COMPUTER HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS: M. Breuer, R. Hartenstein (eds) North-Holland Publishing Company

COMPUTER AIDED SYNTHESIS OF DIGITAL-SYSTEMS

Gerhard Zimmermann

Corporate Computer Sciences Center
Honeywell Incorporated
Bloomington, Minnesota
U.S.A.

INTRODUCTION

The design of a digital system can typically be split into phases, which are defined here as:

Architectural Design: Specification of the behavior and design of the system as built from subsystems.

Subsystem Design: Transformation of the behavior into a register transfer level structure.

Logic and Circuit Design: Refinement of register transfer level modules down to the gate and circuit level.

Physical Design: Transformation of the structure into a physical implementation.

Software design is excluded here. Firmware design is viewed as part of the control structure design, and thus, is done during the subsystem and logic design phases.

Due to the complexity of all of these tasks, there is a strong need for computer aids. Historically, the development started with simulators for all phases, and graphical aids for the physical design. The first synthesis tools were developed for printed circuit board layout. Currently, an enormous effort is devoted to synthesis tools for LSI and VLSI. The total physical design area has become of prime interest for CAD.

In the other design phases, the development of synthesis tools has already begun. PLA synthesis is a well-known example for logic design aids. Subsystem synthesis is in progress in a few laboratories; architectural synthesis is still far-off.

Although, or maybe because, the architectural and structural design phases are critical for the performance and cost of a digital system, designers are hesitant to believe in synthesis aids. I will, therefore, try to explain what synthesis in these two phases can do for the designer, where the current limitations are, and what the topics for further research are. First, the structure of the design process and synthesis in general will be introduced. Then an overview of the current status and existing problems in the architectural and subsystem design phases will be given.

The first three design phases contribute more than 60% to the total design costs, on the average. Since in VLSI the design costs form a considerable part of the product cost, small savings made by using design aids can result in considerable savings in the product cost.

In addition, decisions made at the architectural or subsystem level strongly influence the manufacturability of a system. "Front-end" tools at these levels can help make the right decisions, and reduce the necessity for design iterations through the total physical design process. Synthesis offers "front-end" capabilities. Expensive design iterations also result from functionality and performance failures. Verification by construction (synthesis) is a much stronger tool than by analysis (simulation) for detecting failures early in the design process. Thus, computer aided synthesis can provide considerable aids for the designer, in reducing the design costs or achieving better products.

STRUCTURE OF DIGITAL SYSTEM DESIGN

From the idea to the product, three activities are necessary:

SPECIFICATION DESIGN FABRICATION

DESIGN can again be split into

SYNTHESIS, ANALYSIS, SUPPORT

ANALYSIS is the evaluation and checking of the implementation against the specification. Typical tools are: simulators, design rule checkers, timing analyzers, and fault simulators.
SUPPORT functions are necessary to speed up timeconsuming tasks. Examples are graphical aids, language translators, macro expanders and schematic generators.

SYNTHESIS is the actual design, the transf ormaton:

SPECIFICATION ----SYNTHESIS----> IMPLEMENTATION

The attributes are:
- **-Synthesis is creative**
- **-Synthesis generates an artifact**
- **-Synthesis includes design decisions**
- **-Synthesis can be manual, computer aided, or automatic**

The question is: can computer programs have these attributes? Radical approaches in artificial intelligence are under investigation to automate creativity. The more conservative approach is to do the "routine" parts of synthesis automatically, leaving the "ingeneous" parts to the designer. Where we draw this line depends upon our knowledge of the design process, and what is worth the effort to automate. At the low end, synthesis can mean a process without any design decisions or creativity, for example, a library lookup for TTL modules. We should rather call this mapping. Synthesis can be more intelligent than mapping, as for example, a module binder that selects an optimal entry in a module library, or creates module structures if no match is found.

To bridge the gap between the original specification and fabrication, we must divide the design process into smaller steps. In a structured hierarchical design, these steps are synthesis steps. The description of each of these steps and their interface will be explained in terms of description levels. A digital system can be described in three major ways, which we will call description DOMAINS, in using the Caltech /1/ or M.I.T. /2/ terminology, which

appears to be widely accepted now.

DOMAINS: BEHAVIORAL, STRUCTURAL, PHYSICAL

A BEHAVIORAL description is purely functional. For example, the instruction set manual of a microcomputer describes its behavior without revealing almost any information about its internal structure. The behavior of a logic black box can be described by its truth table.

A STRUCTURAL description contains only information about named boxes and their interconnections. Only if we know the behavior of a box can we derive the behavior of the described subsystem. Schematics are examples of structural descriptions.

A PHYSICAL description is the information about the physical implementation. It is mainly geometric information, as for example, mask data.

The interesting relation between the domains is that for a specified behavior, many structural implementations exist, but for a specified structure, there is only one behavior. The same relation exists between the structural and physical domains. Thus, synthesis involving design decisions only proceeds in the direction:

Synthesis: BEHAVIORAL--->STRUCTURAL--->PHYSICAL

In each domain we may have subdomains. For instance, the behavior can be expressed in applicative or procedural languages, as formal specifications or as state transition graphs. Translations between different subdomains involve degrees of freedom and thus, can be called synthesis.

In every domain or subdomain, we have another dimension: The LEVEL OF DETAIL. In structured programming, we make use of a hierarchy of levels of detail to reduce the complexity of every design step. In hardware we do the same. For example, in the physical description there are some natural levels, such as cabinet, rack, board, carrier, chip, and some artificial, such as macro- or microcell. The choice

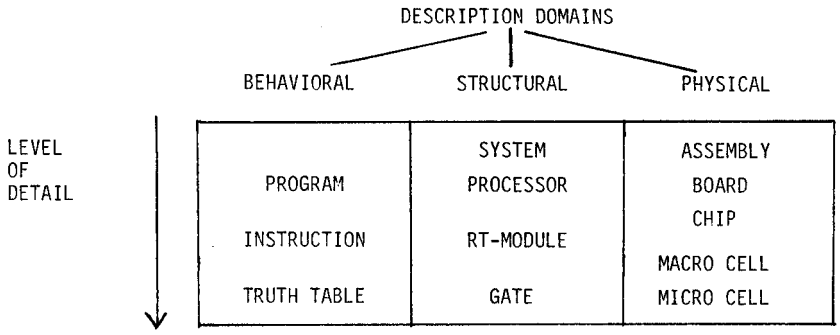


Figure 1
Description Structure

of the artificial levels depends upon the application and comprehensiveness. The register transfer (RT-level) for instance, has become a useful level throughout behavior and structure, and is supported by many languages. Figure 1 gives an overview of the total description structure. The levels of detail are only examples, and are not meant to be complete.

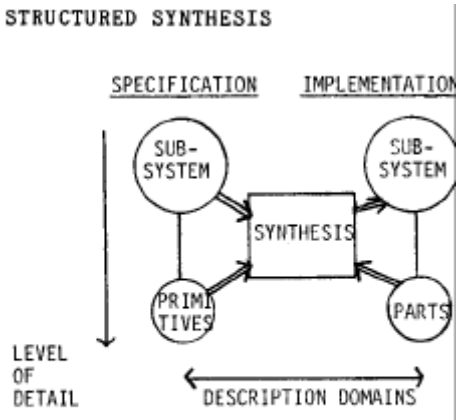


Figure 2
Synthesis Parameter Definition

Applying the structure of description domains and levels of detail on the specification of a synthesis step, we can talk about four parameters (see Figure 2). The SUBSYSTEM is the module we are dealing with at this point of the design. It may be the total system or any piece of it. We can locate it at a certain level of detail which it represents. The SPECIFICATION of the SUBSYSTEM is composed of PRIMITIVES, which can be located in a domain and at a level of greater detail than the subsystem. The SYNTHESIS creates an IMPLEMENTATION of the same subsystem. The subsystem is now described by primitives or PARTS in another domain and level. The level of detail of the parts

cannot be greater than that of the primitives, without additional information.

Locating these four parameters in the description space helps us to understand what inputs synthesis requires, what outputs it provides, and how several different syntheses can be connected in a structured design process.

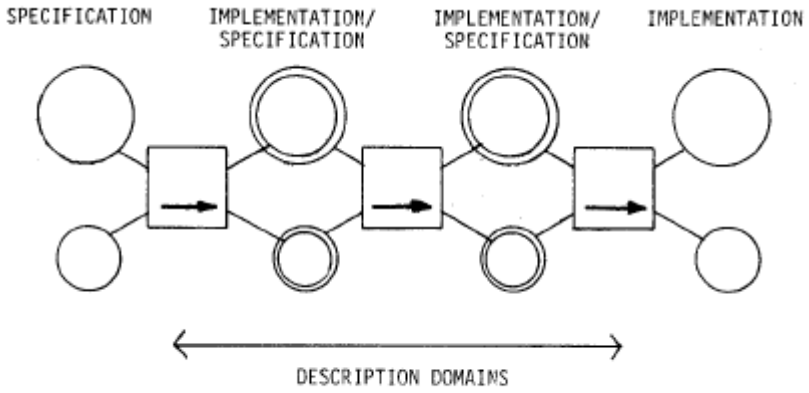


Figure 3
Design Flow Through Domains

We can proceed with the design through subdomains and domains, if we use the implementation of one synthesis as specification of the next one, as shown in Figure 3.

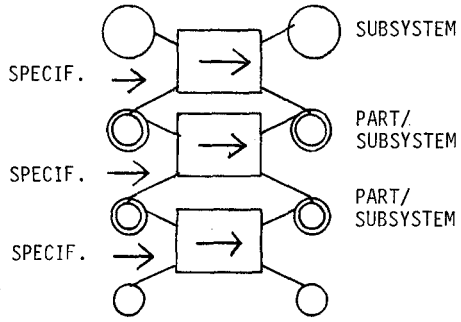


Figure 4
Hierarchical Refinement

We can refine an implementation by making the parts of a synthesis the subsystem of the next, as shown in-Figure 4. It should be noted that we have to specify every level. Thus, refinement cannot be done without additional information) This is often provided by libraries, by manual interaction, or by other syntheses. Synthesis cannot invent the refinement!

So far, we have simplified the specification.We have only used the part that can be transformed. Areal specification also contains other parameters, such as constraints, design objectives, design rules, and predefined parts. These parameters are defined in different locations of the description space, or even at the product level. However, their influence on design decisions is not local. For instance, a specified chip area and layout rules are important criteria for evaluating results from system synthesis. This means that the synthesis steps in a structured design are highly related to each other.

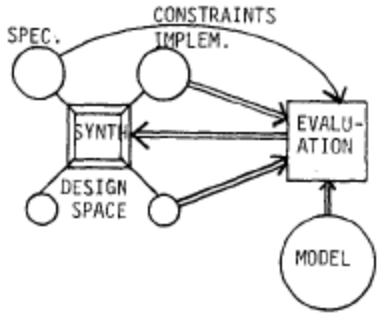


Figure 5
Design Space Control

In order to make the synthesis steps as independent of each other as possible, and yet consider the relations, we add a special evaluation capability to the synthesis that controls the use of the design space. Figure 5 Shows the resulting feedback loop. The evaluation is based on a model of the rest of the design process. This model can be analytical, can be based on statistical evaluations of similar designs, or it can be a fast simulation of the following design steps (fast prototyping). In either case, we get an approximation of parameters that cannot be directly measured at the level of the synthesis step. Such an approximation is normally enough to select a feasible or best version out of several trials, or to control the synthesis process directly. This is a problem of global optimization /3/.

Another problem we have not attacked here, is that the levels of detail do not always match across the description domains. The physical implementation enforces some packaging levels that may be different from useful behavioral levels. This problem can be solved by introducing some intermediate levels at the interfaces between domains, with the help of partitioning and composition steps.- These

can be seen as special syntheses. Partitioning and composition are nontrivial and need additional research. This problem shows that the breakdown of the design problem into connected synthesis steps is not as regular and straightforward as it may seem from looking at Figures 3 and 4. Figure 6 is more realistic, but still a feasible example.

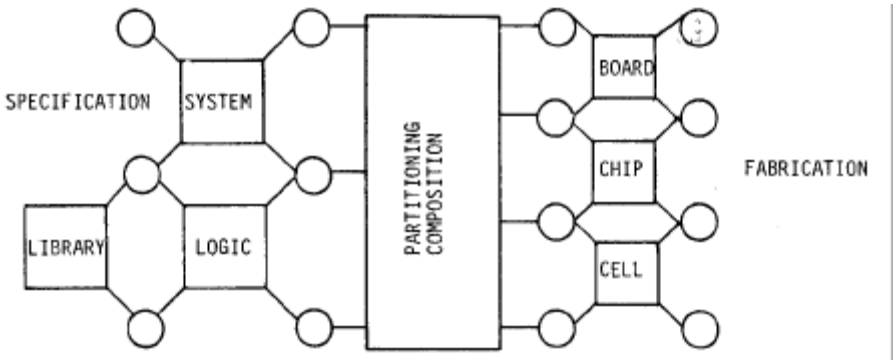


Figure 6
Structured Synthesis

SUBSYSTEM CLASSIFICATION

In the introduction we used the terms architectural, subsystem, and logic design. These terms make sense in the context of one system, but they mean different synthesis problems if we talk about different systems, as for example, a distributed system or a traffic light controller. Therefore, we classify the synthesis problem by system or subsystem characteristics, rather than by the mentioned terms. If we restrict ourselves by excluding logic and physical design, two major classes remain:

CONCURRENT SYSTEMS
SYNCHRONOUS SYSTEMS

CONCURRENT SYSTEMS include distributed systems, multiprocessors, data flow machines, and macro pipelines. Primitives are at the level of processing units, memories, devices and communication channels.

SYNCHRONOUS SYSTEMS include processing units, controllers, memory units, device drivers, and communication interfaces. Internal synchronous and asynchronous parallisms, like pipelining, are possible. Primitives are typically at the register-transfer level.

We will deal with these-two classes separately. In the following chapters, we will show the state-of-the-art, and pose some questions to our knowledge. In doing so, we will only refer to selected publications, without covering the entire area.

CONCURRENT SYSTEMS SYNTHESIS

In determining the synthesis problem, we must first discuss the DESIGN SPACE. One degree of freedom is the choice of architecture. The decision between central and decentral control in distributed systems is only one example. It is already difficult to even specify

the behavior of a system, independent of the architecture. Therefore, we are far from considering the automation of this step, but the provision of analysis tools would be a very appropriate step forward.
Another degree of freedom is the partitioning of the behavior into concurrent subtasks. We have the same problem as above: to find an unpartitioned specification for the behavior. Languages like CAP /4/, SARA /5/ and CSDL /6/ assume a partitioning. An achievable goal would be the evaluation of a partitioning, and the use of a synthesis tool to repartition. Synthesis tools could be used to generate model structures for the evaluation.

Accepting the behavioral partitioning and the communication between subtasks, the mapping on a structure allows considerable design space. If the cited concurrent system description languages prove to be general enough, they can be used as input specification for a synthesis that does this mapping. Again, evaluation of the performance and cost of the synthesized structure is very important and can be supported by a synthesis tool for the parts of the structure. These parts are normally in the class of synchronous systems.

Let us explain this evaluation, using Figure 7. Let I be the synthesis that creates the structure we want to evaluate. Its parts are at the processor level and we cannot say much about their parameters in general. In a top-down design, their parameters are typically determined by the next design step. Evaluation I can either assume parameters and pass these in the specification to the next design step, or, as in Figure 7, we can use a system specification that is refined beyond the processor level. Synthesis I determines an intermediate level (dashed line). Synthesis II generates structures for the parts based on the information in the original specification. Evaluation II provides rough estimates of the parameters of the parts of Synthesis I. Synthesis II does not try to optimize, it is only used for fast prototyping. The parameters can be corrected for average optimization, but they are normally precise enough to control the design space of Synthesis I. Otherwise, this method can be extended hierarchically.

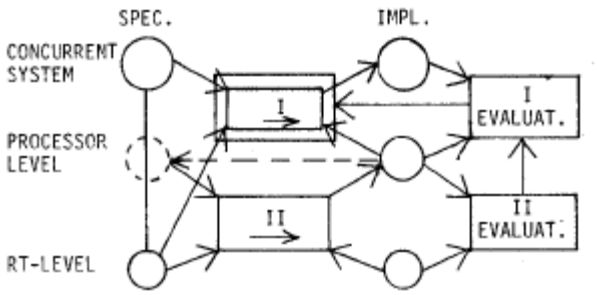


Figure 7
Evaluation Flow

By emphasizing the evaluation first, and then the synthesis, we can learn from the human designer, before we try to automate his reactions. Much more experience with concurrent systems is necessary before synthesis can become realistic.

Simulation is another possibility for evaluating designs, but certainly not as efficient as analytical evaluation, together with synthesis. Simulation is also useful for design validation. Combined with constructive correctness by synthesis, validation of

concurrent systems designs seems to be in reach.

Much work is still needed in the specification of concurrent systems. The current description languages still include too many implementation details to really be able to use the full design space. Also, the problem of correctness proof of the specification has not yet been sufficiently solved.

The state-of-the-art is the manual partitioning of concurrent systems into synchronous subsystems, and this leads to the next class of problems.

SYNCHRONOUS SYSTEMS SYNTHESIS

Synthesis of synchronous systems is much more advanced than that of concurrent systems. This is due to the fact that many more manual designs have been done, and form a basis of design experience for automation. The two oldest implementations of synthesis aids are the CMU-DA system /7/ and the MIMOLA Software System (MSS) /8/.

We will use the two cited systems as examples. Barbacci already showed the importance of the design space, in 1973 /9/. We will, therefore, start with its exploration.

The "user" machine can be seen at many different levels. Its behavior can be its ability to solve problems, or to execute a high level or intermediate language; its behavior can be its instruction set or its state transition diagram. These levels form a hierarchy with considerable design space between them. The available design space depends upon the application, but it also depends on our specification capabilities to be able to make use of the design space in synthesis.

The current capabilities are procedural or nonprocedural CHDL's. ISPS /10/ was designed to describe instruction sets. MIMOLA is a high level microprogramming language /11/. The uses of applicative languages, petrinets based languages, p-notation, calculus-based specification or relational descriptions have not yet been successfully demonstrated in synthesis. None of these languages solve the problem of describing behavior in a canonical form. There are, for instance, many different ways to describe an instruction in ISPS, although the description in the processor manual may be quite unambiguous. Only at the state transition level can we use truth tables as canonical descriptions of behavior.

One technique to decrease this problem is to allow for behavioral transformations, which are very similar to program optimizations. The CMU-DA system uses the optimization of the Value Trace /12/ for this purpose. Another technique is the specification rule in the MSS, to describe algorithms with much higher parallelism than will be implemented. The compiler then transforms from there to a more sequential level, during resource binding.

A description at the "problem solving" level opens the largest design space. Problems are solved by algorithms, with computers. Currently, this transformation from the problem to an algorithm is done manually. We call the output an "algorithmic description." In MIMOLA, HLL features, such as loop constructs, procedure calls, conditional statements, and user defined macros, support algorithmic descriptions. We call all of them macros, because they do not represent register-transfers. Only register-transfers can be mapped

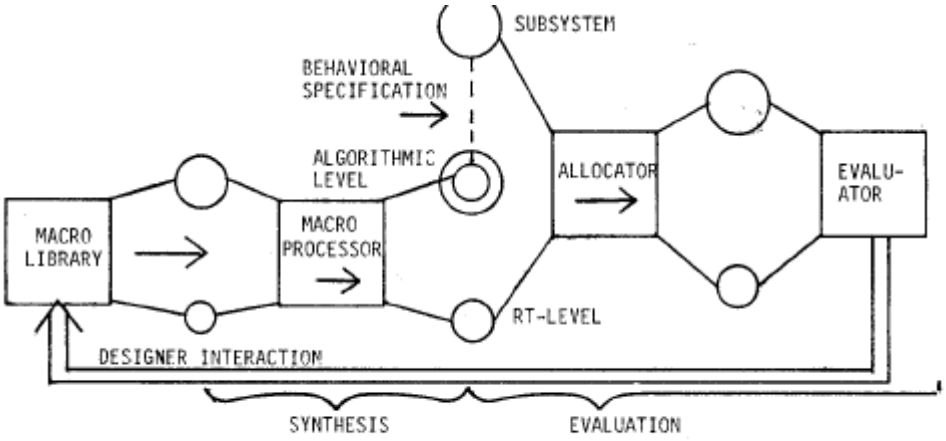


Figure 8
Algorithmic Level Refinement and Evaluation Loop

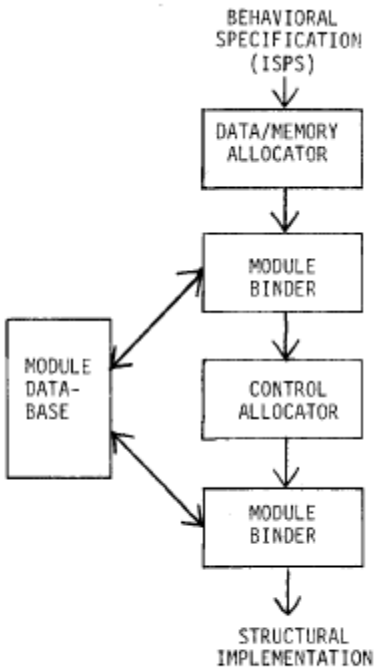


Figure 9
CMU-DA Synthesis Flow

onto structures. Macros must be expanded.
In MIMOLA the semantic of the macros is design space, which is determined by the designer. Additional degrees of freedom at the algorithmic level are, for example, the selection of addressing transformations, complex function implementations, data representations and event handling.

The transformation from the algorithmic to the register-transfer level is a refinement .step in the behavioral domain. At the moment, the only known synthesis tool is the type of macro processor such as that of the MSS (Figure 8). We call this transformation mapping, because no design decisions are made. The specifications of the macros are provided by the designer or by a library. The interesting feature here is again the evaluation that makes use of fast prototyping by synthesis. This allows the designer to control the design iterations in the feedback-loop. The next design step is the transformation of the RT-level behavioral description into a structural implementation. This can be done by either the CMU-DA system or the MIMOLA Software System (MSS).

The CMU-DA system is shown in Figure 9. As can be seen, it is divided into four steps, each of which is a synthesis. The separation into data path and control path has grown historically. The problem with this separation is that the. performance of the subsystem highly depends on both paths, and they should, therefore, be optimized in parallel. This will be done in a future version of the DA system /7/.

The MSS is shown in Figure 10. Data path and control path syntheses are not separated. The MSS has some internal optimization capabilities, and also provides evaluation of generated structures for an external feedback loop. It also accepts structural specifications for predefined structural details. This input is also used to control the synthesis process, without changing the

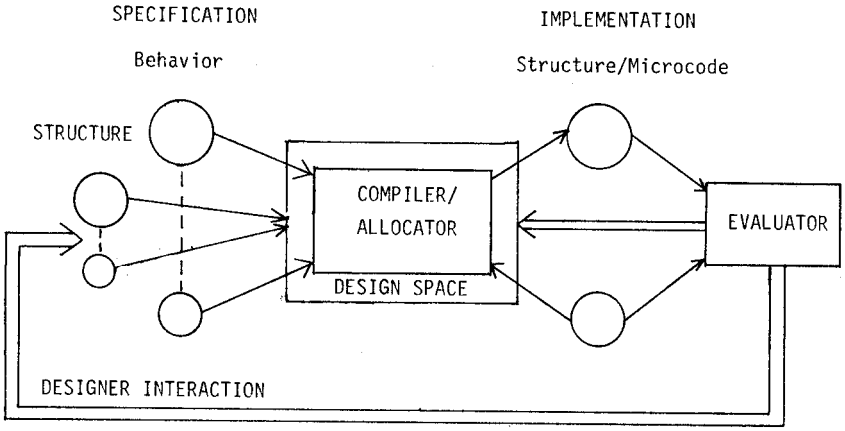


Figure 10
Structural Synthesis Loop of the MSS

behavioral specification. Another feature is the generation of microcode as part of the controller implementation. This feature can be useful beyond the design, in using the MSS as a microprogram compiler with HLL input. We will use this system as an example to explain a synthesis tool, and show its positive features and its problems. More details can be found in /8, 13, 14, 15, 16, 17/.

Let us again first look at the DESIGN SPACE. Not all of it will be available in all designs, but very often a portion of design space is traditionally not used because of cost or interconnect limitations, unavailability of off-the-shelf parts, or other reasons. Especially using LSI or VLSI techniques,- many of the limiting parameters have changed and we should evaluate the total design space for every new design. Synthesis and evaluation tools help the designer to do this quantitatively and quickly.

The DESIGN SPACE of the register-transfer level can be grouped mainly into MEMORIES, OPERATORS, CONTROL and INTERCONNECTS. The following lists are meant only to be examples, and are not consistent or complete.

MEMORIES:
Type: register, randomly, content, or serially addressable,

stack, FIFO
Structure: interleaved, multiport, dimensions, banks
Speed: technology, register file, cache, mass memory
Allocation of variables to memories
Addressing of variables in memories

OPERATORS:
Speed, functionality, multiplicity

CONTROL: Type: Moore, Mealy, synchronous, asynchronous, pipelining, programmability Interpretation hierarchy Control vector: width, decoding, overlapping Condition selection

INTERCONNECT:
Selection: bus, multiplexers, time multiplex
Speed and width

This is a large design space, even if, for example, the behavior is specified by a given instruction set. Let us look only at registers. The instruction set normally specifies a set of user-accessible registers, but we still have the choice of implementing these as individual registers or as addressable memories (register file). Registers can perform functions; register files can have single or multiple access (ports). The speed of the registers can vary. This small example already shows the manifold of decisions that a designer or a synthesis program must make.

The problem in managing the design space is that the dimensions of freedom are not orthogonal. The decision of how registers are implemented influences the data paths, the microprogram, the utilization of parallel operators, and the number of internal registers that are necessary. This decision will influence the location of critical path and the structure of a pipeline, for instance. It is very difficult, even for an experienced designer, to watch all of these side affects and make the quantitatively best decision. Computer-aided synthesis can help the designer get closer to the best decision, or do it faster. Synthesis cannot make the optimal decision, because many parts of the optimization problem have been proven to be np-complete, and are out of the reach of even fast computers.

STATE-OF-THE-ART OF SYNCHRONOUS SYSTEMS SYNTHESIS

In analyzing the state-of-the-art of computer aided synthesis, we distinguish between four tasks:

BEHAVIOR-TO-STRUCTURE TRANSFORMATION Transform any behavioral specification into a structure (regardless of physical limitations).

DESIGNER CONTROL Give the designer control over the design space and the synthesis process.

AUTOMATIC CONTROL
Give automated synthesis control over the design space, in order to meet constraints and design goals.

EVALUATION Verify that the generated structure meets all requirements of the specification (functionality, performance, manufacturability, testability). Determine unspecified parameters, and aid decision making.

BEHAVIOR-TO-STRUCTURE TRANSFORMATION

The scope is defined by the register-transfer level behavioral specification. The only limit we see in the CMU-DA and the MSS is what can be expressed in the languages (ISPS and MIMOLA). Some of these limits have been put in to encourage structured design and to enforce certain design rules. MIMOLA, for instance, enforces a synchronous clock for all storage devices. This is compatible with typical design rules for non-functional testing, using the shift register technique. Principally, extensions of the scopes of the specification languages will not cause a problem.

In the MSS, the design space is modelled as RESOURCES. The synthesis process is the ALLOCATION or binding of RESOURCES to register-transfers in the behavioral specification. Figure 11 shows a simple example. The two PASCAL statements are manually transformed

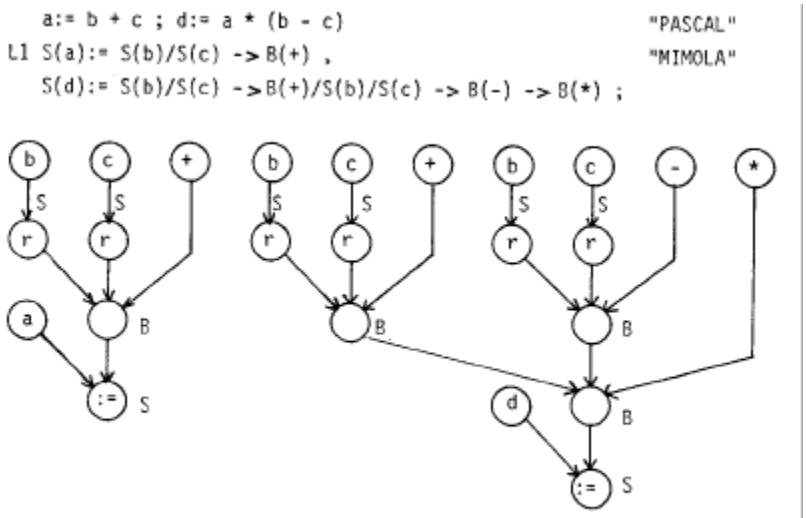
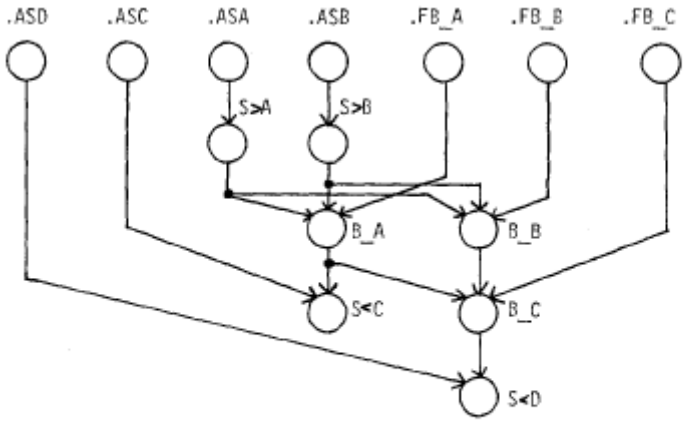


Figure 11
Behavioral Specification and Flow Graph

into two MIMOLA statements that are executed in parallel! The data dependency has been resolved by inserting the first expression into the second statement. The syntax of the MIMOLA statements can be easily understood by comparing it with PASCAL, and recognizing the postfix notation for expressions in MIMOLA. S stands for storage, and B for a dyadic operator.

The transformation into the Flow Graph is done by the MSS. It can also be easily understood. The nodes are RT-modules; the arcs are interconnections. The nodes can contain control information, which can be either constant or variable (leaves of the graph). The graph can be seen as an abstraction of the specification. It is nearly independent of the description language. All transformations are done in this graph, which also represents the internal data structure of the MSS. '

Assuming unlimited resources, the ALLOCATION process is straightforward. Scanning the graph from top to bottom and from left to right, the MSS binds resources to nodes and arcs. This is documented by attaching names to the nodes that identify the resources. In doing so, the MSS shares resources as much as possible. For example, only two output ports are necessary to read (r) all variables (b,c) from memory (S), so the MSS assigns port S>A to address b and S>B to c. In Figure 11, there are two operators (B), with exactly identical inputs; the MSS assigns an adder B_A. The third operator has a different function, thus, a second module B_B is created. Likewise, fields in the microinstruction word I are created. Figure 12 shows the synthesized structure after the allocation process. In the graph, shared resources have been collapsed. Figure 12 also shows the BOUND PROGRAM. This represents all structural information contained in the graph and the MICROPROGRAM. By a simple assembly step, the microprogram can be converted into microcode for the control store, or a PL A.




```
L1  S<C(a):= S>A(b)/S>B(c) -> B_A(+) ;
    S<D(d):= S>A(b)/S>B(c) -> B_A(+)/S>A(b)/S>B(c) -> B_B(-) -> B_C(*) ;
```

Figure 12
Allocated Structure and Bound Program

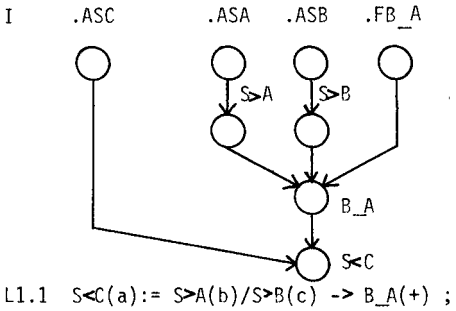
The handling of conditional statements is done in nearly the same way, as is shown in /8/. The difference is that the detailed implementation depends on the chosen control structure. Standard solutions are provided, but are not always satisfactory.

Sequential use of resources is accomplished by adding multiplexers to module inputs, as necessary.

DESIGNER CONTROL

Manual control of the design space is mandatory, since full automatic optimization is not possible. In the CMU-DA system, the designer has some control, in the way he writes the behavioral specification and by changing the Module Data Base. In the MSS, the designer can specify resources, put limitations on the number of resources, or specify parts of the structure. This gives him nearly complete control over the design space that can be expressed in the language. Still, we have to work on the interface between designer and synthesis to free him from details and allow him to express directly and comprehensively what he wants to control. This is a problem of the right design environment for language solutions which allow direct interaction. The "Smalltalk" environment /18/ may be a good starting point.

Figure 11 may again serve as an example for the control in the MSS. If the designer limits the number of operators to one (instead of the three necessary), the statements cannot be executed in the parallel fashion. The only operator, B_A, would be used sequentially by switching sources and functions. Intermediate values are automatically stored in memory locations or registers. Figure 13 shows a possible structure and the generated microprogram. The result is a much smaller structure, but three state transitions instead of one, to execute the same function as in Figure 11. Thus, the designer's decision to use only one operator has resulted in trading cost VS. performance. The MSS immediately shows him this result.




```
L1.1  S<C(a):= S>A(b)/S>B(c) -> B_A(+) ;
L1.2  S<C(1):= S>A(b)/S>B(c) -> B_A(-) ;
L1.3  S<C(d):= S>A(a)/S>B(1) -> B_A(*) ;
```

Figure 13
Limited Allocated Structure
and Bound Space

The designer can control the design space in many other ways. Limitations again exist in the scope of the languages and the ability to automatically react to all commands with valid generated structures. This is still a field of intense research.

AUTOMATIC CONTROL

Automatic control of the design space is valuable, where it frees the designer from decisions that involve many details, but that do not need intuition. Such decisions are typically based on statistical evaluations.

In the MSS, the statistical basis is the frequency of use of resources. In the behavioral specification, the relative frequency of occurance of every instruction can be specified, thus showing an approximate dynamic behavior in an assumed environment. Resource usage or utilization is thus evaluated dynamically. The strategy for

selecting resources is principally to use the most highly utilized first. The step from zero to greater zero has highest priority, if it involves a new resource. Since resources are bound into a structure, this also includes looking at connected resources. If we are, for example, requesting an operator, the most utilized may have zero utilized connections to the sources. The use would create additional interconnections and cost. Avoiding additional cost always has priority. This strategy has the ambiguity as to which resource request should be considered first. One option is to take the most expensive first. Another is to start with the critical path. Another ambiguity lies in the order of statements in the specification. In resource selection it may be appropriate to start with the most critical first. These may be either those with the highest frequency or those with the heaviest time constraint.

The utilization strategy and the "look ahead" to correlated resources automatically detects common subexpressions. We have used this feature in Figure 12. Sharing resources is a mayor cost saving factor in the automatic control of the design space. The compiler also utilizes optimization techniques. It tries to minimize the number of necessary state transitions and storage cells. Another strategy would be to minimize the execution time of individual state transitions.

These examples show that many options and strategies exist to control design space automatically. Some options can be selected by the designer in the MSS. It would be desirable to let the designer express the strategy. At the moment, however, this does not seem feasible, if we look at the programming effort to implement only one strategy. Again, we can hope that new programming environments may change this.

EVALUATION

Evaluation during the synthesis is used to control the design space automatically. Evaluation of the result of synthesis is used to compare manually controlled trial designs and to check them against the specifications.

The check for functionality is unnecessary, if we assume a correct specification and a correct synthesis. We call this correctness by construction. Since we cannot rely on either completely, simulation is used for the behavior and structure, to increase confidence in the design. Formal proofs would be desirable and seem to be feasible.

Performance evaluations can be done relatively precisely, if the delays of the resources are known. The MSS calculates execution times by automatically determining the critical path and applying usage statistics. Thus, the dynamical timing behavior is measured. Resource delays are not really known before the fabrication is completed, but good estimates can normally be achieved. The biggest problem is introduced by the uncertainty of media delays in very high speed logic. These data are partioning and layout dependent, and we have to rely on models derived from experience. These models are an open research problem.

Manufacturability also has to rely on models. Typical limitations are packaging, chip area, power dissipation, and cost. Fast prototyping capabilities for physical design are the most promising ways to get accurate figures before starting an expensive layout cycle. Since technology is changing at a very high rate, this area

will be a continuous research problem.

Currently, testability measures are not really existent at the RT-level. The only firm basis is the stuck-at fault model. Test generation time or the number of test vectors could be used as testability measures. Both are very test generator dependent, and rely on gate-level refinements. Better measures must be found.

Parameters that aid design decisions are all of `t he above, plus resource utilization figures. The MSS strategy to prioritize the highest utilization in the selection of resources has little meaning for the actual usage of a structure, but it clearly shows the designer the barely utilized resources. Utilization is an indicator of the influence on the overall performance of a device in the case of its deletion. Statistics about joint usage of resources point towards possible replacements of resources, without performance decrease. Overlay of microinstruction fields is a typical example. Thus, these statistics help the designer to make his decisions towards meaningful trial designs. In that way, random scanning of the design space is avoided. This is very important, since the space is much too large to be covered completely.

CONCLUSION

It has been shown that behavioral to structural synthesis still has many limitations, and needs additional research efforts. In the case of synchronous systems, however, two implementations already offer considerable help to the designer. Although it is an additional effort for the designer to become accustomed to this new way of thinking about designing, we think that it is now necessary to apply these systems to practical problems. Feedback is necessary to evolve these prototypes in the right direction, and in Honeywell, we have started to transfer the MSS to subsystem designers to let this process begin.

I would like to thank all of the contributors to the MSS at the University of Kiel, and my collegues in Honeywell who helped in the difficult technology transfer. I expecially have to thank Beth Wolf, who edited and generated the manuscript.

References

/1/ Trimberger, S., et. al., "A Structured Design Methodology and Associated Software Tools," IEEE Trans. CAS-28, 7, pp. 619-634 (1981).

/2/ Allen, J., Penfield, P. Jr., "VLSI Design Automation Activities at M.I.T.," IEEE Trans. CAS-28, 7, pp. 645-653 (1981).

/3/ Zimmermann, G., "VLSI Design with the MIMOLA Design System," IEE Conf. Publ. Number 200, Electronic Design Automation, pp. 277-280, Brighton 1981.

/4/ Dachaner, R. J., et. al, "The CAP/DSDL System: Simulator and Case Study," 5th Intl. Conf. Proc. Computer Hardware Description Languages, pp. 213-227, Kaiserslautern 1981.

/5/ Penedo, M. H., Berry, D.M., "The Use of a Module Interconnection Language in the SARA System Design Methodology," Proceedings of the 4th Software Engineering, Munich 1979.

/6/ Wood, W.T., et. al., "Concurrent System Description Language," Honeywell Report HR-81-271:17-38, 1981.

/7/ Director, S. E., et. al., "A Design Methodology and Computer Aids for Digital VLSI Systems," IEEE Trans. CAS-28, 7, pp. 6 3'+-6 45 , (19 81) .

/8/ Zimmermann, G., "MDS - The MIMOLA Design Method," Journal of Digital Systems Volume IV, Issue 3, pp. 337-369, (1980).

/9/ Barbacci, M. R., "Automated Exploration of the Design Space for Register-Transfer (RT) Systems," Ph.D. Dissertation, CMU 1973.

/10/ Barbacci, M. R., "Instruction Set Processor Specifications (ISPS): The Notation and its Applications," IEEE Trans. Comp., Vol C-30 , pp. 24-40 (1981) .

/11/ Marwedel, P., Zimmermann, G., "MIMOLA Report and MIMOLA Software System User Manual," Report Nr. 2/79, University of Kiel, 1979.

/12/ Snow, E. A., "Automation of Module Set Independent Register-Transfer Level Design," Ph.D. Thesis, CMU, 1979.

/13/ Zimmermann, G., "The MIMOLA Design System: A Computer Aided Digital Processor Design Method," 16th Design Autom. Conf. Proc. 1979, pp. 53-58.

/14/ Marwedel, P., "The MIMOLA Design Method: Detailed Description of the Software System," 16th Design Autom. Conf. Proc. 1979, pp. 59-63.

/15/ Zimmermann, G., "Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down Design Method," Proc. 4th Intl. Symp. on Computer Hardware Description Languages, Palo Alto 1979, pp. 33-39.

/16/ Zimmermann, G., "Computer-Aided Design of Control Structures for Digital Computers," Proc. IEEE Intl. Conf. on Circuits and Computers, 1980, pp. 103-106.

/17/ Marwedel, P., "The Design of a Subprocessor with Dynamic Microprogramming with MIMOLA," Structor and Betrieb von Rechen Systemen, Informatik Fachberichte 27, 1980, pp. 164-177.

/18/ Goldberg, A., "Introducing the Smalltalk-80 System," BYTE August 1981, Special Issue.