System Level Hardware Module Generation
Mani B. Srivastava, Member, IEEE, Brian C. Richards, and Robert W. Brodersen, Fellow, IEEE

Abstract—In complex modern day electronic systems, far more time is spent in designing the boards, writing the software to drive and integrate the hardware, and other system level issues, than is spent in designing any application-specific IC's. The design process needs to be much more efficient than at the chip level. Unfortunately, most of the research in computer-aided design has been focused on the more glamorous ASIC design problem, as a result of which the design methodologies and tools at the system level are much more primitive than at the chip level.

We have developed a design framework for application-specific systems, called SIERA, that addresses the higher level aspects of system design, including multichip design issues at the board-level, and hardware-software co-design and integration, in addition to the design of individual ASIC's. SIERA allows rapid prototyping of multiboard systems where the functionality is implemented using a mix of dedicated hardware modules and ASIC's, as well as software running on programmable hardware modules.

A key step in this design methodology provided by SIERA is that of generating the physical implementation of the system hardware from a description of the system architecture. The problem of this paper is that the chip level is referred to as silicon assembly or silicon compilation. In this paper we address this problem at the system level, and describe how the generation and interfacing of board-level modules, board-level physical design, simulation of custom boards, and the overall management of board design are handled in SIERA. While some of the problems could be solved by adapting or extending techniques from the existing ASIC design tools, others required new approaches. Case-studies of several real-life applications are also presented to demonstrate the effectiveness of the board-level physical design methodology embodied in SIERA compared to the traditional PCB design systems.

I. INTRODUCTION

Silicon compilers and silicon assemblers embodying techniques like logic synthesis, parameterized module generation, and reusable cell libraries have considerably shortened the design time of application-specific IC's (ASIC's), and have almost completely automated the ASIC design process [1]. In certain application domains it is even possible to automatically generate ASIC's from a behavioral description of the algorithm using techniques like synthesis [2]–[5] and retargetable compilation to an architecture template [6]. In short, there has been a dramatic reduction in the design time of ASIC's at all levels of the design process.

However, the same cannot be said about system level design. The functionality of most systems is implemented by a mix of hardware and software, and off-shelf IC's and ASIC's, and spans several levels of packaging, including chip carriers, MCM's, PCB's, and back-planes. Our experience with several real-time application-specific systems has shown that most of the design time is spent in system level issues, such as design of boards, software design, integration and testing of system hardware and software, etc., and not in the ASIC's themselves. These higher-level aspects of system design have not been adequately addressed in CAD research. Not only are some of these problems quite difficult, but also problems like board-level physical design have been considered to be less glamorous than the corresponding problems at the chip level, and therefore have been neglected. As a result, even structural/physical design techniques for PCB's are quite primitive when compared to silicon assemblers and compilers available for ASIC design.

We have developed SIERA, a system design environment, embodying a design methodology that addresses these higher level aspects of system design in addition to the design of individual IC's. Taking a unified view of the hardware and software components constituting a system, SIERA implements the system functionality, described by a network of communicating state machines (processes), as a mix of hardware and software on multiple custom printed-circuit boards. The custom boards themselves are composed of dedicated as well as software-programmable hardware modules, using both off-the-shelf IC's and ASIC's.

An important step in the system design methodology in SIERA is that of generating the physical implementation of the system hardware from a description of the system architecture that is generated by the higher level architecture generation and synthesis tools. The analogue of this hardware module generation problem at the chip level is referred to as silicon assembly or silicon compilation. In this paper we address this problem at the system level, and describe how the generation and interfacing of board-level modules, board-level physical design, simulation of custom boards, and the overall management of board design are handled in SIERA. Although a brief overview of SIERA is presented later in the paper, the reader is referred to [7], [8] for a detailed description of the other higher level steps, such as architecture generation, software module generation, and interface synthesis, in SIERA's system design methodology.

II. PREVIOUS WORK IN HARDWARE MODULE GENERATION

Most of the research work in computer aids for hardware module generation has been done at the chip level where many general-purpose as well as specialized module layout...
and synthesis tools are available, both from the universities and the industry. Examples include stand-alone layout tools such as [9] and sequential and logic synthesis tools such as [10], [11], as well as module generators like datapath compilers and controller synthesis that are integrated in CAD environments such as LAGER [1], [12], CATHEDRAL [4], and Bit Serial Compiler [13]. These module generators range in sophistication from generating the physical implementation from a parameterized or procedural connectivity and placement specification, to generating the physical implementation from higher-level specifications such as logic, register-transfer level, and timing diagram descriptions. Due to the availability of these module generators the problem is largely considered solved at the chip level, except for niche areas like analog chips.

In contrast to the variety of sophisticated module generation aids available to chip designers, the design tools available to board designers are quite rudimentary. Typically the only design aid which board designers currently have are the commercial PCB schematic capture, placement, and routing packages available from vendors such as OrCAD, Valid, Racal-Redac, etc. These board design packages typically provide the designer with the ability to enter the design as a netlist of chips, and then place and route the flattened netlist.

However hardware design at such a low level is not sufficient, as is indicated by the experience at the chip level. Describing a complex board as a hierarchical netlist of chips is extremely cumbersome—it is equivalent to designing a chip using a library of primitive gates such as NAND, OR etc. The answer at the chip level has been to raise the level of abstraction, and view a chip to be composed of high-level modules that either come from a library of parameterized reusable modules (adders, multipliers, RAM’s, etc.), or are synthesized from a behavioral description (FSM controllers, decoders etc.). This two-pronged strategy has proven very successful for the design of ASIC’s because it raises the design abstract to a sufficiently high level while providing the benefits of reusable and possibly hand-optimized library modules, as well as the use of synthesis wherever appropriate.

Board-level design tools have lagged behind this use of higher levels of abstraction. The key requirement for realizing such a board design philosophy are an extensive library module, a variety of layout and synthesis tools, and a design management framework. Our focus is on the solutions to these three problems as implemented in SIERA.

One existing board-level tool that needs to be mentioned here for its ability to allow board design at a higher level of abstraction is the microprocessor “configurator,” or Micon, system from CMU [14]. It is a CAD environment that uses an AI expert system to generate a single-board computer. The computers designed by Micon consist of a microprocessor, ROM, SRAM or DRAM, cache memory if supported by processor, serial and parallel I/O, standard bus interface, and circuits for support functions such as address decoding, clock generation, etc. The user specifies the type of the microprocessor, amount and type of memory, and the number and type of I/O devices. The knowledge about the components and the microprocessor system structure in the rule base is then used to generate a design satisfying board size, cost, and system reliability. Although Micon raises the level of design, its use is restricted to single-board computers with a restricted architecture. SIERA, in contrast, addresses a much wider class of boards than just single processor computer boards. The application domain specificity of Micon allows it to use knowledge-based optimization techniques. On the other hand, SIERA leaves optimizations requiring application-specific knowledge to the designer while providing support for general module generation, synthesis, and parameterization.

III. SYSTEM DESIGN METHODOLOGY

IN SIERA: AN OVERVIEW

SIERA provides a vertically integrated design methodology that supports all stages involved in the development of application-specific systems from high-level description to board and software generation. Due to our “application-driven” approach, we placed much more emphasis on the ability to model and implement the system being designed than on its analysis. As shown in Fig. 1, the design of a system in SIERA is divided into two distinct phases—architecture generation and module generation.

Architecture generation refers to the process of generating a suitable architecture for the system starting from a high level description. In SIERA, the system is specified by the user as a parameterized network of concurrent processes that communicate using single-reader and single-writer FIFO channels that connect input and output ports of processes. The ports are characterized by a protocol that specifies the behavior in case the corresponding channel is not ready for communication, and the channels are characterized by a buffer depth. A multiplicity of languages including suitably adapted versions of Promela [15] and Silage [16] can be used to describe the behavior of the processes, which is then translated to an intermediate format. The topology of the communication between the processes is entered textually using the language SDL, or schematically as a block diagram—both of which are also translated to the intermediate form. A set of VHDL packages is used to simulate the network of processes specification according
to the underlying abstract computation model. The network of processes, after suitable optimizing transformations, is manually partitioned on to a layered architectural template. The architecture template defines a scalable and parameterized organization of the dedicated hardware modules, and software modules running on programmable processors, in a system as a multilayered hierarchy of busses implemented across multiple application-specific custom boards.

Module generation refers to the physical implementation of a system given its architecture as a composition of hardware and software modules, such as may be generated by the architecture generation phase of SIERA. At the system level, unlike at the chip level, there are two aspects to the module generation problem—hardware and software. From a hardware perspective the term architecture refers to a set of hardware modules (processors) interconnected in a specific topology. From the software perspective the term architecture refers to a set of software modules (processes) that communicate in a specific topology, and are usually mapped many-to-one to the hardware modules. A complete physical implementation of a system requires that both the hardware and software views of system architecture be implemented. The resulting subproblems of hardware module generation and software module generation, although not completely isolated in reality, are addressed separately in SIERA. A mix of parameterized board-level hardware module libraries, and pre-existing synthesis tools is used in the hardware module generation phase to generate the board netlists. Tiling, user-defined and commercial tools are used for the floorplanning, placement and routing of the boards. The software modules are implemented in the software module generation phase as processes running under the control of real-time OS kernels on programmable processor modules. Libraries of device drivers and interprocess communication mechanisms are used for this.

An important point that needs to be emphasized is the separation of the module generation phase from the architecture generation phase. The former problem is easier in many ways, and one expects the design aids and methodologies for it to be relatively general-purpose. This separation of the two phases allows the module generation aids to be used directly by the user, or as a back-end by other higher-level tools for architecture generation.

The emphasis in SIERA is more on facilitating the management of the system design complexity by discouraging the designer from adopting ad hoc low-level approaches, and supporting modularity, reusability, uniformity, and abstraction [17]. One of the key techniques adopted to enforce these design principles is the extensive use of libraries which serve as repositories of optimized and reusable hardware and software modules. The policies associated with each library enforce the uniformity of its member modules. In order to minimize the size of these libraries, extensive use is made of parameterization. For example, board-level hardware modules such as embeddable computer core modules are designed so that the memory size, I/O interface, and other attributes can be varied by passing appropriate parameter values.

Since ASIC's are an important part of many systems, a seamless integration of board-level hardware design with that at the chip level is also considered important in SIERA. This is accomplished by using database mechanisms and policies that are consistent with those used in LAGER [1, 12], our CAD environment for ASIC design. As a result SIERA presents the designers with the same structure design environment at the board-level as at the chip level.

SIERA is an open system in that new modules can be added to the libraries, and new tools can be added to manipulate the design. This is intended to be done by expert users who are familiar with the low-level design issues in hardware or software design, the design constraints, and the underlying database policies for tool integration.

The remainder of this paper presents a detailed description of and techniques used in the Hardware Module Generation phase of SIERA. A detailed description of the Architecture Generation and Software Module Generation phases of SIERA is available in [7].

IV. GENERATION OF HARDWARE MODULES IN SIERA—THE DESIGN FRAMEWORK

The design abstraction of a hierarchical netlist of individual chips that is provided by the currently available board design packages is at too low a level for describing complex boards. Whereas chips are routinely designed by composing high-level modules that either come from a library of parameterized reusable modules (adders, multipliers, RAM's etc.), or are synthesized from a behavioral description (FSM controllers, decoders etc.), board-level design tools have lagged behind this use of higher levels of abstraction. In SIERA these lessons learned at the chip level have been applied to the board-level hardware module generation problem. Of course, different layout and synthesis tools, and libraries are needed at the board-level. The resultant methodology for the generation of board-level hardware modules in SIERA is shown in Fig. 2. The key requirements for realizing this design philosophy are an extensive module library, a variety of layout and synthesis tools, and a design management framework. SIERA's design management framework, which is an extension of the chip-level design management framework in LAGER, has three key components:

— a database with a procedural interface for information access by the tools
— standard policies to represent various types of design information in the database
— a design manager to provide orderly access to the database by the user and the tools.

A. The Design Database

SIERA uses OCT [18], an object-oriented database from Berkeley, for storing persistent as well as transient information about the various aspects of an evolving system design. While hiding the actual storage mechanism, OCT provides procedural interfaces in various programming languages, such as C and C++, using which the tools in SIERA store and retrieve design information. Object-oriented databases other than OCT can be used—we chose OCT because of familiarity. Similar to other object-oriented databases specialized for electronic design, OCT represents a design as a collection of objects.
such as cells, facet, instances, nets, terminals, properties, bags, layers, geometric objects, etc., with the relations between objects being defined by attaching one object to another to form a directed graph.

B. SIERA's Policies for Design Representation in the Database

A policy specifies what directed graph structures of database objects are legal, and the semantics of these legal structures. Although OCT provides a set of standard schematic, symbolic, and physical [19] policies, they were developed for nonparameterized chip design, and proved inadequate at handling system level parameterized hardware design in SIERA.

1) The structure.master Policy: A hierarchy of database views following the structure.master policy is the initial representation of the system hardware in SIERA as stored in the database. This policy has a close correspondence to the language SDL (structure description language) which is used in SIERA to specify structures of parameterized designs. Parameterization is an efficient way for representing a class of modules in a well-defined manner and enhances design reusability. For example, a memory module can have the word width and memory size as parameters. When a parameterized module is instantiated, each parameter is bound to a specific value.

The structure.master view of a cell can have parameters defined for it. These parameters, called formal parameters, have optional default values defined for them. When a cell is instantiated as a subcell in another cell higher up in the design hierarchy, values can be bound to these parameters in the form of actual parameters. Parameter values in general are functions of the parameters of the container cell, thus allowing an entire design hierarchy to be parameterized in a top-down fashion. Some parameter names are reserved in SIERA to provide uniform module interface and for passing information to tools operating on the design. For example, PACKAGENAME is reserved to name the physical package for a module. Most module parameters are, however, defined by the designers. The designer of a processor module might parameterize the design along dimensions such as RAM/ROM size, number of I/O ports etc.

Attributes, with optional values that are functions of the cell parameters, can be attached to the facet of a cell, to its formal parameters, to its formal terminals, and to nets and instances contained in the facet. While parameters are used to pass information across the design hierarchy, attributes are used to annotate design objects within a cell with complex data structure values. SIERA uses several reserved attribute names with special meaning to various tools. For example, the DIRECTION attribute is used to specify the direction of a terminal. Similarly, X, Y, and T attributes are used to specify the position and orientation of a subcell.

Cell parameters and attributes can also be used to affect several other aspects of a design. For example, parameterized indexed sets of instances can be created allowing a single description to describe a class of circuits. The connectivity specification can also be parameterized by associating an arbitrarily complex mapping between terminal and net indices to every terminal-net connection. The SDL language provides syntactic mechanisms for expressing this mapping.

A key to representing parameterized structure using the structure.master policy is the expression of actual parameter and attribute values as functions of parameters of the container cell. We encode these functions using a subset of Lisp, together with a designer-extensible library of functions useful for expressing parameterized hardware structures. A Lisp interpreter evaluates these functions at the run-time. This allows complex computations, and OS/filesystem interaction to be done as part of parameter and attribute evaluation, and enables module generation and optimization algorithms that make use of application-specific knowledge to be embedded in a SDL file. For example, the SDL file for a processor module may include a rule-based selection of parts based on cost and performance constraints specified via parameters.

In addition to the structure of the cell, the structure.master policy also allows the view to be annotated with the names of tools that would generate an implementation of the cell. There are two types of tools: STRUCTURE.PROCESSOR tools that create or modify the structure of a cell, and LAYOUT.GENERATOR tools that generate the physical implementation.

2) The structure.instance Policy: This policy is used to represent a nonparameterized design hierarchy. It is obtained from a parameterized design hierarchy following the structure.master policy by evaluating all the parameters and attributes, and also expanding all the terminal and net mappings. The resulting structure.instance hierarchy represents nonparameterized structure with all the bus nets fully expanded, and contains all the information needed to generate a physical implementation.

3) The physical Policy: This is the simplest of the three policies—its purpose is to represent the physical implementation of the design as a geometrical structure consisting of terminal, layer, geometric, and instance objects. A pure geometric representation is usually not sufficient because the design may be only partially implemented even after a layout-generator has been run. Therefore connectivity information for unimplemented nets is also retained in a physical view in SIERA. This allows boards to be designed in SIERA by using hierarchical placement via intermediate specialized layout generators, and then doing a one-shot routing of the flattened hierarchy at the end. This approach has proven to be much more convenient to the designers than the nonhierarchical placement approach adopted in commercial board design systems.
C. Design Manager *DMoot*

SIERA's central design manager is *DMoot* which manages access to the design database by the user as well as the module generation tools. *DMoot* automates the generation of a hierarchical design which may require different tools to create different parts of the hierarchy by calling the required tools in the correct order while making use of time-stamp checking and parameter comparison techniques to avoid regeneration. Most importantly, it presents the designer with a common front-end to all the tools.

Fig. 3 shows the design flow as orchestrated by *DMoot*. It transforms the initial hierarchical textual representation of the parameterized design in SDL to a hierarchy of *structure_master* views. Next, with the aid of a built-in Lisp interpreter and a set of tools called structure-processors, it creates a hierarchy of *structure_instance* views. This is done by passing parameter values down the *structure_master* hierarchy and creating the *structure_instance* views in a bottom-up fashion by expanding subcell arrays and composite nets or busses. At each stage of the hierarchy after creating the *structure_instance* view *DMoot* cells special tools, referred to as structure-processors, to modify that view or the *structure_instance* subtree rooted at that view. Often these tools are synthesis tools that use a behavioral description to generate a net-list that implements the functionality. Optionally *DMoot* can also flatten the *structure_instance* hierarchy at any given point. Finally *DMoot* traverses the resulting *structure_instance* hierarchy in a bottom-up fashion and calls special tools, referred to as layout-generators, at each stage to generate a physical implementation. In case of boards these tools do the placement and routing to create the photo-ploter files required for board fabrication. *DMoot* provides hooks to let the designer force the generation of or ignore the generation of a selected part of the hierarchy, thus by-passing its built-in time-stamp checking and parameter comparison mechanism. A sophisticated search mechanism is used by *DMoot* and other tools to access files from libraries.

1) Handling of System Packaging Hierarchy by *DMoot*:

A key distinction between systems and chips is the existence of multiple types of packaging in systems. Multiple levels of packaging require special handling during the design management process. A chip package, for example a PGA package, is usually the lowest level of packaging—it contains a silicon die and uses bonding wires and pins to connect pads on the die to copper traces on a board. For improved electrical performance, spatial efficiency and economic reasons complex systems make use of a variety of packaging levels organized in a packaging hierarchy. According to the industry standard jargon, the chip die is the level 0 packaging. The level 1 packaging is formed by the packages for single chips and discrete parts. MCM's composed of multiple silicon dies that are mechanically and electrically bonded on a substrate using a variety of emerging technologies form what is informally referred to as level 1.5 of the system packaging hierarchy. Level 2 of the packaging hierarchy is formed by printed circuit-boards with copper traces in an epoxy laminate to electrically connect lower level packages that are soldered to the boards. Level 3 packaging, usually the top level of the system packaging hierarchy, is the card-cage with a backplane bus or cables to interconnect multiple boards.

Different tools treat a design with a multilevel packaging hierarchy differently. A board placement tool treats MCM’s and chips as primitives, and requires that the design hierarchy below these packaging levels be truncated. Similarly a chip router tool connects modules inside a chip package—this requires that the design hierarchy above the chip level packages be ignored. A simulator on the other hand is not concerned with the packaging hierarchy at all—it is concerned only with the functional hierarchy. This suggests that in addition to super-imposing a packaging hierarchy on top of a design hierarchy, one also needs the ability to extract parts of a design hierarchy that are below, or above, or between, certain packaging levels.

*DMoot* provides support for handling of packaging hierarchies by following a similar standard for expressing levels of packaging as described above. Design cells that correspond to mechanically packaged entities are required to have an attribute called PACKAGECLASS, and an optional parameter called PACKAGE_NAME. The value of the PACKAGECLASS attribute indicates the level of packaging hierarchy whereas the value of the parameter PACKAGE_NAME is used to select a specific package. For example, the PACKAGECLASS may be PCB to indicate that the package is meant to be contained by a board, and the PACKAGE_NAME may be DIP20 to indicate a particular package. Using the PACKAGECLASS attribute *DMoot* can be forced to ignore the design hierarchy below a certain packaging level. This is done by ignoring the sub-structures inside cells that have PACKAGECLASS attributes belonging to a specified set of values. Similarly, selected portions of the design hierarchy can be flattened down to a specified packaging level. For example, a multiboard system can be flattened to the level of gates. This is the opposite of partitioning—synthesis and partitioning tools create these packaging hierarchies.

V. BOARD LEVEL LAYOUT GENERATORS IN SIERA

Board-level layout generators are tools that given a net list of chips or macro modules, do the placement and/or routing. The open architecture of the design management framework provided by *DMoot* allows different layout-generators sup-
porting different layout styles to be easily integrated. Just as at the chip level, different placement and routing algorithms are appropriate for different types of modules. For example, a memory module is very regular, and a tiling based placement based on abutment of subcells might be appropriate for it. On the other hand, at higher levels of the hierarchy one typically has a collection of polygonal macro blocks, for which a more general-purpose placement strategy is needed.

The set of layout generation tools currently available in SIERA is small but nevertheless provides capabilities superior to existing board design packages. The current set of tools supports a hierarchical placement of the design together with one-shot routing of the flattened and placed design hierarchy.

A. psx: Package Symbol Generator

Psx is the layout generator used to generate physical geometries for board-level components such as chips, discrete parts, MCM’s, SIMM’s, etc. The functionality of psx is quite simple—it uses the package name specified by the facet attribute PACKAGENAME to search for the physical package information (copper shapes on various board layers, pin locations, package bounding box and package height) in the package library corresponding to the facet attribute PACKAGECLASS, and adds this information to the structure instance view of the component. Psx allows encapsulation of the physical package information in a separate library and thus simplifies design management when many different components use the same package.

B. pfp: PCB Floor-Planner and Placement Tool

The board floor-planner pfp is the most important of the available layout-generators. It is used to place the submodules within a module according to a variety of methods:

1) Absolute placement by specifying the position (X, Y) and rotation (T) of subcells of a module.
2) Relative placement—tiling of subcells in a row-major or a column-major fashion—by treating X and Y to be relative coordinates.
3) Automatic placement using simulated annealing.
4) Placement using previously saved floorplans.
5) Interactive placement and modification of an initial placement obtained using the above four techniques.

VEM, the graphical editor for OCT, is used to manipulate the placement.

The real power of pfp comes from the fact that the attributes X, Y and T for specifying subcell placement can be arbitrary Lisp expressions that use the parameters of the module. Using this mechanism sophisticated absolute or tiling based parameterized placement can be specified. In effect this allows custom module placement algorithms to be embedded in a SDL file. Pfp can optionally “flip” the resulting floorplan horizontally or vertically— unlike the simple “mirroring” transformation which is not meaningful in board layout, “flipping” ensures that the subcells are not mirrored while the floorplan is mirrored. Pfp also provides options that allow control over router constraints, such as ensuring that the pins of the parts lie on a routing grid.

C. oct2pcb: An Interface to PCB Routers

At present there is no built-in board-router available in SIERA. Instead commercial printed circuit-board-routers from Racal-Redac, Valid, and SciCards have been integrated through an interface provided by oct2pcb which can be used as a layout generator on its own, or can be accessed via pfp. Since the commercial routers cannot do hierarchical routing, oct2pcb first flattens the design hierarchy and then converts the resulting flattened netlist into ascii files required by the routers. In this process oct2pcb also maps the vendor-independent packages in SIERA to vendor-specific packages required by the commercial router. In addition to mapping the names, this process also involves geometric translation and rotation, as well as a transformation of the pin numbers from SIERA’s package library to the vendor package library. Oct2pcb also generates part list and connectivity reports, and detects some types of netlist errors.

A one-way interface to routers is by no means sufficient—one also needs the ability to obtain physical information generated by the router, and put it in the design database so that tools such as extractors can access it in a uniform fashion. This task is made difficult because there is no standard interchange format that is used by commercial routers. SIERA has a limited capability to back-annotate the database from the outputs of Racal-Redac and Valid routers.

VI. MODULE GENERATION FROM BEHAVIORAL SPECIFICATIONS

Many modules in the top-level architecture of a board are more conveniently described behaviorally, and it is often possible to synthesize efficient hardware for them. This is particularly true for modules such as random logic, memory address decoders, bus-interface logic, etc., for which special-purpose synthesis tools can be used to transform the behavioral representation into a netlist of chips at the board level. Several such module generators are available in SIERA for use as structure-processors to synthesize the structure of a design entity. The tools described here are used at lower levels of the structural hierarchy of the system—while they cannot synthesize the system hardware as a whole, they can certainly synthesize parts of the system from a suitable behavioral description.

A. Mapping Random Logic to PLD's and FPGA's

Board-level modules are often best implemented on field programmable devices such as PAL's, PLD's and FPGA's (for example the ones from Actel and Xilinx). Modules such as address decoders, bus control logic, glue logic, interface logic, etc., fall in this category. Implementing such modules usingSSI parts is too inefficient in board area while implementing them as ASIC's is usually not cost effective. Field programmable devices provide a nice compromise and the ability to map a behavioral level description to a net-list of these devices is very desirable.

2 The integration of the Finesse router from SciCards into SIERA was done by John Granacki at USC’s Information Science Institute.

3 This seems to be typical of all available PCB routers.
With either synthesis strategy, the output is a netlist of basic blocks. This netlist is then merged with the structurally specified part of the initial description, and the resultant flattened netlist of basic blocks is partitioned to a board-level network of identical chips (FPGA's, PAL's, PLD's, etc.). The output is a list of these chips together with specifications for each one of them in an appropriate format, such as ABEL for PAL's, XNF for Xilinx FPGA's, ADF for Actel FPGA's, and structural VHDL. Some FPGA devices also provide special basic blocks—such as the decoder blocks on Xilinx's XC4000 series FPGA's. The mixed behavioral and structural description allows these special blocks to be used structurally through the macro block library.

B. Generation of ASIC's from Behavioral Description

Many system level functions, such as DSP subsystems and communication controllers, are best implemented as ASIC's because of performance and cost reasons. Therefore it is very important that a system or board-level hardware module generation framework provides the ability to implement board-level modules as ASIC's. Due to the commonality of the database and the design manager used in the board-level hardware module generation in SIERA with those used in the LAGER environment for ASIC design, the various ASIC design tools that are part of LAGER are also available to users of SIERA. These tools include:

1) LAGER silicon compiler [1] for high-performance ASIC's from architecture descriptions.

2) C-to-silicon compiler [1], [6] to generate microprogrammed ASIC's from algorithms described in a C-like language. This approach is good for control dominated, low data rate applications.

3) HYPER behavioral synthesis system [2] to generate medium to high data rate DSP ASIC's with simple control from SILAGE descriptions. Using an approach based on transformational synthesis, HYPER allows the algorithm and architecture design space to be explored in order to optimize for throughput, area, power, etc.

These ASIC's can be simulated together with the other modules on the board for a complete system-level simulation.

C. Synthesis of Interface Logic

A very important class of board-level hardware module is that of Interconnect Modules—glue logic that physically links data processing modules, such as processors and memories, while meeting I/O protocol and timing constraints. Due to the wide variety of I/O protocols encountered at the system level, a library of interconnect modules is of limited scope. The ability to synthesize these interconnect modules from a high-level description is the key to having a library of reusable subsystem level data processing modules, and is particularly desirable at the system level where off-the-shelf components with a variety of I/O interface protocols play an important role.

SIERA uses the ALOHA interconnect module synthesis system that has been developed by Jane Sun at Berkeley [21]. A pictorial overview of ALOHA is presented in Fig. 5. The interconnect module is specified at a high-level in a language.
called IDL (interface description language) in which only the data flow in the interconnect module is expressed together with the appropriate protocol names. The IDL description essentially specifies the temporal and spatial mapping of the source data streams to the destination data streams in a protocol and technology independent manner. The details about the I/O protocols of the modules being interconnected are described separately and stored in the module library. The event graphs (also called signal transition graphs or STG’s) corresponding to the I/O protocols for the various modules are merged according to the data-flow described in the IDL description to produce a STG for the entire interconnect module. Then lower-level foreign tools, such as Async [22] for synthesizing asynchronous finite-state machines (FSM’s) from the STG description, are used by ALOHA to produce a gate level net-list together with timing constraints. The gate level net-list is then implemented as an ASIC or as FPGA’s or PLD’s using the tools described earlier.

Interface synthesis as implemented in ALOHA works best for asynchronous interfaces that use protocols with full handshaking such as is found on VME bus, for example. On the other hand interfaces requiring arbitration or using signalling protocols that are not speed independent are harder, and in some cases impossible, to specify and synthesize automatically. Nevertheless, the above synthesis approach to interconnect modules has been used by us for diverse examples ranging from bus interfaces such as TMS320C30 memory interface, VME bus slave interface, SBus slave and SBus DMA interfaces, etc., to complicated protocol processors for a packet oriented fiber optic communication link in a robot control system.

VII. LIBRARIES

A key aspect of the hardware module generation strategy is the library of reusable components. The purpose is to encourage reuse of design effort, and modularity. The library members can use the same layout generators and synthesis tools as described in the previous subsection. However, in order to make them reusable, the modules are usually parameterized and can be customized with suitable values of the parameters.

There are three distinct types of libraries that are needed corresponding to three different types of reusable elements. The first two libraries, package library and primitive component library, are similar to what typical board placement-and-routing systems provide. The third library, subsystem module library, is an interesting part of SIERA because this library provides system designers and higher level tools for system architecture synthesis and exploration with complex but well characterized system level building-blocks in the form of reusable subsystems.

A. Package Library

The package library contains the physical information about the various board-level packages. There is no analogue to the package library at the chip level. The reason is that at the chip level every primitive component, for example a leaf cell such as a NAND gate, has a distinct physical layout. On the other hand, at the board-level many different primitive components may be housed in packages with the same physical characteristics. From a data management perspective it is better to keep such shared information centralized in a single copy.

The library includes a variety of packages including DIP and SIP packages, PGA packages, packages for discrete parts, surface-mount packages such as SOIC, PLCC, etc. Each package is specified by a physical view in the OCT database. The physical view contains all the geometry corresponding to the package in the various layers of the board. In addition, a special placement layer is used to specify a placement boundary. Physical geometry is also attached to formal terminals that indicate the pins using which a package connects to the signal traces on the board.

Using an interface to the OCT database in the interpreted language TCL [23], scripts have been written for procedural generation of common package geometries such as DIP, SOIC, PLCC, PQFP, etc., thus considerably simplifying the creation of new packages.

The library also maintains information about transforming a package name, geometry, and pin numbers to that of commercial package databases—this is used by tools such as oct2pcb for generating net-lists for foreign routers.

B. Primitive Component Library

The primitive library is akin to a leaf cell library at the chip level. It contains information about board-level primitive components—chips, connectors, discrete components, etc. The distinguishing characteristic of these primitive components is that a package name must be specified for them in order for the tools to access the physical information about the component. Table I shows a partial listing of contents of the primitive component library.

Each primitive component in the library requires a SDL file written according to a specific policy, and additional optional files for simulation model, documentation, etc. The SDL file for a primitive component, an example of which is shown in Fig. 6 for a TTL 74X00 part, gives a black-box description of the part by listing the formal terminals of the part, and the standard terminal attributes PINNUMBER, TERMINAL, and DIRECTION; the latter two being used to facilitate checking the final design for electrical problems such as open inputs and shorted outputs. The facet attribute PACKAGECLASS with value PCB is used to indicate that it is packaged so as to be contained by a printed-circuit
Table I

<table>
<thead>
<tr>
<th>PART CATEGORY</th>
<th>PART NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top Level</td>
<td>TIL, parts to 32, 64, 128, 256, 512, 1024, 1284</td>
</tr>
<tr>
<td>Memory</td>
<td>CYM1632, CYM1936, CYM2563, CYM4096, 8275256, 8275728</td>
</tr>
<tr>
<td>Processors</td>
<td>DSP450, TMS320C31, MC68000, AT89C51 (Hitachi)</td>
</tr>
<tr>
<td>Programmable Logic</td>
<td>PLA (XILINX), PLD (ALTERA, Lattice)</td>
</tr>
<tr>
<td>FPGA</td>
<td>ACTX00105, ACTX0105, Agilent, Xilinx (XC9500, XC9550)</td>
</tr>
<tr>
<td>I/O Devices</td>
<td>YMC interface (XMC3500, XMC3502, XMC3510, XMC42000, XMC62100), UARTS (RC52001, RC52005, RC52007), IDE Controller (AD9020A)</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>TTL Decoders, DCL, DTL, current-measuring HC1100, clock drivers HC2050, op-amp buffer HCPL312, RS232 Bridge (MAX2553)</td>
</tr>
<tr>
<td>Analog</td>
<td>Op amp gains, capacitors, resistors, others, bands, op am, converters, (TE01, TEL00, ATL00), AD (AD780, AD781), Optical driver and receiver (LCX7000/10, LCX9000), Transistor arrays (MC1451, MC1454), Diode Lines, LMC355, voltage regulators (LM7800, LMC1940)</td>
</tr>
<tr>
<td>Packaging</td>
<td>Variety of headers, connectors, sockets, terminal blocks</td>
</tr>
</tbody>
</table>

Fig. 6. Sample SDL file from the primitive component library.

C. Subsystem Module Library

This library is analogous to a macrolevel library at the chip level. At the chip level such a library may contain adders, multipliers, RAM's, FSM's, etc. At the board-level an adder or a multiplier is functionally at too low a level—in fact a single chip is usually far more complex. Complete subsystems, such as embedded programmable computers, I/O interfaces, signal processing subsystems, etc., are more appropriate modules for representing hardware architecture of boards.

An extensive library of such reusable subsystem level modules has been created by using the tools described earlier. The library II is a partial list of the subsystem modules available in the library. Many modules in the library are parameterized so that they really represent a class of subsystems from which a particular module that is tailored to the needs of the application can be instantiated. The library has a variety of memory subsystems, complete embedded computer modules based around different processor chips, bus-interface modules, data acquisition modules, etc. The library has been developed over the course of several board designs and continues to grow as more systems in diverse application areas are being designed. For example, some of the modules contributed by other on-going design projects include SBUS interface module, video frame-buffer module, JTAG test controller module, etc. This library of subsystems is a major contributor to the goal of reusability.

These subsystem modules are usually composed of other modules and/or primitive components. As such a subsystem module is usually specified by a hierarchy of SDL files describing the structural interface between its building blocks which may be available as an off-shelf primitive component or may be behaviorally specified using one of the tools described earlier. The SDL files also contain floorplanning information. In addition optional files for the simulation model, I/O protocol and documentation are also present in the library for each subsystem module. The I/O protocol information is particularly important for many subsystems. It describes the timing diagrams relating the events on the various formal terminals of the subsystem when it interacts with other subsystems. Several different protocols corresponding to different types of transactions may be defined for a given subsystem. The timing diagram is represented by an event graph that describes the signalling protocol in terms of events or transitions on the I/O signals. The causality relationships and timing constraints between these events are represented by directed edges in the event graph with the events themselves being represented by the nodes. The event graph is represented in the library by a text file in aff format as described in [24].

1) Subsystem Module Example #1: A TMS320C30 Based Processor Module: A good example of a subsystem level module is proc30, a TMS320C30 based processor module listed in Table II. It provides a complete microcomputer based around a powerful 32-bit digital signal processor, and can be used wherever a powerful, software programmable,
embedded computation core is required on the board. The configuration of the module in term of its memory organization, I/O devices, etc., is specified through parameters. For example, three parameters NSRAM64C, NSRAM64M, and NSRAM256M specify the number of fast-and-big 64 K × 32, slow-and-compact 64 K × 32, and slow-and-compact 256 K × 32 SRAM modules respectively. All the blocks for making a self-sufficient computer, such as the memory, address decoder, clock generator, wait-state generator, host interface, I/O device controller, etc., are included. Some of these building blocks make use of field-programmable devices such as PAL’s and PLD’s. The implementation information for these devices is also generated as part of the module generation process. The address and data buffers are also tailored according to the amount of the memory and I/O devices. The module is also placed giving a fairly good first-cut placement. Fig. 7 shows the basic architecture of the processor module, while Fig. 8 shows an example instance of this processor module. Several different instances of this module have been fabricated and tested. The module operates flawlessly at the maximum speed of 33 MHz.

2) Subsystem Module Example #2: An Asynchronous Bit-Parallel Fiber Optic Receiver: This example is a complete bit-parallel optical receiver section. Together with its dual opticalT it provides the ability to send a multibit word at one end of an optical communication link and receive it at the other end at data rates up to 125 Mbits/sec together with a bit-error detection capability. The module is based around commercial parts. The two main parts are Am79h1000R and Am7969. The first part is a optical data link receiver which converts the optical signal received through an optical fiber with a ST connector to an electrical signal. The second part, popularly known as the TAXI receiver, is a high-speed serial-to-parallel receiver. It asynchronously decodes the serial bit-stream that has been encoded by its sister chip Am7968 TAXI transmitter that is used in the opticalT module. The remaining parts in this module are there for noise considerations. Since the optical data link receiver deals with very small currents (nano amperes) to distinguish between 0 and 1 bits, noise is a major consideration. The module uses ferrite bead filters at strategic locations and in addition decouples the optical data link receiver from the rest of the system by optionally using a.c. coupling which is selected through a parameter. Encapsulating these details about design for noise into a robust module makes a high-speed point-to-point optical communication link subsystem easily accessible to a system designer in SIERRA.

Fig. 9 shows the black-box picture of the module which interfaces electrically with the rest of the system through a set of signals over which two types of transactions are defined. The two types of transactions correspond to the arrival of a data word and the arrival of a control word—the communication link implemented by the module has the provision for a 8/9/10-bit data channel and a 4/3/2-bit control subchannel. The signalling protocols for the two types of transactions are pictorially represented in Fig. 10 from which it is clear that the transactions are distinguished by the strobe that is asserted by the module—DSTRB or CSTRB. These event graphs are stored in the library in the alt format mentioned earlier. Fig. 11 shows the alt description for the event graph corresponding to the arrival of a data word. The module structure is expressed as a parameterized netlist using the SDL syntax similar to Fig. 6.
Fig. 11. Event graph for opticolor subsystem module in aff text format.

Fig. 12. Design flow for a custom board with dedicated architecture.

VIII. SIMULATION AND NETLIST CHECKING

Although the process of board hardware generation in SIERA is quite automated, the integrity of the resulting implementation is never quite guaranteed because of the inevitable presence of "bugs" in the module generators and libraries. Therefore it is desirable that one be able to verify that the resulting implementation has the required functionality, and that it is free of electrical problems.

Three tools are provided in SIERA for this purpose. The first is SIVcheck which does a static analysis of the netlist of a board, and tries to check for electrical correctness. It discovers problems such as multiple outputs driving the same net, or a floating input. This is accomplished by using the DIRECTION and TERMMODE attributes attached to the formal terminals of the individual chips. Experience has shown that such a static analysis of the final netlist catches a vast majority of design problems—and should be done before costlier dynamic techniques such as simulation are attempted. The second tool is DMverify, from Prof. Rajeev Jain’s group at UCLA, which compares the initial netlist to an extracted netlist of the design. The goal of this netlist comparison is to audit the layout-generators themselves and guard against software bugs—for example, to detect shorts created by a bug in the router.

Static analysis tools such as SIVcheck and DMverify are, however, not powerful enough to detect logical and timing problems in the design. In the absence of formal verification techniques, such problems are best detected by doing a bit-level simulation of the entire board. Support for such simulation is provided in SIERA using the THOR event-driven simulator from Stanford [25]. THOR models digital system at a functional level as a hierarchy of modules described using the C language with some extensions. All the modules in the board-level primitive component library have associated THOR models that are stored in the THOR.MODEL attribute in the corresponding structure.master view. A THOR model for bit-level simulation of the entire board design is generated by composing these lower level models in the libraries. Although THOR is the simulator that is currently used, the library organization and tool policies are modular enough that other simulators, such as VHDLC, can also be used.

IX. APPLICATION OF SIERA’S MODULE GENERATION FRAMEWORK TO SYSTEM DESIGN

SIERA, as described in Section III, provides a complete vertical path for system design including architecture generation and software module generation in addition to hardware module generation, although only the latter aspect is described in this paper. However just the hardware module generators can often be used fruitfully in isolation from the software and architecture generation utilities available in SIERA. In particular the hardware module generation part of SIERA can be used to prototype custom boards with dedicated hardware and software programmable processors—in such a case the role played by SIERA is analogous to silicon assembly/compilation at the chip level.

Even in this limited role SIERA provides a design environment which is more sophisticated than that provided by commercial board schematic capture, placement and routing tools. The extensive library of parameterized modules, module generators, and tools for variety of placement styles enable quick design iteration in the case of architectural changes. The
top level design flow from the user’s perspective is shown in Fig. 12. The design is expressed as a hierarchical netlist of parameterized subsystems that are either from the extensive subsystem library that comes with SIERA (see Table II) or that are designed by the user using SIERA’s suite of module generators. Typically subsystems such as microprocessor or DSP based embedded compute cores, network interfaces, bus interfaces, signal processing modules, etc., are available in the library and only specialized I/O interfaces or glue logic needs to be designed. In fact, our experience has shown that often system designers will tailor the architecture so as to be able to take advantage of SIERA’s subsystem library and thus saving themselves an enormous amount of hardware and software (in case of processor modules) design and debugging time. Next the design manager DMoct is run on the root of the design. DMoct traverses the design hierarchy, and runs appropriate structure-processors (Section V) and layout-generators (Section V) as needed. The resulting fully placed netlist of chips and other board-level components is then routed to generate GERBER mask files for board fabrication. Post-fabrication debugging and development is done by using software modules from SIERA’s library, such as multitasking kernels, inter-process communication and synchronization primitives, and run-time utilities for processor cores that may be embedded in the board.

A. Examples of Boards Designed Using SIERA

1) Robot Peripheral Interface Board: This board is part of a multiboard robot system [26] that was designed using SIERA—it provides an interface to the robot joint motors, and brakes. Its task is to receive voltage or current values from the controller board and apply them to the robot motors, to sense the motor current, and to apply the brakes in case of stalls or when commanded. It is a mixed analog-digital board. For reasons of electrical noise isolation, and to avoid mechanical problems with thick cables for carrying signals to the controller, a duplex optical fiber with a custom communication protocol is used as the system-level interconnect between this board and the controller board. The optical fibers carry the data serially and the high bandwidth (125 Mbits/sec) allows protocols with low latency which is important in this case because the optical fibers are part of the controller feedback loop.

Fig. 13 shows a block diagram and photograph of the board. The board uses A/D, D/A, and optical communication modules from the subsystem module library. The protocol processors for implementing the custom packet communication protocol over the fiber-optic links are synthesized using the ALOHA tool and implemented using two Actel FPGA’s. The six-channel digital pulse-width modulators are also implemented using Actel FPGA’s from a mixed structural and combinational behavioral description. Only a small analog portion of the board (opamp based filters) had to be custom designed for this board—the rest was either automatically generated or instantiated from the reusable parameterized subsystem module library. As a result of this level of automation the entire development cycle from input description to the working board was less than two months. Similar boards for robots with different numbers of joints or different amplifier interfaces can be generated in a very short time. Table III lists the salient features of the board.

The top level description of the board was written as a SDL netlist of the following subsystems: a fiber optic transmitter, a fiber optic receiver, a transmit protocol processor, a receive protocol processor, a six-channel A/D, a six-channel D/A, a six-channel digital pulse-width modulator (PWM), and a robot-specific module for driving and receiving opto-isolated signals to and from the H-bridges amplifiers and relays inside the robot. Of these subsystems the fiber-optic transmitter and receiver, and the six-channel A/D and D/A were used straight from the library with appropriate parameters. In addition to parameterized netlist for these modules, the library also contains event-graph based protocol descriptions of the data busses on these modules. Therefore the two protocol processor modules were behaviorally specified using ALOHA’s (Section VI-C) interface description language (IDL) and the synthesized register-transfer level descriptions (SDL + BDS as mentioned in Section VI-A) were automatically mapped on ACTEL devices using the PLDS module generator described in Section
VI-C. The six-channel digital PWM subsystem was manually specified at the register-transfer level in the form of a hybrid SDL + BDS description and automatically mapped to an ACTEL device by the PLDS module generator.

The only subsystem for which a low-level manually designed structure description was needed was the robot-specific driver module that primarily consisted of opto-isolators, drivers, opamps, and various discrete components. In fact, the bulk of the SDL code that had to be written for the board corresponded to this module—it's mix of analog and glue-logic type functionality made it impossible to specify it at a higher level. Given the large number of odd-shaped components in this module, we also had to resort to manual placement of this module.

In terms of design time and complexity the most difficult modules proved to be the two protocol processors. Their functionality was too complex to handle easily in the asynchronous handshake model that is preferred by the ALOHA interface synthesis tool. In part the complexity grew from our attempt to treat the entire protocol processors as interfaces when in reality they do much more than simple low-level handshake signalling. For example, the transmit protocol processor monitors various signals on the board and follows a simple algorithm to generate appropriate data and control packets to be sent over the fiber link. The receiver protocol processor similarly reads the incoming data words to decode packet structures and then interprets the information contained in the packets to take actions like downloading new values to the PWM module, shut the robot down, etc. Our initial difficulties led to a major enhancement in ALOHA's model of an interface—instead of just an asynchronous FSM an interface is now partitioned into a top-level synchronous interface controller that in turn controls a low-level asynchronous protocol controller FSM and a low-level datapath. The high level functionality of the protocol processors (such as packet decoding and encoding) were synthesized by ALOHA into the top-level interface controller whereas the low-level data and control bus signalling was handled by the asynchronous protocol FSM.

2) DSP Multi-processor Board: This example is an interesting one because it was done in less than three months by a graduate student researcher from the DSP group at Berkeley who had little previous board or system design background. The three month time period included the learning curve associated with getting familiar with the various tools. The board is part of a multiprocessor system with a special shared memory architecture called ordered memory access described in [27]. The system developed by the DSP group is made up of one or more identical interconnected boards. The key feature of the architecture is that the access to the shared memory is granted to the processors by a central controller (called MOMA) according to a static schedule. This is in contrast to conventional shared memory architectures where the processors request access to the shared memory. Once access to the shared memory is granted to a processor, it is not released until the processor has completed a shared memory transaction. The order of accesses to the shared memory is determined by a fully static scheduler and loaded into the central controller as a list of memory transactions. An advantage of such fully static scheduling is that no hardware or software semaphores based synchronization is required. Such fully static scheduling is possible for a subclass of dataflow graphs called Synchronous Dataflow Graphs that lack data dependency. Many important algorithms in digital signal processing belong to this category. The ordered memory access architecture offers an efficient and low-cost architecture for such applications.

Fig. 14 shows a block diagram of the board which consists of four processor modules using the MC96002 32-bit digital signal processor as the CPU. Each processor module contains 1 Mbyte of fast SRAM and some glue logic for initializing the CPU mode. This particular CPU has two identical memory ports, one of which is used for the local SRAM mentioned above and the other is brought out of the processor module as a global bus. The global busses of each of the four processor modules are tied together and connected to 2 Mbytes of shared SRAM. A Xilinx XC3090 FPGA is used to implement the memory access controller which controls the bus grant line of each of the processor modules. An external 16 K x 8 SRAM module is used to store the memory access schedule which is loaded from a workstation host. The Xilinx FPGA is also connected to the global bus for loading the program code into the processor modules on initialization. The host interface is also mapped to the Xilinx FPGA. Finally, multiple boards can be connected in a chain by modifying the logic in the Xilinx FPGA's so that they act as gateways.

This design project made use of some of the existing memory modules and in turn contributed the MC96002 based processor module which is easily reusable in other designs. More importantly, variations of the board with different numbers of processors can be generated in a very short time. The physical layout of the board is shown in Fig. 15. As is evident from the figure an extensive use of tiling-based placement was made in the design of this board. Further, the four processor modules have horizontally and/or vertically flipped versions of the same floorplan, resulting in a very compact and efficient layout. Table IV summarizes the characteristics of the board.

B. Successes and Failures of SIERA's Module Generation Framework

Unlike specific CAD tools or algorithms, it is not only difficult to measure and quantify the impact of a design system
like SIERA but it is also not possible for us to directly and meaningfully compare against others. While the two examples in the preceding section give ample indication of the complexity of designs that can be handled by SIERA and the ease or complexity of the process of board design in SIERA, the examples alone do not provide sufficient evaluation of our approach. We therefore resort to the following discussion of the successes and failures of our approach to system level module generation.

Our module generation approach can be evaluated in two contexts: as a rapid-prototyping environment for application-specific boards, and as a general purpose board design environment with synthesis tools integrated into it. Our primary goal with SIERA was the first one—a framework where realistic prototypes can be designed and implemented with a short turn-around time. In this role we found that SIERA has been quite successful. For example, thesis projects in our group that normally would have stopped after fabricating an ASIC or after an algorithm level design of the system now routinely take the extra step of fabricating a board to implement a prototype system. The subsystem libraries and the module generators reduced the design time enough so that the advantages of a realistic working system prototype outweigh the increase in the length of a research project.

While SIERA's approach succeeded in the primary goal of reducing the design time for a fairly diverse class of boards, the same cannot be said about SIERA’s performance when design quality and not turn-around time is the sole metric as would be the case with high volume systems. The reason is that the key design philosophies adopted by us, modularity and synthesis, are conducive to short turn-around time but often cost in terms of design quality because the current tools and algorithms are not mature enough. This situation is however no different than what has been experienced at the chip level where higher level CAD tools were initially used for ASIC's where short turn around time was critical and only recently have high-volume commodity chips have begun to adopt techniques such as logic and RTL synthesis.

Following are some specific areas where our module generation approach needs improvement:

1) SIERA’s heavy reliance on parameterization and behavioral description makes board design easy but makes board debugging hard. This is mostly because debugging using logic analyzer or oscilloscope probing is by far the norm at the board-level, and the results of synthesis and automatic placement lack placement symmetry and orderliness. The textual and behavior level specification of some modules only exacerbates this problem because no nice schematics exist for such modules. The real problem however is that board level debugging methodologies are still not advanced enough.

2) The hierarchical modular placement strategy adopted in SIERA simplifies placement and allows library modules to be pre-placed in a parameterized fashion but sometimes results in poor quality of placement in the form of unused space and/or increased number of layers. Also, SIERA lacks automatic placement for those surface mounted boards that have components on both top and bottom sides. Placing components on both sides of a board is sometimes needed in space-constrained designs such as PCMCIA or SBUS boards.

3) Interface module synthesis using ALOHA works well for cases where the underlying signalling is asynchronous and uses full handshaking. In other cases it does not work as well or not at all. For example, interfaces with protocols that are not speed independent cannot be handled easily. Also, the handling of timing constraints during synthesis is not very precise. The synthesized interface therefore needs to be checked for violations of detailed timing constraints—ALOHA provides a formal verification tool to aid in this.

4) A disadvantage of SIERA’s reliance on a foreign router is that the extraction of information such as capacitance, inductance, and resistance values from the physical layout is quite difficult. In part the problem is that the GERBER format that is used to express board layout is not as sophisticated as the CIF format used for chips. As a result, SIERA’s module generators cannot be reliably used for very high frequency systems where transmission line analysis and simulation from extracted data is critical.
5) Another problem arising from SIERA’s routing stage not being closely integrated to rest of the design trajectory is the inadequate handling of subsystem modules that require special layout considerations. For example, the fiber optic receiver module of Section VII-C2 requires special ground and supply plane isolation for the optical data link receiver chip leading to split ground and supply plane geometries. At present there is no automatic way of passing such information to the router and this necessitates manual intervention at the routing stage for such modules.

6) As at the chip level, the synthesis, placement, and routing tools are good at synthesizing the functionality but not at handling timing constraints, if at all. The problem becomes worse at the board-level where different logic families with different drive capabilities and input capacitances are mixed. At present much of the design effort in SIERA goes toward ensuring that timing and drive constraints are not violated.

7) The mechanical aspects of a board are quite important but are ignored in SIERA. In particular, the three-dimensional geometries of the components and the mechanical interface between the board and the backplane chassis or package are important in real systems, as are related issues of heat dissipation and electromagnetic interference.

Although the above problems are present in SIERA, it needs to be emphasized that these problems are largely unaddressed in current board design environments as well.

On the positive side, the emphasis on modularity and synthesis worked quite well in reducing the design time. In particular we found that the subsystem library is extremely useful. Unlike in ASIC’s, the modules at the board-level can be quite complex because they may be based on complicated microprocessors, DSP’s, etc. The subsystem library therefore has become a repository of a large amount of accumulated design knowledge and design experience in our group. The other aspect of SIERA that has much potential is the use of interface synthesis to generate interconnect modules although, as mentioned earlier, the underlying synthesis approach does not yet work for all types of interfaces. Nevertheless given the complexity of system level I/O protocols, interface synthesis is essential for rapid-prototyping. Therefore one lesson that can be drawn is that chips should be designed such that their bus I/O protocols are well-behaved and amenable to interface synthesis algorithms.

X. CONCLUSION AND FUTURE WORK

We presented the system level hardware module generation techniques used in the SIERA framework for rapid-prototyping of hardware and software of board-level systems. The hardware module generators in SIERA for the first time provide board designers with the same type of synthesis and parameterized module library based design methodology as is available to chip designers using silicon compilers and assemblers. In contrast, the current board design packages are primarily schematic capture tools with placement and routing of a flat netlist. SIERA’s module generation framework is now quite mature, having been used for several board designs—two of which were described. The main successes of SIERA’s approach to hardware module generation have been the reduction in board design time and design at a level of abstraction much higher than just a netlist of chips. The availability of a subsystem module library and a suite of module generators played important roles in this success as did the approach advocated in SIERA of using interface synthesis to synthesize modules that interconnect the various subsystems. However, as we pointed out earlier, our approach to system level hardware module generation is not perfect. For example, one deficiency of the current hardware module generation environment in SIERA is that has become evident with use is the lack of accurate performance analysis of the final placed and routed board. Due to increasing clock frequencies, transmission line effects have become important so that the ability to extract capacitance, inductance and resistance to do a transmission line analysis is important.

The hardware module generation in SIERA is also closely linked to the software module generation and architecture generation. For example, the various processor modules that are part of the hardware subsystem module library come together with operating system kernels, communication software interfaces, and other supporting software. Substantial software, debugging, and testing infrastructure—and not just board hardware module generation capability—is thus available to a designer in SIERA. This systems view of the design process is crucial to reducing the design time.

Since SIERA’s hardware module generators have largely met the main goal of a short board design time, the new research activity in SIERA is primarily focussed at the software module generation and system architecture generation aspects of SIERA because they are considerably less automated than the hardware module generation aspect described in this paper. Still, SIERA’s hardware module generation environment continues to evolve. In addition to a rapidly growing subsystem library with modules contributed by various users, continuing research at Berkeley is concentrated on achieving a tighter integrating between the module generators and the physical routing phase, and on coupling SIERA’s board design environment to a mechanical computer-aided design and analysis environment so as to allow mechanical, thermal, and electromagnetic analysis of the board together with the system packaging. This extension of SIERA to address mechanical aspects that are important to electrical design together with our research on software and architecture synthesis aspects will bring us closer to a true software-hardware-mechanical codesign environment.

ACKNOWLEDGMENT

Many people have directly or indirectly contributed to SIERA, its tools, and its libraries. J. Sun contributed ALOHA, the interface circuit synthesis tool which is so critical to system design in SIERA. S. Mollers, and before her P. Schrupp, helped in setting up the board fabrication and testing laboratory, and maintained the Racal-Redac router. S. Sriram from Prof. E.
Lee’s group provided the DSP multiprocessor example, while T. Blumenau was a tremendous help with the robot example. J. Granacki at USC’s ISI provided the extensions to hook SIERA’s hardware module generation environment with the Finesse printed-circuit board router from SciCards. We would also like to thank all the people behind the LAGER chip design framework which provided much of the basis for this work.

REFERENCES


Brian C. Richards received the B.S. degree in electrical engineering from the California Institute of Technology in 1983, and the M.S. degree in electrical engineering and computer science from the University of California, Berkeley, in 1986. In 1986, he joined the Technical Staff at the University of California, Berkeley. He is maintaining and continuing the development of several VLSI and system design CAD tools. Current projects include supporting various research efforts related to implementing a wireless multimedia terminal for the InfoPad Project.

Robert W. Brodersen (M’76–SM’81–F’82) received the B.S. degrees in electrical engineering and in mathematics from the California State Polytechnic University, Pomona, CA, in 1966, and the M.S. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1968 and 1972, respectively.

He was a Technical Staff Member with the Central Research Laboratory at Texas Instruments, Inc., Dallas, TX, from 1972 through 1976, engaged in the study of operation and applications of charge coupled devices. In 1976, he joined the Electrical Engineering and Computer Science faculty of the University of California at Berkeley, where he is currently a Professor. He is also involved in the research of new applications focused on the area of signal processing and the development of CAD tools necessary to support this activity.

Dr. Brodersen received the EASCON’s best paper award in 1973. He was presented with the International Solid State Circuit Conference best paper award in 1975 and 1978, respectively. He was named the 1978 outstanding engineering alumnus of California State Polytechnic University. In 1979, he received the W.G. Baker award for the outstanding paper for the IEEE journals and transactions, and in 1983 was presented the IEEE Circuits and Systems Society Technical Achievement award. He was elected to the National Academy of Engineering in 1988. In 1991, the IEEE Signal Processing Society presented him with the Technical Achievement Award for innovative and pioneering contribution to the design and VLSI implementation of signal processing systems. In December of 1993, he received the Globetron Society’s award for Best 1992 Tutorial Paper for the IEEE Communications Society Publications for “A Portable Multimedia Terminal,” IEEE Communications Magazine, December 1992. In February of 1994, with copepologists on the panel “The Future Role of the Analog Designer” at ISSCC ’93, he received the 1993 IEEE International Solid-State Circuits Conference Award for Outstanding Panel Presentation.