PARAMETERIZED SCHEMATICS

Richard Barth, Bertrand Serlet and Pradeep Sindhu
Xerox PARC Computer Science Laboratory
3333 Coyote Hill Rd., Palo Alto, CA 94304

Abstract: This paper presents a design capture system that allows parameterized schematics and code to be intermixed freely to produce annotated net lists. A key feature of the system is its extensibility. It provides a small set of powerful abstractions for design description that can easily be extended by users. The system also allows convenient graphical specification of layout generators, and has been used to produce several large VLSI chips.

1. Introduction

Traditional design capture systems are typically either schematic-based or text-based. Most of the schematic-based systems represent designs as static entities in which decisions such as the width of buses and the size of memories are fixed, thereby greatly limiting the potential expressive power of the graphical description. The text-based systems use either a conventional programming language, or a specialized hardware description language [1, 5, 8, 9, 10]. These systems offer great flexibility, but a textual description of structure is often harder to understand and manipulate than a graphical one. More recent work has attempted to overcome these drawbacks by defining graphical languages that permit flexible specification via schematics [6, 7]. Typically, the graphical language provides iterators and conditionals that are translated into a textual description. While this approach provides parameterized schematics, it fails to achieve a synthesis of graphical and textual descriptions in which either may be used with equal facility.

This paper describes an integrated text and graphics design capture system that has been used to produce several large (> 50,000 transistor) VLSI chips. The central thesis of this system is that permitting a designer to freely intermix graphical and textual specifications offers significant advantages. The description tends to be more compact and comprehensible because the designer can choose the most appropriate way to express each piece of his design; consequently, it is easier to create, modify, and maintain as well. The intermixing naturally allows schematics to be parameterized, thereby offering all the benefits of better abstraction. It also permits convenient graphical description of layout generators via schematics, an aspect that is explored in detail in a companion paper [2].

The particular way in which we have integrated text and graphics was influenced strongly by our programming environment. Rather than design a new graphical programming language, we chose to provide tight links between graphical objects and programs written in a strongly typed, block structured programming language. This has had the obvious benefit of avoiding a proliferation of concepts, but its two other effects have been perhaps even more important: it has resulted in an *extensible* system, one in which there is no distinction between built-in abstractions and user-defined ones and where new abstractions may be added easily, and, it has made graphical and textual descriptions symmetric and interchangeable, permitting the designer to freely intermix the two in a given design.

A conceptual model that will facilitate discussion through the remainder of this paper is that intermixed specifications are *net list generators* that produce annotated net lists when evaluated. A net list generator is like a layout generator, except that it operates at a higher level of abstraction. In fact, the output of a net list generator typically forms the input to a layout generator. We believe that this notion represents the appropriate next higher level of abstraction for design systems because it simplifies description while retaining the tight control over the result provided by layout generators. Accordingly, the next section begins with the model for net list generation. Subsequent sections provide implementation detail, describe our experience with using the system to design a number of large chips, and point out directions for future work.

2. Net List Generation Model

2.1 Computational Model

A *net list* is a hierarchy of instantiated *devices* connected by *nets*. Some devices are primitive, while others are compositions eventually built using primitive devices. Net lists may be *annotated* with arbitrary properties such as name, transistor size, and net capacitance.

A *net list generator* is a function that takes arbitrary parameters (integers, geometrical objects, net lists, or other functions, for example) and returns a net list when evaluated. The function is represented concretely either by code that will be executed, or a schematic that will be interpreted, or *extracted*. When a net list generator is evaluated, it either returns a primitive device, or it merges the results of subsidiary net list generators into a net list and returns it. Thus, the evaluation of a single net list generator may entail many levels of schematic extraction and/or code execution, possibly interleaved. Code and schematics are tightly linked, and the linkage works both ways.

2.2 Schematic Extraction

A *schematic* is a hierarchy of instantiated graphical objects composed of rectangles, icons, satellites, and compositions. Each instance of these graphical objects has a procedure, called the instance's *extract method*, that knows how to interpret it in terms of a net list. This extract method provides the link from geometry to code. The reverse link is provided simply by making the extractor available as a function that can be called from user code.

Rectangles usually become nets, with touching rectangles inside a composition forming a single net. An *icon* is a graphical abstraction of a net list. Typically, it has some named rectangles to which connections are made, and some arbitrary pictorial geometry that denotes its function. *Satellites* are textual expressions attached to one of the non-text objects or its instance; the entity to which a satellite is attached is called its *master*. Satellites are evaluated during schematic extraction, and serve principally to pass parameters to their master's extract method. Figure 1(a) shows an icon whose net list happens to be represented by a simple schematic. The schematic is a composition that contains transistor icons, an inverter icon, the rectangles that connect them, and satellites that serve to name nets and specify transistor sizes. Figure 1(b) shows a more complicated example.

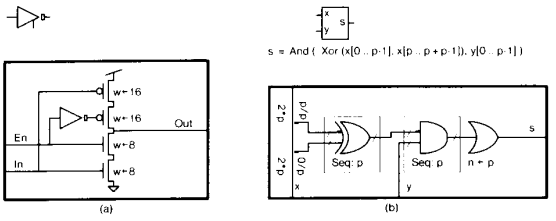

Figure 1 Examples of schematics and their icons

Our model of schematic extraction can be described best by capitalizing on the close analogy between a schematic and a program in a dynamically-scoped block-structured programming language: Icon instances are analogous to procedure calls, compositions to blocks, and satellites to variable declaration and variable assignment. Variables declared and/or assigned in satellites have scopes determined by the geometric and icon hierarchies, analogous to scopes determined by the dynamic nesting of blocks and procedures. Schematic extraction, then, is analogous to interpreting a program in the above language.

Extraction is a recursive process in which each step computes the net list for some icon or composition instance. This step breaks down into two parts, the first of which evaluates the satellites associated with the instance, and the second of which invokes the instance's extract method. The parameters passed to this method include the set of variable definitions visible in this scope - the *context*. This context is computed by applying the results of satellite evaluation to the old context - the application being side-effect free in that the old context is restored at the end of the current recursion. This last property is important because it results in a simple, functional model of extraction.

2.3 Attributes of the Model

The combination of code and schematics embodied in this model can be thought of as a more powerful language for describing designs. This language has three desirable attributes: *extensibility*, *orthogonality*, and *homogeneity*.

The language is *extensible* in the sense that there is a small set of abstractions for design description that may be easily extended by users for widely different styles of schematics. This extensibility has been facilitated primarily by the fact that we concentrated on providing close links from graphical objects to code. Icons, for example, provide a direct graphical link to arbitrarily complex user procedures. Users can exploit this connection to easily define simple constructors, such as arrays of cells and subrange selectors from a bus, or more complicated ones, like data path and finite-state-machine generators.

The language is *orthogonal* because we avoided duplicating those concepts in graphics that are better handled by code. We concentrated on the declarative power of schematics by introducing the equivalents of variables, blocks, and functions, but left constructs such as general iterators and conditionals to code. This works well since net list generators that capture structural decompositions are often declarative, so they are better expressed pictorially, while generators that reflect algorithms are more easily expressed by code.

The language exhibits *homogeneity* in the sense that code and schematics may be used with equal ease, and may be intermixed easily. This attribute frees the designer to choose the best way to express a given piece of his design based on whether it is *inherently* easier to express algorithmically, by schematics, or by an appropriate mixture of the two. Homogeneity derives from the bidirectional linkage between geometry and code.

3. Implementation

3.1 Net List Representation

Net lists are represented using the four data types: *property*, *wire*, *cell type*, and *cell class*. These basic types allow us to define an extensible set of abstractions that can represent a design at any desired level of detail. Extensibility derives from the ability to define new cell classes without affecting the code for those that already exist. A more detailed treatment may be found in a companion paper [3].

3.2 Creating Net Lists by Program

The system provides a complete set of utilities to help create and manipulate net lists by program. These utilities consist of modules for creating cell types and wires and specifying interconnections by name, layered on top of more primitive creation functions. The higher level modules avoid cumbersome declarative specifications that are typical when creating net lists using the primitive functions. When a composite cell type is created, the code performs a number of checks to ensure that invariants required by the net list data structure are maintained. An example of code can be found in [2].

Although this approach of providing creation utilities for textual specification is probably more verbose than a specification based on a hardware description language, it has some advantages. Namely, it avoids a separate language for hardware design, so it is easier to implement. Also, utilities may be added and improved piecemeal, allowing an evolutionary system development which is much harder to do in a language-based system once the language is fixed.

3.3 Schematic Extraction

The task of the schematic extractor is to turn a combination of pictures and code into a net list. As outlined earlier, the extractor proceeds top-down in a recursive context-sensitive manner starting at the instance to be extracted. Each recursion computes the net list for some icon or composition

instance in two steps. The first evaluates satellites associated with the instance, while the second passes the context containing the new evaluations to the instance's extract method. To speed up extraction for incremental changes, the extractor caches results on instances and objects. Included in the extraction context are variables defined at higher levels in the recursion, as well as certain procedures and global variables accessible in the extractor's run time environment.

Satellite evaluation itself proceeds in two phases. Satellites bound to the instance's object are evaluated first, while those bound directly to the instance are evaluated second. This order allows defaults to be specified by object satellites and overridden by instance satellites. Table 1 provides examples of satellites.

Syntax	*Semantics*	*Examples*
<variable> ~ <value>	variable definition and initialization	lines ~ 42 lines ← maximumWidth/lineWidth
<variable> ← <value>	variable assignment	w ← 2 I ← 4 ratio ← IF lines<3 THEN 2.5 ELSE Ratio[Lines]
<key>: <value>	net list annotation	Layout: Abut Simulation: "CombinatorialEval" width: DesignRules.MetalWidth["CMos"] + 2*lambda
other	names satellite's master	arbitraryName

Table 1 Satellite Usage

The actual net list computation for an instance is performed by its extract method. For icons, the extract method proceeds differently depending on whether the icon is associated with a schematic or a user-defined procedure. In the former case, the method invokes the extractor on the schematic; in the latter, it simply calls the user-defined procedure. In either case, the method subsequently checks that the interface of the resultant net list conforms with the interface of the icon. The result of extracting an icon is either a cell type that gets instantiated within the parent composition or a wire that gets used to make one or more connections.

The extract method for compositions is considerably more complex, because it is here that most of the geometric work of extraction takes place. For each icon or subcomposition it encounters, the method recursively invokes the extractor, and then instantiates the result within the cell type it is constructing. For icons, this result is a cell type or a wire, but for compositions it is always a cell type. When instantiating a cell type, the method determines how the cell instance will be connected to other nets. Connectivity is computed by intersecting rectangles that represent connection points on the instance with rectangles in the parent composition. When the method encounters a rectangle, it creates a new wire and merges it with an existing wire if rectangle intersection indicates the two wires are connected. The geometrical engine used for this purpose is the same as for layout, since much of the code can be shared. In schematics it is convenient to specify connectivity based on name equality as well as geometrical intersection, so the method also merges together wires with the same name.

3.4 User Interface

Each net list generated by the extractor is annotated with all the graphical objects which were part of its schematic. These annotations establish a link between a wire and its rectangles, or a cell and its corresponding graphical object. Tools that require the specification of a wire may ask the user to designate a rectangle with the graphical editor by pointing to it with the mouse. Similarly, tools that specify a wire or a cell as output - for example, when an error occurs - highlight the corresponding rectangles or cells. This approach, analogous to symbolic debugging of source code, permits real time interaction and avoids tedious names. It has proved extremely effective for reducing the extraction-simulation-debug loop. For example, it allows the implementation of a command that adds a selected wire to the oscilloscope-like output of the logic simulator.

This tight integration between the graphical source and its net list stops, of course, whenever schematics refer to a net list generator expressed by code. A powerful disambiguation mechanism was introduced for cases when there is not a one-to-one correspondence between the net list and the schematic, for example, when an icon that refers to the same net list is used several times.

4. Results

Since its initial implementation in early 1986, the system has been used to describe around a dozen VLSI chips of varying sizes and complexities. Some of these chips have been fabricated while others are in the final stages of design.

The remainder of this section provides data that characterize our experience with the system to date. To put this data in perspective, we first describe how the system fits within the larger context of a chip's overall design cycle, and then go on to provide the numbers which form the basis of the discussion in the retrospective section.

4.1 Chip Design Cycle

Figure 2 shows a highly idealized view of the design cycle of a VLSI chip. The bold boxes indicate stages that use the design capture system. The detailed specification stage involves paper design and/or high-level simulation. Upon its completion, the designer has a specification detailed enough to enter pieces of his design into the design capture system. The design of a real chip rarely follows such a simplistic path. More often than not, there is considerable overlap between stages, the design involves more than a simple two-level description hierarchy, and the order of the stages is hard to determine. Nonetheless, this figure provides us with a useful basis for discussion by identifying significant tasks involving the design capture system.

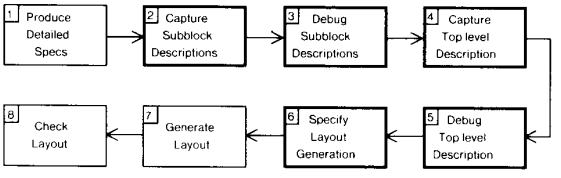

Figure 2 An Idealized View of the Chip Design Cycle

Often, a significant fraction of the overall design time is spent in debugging a description once it has been entered into the system (stages 3 and 5 in the figure), so it is useful to examine debugging more closely as shown in figure 3. This process closely resembles the load-run-think-modify-compile cycle so familiar to programmers. Ideally, as with programming, one would prefer a system in which most of the

time around the loop is devoted to thinking about solutions (stage C).

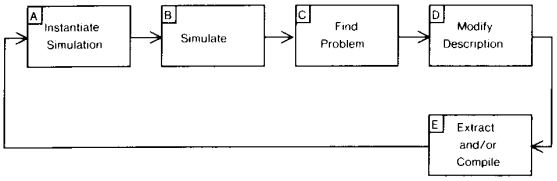

Figure 3 The Debug Loop

4.2 Chip Designs and Statistics

Table 2 shows the layout methods and development times for chips that have been designed using the system (the numbers in parenthesis indicate steps in the design cycle). The figures in the table are subjective measures obtained by interviewing designers, and as such are only approximate. The layout–methods vary all the way from program generated, through standard cell, to full custom (FC), while the design times range from a day to over two years. Of the 14 chips represented, six have been fabricated and tested (at least for functionality); four are currently under test; three are just entering the layout production and verification stage; one will be sent to fabrication soon. A figure that helps put the design capture system in proper perspective is the fraction of time spent in design capture/simulation/debugging over the entire design cycle. For our sample, this figure ranges from around 20% to around 75%, with the average being close to 50%. This indicates that further improvement in the design capture system would have a significant impact on the overall design time.

Chip	Methodology	Current Status	(1) Detailed Specs	(2, 4) Design Capture	(3, 5) Debugging	(6, 7, 8) Layout
C1	Std Cell	Fabricated	2 months	1.5 months	1 month	1 month
C2	Std Cell + FC	Tested	6 months	1 month	1 month	1 month
C3	FC	At Layout	4 months	3 months	7 months	5 months
C4	Pgm Gen	Tested	2 hours	3 hours	.	.
C5	FC	Tested	3 days	15 days	15 days	2 months
C6	Std Cell + RAM	At Layout	2 months	2 months	4 months	.
C7	FC	Tested	3 months	4 months	1 month	6 months
C8	FC	Tested	9 months	2.5 months	2.5 months	1 year
C9	Std Cell + RAM	Fabricated	3 months	3 months	6 months	2 months
C10	Std Cell + FC	Fabricated	1 month	1.5 months	3 months	2 months
C11	FC	At Layout	9 months	5 months	6 months	7 months
C12	FC	Tested	15 days	1 month	1.5 months	2 months
C13	Std Cell + RAM	Waiting Fab	1 week	2 weeks	1 week	1 week
C14	Std Cell	Fabricated	2 weeks	1 week	1 week	1 week

Table 2 Layout Methods and Development Times

Table 3 shows various size statistics for the 14 chips. The area and transistor counts indicate that several of the chips are moderate to large by today's standards, thus constituting a serious test of the design system. The schematic and code sizes show that while most of the descriptions are dominated by schematics, sizable fractions of a few descriptions are in code, and most utilize some code.

Chip	# Transistors	Area (sq. mm)	Schematic Size (bytes)	Code Size (bytes)
C1	30K	70	35K	9K
C2	11K	45	110K	0
C3	340K	150	385K	27K
C4	1.5K	9	0	9K
C5	150K	42	135K	8K
C6	188K	156	115K	0
C7	67K	100	124K	45K
C8	61K	110	304K	150K
C9	36K	120	325K	8K
C10	73K	108	250K	67K
C11	130K	80	367K	92K
C12	13K	20	332K	0
C13	170K	110	103K	0
C14	48K	80	80K	0

Table 3 Chip Size Statistics

Finally, Table 4 provides data that helps give a feel for the time around the debug loop (the letters in parenthesis refer to stages in the debug loop). The first column gives the time to produce a net list from scratch, while the second gives the same time after a "typical" change made during debugging. The next two columns indicate the time to set up a simulation, and the number of clock cycles of the chip simulated per minute once the simulation is under way. Adding a selected wire to the oscilloscope-like output of the logic simulator typically takes less than a second. These numbers are for complete chips rather than for chip subblocks.

Chip	(E) Extraction	(E) Incr Extraction	(A) Simulation Set-up	(B) Simulation Speed
C1	20 min	5 min	10 min	60 cycles/min
C2	8 min	6 min	3 min	120 cycles/min
C3	25 min	5 min	12 min	4 cycles/min
C4	1 min	1 min	.	.
C5	15 min	5 min	45 min	60 cycles/min
C6	15 min	5 min	10 min	4 cycles/min
C7	15 min	4 min	30 min	5 cycles/min
C8	45 min	30 min	60 min	5 cycles/min
C9	20 min	4 min	6 min	12 cycles/min
C10	20 min	3 min	5 min	24 cycles/min
C11	40 min	30 min	20 min	6 cycles/min
C12	10 min	2 min	.	.
C13	10 min	2 min	15 min	60 cycles/min
C14	10 min	2 min	20 min	80 cycles/min

Table 4 Interaction Times

It is clear that the system's speed is not really adequate to debug large designs in a completely interactive manner. Extraction and simulation set up could each use an order of magnitude of speed improvement before they become comparable to the time spent thinking. Caching already buys a factor of around 4 over extraction from scratch, but as can be seen from the figures for C2, C8 and C11, the gains are sometimes considerably less. The current implementation, which caches results on geometric objects, is inadequate in two respects. First, if a design description is relatively flat (the case with C2), then the system has to redo most of the work. Secondly, if a description uses code to define a piece of the design (the case with C8 and C11), then this piece is sometimes recomputed even if it doesn't need to be. Improving incremental extraction is only part of the problem, however, since simulation setup also accounts for a significant fraction of the time. Moreover, our system currently cannot do the setup incrementally so the gains to be had are large. Finally, the speed to display signals is adequate, though some improvement would help, especially for the larger designs.

5. Retrospective

We had to make the system work quickly since other researchers depended on the results of our work. This caused us to decide to use the existing graphical editor, and to postpone building a completely incremental system. This section discusses some of the system's basic design decisions, noting which ones were successful, which ones weren't, and where we encountered unexpected difficulty.

5.1 Parameterized Schematics

Our provision of parameterized schematics works well. Parameterization allows our designers to delay decisions such as the number of lines in a processor cache, the number of bits per line, and even the number of bits in a data path. Designers can therefore debug their design with the smallest feasible size, and confidently expect that the full-size version also works. For chip C3 this technique changes the time around the debug loop by an order of magnitude. Delaying size decisions has the additional side benefit of allowing processor cache capacity analysis to proceed parallel to the description - potentially speeding up the schedule.

Parameterized schematics also permit us to annotate schematics with layout information. Typical industry practice consists of describing the layout via a floor plan that is more or less independent of the schematic (the floor plan describes the placement while the schematic specifies the connectivity). We have successfully linked net list and layout generators so that designers' schematics drive layout synthesis. This approach is presented in a companion paper [2].

Parameterization did, however, complicate the system's implementation. Generating net lists incrementally is considerably harder because determining what changed and what did not is more difficult. Parameterization also made it more difficult to implement commands that use connectivity (for example the command to highlight a whole net) because this information is only available after a parameter-dependent extraction, and cannot be derived simply from the geometry.

5.2 Extensible System

Our decision to build an extensible system has also worked well. The extractor has been implemented using an object-oriented model in which each instance of a graphical object has its own procedure to interpret it. This is an advantage for the CAD programmer who maintains the system since it results in a simple structure. It is also beneficial for the VLSI designer because it keeps the mental model of the extractor quite simple. The key aspect, however, is that this model has proved easy to extend and we have added several new graphical items since the system's initial design.

For example, we have introduced a variety of graphical items which generate Spice primitives such as diodes, resistors, voltage or current probes, and voltage generators. We applied the same principle to driving our mixed mode simulator. Test vectors are applied by using digital signal generator icons or supplying the name of a test vector file in a special icon. The graphical appearance is similar to a physical electronics workbench. This eliminates the need for hard to read command files and keeps the testing information bundled with the design source.

An advantage of extensibility is that it frees us from having to write a large number of graphical primitives before the system is useful. It allows these primitives to be added on demand and provides freedom to experiment with alternatives before selecting one. We built several powerful operators such as one and two dimensional sequencers, a data path generator, a finite automaton generator, and a rich set of wire structure manipulators.

It is difficult to design a unified sequence operator that is general enough to cover all of the applications on hand. Part of the problem is determining how close to the layout geometry one wants the schematic representation. One extreme is the *geometric sequencing* model, in which the sequencing is identical to the layout. The other is the *logical sequencing* model where no attempt is made to conform to the structure of the layout and all inter-cell interconnection is specified by appropriate manipulation of structured wires. The geometrical sequencing model is conceptually very simple, though we find it more difficult to provide generality in this model. On the other hand it is easier to provide generality in the logical sequencing model but it is more cumbersome to use.

5.3 Extractor Implementation

The schematic extractor and the layout extractor use a common geometry engine. This may seem strange because layout extraction is context-independent while schematic extraction is strongly context-dependent. As it turns out, it is useful to do layout in a context-dependent manner because it enables neighbor overlaps to be considered or not. It also allows net list construction to depend on the names of wires as well as on geometric features. On the other hand, layout extractors have fairly sophisticated rectangle intersection algorithms which may appear to be overkill for schematic extraction. In our experience, sophistication is required since some schematics are quite large. The use of a common geometry engine is also valuable from a system-structuring standpoint because it avoids redoing a substantial amount of fairly intricate code.

In implementing the connection of graphics to code, we utilize an interpreter for the local programming language to evaluate satellites. This use of the interpreter provided us with a great deal of flexibility, since it allowed general expressions to be used directly within schematics. Variables referenced in these expressions could be either variables in the context, or in the load state of the machine, thereby providing direct graphical access to any loaded program. Along with this flexibility also comes a speed liability. Currently, about half of the extraction time is spent in the interpreter. Straightforward use of the interpreter means that every time a schematic is reextracted expressions have to be reparsed and reevaluated.

In our object-oriented model, the order in which instances are extracted cannot be controlled. This causes the same information to be specified more often than necessary. For example, when we select a component wire from a bus, the size of the bus must be specified within each selector. This is needed because there is no way to guarantee that the bus size is known when the extractor encounters a particular selector. The selector might, in fact, be the first instance the extractor encounters within a composition.

In the first implementation we divided satellites into

parameter expressions and result expressions depending on their syntax. The parameter expressions computed values for the parameters, while the result expression called the extract method for the instance being extracted. This approach was unsatisfactory because as we added new classes of objects the syntax of expressions would have to be embellished. This approach was also inefficient because the interpreter was invoked more often in calling the extract method. Our second attempt eliminates result expressions and has hidden properties from which the extract method can be inferred. This solution is not very satisfactory either, because it does not allow the extract method information to be easily seen. At least part of our trouble here is the use of an existing editor.

5.4 Existing Editor

When we started building the system, we had a high quality layout editor available to us. In one respect this was a big plus because it left us free to implement higher levels of the design capture system. The editor had some drawbacks, particularly with regard to displaying annotations. We found it useful to provide designers the option of hiding information to avoid clutter. However, it was extremely awkward if the system required *all* information to be hidden. Our editor did impose this constraint, and so we implemented satellites, i.e., properties that could be made visible or not and whose position and looks could be controlled entirely by the designer.

The editor does not impose any semantics on geometry, permitting the same editor to be used for both layout and schematics. This simplifies both the user model and the implementors' task, but it also implies that we have to reintersect rectangles each time an object is extracted. The slow down caused by reintersection is ameliorated by result caching but, as we pointed out earlier, caching does not work as well as we would like. Another problem with the editor not knowing about schematic semantics is that the editor cannot discover syntactic errors interactively. In our case, the extractor locates some of these errors and a separate checker weeds out the rest.

6. Future work

6.1 Operator Extension

We have found that the design of operators that succinctly express the structure of VLSI is a difficult task, particularly within our simple bottom-up construction model. We do not yet have a common set of operators for all our designs because extending the operator set is a tedious cut and try process which is best done in the context of many real designs.

The cell sequence operator has received a great deal of attention but we have yet to find an expression of sequence that both covers the frequent tilings of a two dimensional plane with a single cell type and combines power with conceptual simplicity. Decoders are obvious examples of such tilings. The wiring between cells, aggregation of wires into buses, and index dependence are just a few of the issues which must be considered.

6.2 Refinements to the Existing Model

When we designed the computational model we decided not to consider the frequent updates that occur during debugging. For instance, the model allows arbitrary side effects to occur within procedures called during extraction. We have added manual specification of procedure arguments and a cache mechanism so that a procedure need not be reexecuted every time it is called. Really fixing this defect requires fundamental changes to the system. The underlying programming language needs to be modified to allow only side-effect-free procedures, and the manipulations of the context by the extractor need to be changed to fit the functional model.

The greatest limitation to extraction speed is the interpreter, closely followed by rectangle intersection. We have observed that rectangles that make up the picture always intersect the same way even when the parameters of a cell are changed. We could avoid reintersecting each time if we maintained another intermediate representation. Avoiding rectangle intersection and enhancing the interpreter to decouple parsing from evaluation could improve the speed by an order of magnitude.

Even with the existence of satellites, there is some information that guides the extraction process that is always hidden. This is a result of the fact that the concept of satellites appeared after the editor was written, and therefore it is not well integrated. The right way to allow users to control the disposition of useful information would be to add properties that have user-definable display procedures.

6.3 A New Computational Model

The computational model that we currently use is batch oriented. Whenever a new net list is built, the top level net list generation function is called. There is caching that makes this process partially incremental, but the model is still one of complete reexecution of the generation function. An alternative model would be to change the contents of the net list incrementally when the net list generator is changed. In our current model, every cell can be an arbitrary function of the subcells. Thus, every cell type, starting from the one that is changed all the way to the root cell type, must be rebuilt. It is frequently the case that a change in a low level cell type does not really require changes all the way to the root. Differentiating the portions of the net list that need to be changed from those that do not, requires a fundamentally different computational model.

A much more restricted functional model of evaluation would facilitate the definition of such a model and make it easier to construct an incremental system. The chief benefit of such a system would, of course, be its interactive nature.

6.4 Editor Framework

A schematic-specific editor, rather than a general purpose editor followed by an extractor, would allow more effective capture and feedback mechanisms. The decision to use an existing editor and translate afterwards was a sound one because we have sorted out many thorny semantic issues but we now need to revisit this decision.

We would like to integrate representation specific editors into a single editor framework. Currently, the granularity at which we can intermix information is whole documents. We need to reduce this granularity so we can have a single document that allows us to freely mix commentary, code, schematics, layout, timing diagrams, and other information. This has the added advantage that it couples well with ongoing

document management work at our research center.

7. Summary

We have described an extensible design capture system based on the notion of net list generators. Permitting generators to be specified with a combination of both traditional schematics as well as programs in an existing language, leverages the familiarity that designers have with these modes of expression. We have discussed some of the issues that arose during the construction and use of this system. Finally, we have sketched both incremental and radical changes that need to be made for improving the system.

8. Acknowledgments

The contents of this paper overlaps with [4]. The material is presented here to tie together [2] and [3] and to reach a larger audience.

Many people have contributed to the evolution of graphical design capture systems within the Computer Science Laboratory. The members of CSL provided the Cedar environment which enabled rapid development and provided a solid foundation. The Dragon designers supplied feedback and code which enriched the system. Finally, none of this work would be possible without the research environment so graciously provided by Xerox.

9. Bibliography

1. M. R. Barbacci, "Instruction Set Processor Specification (ISPS): The Notation and its Applications," IEEE Transactions on Computers, 30(1), pp. 24-39, January 1981.
2. R. Barth, L. Monier, B. Serlet, "PatchWork: Layout from Schematic Annotations," 25th ACM/IEEE Design Automation Conference, June 1988.
3. R. Barth, B. Serlet, "A Structural Representation for VLSI Design," 25th ACM/IEEE Design Automation Conference, June 1988.
4. R. Barth, P. Sindhu, B. Serlet, "Integrating Schematics and Programs for Design Capture," Advanced Research in VLSI, proceedings of the 1988 MIT conference, MIT Press.
5. S. German, K. Lieberherr, "Zeus: A Language for Expressing Algorithms in Hardware," IEEE Computer, pp. 55-65, 18(2), February 1985.
6. K. Lieberherr, "A Two-dimensional Hardware Design Language for VLSI," North Holland, proceedings, EUROMICRO Symposium on Microprocessing and Microprogramming, pp. 131-142, March 1984.
7. J. Nash, S. Smith, "A Front End Graphic Interface to the First Silicon Compiler," proceedings EDA, pp. 120-124, March 1984.
8. R. Piloty, D. Borrione, "The Conlan Project: Concepts, Implementations, and Applications," IEEE Computer, pp. 81-92, 18(2), February 1985.
9. M. Shahdad, R. Lipsett, E. Marschner, K. Sheehan, and Howard Cohen, "VHSIC Hardware Description Language," IEEE Computer, pp. 94-102, 18(2), February 1985.
10. M. Shahdad, "An Overview of VHDL Language and Technology," proceedings DAC 1986, pp. 320-326, July 1986.