as the ability to embed a chip in a software/hardware environment that simulates the actual environment in which the module is intended to function.

4.4 Configuration and Version Control

As any nontrivial system evolves, it typically goes through a cycle of design, implementation, and redesign. Besides being intrinsically quite complex, such a system consists of nontrivial subcomponents that are developed by several people and updated by several others. Further, there are usually multiple versions of a system in existence at any one time, both in the hands of different end users and in use by the designers themselves during the development phase. The utility of tools (even very primitive ones) that aid in keeping track of different versions and configurations of software systems, e.g., various releases to different end users, has been established beyond doubt. We believe that it is important that a richer set of support facilities than those existing in current version control systems like SCCS is needed. Some examples of such systems are the per-

sonal information environment developed by Bobrow and Goldstein [17] that support a layered network model of software systems (including various perspectives of a program), and systems like SOLID (an evolution of the programmers workbench at Bell Laboratories that facilitates in installing different versions of large systems). The initial experiments with such environments to support evolution of designs have been that a methodology for using the underlying tool is needed for efficient usage (e.g., if a layered network model is to be prevented from becoming a very expensive file system). Of course, it is desirable that such a set of facilities be reasonably well integrated into the remainder of the development environment.

4.5 File and Operating System Services

The development of a sophisticated system environment is critically dependent upon the underlying file and operating system capabilities available. In this sense, such capabilities are so basic that a certain minimal standard is indispensable to enable the development of good environments. On the other hand, once such capabilities are provided, they can be (and are) taken for granted for the most part.

In order to ensure that a system scales smoothly as its functions grow or as the size of the problems it can be applied to grows, it important to have a large enough address space (at least 24 bits, and preferably 32 bits of addressable code space). It has been experienced that space, rather than time, is the limiting factor in expanding existing systems, in the sense that once the available space limits have been reached, a radical redesign is needed. Of course, faster machines lead to improved response times and in turn to improved programmer productivity.

File support facilities are also important, particularly the ability to have segmented files and some form of direct accessing mechanism. Here, the addressing capabilities needed are typically larger: 32 bits may be enough for all of the code in a system but not for its data, e.g., dictionaries and medical and legal data bases can contain several billion characters very easily. Robustness of the underlying file system and storage mechanisms then becomes quite important.

Various support features of a similar nature, whose main objective is to free the system designer from low-level considerations while providing him with a reasonably powerful set of primitives include: good process and memory management, swapping facilities, object management support utilities (e.g., garbage collection, reference counting), interrupt facilities, exception handling, software and hardware dynamic monitoring functions for various performance measurements, etc. In addition, it is important that any software support routines already existing (or that are written in different languages) be easily integratable; in order to facilitate this, mechanisms to enable interlanguage communication must be provided. Microprogrammability, and the ability to speed up some of the basic support tools by casting them in hardware must also be feasible to some reasonable degree.

4.6 Database / Library Services

Several of the design functions may be viewed as appropriate manipulations being performed on relevant sets of objects. The integration of various tools therefore involves sharing some of these sets of objects. While it is possible to view some of these functions as being provided by a database management system (e.g., [20]), whether or not they are explicitly labeled as such is largely a matter of taste and/or historical precedence. Examples of the objects manipulated by a system would include libraries of programs (indexed by relevant attributes of the implementations), available implementations of standard (or widely used) hardware modules, cell sets implementing primitive functions and tailored to various technologies, etc. The database facilities should provide for accessing the various library entries when indexed by the relevant attributes. This system should support the evolution of the entries themselves and the existence of several versions of objects that perform similar functions. It is obvious that the system include many of the features of version control systems and other environments like PIE [17].

4.7 Implementation Services

When the initial startup cost for some component of a development environment is capital intensive, a centralized set of services may be made available to a reasonably circumscribed community of users. A successful example of such a service is the MOSIS facility provided by DARPA (through ISI) that enables members of the DARPA community to have their designs fabricated despite the lack of an in-house fabrication facility. This centralization has the benefit of localizing the choice of vendors and insulating the user from details of the fabrication process (to some extent). The emergence of such centralized silicon foundries is analogous to the development of university wide computing centers in the 60s and early 70s (in the absence of every department or research project having its own computing facility). A more geographically centralized example is the MACSYMA consortium, which provides access to symbolic computation facilities (located at

MIT) to members of the consortium; these facilities are accessible through various international networks.

One obvious advantage of such centralized facilities is that the cost per user is reduced. Further, in some cases, the existence of a single or limited number of such centers either implicitly or explicitly imposes some community wide standards, e.g., the use of CIF files to describe fabrication masks. A potential disadvantage of this is the fact that undesirable features may creep into such standards prematurely, thus harming the community in the long run. The existence of such a centralized facility cannot obviously be used to pursue research in the service provided by that facility and therefore does not eliminate the need for research centers to explore improved fabrication methods, symbolic computation methods, etc.

5. DIFFERENCES AND SIMILARITIES BETWEEN VLSI AND SOFTWARE

The similarities and differences between software and VLSI Systems can be seen by examining what is underdeveloped in both areas and what is being done at present.

5.1 Similarities

The complexity of systems under development and the number of people participating in those developments are increasing. Subsequently, the costs of developing these complex systems are rising. How to best organize and perform these development projects with the people and resources at hand is not well understood. However, producing and cultivating such an understanding are key factors in overall system development productivity [10, 35]. Use of automated environments that support comprehensive system life cycle engineering is likely to be another factor as they become available for software and VLSI.

New system development environments for VLSI and software applications increasingly rely upon formal notations or language-based system descriptions to support automated processing of each system life cycle activity. Because of this, a common set of automated tools that can be specialized to process these descriptions can be developed.

In general, there appear to be enough similarities between the development process for complex software and VLSI systems that automated environments, engineering methodologies, and project management strategies tailored for one technology may be applicable to the other. However, differences in the development process for software and VLSI must be supported before comprehensive hardware–software "factories" can be built.

5.2 Differences

Although limited, we have more experience with software environments than those for VLSI. VLSI is a younger technology: tools and engineering environments for VLSI are still quite new. Software environments such as UNIX-PWB and Interlisp have a large user base [45]. However, environments supporting the development of real-time software systems or complete system life cycle engineering are not widely used, if they are even available.[3] In addition, the importance of life cycle engineering efforts is not yet recognized throughout the VLSI development community.

More substantial difference between VLSI and software system development appears in later stages of their system life cycles. However, we should expect this since VLSI systems require a physical fabrication. This physical fabrication does not occur with software.

Clearly, there can be differences at each stage of the life cycle for VLSI and software systems. But these differences are subtle and usually specific to the system application domain or to the system development methodology in use.

With VLSI, preliminary circuit implementations must be made to test circuit operation with different fabrication constraints or circuit layouts to improve production yield. To minimize the chance of discovering fatal circuit design errors at such a late development stage, many VLSI simulation tools are used. Simulators for fabrication process characteristics are employed to either reveal electrical faults or verify circuit designs.[4] VLSI systems also undergo physical inspection and test upon fabrication and packaging. These evaluations are usually performed with expensive, special-purpose testing equipment. Since there can be fabrication or packaging variations (e.g., flaws) in individual copies of the same mass-produced circuit, statistical samples of production chips must be evaluated to determine current yield and ways to improve it. Widely used software systems are not tested in this manner.

Design libraries are used differently in the two disciplines. Some VLSI design methods (e.g., standard cell, gate arrays) consist of selecting the implementation of functions from a library and wiring them up to

[3] Real-time software systems often have performance requirements for timing behavior similar to those for VLSI systems.

[4] A current stumbling block in the use of VLSI simulation tools is the lack of a common language or engineering methodology that coordinates interaction between them.

form the system. While software systems make extensive use of standard subroutine libraries, the design of the system is not influenced by the libraries to the same extent. Note however that early link and load systems were not unlike modern gate array systems, in that standard functions were selected from libraries and linked together to form an executable program.

In summary, the similarities between software and VLSI systems arise from the ways system engineering work is organized and performed. The development of either software or VLSI systems usually requires the articulation, transformation, and evaluation of various system descriptions within a single computational medium. On the other hand, the differences between VLSI and software systems arise because VLSI circuit descriptions must also be transformed (fabricated) and evaluated across computational and physical media.

NEAR TERM PROSPECTS AND FUTURE SCENARIOS

6.1 Available Now

For software systems, there already exist good operating and programming system environments. These include Bell Lab's UNIX [33], the Programmer's Work Bench [14], and the InterLisp System [42]. Other systems supporting various activities in the software life cycle can be found in currently available surveys [19, 45]. A comprehensive, up-to-date survey on available software development methodologies can be found in [15].

For VLSI, many integrated environments for circuit design exist, but these support a fixed, and often proprietary, methodology (e.g., IBM's Engineering Design System and various vendors' design systems). These support the entire range of circuit design activities, but are often tuned for a particular implementation environment, thus limiting the transportability of designs. To our knowledge, there is no integrated CAD environment that provides either complete system life cycle coverage or independence from circuit design methodology.

On the other hand, workstation-based systems are beginning to emerge that place complete sets of design tools in the hands of individual designers [36]. However, the tools are presented as a loose confederation, and frequently lack integration. Workstation system designers are directing a great deal of effort towards building their tool sets around an integrated system development database to reduce these problems.

Except for the earliest and most strategic activities of the development cycle (product requirements, system specification, etc.), where the most commonality in development approaches is to be found, little has been done to combine the hardware and software development environments. As has been mentioned in Section 3 above, tools are needed for the combined environment to aid in partitioning a system into its hardware and software components. Fundamental research remains to be done to better understand the process of system partitioning.

6.2 Desired Future Capabilities

Expert systems technology as well as other artificial intelligence techniques will be applied in future development environments. The environment will not only adapt to the system developer/user as she/he progresses from novice to expert, but it will also take a more active role in aiding system development decisions. The system developer/user can concentrate on the higher level strategic issues of design, such as specification of functions to be implemented, while the environment does the low-level tasks, such as choosing detailed structures for implementing these functions. Additionally, the environment should be able to keep track of the developer/user interactions with various system components and provide a range of problem-solving or diagnostic help services if needed.

A major problem with current development environments is that there is little support for making system components reusable across a wide spectrum of applications. Examples of reusable components include technology-independent cell libraries for VLSI circuits and machine-independent subroutine packages for software. Continued research must be directed towards the issues of how to make portable systems. In particular, we must develop ways to specify the behavior of a system component in a technology independent manner.

Related to the issue of technology independence is support for families of implementations and families of algorithms. Given a specification of the performance requirements of a system, a development environment will eventually be able to select among alternative implementations of the same functional unit. The environment should enable a designer to explore the space of alternative implementations of his design, with different combinations of performance metrics, such as area, speed, and power tradeoffs.

Finally, when all of these facilities are incorporated into the "ultimate" development environment, the question arises as to how to keep the whole thing manageable. The performance and usability of the system must remain reasonable, even as more powerful facilities are

being added. An ability to tailor the development environment to specific projects and technologies should help to maintain adequate performance. Techniques for adapting the system to the expertise of system engineers, project managers, and other users must be developed, to avoid overwhelming them with the sheer complexity of the computational environment.

7. SUMMARY

Developing larger, more complex VLSI or software systems will remain very costly. Automated environments are expensive and resource intensive [32]. Accordingly, automated environments which support the life cycle engineering of complex systems will probably not reduce these costs; instead they may help keep them from rising too quickly. Substantial cost savings and productivity boosts are still elusive. Thus, we might look beyond automated system engineering tools to find the breakthroughs we seek. Perhaps the answers lie in discovering new ways to organize and perform system development work, and to manage the flow of production resources. Nonetheless, the tools we can identify are becoming more important.

In this report, we described the desirable features of environments for system (hardware and software) development. We emphasized the need for development environments to support the entire system life cycle, and discussed the significant events that constitute that life cycle. Important components of the environment were identified. We also described the large degree of similarity among development environments, and pointed out the few differences. Capabilities of existing development environments were reviewed, and we offered speculations about the future.

One of the goals of the workshop was to identify where VLSI and software engineering intersect. With respect to development environments, we discovered that the intersection was large, and that systems that support VLSI and software development share many of the same requirements and need to provide the same solutions. Development environments of the future will support complete system life cycle engineering, where components of the system can be realized as either hardware or software subsystems. Future environments will take a more active role in the development process, and will support families of alternative system implementations, with a greater emphasis on portability of designs. Builders of such environments will be faced with the challenge of providing a powerful, flexible system that is both easy to use and does the job with little overhead.

ACKNOWLEDGMENTS

Many people contributed to the lively discussions that led to this report. The members of the Development Environments Working Group included Duane Adams, Marc Davio, Jacob Katzenelson, John Kellum, Steven Reiss, Gilles Serrero, and Jack Thomas.

REFERENCES

1. J. Allen, Requirements for a Research-Oriented IC Design System, *Proceedings of the Caltech Conference on Very Large Scale Integration,* California Institute of Technology, 1979, pp. 253–258.
2. J. Allen and P. Penfield, Jr., VLSI Design Automation Activities at M.I.T., *Proceedings of the 1981 IEEE International Symposium on Circuits and Systems,* IEEE Circuits and Systems Society, 1981, pp. 648.
3. R. Ayers, A Language Processor and a Sample Language, Silicon Structures Project File No. 2276, Computer Science Department, California Institute of Technology (June 1978).
4. R. Ayers, IC Specification Language, *Proceedings of the 16th Annual Design Automation Conference,* 1979, pp. 307–309.
5. R. M. Balzer, Transformational Implementation: An Example, *IEEE Transactions on Software Engineering* SE-7, 3–14 (Jan 1981).
6. J. Batali and A. Hartheimer, The Design Procedure Manual, V.L.S.I. Memo 80-31, Massachusetts Institute of Technology (Sept 1980).
7. F. van den Bosch et al., Evaluation of Software Development Life Cycle Methodology Implementation, *Software Engineering Notes* 7, 45–61 (Feb 1982).
8. R. E. Bryant, A Switch-Level Simulation Model for Integrated Logic Circuits, LCS/TR-249, Massachusetts Institute of Technology (1981).
9. J. N. Buxton, An Informal Bibliography on Programming Support Environments, *SIGPLAN NOTICES* 15, 17–30 (Dec 1980).
10. IEEE Transactions on Circuits and Systems, Special issue on Computer-Aided Design of VLSI Systems (July, 1981).
11. E. Cohen, Program Reference for SPICE2, Memorandum No. ERL-M592, Electronics Research Laboratory, University of California, Berkeley (June 1976).
12. E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1976.
13. Department of Defense, Reference Manual for the Ada Programming Language, Proposed Standard Document July 1980, U.S. Government Printing Office, Order No. L008-000-00354-8.
14. T. A. Dolotta, R. C. Haight, and J. R. Mashey, The Programmer's Workbench, *Bell System Technical J.* 57, 2177–2200 (July–Aug 1978).
15. P. Freeman and A. I. Wasserman, Software Development Methodologies and Ada(tm), technical report, University of California, Irvine, CA (1982).
16. N. Goldman and D. Wile, GIST (Internal Report), un-

published, USC Information Sciences Institute (Sept 1980).
17. I. P. Goldstein and D. G. Bobrow, A Layered Approach to Software Design, Xerox PARC CSL-5-80, submitted for publication.
18. J. Guttag, E. Horowitz, and D. Musser, Abstract Data Types and Software Validation, *Commun. ACM* 21, 1048–64 (1978).
19. H. Hunke, (ed), *Software Engineering Environments,* North-Holland, New York, 1981.
20. R. H. Katz, A Database Approach for Managing VLSI Design Data, 19th ACM/IEEE Design Automation Conf., Las Vegas, NV (June 1982).
21. R. Kling and W. Scacchi, Recurrent Dilemmas of Computer Use in Complex Organizations, National Computer Conference, AFIPS Press, 1979, Vol. 48, pp. 107–115.
22. R. Kling and W. Scacchi, Computing as social action: The social dynamics of computing in complex organizations, *Advances in Computers,* Academic, New York, 1980, Vol. 19, pp. 249–327.
23. R. Kling and W. Scacchi, The web of computing, *Advances in Computers,* Academic, New York, 1982, Vol. 21, pp. 3–85.
24. H. T. Kung, Let's Design Algorithms For VLSI Systems, *Proceedings of the Caltech Conference on Very Large Scale Integration,* Computer Science Department, California Institute of Technology, 1979, pp. 65–90.
25. L. W. Leyking, Data Base Considerations for VLSI, *Proceedings of the Caltech Conference on Very Large Scale Integration,* California Institute of Technology, 1979, pp. 275–302.
26. J. McCarthy, Comments on 'The State of Technology in Artificial Intelligence,' *Research Directions in Software Technology,* M.I.T. Press, Cambridge, MA, 1979, pp. 814–815.
27. D. W. McSweeney, Timing Verification of VLSI Logic Circuits, in (Paul Penfield, Jr., ed.), *Proceedings, Conference on Advanced Research in VLSI,* Massachusetts Institute of Technology, Artech House, Inc., 1982, pp. 63–66.
28. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems,* Addison Wesley, Boston, 1980, pp. 115–127.
29. J. K. Ousterhout, CAESAR: An Interactive Editor for VLSI Layout, Digest of Papers: CompCon Spring 82, IEEE Computer Society, 1982, pp. 300–301.
30. P. Petit, Chipmonk: An Interactive VLSI Layout Tool, Digest of Papers: CompCon Spring 82, IEEE Computer Society, 1982, pp. 302–304.
31. A. Pnueli, The Temporal Semantics of Concurrent Programs, *Semantics of Concurrent Computation* (Kahn, ed.), Springer Lecture Notes in Computer Science, Springer, New York, 1979, pp. 1–20.
32. D. Prentice, An Analysis of Software Development Environments, *Software Engineering Notes* 1981, vol. 6.
33. D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, *Bell System Technical J.* 57, 1905–1930 (July–Aug 1978).
34. W. Scacchi, *The Process of Innovation in Computing: A Study of the Social Dynamics of Computing,* Ph.D. dissertation, Dept. of Information and Computer Science, University of California, Irvine (1981).
35. W. Scacchi, Managing Software Engineering Projects: A Social Analysis, *IEEE Trans. Soft. Engr.,* Vol. SE-10, (Jan 1984).
36. M. Schindler, Computer-Aided Engineering Comes of Age, *Electronic Design* (Nov 11, 1982).
37. P. A. Subrahmanyam, A Basis for a Theory of Program Synthesis, *Proceedings of the First Annual National Conference On Artificial Intelligence* (AAAI) (August 1980), pp. 74–76.
38. P. A. Subrahmanyam, Automatable Paradigms for Software-Hardware Design: Language Issues, J. Rader (ed.), *IEEE Workshop on VLSI and Software Engineering,* IEEE, 1982. Also available as University of Utah Technical Report UTEC-82-096, (Sept 1982).
39. P. A. Subrahmanyam, *An Automatic/Interactive Software Development System: Formal Basis and Design,* North-Holland, Amsterdam, 1982.
40. P. A. Subrahmanyam, From Anna+ to Ada: Automating the Synthesis of Ada Package and Task Bodies, Technical Report Internal Report, University of Utah (March 1982).
41. P. A. Subrahmanyam, On Automating the Computation of Approximate, Concrete, and Asymptotic Complexity Measures of VLSI Designs (to appear), Technical Report UTEC-82-095, Dept. of Computer Science, University of Utah (Oct 1982).
42. W. Teitelman and L. Masinter, The Interlisp Programming Environment, *IEEE Computer* 14, 25–33 (April 1981).
43. W. M. vanCleemput, Hierarchical Design for VLSI: Problems and Advantages, *Proceedings of the Caltech Conference on Very Large Scale Integration,* California Institute of Technology, 1979, pp. 259–274.
44. A. Vladimirescu and D. O. Pederson, A Computer Program for the Simulation of Large-Scale-Integrated Circuits, *Proceedings of the 1981 IEEE International Symposium on Circuits and Systems,* IEEE Circuits and Systems Society, 1981, pp. 111–113.
45. A. I. Wasserman, *Software Development Environments,* IEEE Computer Society Press, Los Alamitos, CA, 1981.
46. D. Wile, POPART: A Producer of Parsers and Related Tools, System Builder's Manual, Internal report, USC Information Sciences Institute, Marina Del Rey, CA (June 1980).