Automating the Design of Computer Systems

Anurag P. Gupta, Student Member, IEEE, William P. Birmingham, Member, IEEE, and Daniel P. Siewiorek, Fellow, IEEE

Abstract—Rapid advances in semiconductor technology are motivating the development of computer-aided design tools to assist human designers at higher levels in the design process. In particular, there is a need for tools that aid system-level design—the process of producing a computer system satisfying high-level specifications. The design of these systems requires a unique synthesis approach. A hierarchical select-and-interconnect methodology is described that extends the flexibility of previous approaches by allowing dynamic subproblem ordering, which is essential for this domain. M1 is a knowledge-based system that implements this approach for small computer systems. M1's design space covers five microprocessor families, and it has generated hundreds of designs, three of which were built. A set of experiments are described that show that M1 can successfully design in this complex domain, and provides a substantial increase in designer productivity.

I. INTRODUCTION

Computer-aided design (CAD) tools are essential for managing the complexity of designing integrated circuits (IC's). As the demands on a designer become greater—more function for lower cost in a shorter design time—the demand for greater designer productivity is more evident. An important technology for enhancing designer productivity is automated design synthesis tools. We characterize synthesis as a process that maps abstract specifications into detailed hardware implementations. These tools raise the level of abstraction seen by a designer, thereby reducing the complexity of the design process, and producing a number of benefits, namely reducing product-development time and the number of design errors.

Some researchers [8], [17] have used the level of abstraction of design activity to classify various synthesis tasks. For example, logic synthesis maps a set of logic equations into gates [16]; behavioral synthesis or high-level synthesis maps an algorithm, control-flow behavior, or register-transfer behavior into a register-transfer structure [13]. Several logic-synthesis systems have gained acceptance in industry, and it is expected that behavioral synthesis tools will soon become commonplace. By extrapolating this trend, it can be seen that the system level is an important area for synthesis activity. We loosely define system-level synthesis as a process that maps an abstract description of a complex system into a set of components. The components can be either IC's or cells in an application-specific IC library. Example systems include computers (e.g., workstations), embedded controllers, and avionic systems.

This paper discusses a synthesis tool called M1 (MICON Synthesizer Version 1) [9], a part of the MICON system [1], [2]. M1 has designed systems ranging from small single-board computers to sophisticated “motherboards” that contain high speed, bus-snooping caches, and complex-bus interfaces. Section II describes the system-level synthesis problem and highlights its differences from other synthesis problems. Section III describes M1's model of design based on a novel hierarchical select-and-interconnect synthesis approach. Section IV discusses the implementation of M1 as a knowledge-based system. Section V presents a set of experiments validating the operation of M1. Section VI discusses M1 in relation to other synthesis tools. Finally, our conclusions are given.

II. THE SYSTEM-LEVEL SYNTHESIS PROBLEM

The objective of system-level synthesis is to create a complete and operational computer system capable of performing general-purpose or special-application computing. System-level synthesis, therefore, requires a solution that typically includes the following subsystems:

- central-processing unit (CPU), which is realized by a microprocessor in our case;
- memory systems that may contain: caches, static random-access memory (SRAM), dynamic random-access memory (DRAM), and programmable read-only memory (ROM);
- input/output (I/O) that may contain: serial ports, parallel ports, direct memory access (DMA);
- bus interfaces;
- a wide variety of support circuitry, ranging from address-decoding logic to power-on reset and power failure detection.

Manuscript received November 5, 1990; revised May 18, 1992. This work was supported in part by the National Science Foundation under Contract MIP 8905781 and Grant DMC-8405136 to the Demeter Project, the Engineering Design Research Center, Carnegie Mellon University, an NSF engineering research center supported by Grant CDR-8522561, and the Semiconductor Research Corporation under Contract 90-DC-068. This paper was recommended by Associate Editor A. Parker.

Anurag P. Gupta is with the Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA 15213.

William P. Birmingham is with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48105.

Daniel P. Siewiorek is with the Electrical and Computer Engineering Department, and the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 9205511.
In addition, fault-tolerant and testability techniques may be required on the entire design, or portions of it.

The overall problem description is as follows. Consider an application that requires an embedded controller. A user might generate the specifications in Table I, which requires a Motorola 68008-based single-board computer containing SRAM, ROM, a parallel input/output (PIO) port, and an RS-232 compatible serial input/output (SIO) port. From this input, M1 produces a complete part and interconnection list, known as a netlist, which can be automatically converted to a set of schematics. The part list contains, in addition to all the IC's required for the design, all necessary electrical components (resistor, capacitors, etc.) and connectors. A distinction must be noted between the two ways a designer may interact with M1. In the example given here, the designer is a user, using M1 to generate a design. As explained later, M1 is a knowledge-based system; the second way a designer interacts with M1 is as a domain expert, when design knowledge is added into M1 using a specialized-knowledge-acquisition tool, CGEN [3]. The user may be a novice who exploits knowledge added to M1 by one or more domain experts.

The input specifications are based on a functional description of the elements comprising a system. The function of each subsystem is described at a high level. System-level synthesis differs from lower-level synthesis tasks in that it is loosely defined. There is no equivalent of a hardware-description language or a finite-state-machine model to clearly define the function of the artifact being designed. Using either of these representations would result in an unnecessary explosion of detail. For example, the input specification in Table I merely indicated an SIO device with an RS-232 interface; the interpretation of both "SIO" and "RS-232" was implicit in the specification; no details of the components, the algorithms they execute, or the interface protocols are specified.

Another aspect that distinguishes system-level synthesis from lower level synthesis tasks is the lack of a complete and well-defined theory for relating behavior and structure. For example, given an interconnection of logic gates, Boolean algebra can be used to compute the logic function implemented by the set of gates. Given an interconnection of adders, registers, and buses, the register transfer operations can be determined. Thus, in logic and behavioral synthesis, analytic approaches based on this theory can be used. System-level synthesis deals with relatively more complex components. For example, given an interconnection of an I8251, a MAX232 (a voltage level converter), and a connector, a general and complete theory that determines the function of this structure does not exist. The lack of a theory linking behavior and structure rules out analytic approaches for system-level synthesis. Also, system-level synthesis concentrates on producing a complete system, including components such as connectors, optical isolation buffers, voltage converters, bypass capacitors, etc., which cannot be adequately handled by existing logic and behavioral synthesis approaches.

Other important features of the system-synthesis task that influenced the design of M1 are the following:

**Large Design Space:** The design space (the set of possible designs) is very large, as parts can be used in a va-

---

<table>
<thead>
<tr>
<th>Query</th>
<th>Inputs</th>
<th>Query</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Constraints</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Board area (sq. ft.)</td>
<td>100</td>
<td>Number of PIO units</td>
<td>1</td>
</tr>
<tr>
<td>Amount of board cost (dollars)</td>
<td>250</td>
<td>PIO chip</td>
<td>1</td>
</tr>
<tr>
<td>Amount of power (mW)</td>
<td>1000</td>
<td>PIO generate interrupts</td>
<td>1</td>
</tr>
<tr>
<td>System modules</td>
<td></td>
<td>Interrupt priority level</td>
<td>0</td>
</tr>
<tr>
<td>Is a CO_PROCESSOR needed</td>
<td>N</td>
<td>PIO port direction</td>
<td>IN/OUT</td>
</tr>
<tr>
<td>Is PIO needed</td>
<td>Y</td>
<td>PIO connector type</td>
<td>MALE</td>
</tr>
<tr>
<td>Is SIO needed</td>
<td>Y</td>
<td>PIO connector size</td>
<td>9</td>
</tr>
<tr>
<td>Is TIMER needed</td>
<td>N</td>
<td>PIO port drive</td>
<td>NONE</td>
</tr>
<tr>
<td>Is DRAM needed</td>
<td>N</td>
<td>SIO</td>
<td></td>
</tr>
<tr>
<td>Is SRAM needed</td>
<td>Y</td>
<td>Number of SIO units</td>
<td>1</td>
</tr>
<tr>
<td>Is ROM needed</td>
<td>Y</td>
<td>External interface</td>
<td>RS-232</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td>Baud rate (Hz)</td>
<td>9600</td>
</tr>
<tr>
<td>Processor name</td>
<td>68008</td>
<td>SIO generate interrupts</td>
<td>1</td>
</tr>
<tr>
<td>Minimum clock spec (MHz)</td>
<td>8</td>
<td>Interrupts priority level</td>
<td>0</td>
</tr>
<tr>
<td>ROM</td>
<td></td>
<td>External port size</td>
<td>5</td>
</tr>
<tr>
<td>ROM BYTES needed</td>
<td>2K</td>
<td>Type of port</td>
<td>DTE</td>
</tr>
<tr>
<td>Name of ROM chip</td>
<td>?</td>
<td>Communication mode</td>
<td>ASYNCHRONOUS</td>
</tr>
<tr>
<td>SRAM</td>
<td></td>
<td>Connector polarity</td>
<td>MALE</td>
</tr>
<tr>
<td>SRAM BYTES needed</td>
<td>48K</td>
<td>Connector size</td>
<td>25</td>
</tr>
<tr>
<td>Name of SRAM chip</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
riety of ways to fulfill functional specifications. The problem, however, is to find the proper combination that does not violate any design constraints. This leads to combinatorial complexity for general configuration (synthesis) tasks, as has been formally noted [14].

Interaction between Design Subproblems: M1 divides the synthesis problem into a set of subproblems. There are interactions between these problems that cannot be predicted before the synthesis process begins. In addition, there is no good evaluator that will determine whether an evolving design is feasible. Thus it is not possible to determine that the design for a subsystem will yield a satisfactory complete design.

Criticality of Structure: Component interconnection is complex and can change depending on a number of factors, including the way in which a component is programmed.

Rapidly Evolving Technology Base: New ICs are constantly being developed, leading to improved components and new design styles. In order to maintain high-quality designs, M1 must be able to easily integrate new components and design styles.

III. THE M1 DESIGN MODEL.

The design model that formalizes M1's design process is presented in this section.

3.1. Overview

Designing computer systems requires selecting and interconnecting a suitable set of components from a library. In our model this process is viewed as two conceptually distinct, but closely related, searches:

- search for function (SFF): selecting components that satisfy the specifications for the design;
- search for structure (SFS): interconnecting the selected components into the design.

The issues involved in both searches are significantly different, as will be shown later. Also, the searches consider interactions between components as discussed in Section 3.2.4.

The parts in the component library are abstracted and organized into a functional hierarchy; SFF is performed along this hierarchy. After SFF selects successor part(s) in the hierarchy, SFS is done by invoking a template that is essentially a fragment of a schematic drawing that indicates the appropriate connections for the part. For example, consider a design composed of parts A, B, and C, where i ∈ {1, 2}. Fig. 1 shows the functional hierarchy for P, the abstract part for the complete design. The overall design process is illustrated as a sequence of steps. SFF for P selects A and Q; SFS for P interconnects them; similarly SFF and SFS for A and Q are performed in sequence. Note that the process alternates between SFF and SFS.

The select-and-interconnect process successively refines the functional hierarchy one level and one part at a time. Each step involves SFF followed by SFS, which alternate until the entire design has been refined into the leaf parts in the hierarchy. A traversal step starts with computing requirements using current design information, and ends with generating new relevant design information. This action is embodied in the design cycle algorithm. The ordering of the design cycles is a nontrivial task performed in an opportunistic manner. The following sections describe each of these concepts.

3.2. Design Representation

All design information is obtained from one or more hardware designers, who are called domain experts. The design model provides a framework for representing this information as described next.

3.2.1. Functional Hierarchy

A brute-force search for function is not suitable, as it would take an unreasonable amount of time due to the large search space. Two techniques are employed to reduce the size of the search space: abstraction and problem decomposition. Since the function of the part is the criterion used for abstraction and problem decomposition, the resulting hierarchy is termed the functional hierarchy.

The functional hierarchy is a directed, acyclic graph, where each node represents a part. The leaf nodes correspond to available components, and are termed physical parts. All other nodes correspond to a function, and are formed by abstracting the nodes beneath them in the hierarchy, and so are termed abstract parts. The SFF traverses the hierarchy. Nodes where an abstract part is decomposed into several successor parts are termed AND nodes; no selection is needed at these nodes. Nodes where an abstract part is refined into one of the successor parts are termed OR nodes; the successor part that best meets the input specifications is selected here.
An example functional hierarchy is shown in Fig. 2. Nodes are depicted with an arc through all links to the successor nodes. The single-board computer (SBC_0) is decomposed into the processor (PROC_0) subsystem, the memory (MEM_0) subsystem, the input-output (IO_0) subsystem, the address decoder (ADDR_DEC_0), and so forth. The generic processor PROC_0 is refined into processor-family specific parts (e.g., 68XX_PROC_0, 80386_PROC_0) that in turn are refined into physical parts (e.g., MC6800, MC6809, MC6502). The IO subsystem is decomposed into serial input-output (SIO_0) and parallel input-output (PIO_0). The dashed lines indicate links to portions of the hierarchy that are omitted from the figure.

In general, the functional hierarchy (FH) used by M1 is denoted by a graph \( (N_{FH}, E_{FH}) \) where

\[
N_{FH} = \{ n_i | n_i \text{ is a node corresponding to part } p_i \} \\
E_{FH} = \{ (n_i, n_j) | n_i, n_j \in N_{FH}, \text{ part } p_i \text{ is either abstracted to or part of part } p_j \}.
\]

The immediate predecessor and successor sets of a node \( n_i \) in the graph are denoted by \( P_{FH}(n_i) \) and \( S_{FH}(n_i) \), respectively.

M1's functional hierarchy shown in Fig. 3 is significantly different from the simpler tree-like structure of Fig. 2 in the following ways:

- While the functional abstraction shown was based on processor-family designs, mixing components from different families is possible. In fact, a design using the Zilog Z8002 processor and Intel peripherals was produced by M1.
- A single part can implement one of several functions, i.e., a part can have more than one predecessor in the hierarchy. For example, the MC6850 can be used as an SIO for either 68XX_SIO_0 or 8008_SIO_0. Thus arcs can extend from a node to any other node, skipping levels. For example, a microcomputer chip directly implements SBC_0, and would have a link to it.
- Some physical parts can have multiple functional units; i.e., a part can perform more than one function, and hence have more than one predecessor in the hierarchy. For example, the 8155 has 256 bytes of SRAM and three PIO ports. Note that the part here has all the functions, and is distinct from the part-satisfying one of several functions.
- There are actually a number of functional hierarchies, where each one represents a "significant" set of functions. For example, there are hierarchies for logic gates and crystals. Separating these functions into distinct hierarchies makes for a more logical component organization.

---

A loose convention has all abstract parts ending with "_0", although this is not required.
(68008, SIO_0, or, 1). Physical parts with multiple functional units have a unique number for each unit. Multiple links with a different functional-unit-number mean that the part can satisfy the function of all of the predecessor parts. For example, the 8155 shown in Fig. 3 has the links (G_SRAM_0, or, 1), (80386 PIO_0, or, 2), (80386 PIO_0, or, 3), and (80386 PIO_0, or, 4).

Pins: the part’s signals represented as the three-tuple (pin-name, pin-isa, pin-index). For physical parts these correspond to actual pin names; for abstract parts these correspond to generic signals or buses. The pins on an abstract part are formulated to be consistent with the functional abstraction of that part. Let Pins(n_i) denote the set of pins corresponding to node n_i. For pins that are collected to form buses, the pin-index is the numeric part of the alphanumeric pin-name; e.g., D0 has pin-index = 0. The pin-isa field indicates the function of the pin, e.g., D0 has pin-isa = DATA_BUS; it is used only in part expansion as described in Section 4.3.

Each manufacturer, technology, and package-specific part has a unique name and a distinct part model. For example, MC6850, MC68A50, MC68B50, and HD6850P are four distinct parts in the database. Each part also has a logical name termed chip-name. Several parts may be logically equivalent, and must have the same chip-name. These parts, n_i, where i is such that chip-name(n_i) is the same, can share the following information:

• Pins (n_i).
• a subset of Char(n_i). For example, (NUM_GATES, 4) is a shared characteristic for all parts that have chip-name = 7400, such as SN74LS00D, SN74F00J, and SB74S00D, while PROP_DELAY is different for each of them.
• templates.

An example part model is shown in Fig. 4.

3.2.3. Design Hierarchy

M1 builds a design by successive refinement along the functional hierarchy. The hierarchy it builds during the design process is termed the design hierarchy (DH). An example design hierarchy for an Intel 80386-based design is shown in Fig. 5. It is distinct from the functional hierarchy in the following ways:

• It is a pruned version of the functional hierarchy; since only one node is selected at an or node, subtrees for all the other successor nodes are eliminated.
• In addition to the corresponding successors in the functional hierarchy, successors of a node in this hierarchy also include any other parts that are used as support circuitry in mapping that node to its successors (i.e., included in the corresponding template, see Section 3.3.2). For example, PROC_0 is succeeded by the 80836_FPROC_0 and the 80386_CLK_GEN_0, the former coming from the SBC_0 functional hierarchy, while the latter is support circuitry that comes from another functional hierarchy. Thus, parts in other functional hierarchies are merged with the main hierarchy to form the design hierarchy. While there are several functional hierarchies, there is just one design hierarchy for a design.

• Each functional unit on a part in this hierarchy is linked to at most one abstract part. For example, in the design hierarchy, the G_SRAM_0 has only one link, in this case to the 80386_SRAM_0.

Parts corresponding to nodes in the design hierarchy have the same part model as described previously. P_{DH}(n_i) and S_{DH}(n_i) denote the set of immediate predecessor and immediate successor nodes, respectively, of a node n_i in the design hierarchy.

3.2.4. Subproblem Interactions

The problem-decomposition strategy divides the problem into smaller, logically organized subproblems, finds a solution to these subproblems, and then integrates the results into a total problem solution. For example, the functional hierarchy indicates that the single-board computer consists of subsystems, such as memory, processor, and so forth. Thus, by synthesizing each subsystem and integrating them, an entire computer system can be created.

Subproblems formulated by problem decomposition usually interact in ways that are not always predictable before runtime. M1 handles subproblem interactions by propagating information from one subproblem to another; that is, one subproblem creates the information that is utilized by another subproblem. Consider the case where the design of abstract parts A_i and A_j are related. Some design information is generated after A_i is designed. This information is subsequently used as a specification for the design of A_j to make A_j’s design compatible with the design of A_i. For example, the value for ACCESS_TIME computed using the equation

\[
\text{ACCESS\_TIME} = \text{PROCESSOR\_CYCLE\_TIME} \ast \text{NUMBER\_OF\_WAIT\_STATES}
\]
is propagated from the processor-design subproblem to the memory- and IO-design subproblems. The SFF for memory (IO) design ensures that the selected memory (IO) chip satisfies the timing constraint imposed by the processor design.

The propagated information is represented as reports. Reports are two-tuples of the form \((\text{report\_name}, \text{value})\), where \text{report\_name} is the name of the propagated variable, and \text{value} is the value of the variable. Reports can be created in one abstract part's design and used to fill specifications of other abstract parts. In this example, the report \text{ACCESS\_TIME} is created in the processor design, and is used to fill the \text{MAX\_ACCESS\_TIME} specification of the memory and IO abstract parts.

In several cases, the name of the propagated variable is not unique, resulting in ambiguity about when it is to be utilized by other subproblems. For example, the \text{NAND\_PROP\_DELAY} report, created by a \text{NAND21\_0} abstract part, may be used to compute the access time of some device. This report, however, may not be unique since there may be several \text{NAND21\_0} gates in the design. In these cases, \text{local-reports} are used for propagating values among hierarchically related subproblems. Local-reports are three-tuples of the form \((\text{local\_report\_name}, \text{value}, \text{instance\_tag})\), where \text{local\_report\_name} and \text{value} are the same as for reports, and \text{instance\_tag} identifies the subproblem that created the variable. The set of local reports created by the design of an abstract part corresponding to node \(n_i\) is represented as \text{Local-report}(n_i).

The decision about which reports should be propagated between subproblems and whether the reports should be local or not is made by the domain expert while adding knowledge to M1. Reports are accessible anywhere, i.e., they have global scope; local reports created for part \(n_i\) are accessible only in the design of \(P_{DH}(n_i)\) and \(S_{DH}(n_i)\); i.e., they have local scope.

3.2.5. Design State

The design state for a node \(n_i\) includes all information relevant to its design. It consists of all global reports and other variables (specifications, characteristics, and local-reports) in the local scope. Thus,

\[
\text{DesignState}(n_i) = \text{Reports} \bigcup_{n_i \in \text{P}} \text{Spec}(n_i) \bigcup_{n_i \in \text{S}} \text{P} \bigcup_{n_i \in \text{I}} \text{Local-report}(n_i).
\]

The use of these variables during the design process is described in the next section.

3.3. Design Process

The design proceeds down one level of the hierarchy at a time, adding successors to the design hierarchy, and creating information to be used later in the synthesis process. At each level, the design process consists of the following steps: SFF, SFS, and communicating information between subproblems. Each of these is described in detail next.

3.3.1. Search for Function

At an AND node no action is taken in SFF as the refinement is done using a template in the SFS phase. At an OR node \(n_i\) the SFF selects only one of the successor nodes \(n_j \in S_{PH}(n_i)\); the objective is to find the \(n_i\) that best satisfies \(\text{Spec}(n_i)\). This process described below uses the functional hierarchy, which reduces the search space by pre-sorting nodes according to function.

\text{Step 1: Utilizing parts with multiple functional units:}

Before searching for a new part from among the elements of \(S_{PH}(n_i)\), a check is made for unused functional units on parts that are already in the design. Using existing functional units are always preferred since it reduces the total number of parts in the design.

Two points need to be considered. First, our approach can result in locally optimal, but globally sub-optimal solutions. While the approach can exploit all units on a part once it has been brought into the design, we do not address the issue of globally selecting an optimal set of parts to be brought into the design (since the parts are selected locally at each node). For example, consider the
functional hierarchy shown in Fig. 3. The 18251 has one 80386_SIO_0 unit, while the 18274 has two such units. If a design requires two SIO ports, the 18274 is a better choice from the global perspective assuming one 18274 is cheaper than two 18251’s. Our approach, however, would design each SIO port independently, selecting an 18251 for both ports (assuming one 18251 is cheaper than one 18274); this is clearly a locally optimal, globally sub-optimal choice. Of course, the user can force M1 to select the 18274 for one of the SIO ports by explicitly specifying the chip to be used; M1 will subsequently use the remaining unit on the 18274 for the second SIO port.

Second, note that even though an unused functional unit exists, it may or may not be found to be suitable. The configuration of other functional units on a part in the design could possibly render the available functional unit to be unsuitable for its intended use. For example, the 8116 contains two clock generators, both sharing the same crystal. After the first clock generator on the 8116 has been used, a frequency is fixed for the crystal thereby constraining the second clock generator to a frequency that is a submultiple of the crystal. If later there is a CLOCK_GENERATOR_0 abstract part to be refined, the second clock-generator unit on this 8116 chip could be used only if the frequency specification of the second CLOCK_GENERATOR_0 is a submultiple of the frequency of the crystal.

Step 2: Building a set of satisfactory candidates:

All parts \(n_j \in S_M(n_i)\) are candidates. From these, M1 rejects candidates that do not satisfy constraints, which are relationships between specifications of \(n_i\) and characteristics of \(n_j\). For example, a constraint may require the ACCESS_TIME characteristic of memory chips to be less than the MAX_ACCESS_TIME specification on the memory abstract part. If no candidates are left, SFF has failed and the failure-handling module [7] is invoked.

Step 3: Selecting one candidate from the set:

If exactly one candidate is left, M1 selects it. Otherwise, M1 computes the penalty points for each candidate \(c\) using the equation

\[
\text{Penalty points}(c) = \sum_{r \in R} \frac{\text{Amount already used of } r_i}{\text{Total amount allocated of } r_i} \times \frac{\text{Amount of } r_i \text{ used by } c}{\text{Total amount allocated of } r_i}
\]

where \(R\) is the set of resources utilized by the part. Resources are things consumed by the design; a ceiling on these is initially specified by the user. Example resources are power, board area, and cost. The function assigns a normalized penalty (second fraction) for each candidate based on the amount of resource it uses. Also, the penalty is weighted by the criticality of the resource (first fraction); if the design is running out of a resource, its criticality increases. M1 selects the candidate with the least penalty points.

An interesting property of the search procedure should be noted. The weight on the penalty for each resource is the only global variable used in the selection process. The effect is dependent on when in the overall design process the selection for an abstract part is done. If the selection is done at the beginning of the design process, the weights for each resource are fairly uniform. Toward the end of the design process, the plentiful resources tend to have lower criticality and lower weight. Hence, in the beginning, M1 might select parts that consume large portions of a critical resource. Later, M1 will make more-discriminating choices. As a result of this property, M1 may make poor component selections in the context of the entire design.

Simple greedy solutions to this problem, such as adding a heuristic that always prefers the least costly part, will not work, because of unforeseen interactions between variables in the design. Other schemes, such as allocating budgets to various subsystems, are also susceptible to the same problem, which is cast as properly allocating resource budgets [4], [15]. The budgets are merely a guess to the actual resource need, and, therefore, may unnecessarily overconstrain or underconstrain a subsystem’s design.

As a result, M1’s selection process can be viewed as intelligent guess-making. Since guesses are sometimes wrong, M1 may have to backtrack to some earlier decision, try another guess, and redesign. As discussed elsewhere [7], this is a property of design problems where interactions between design parameters and design subproblems are not totally predictable.

3.3.2. Search for Structure

SFS structurally integrates the abstract part with the selected successor part(s). The representation of complex structural information is critical to M1’s operation as the system essentially integrates various "pieces of structure." M1 uses a simple, declarative representation for structure called a template. A template is a chunk of knowledge about structural design for a particular part in a particular design situation. An example template detailing the integration of a 68XX_SIO_0 abstract part to a 6850 SIO physical part is shown in Fig. 6. Salient features of a template are as follows.

- **Boundary Pins:** The pins of the abstract part being refined (the 68XX_SIO_0) constitutes the boundary of the template. Integrating the parts in the template with this boundary effectively integrates them into the rest of the design.
- **Major Functional Component:** The selected successor part is the major functional component (only for or nodes). The 6850 is the major functional component in Fig. 6.
- **Support:** In Fig. 6, the RS232_PORT_0 and the CLOCK_GENERATOR_0 support the 6850’s operation.
- **Invocation Preconditions:** Many parts may be used in a variety of ways, each requiring different interconnections or support parts. M1 requires that each
part usage have a different template. To aid SFS, each template is marked with a unique, unambiguous set of invocation preconditions. Invocation preconditions for the template in Fig. 6 are shown in Fig. 7. Note that an RS232 port is required for this template. Another template would be used if an RS422 port was required instead.

- ** Logical Parts:** Templates are organized by part chip-names, since logically equivalent parts have the same structural interconnection.

SFS chooses the proper template for abstract part \( n_i \) from among those stored in M1's knowledge base. The search process is simple: DesignState(\( n_i \)) is compared to the template's preconditions, and the template that matches the design state is selected and incorporated into the design, thereby reducing SFS to match [12]. For example, given that an MC6850 has been selected for 68XX_SIO_0 and an RS232 port is required, the single template that integrates the MC6850 and the RS232 port to the 68XX_SIO_0 is chosen.

If more than one structure matches the design state, more information must be added to disambiguate them. For example, the 6850 can be connected to the RS232 port in two ways depending upon whether the SIO port is being used as a DATA TERMINAL EQUIPMENT (DTE) or a DATA COMMUNICATION EQUIPMENT (DCE). To distinguish these structures, an additional specification, PORT_TYPE, which can be either one of these two values, is added to the 68XX_SIO_0 part. If this cannot be done, then both structures are equivalent in all known respects and M1 needs just one.

Templates represent knowledge of how M1 structurally maps from one level in the functional hierarchy to a lower level. Designs are created by combining templates in such a way as to satisfy the designer's requirements. Thus given a functional hierarchy and a set of templates, the designs that a system can produce are enumerable a priori (except as noted in Section 4.3). In order to cover a wide range of designs, a large number of templates are needed. Note, however, that by organizing the parts in a hierarchy, the number of templates to be entered is substantially smaller than the number of unique designs that can be produced.

**Fig. 7. Invocation preconditions for template mapping 68XX_SIO_0 to 6850.**

### 3.3.3. Information Communication: Report Propagation

As pointed out previously, subproblem interactions are handled by propagating reports (or local-reports) from one subproblem to another. Let the design of \( n_1 \in N_{DH} \) and \( n_2 \in N_{DH} \) interact, and let there be a report that is computed after \( n_1 \) is designed that is used in the design of \( n_2 \). Report propagation consists of the following two steps.

**Report Calculation:** After SFF and SFS, a suitable report \( R \) based on the solution of the \( n_1 \) design subproblem is created and its value is calculated using any of the variables in DesignState(\( n_1 \)).

**Specification Filling:** Before starting SFF for the \( n_2 \) design sub-problem, the report \( R \), together with other variables in DesignState(\( n_1 \)), is used to fill the value of a specification in Spec(\( n_2 \)).

The equations used to compute the value of the report and the specification can be arbitrarily complex functions of the variable in the current design state.

### 3.4. Putting it All Together

In this section a formal model of the design methodology is given.

### 3.4.1. Design Cycle

The design process described above is represented as a sequence of steps termed the design cycle, which forms M1's problem-solving method. The design cycle refines a node \( n_i \) into one or more of the successor nodes in \( S_{FH}(n_i) \), possibly introducing additional support-circuitry parts into the design. Each step is denoted by \( d_{step\_names} \), and is explained below:

**Specification (\( d_{spec} \)):** Each \( n_k \in Spec(n_i) \) is assigned a value using either of the following two mechanisms:

- **Query:** The user is prompted with an appropriate question and the response is used to fill in the specification.
- **Formulation:** The reports, specifications, and characteristics of \( P_{DH}(n_i) \) are used to fill in the specification.

For each specification, the mechanism to be used is given by the domain expert while adding knowledge to the system. For query, the domain expert must provide the appropriate question to be posed to the user. For formulation, the domain expert must provide the appropriate
equation. A design cycle is executed for a node only when all its specifications are filled.

Selection \(d_{\text{select}}\): At an OR node, one of \(S_{\text{fin}}(n_i)\) is chosen; no action is taken at an AND node. This step has been described in detail in Section 3.3.1.

Structural Synthesis \(d_{\text{calc}} + d_{\text{template}}\): A template is asserted for structurally mapping \(n_i\) to one or more of \(S_{\text{fin}}(n_i)\), as described in Section 3.3.2.

Calculation \(d_{\text{calc}}\): Reports are generated in one of the following ways:

- **Immediate calculation \(d_{\text{calc IMM}}\):** Reports are generated immediately. Any of the variables in Designstate \((n_i)\) may be used to compute the value of the report.
- **Delayed calculation \(d_{\text{calc_DELAY}}\):** Structural synthesis may bring several abstract parts into the design, and the reports to be generated at this point may depend on reports generated by the design of those abstract parts. Hence, generation of some reports is delayed until all abstract parts introduced by a template are completely designed (i.e., have been driven to physical parts). Also, the purpose of some delayed reports is to signal the completion of an abstract part’s design.

Step \(d_{\text{select}}\) realizes the SFF, steps \(d_{\text{calc}}\) and \(d_{\text{template}}\) realize the SFS, and steps \(d_{\text{spec}}\) and \(d_{\text{calc}}\) realize the propagation of design information.

When selection fails, the failure-handling subtask involves redesigning \(n_i\) and possibly other parts in the design. This redesign process, which requires a distinct problem-solving method, is performed by another module [7].

### 3.4.2. Subproblem Ordering

The functional hierarchy, while dividing the problem into smaller, simpler subproblems, does not give an ordering for solving the subproblems. For example, the hierarchy does not imply that the processor should be designed before memory, or vice versa. Ordering depends upon the interaction, or dependencies, between subproblems; a report generated during the design of one abstract part may be needed to fill the specifications of another.

In the simplest type of problem, the dependencies between subproblems are determined by analysis of the problem domain, and an ordering is derived such that each subproblem is attempted only after other subproblems that provide information for it have been solved. For this problem type the subproblem ordering is fixed. In a less-restrictive problem class (e.g., [5]), the ordering is fixed, but there is interaction between subproblems, possibly leading to backtracking and redesign. For example, the solution to subproblem \(A\) assumes a value that is computed later in subproblem \(B\). If the assumed value is incorrect, \(A\) can be redesigned.

In our case, ordering is more complex than that for any of the above problem types. It is not possible to find a reasonable, invariant subproblem ordering since the dependencies can change. Recall the formula used to calculate the access time for memory and IO:

\[
\text{ACCESS\_TIME} = \text{PROCESSOR\_CYCLE\_TIME} \times \text{NUMBER\_OF\_WAIT\_STATES}
\]

Note that the \text{PROCESSOR\_CYCLE\_TIME} value depends on the selection of a processor part. So, it is clearly advantageous to wait for the processor to be chosen before deciding on memory and IO. Under normal conditions, memory and IO subsystems can be designed independently. If a particular design, however, requires a direct-memory-access controller, then the access time of IO will affect the access time of memory and vice versa, and the subproblem ordering will change. It is not practical to enumerate all possible subproblem interactions \textit{a priori}, especially considering that as new parts or subsystems are added, unforeseeable interactions may arise.

Inability to order subproblems \textit{a priori} has an important implication for M1. M1 cannot use a fixed control scheme, such as plans, for designing a system; the order in which subproblems are attacked must be dynamically computed. This dynamically determined subproblem ordering is a novel, powerful feature of our design model. The subproblem of designing the part associated with a node is solved by the application of the design cycle to that node. Before a design cycle can be started for a node, its specifications must be filled. Thus, conceptually the algorithm for subproblem selection is given below:

\[
\forall s_i \in \text{Spec}(n_i), \text{if } s_i \in \text{DesignState}(n_i)
\]

then activate design cycle for \(n_i\).

This procedure results in a data-driven search process—when a node is eligible to be designed, it will be. This provides the flexibility needed to add newer parts (subproblems) that might change the dependencies. Also, the partial ordering imposed by the subproblem interactions is satisfied. Note that conceptually, several design cycles could be active simultaneously. To keep the design process manageable in M1, however, nodes that are not yet designed are kept on a queue. A design cycle is scheduled for the first node on the queue until the queue is empty. If no progress is being made, then a deadlock is reported to the user, who may take corrective action, such as changing specifications or modifying the knowledge base.

M1’s subproblem-ordering algorithm in Fig. 8 is an improvement on the brute-force queue-scheduling algorithm outlined above. A queue of only those nodes that can be designed is maintained; sets store other nodes that are waiting for additional design information, or waiting for successors to be designed. \text{DESIGNCYCLE1} and \text{DESIGNCYCLE3} are portions of the design cycle that evaluate the nodes in the waiting sets to check if they are
IV. The M1 System

So far, we have described the M1 design model at a conceptual level. In this section, we present its implementation.

4.1. Knowledge-Based Implementation

A knowledge-based approach to implementing M1 was chosen for several reasons:

- The programming paradigm easily supports data-driven computation. The subproblem ordering task can be conveniently cast in a data-driven approach.
- Rules naturally represent the design knowledge used.
- A knowledge base is easily extensible as compared to modifying procedural code. This allows M1 to track technology changes.

M1 is implemented as a rule-based expert system in the OPS83 [10] language.

Part models are stored in a central, relational database. The knowledge that M1 reasons with while creating a design is stored in the knowledge base. Since the performance of a knowledge-based system depends strongly on both the quality and breadth of its knowledge, especially in knowledge-intensive application areas such as design [11], building the knowledge base is a critical part of the development process. M1's knowledge base has been engineered to support automated knowledge acquisition [3], enabling high-quality knowledge bases to be easily generated by hardware designers unfamiliar with either M1's implementation or rule-based programming techniques. Here we describe the structure of M1's knowledge base.

4.2. Knowledge Base

M1's knowledge base is divided into disjoint partitions, each corresponding to a particular step in the design cycle. The knowledge base consists of a set of rules with each rule residing in one and only one partition. Strict separation of rules eliminates interactions in the rule base and facilitates knowledge acquisition. Each of the following knowledge-base partitions in M1 corresponds to steps in the design cycle as described in Section 3.4.1:

- specification knowledge ($k_{spec}$) supports $d_{spec}$,
- selection knowledge ($k_{select}$) supports $d_{select}$ (includes penalty function in $k_{select\_rule}$),
- structural synthesis (composed of two partitions, see Section 4.3 for details);
- part-expansion knowledge ($k_{case}$) supports $d_{case}$,
- structural-design knowledge ($k_{template}$) supports $d_{template}$,
- calculation knowledge ($k_{calc}$) supports $d_{calc}$.

These partitions are termed design knowledge. Knowl-
edge for the failure-handling subtask is described elsewhere [7].

Each partition is activated in sequence (as given in Fig. 8) to execute several rules to accomplish a specific task. An additional partition, architectural and support knowledge (karch), includes the subproblem-ordering and design-cycle algorithms, and other support functions.

4.3. Structural Synthesis

Structural synthesis instantiates a part chosen in the selection step and integrates that part into the design. The number of parts to be instantiated depends on specifications that are filled at run time. For example, the number of 68XX_SIO_0 parts to be brought into the design depends upon the number of SIO ports desired by the user; the number of SRAM physical parts to be instantiated depends upon the amount of memory required by the user. As the specifications cover a large range of values, the amount of integration knowledge can explode. Fortunately, parts can be viewed as constituting a regular structure, and the knowledge of forming the structure is independent of the knowledge of integrating this structure into the design. For example, the SRAM chips can be viewed as constituting an array structure; the knowledge of designing an array of SRAM’s is independent of the knowledge needed to integrate the array into the memory subsystem. This dichotomy leads to the division of the structural-synthesis step into the following two separate substeps.

Part expansion (d_case). A variable number of successor parts are instantiated and integrated into the structure as described below.

Structural design (demplate). The successor parts in the structure are integrated with the abstract part.

There are two common structures used in part expansion:

- Independent: Each instance of the successor part is independent of the other. For example, each 68XX_SIO_0 instantiated is independently integrated.
- Array: The successor parts are organized as an array with variable dimensions. d_template is used for the boundary parts in the array only. Integration of the other parts of the array is decoupled from d_template, and also from the dimensions of the array. For example, memory chips are organized as an array with dimensions depending upon the data-bus width and the amount of memory required. The address-bus pins and chip-select signals connect across rows, while the data-bus pins connect across columns. This structure is used when the keyword ROWS and COLUMNS are specifications for an abstract part. To make this procedure generic, pin function, represented in part-models as PIN-ISA, is used to express these interconnections.

Forming these structures from templates would require a large number of them. Hence, k_case is represented in procedural form.

V. Experimental Results

M1 is a fully functional system-level design tool, with a knowledge base of over 3700 rules. Experiments in building and utilizing the knowledge base are described below.

5.1. Knowledge Gathering

The experiments described here were designed to simulate as closely as possible the conditions under which additions to M1’s knowledge base would be made in the field. A version of M1 with minimal knowledge, i.e., only k_select, k_case, and k_arch, and no design knowledge, was used as the baseline system. The designers added knowledge about four microprocessor families: the Motorola 6809, 68008, 68010, and the Intel 80386 (without caches or bus-interfaces). These designs were neither developers of M1 nor familiar with AI programming techniques. They used CGEN [3] to build the knowledge base. No direct coding or editing of rules in the knowledge base was required.

In this period, the designers developed the functional hierarchy, portions of which were shown in Figs. 2 and 3. Each designer added knowledge for one of the four microprocessor families. For parts shared between designs, knowledge would be added by the person first encountering a design cycle to refine that part; this knowledge would be made available to the other designers by periodic release of shared knowledge-base files. The results of the experimental period are given in Table II, and substantiate the following conclusions.

- M1 provides an effective design model for representing knowledge for designing single-board computers.
- The model effectively supports automated knowledge acquisition, since a large number of rules could be acquired in a relatively short period of time (about four man-months).
- M1 allows for the sharing of knowledge between designs. For example, fewer rules were added for the 68008-based designs than for the 6809-based designs, since a large number of generic parts added for the latter design could be utilized in the former.

After this experimental period, knowledge about several other designs has been added to M1’s knowledge base. M1 now has 3700 rules, 3600 of which were acquired using CGEN; the rules contain design information for over 600 parts, around 250 are physical parts. The size of M1’s knowledge base is relatively large; most design systems reported in the literature have on the order of 500–1000 rules, or their equivalent. Because of the knowledge base’s highly modular organization, rules can be added until memory limits of the computer are reached.

The number of physical parts is relatively small since we added only a few parts for each function.
TABLE II
KNOWLEDGE GATHERED IN EXPERIMENTAL PERIOD

<table>
<thead>
<tr>
<th>Family</th>
<th>Number of Templates</th>
<th>Number of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>M6809</td>
<td>36</td>
<td>343</td>
</tr>
<tr>
<td>M68008</td>
<td>38</td>
<td>173</td>
</tr>
<tr>
<td>M68010</td>
<td>19</td>
<td>147</td>
</tr>
<tr>
<td>IS0386</td>
<td>79</td>
<td>256</td>
</tr>
<tr>
<td>Total acquired by CGEN</td>
<td>221</td>
<td>919</td>
</tr>
<tr>
<td>Overall Total</td>
<td>221</td>
<td>1050</td>
</tr>
<tr>
<td>Parts (abstract + physical)</td>
<td></td>
<td>382</td>
</tr>
</tbody>
</table>

5.2. Design Space

With its knowledge base and database, M1 is able to create a wide variety of designs. Partial specifications used for some of these designs are shown in Table III. Each of the designs represents a point in the large space of designs that can actually be generated. A number of these designs were produced by our industrial affiliates.

- As expected, the part count, rule firings, and run-times increase for designs with more function (i.e., larger memory, more IO ports).
- Specification Set 1 and 2 (and Set 8 and 9) are the same except for the amount of SRAM memory. To build larger memory, M1 used a memory chip of larger density resulting in the same part count.
- Set 5 has just one more PIO port than Set 4 (similarly, for Set 9 and 10). As the 6821 chip used for the first PIO port has two ports on it, M1 uses the second port for Set 5. Hence, the design for Set 5 has only one more part than that for Set 4, the additional part being a connector for the second port.

It should be pointed out that this table only captures the major parameters of the design space covered by M1. For example, consider the design of an SIO subsystem in a 6809-based design. The table merely lists the number of SIO devices as a parameter, but the actual parameters include the following (see also Fig. 4):

- **External interface**: RS-232 or RS-422 (2 choices);
- **Baud rate**: 300, 1200, 2400, 4800 and 9600 (5 choices);
- **Baud-rate switchability**: fixed by hardware switches, or by software (3 choices);
- **Interrupt structure**: If interrupt-driven, then 3 priority levels (1 + 3 choices);
- **Port type**: DTE or DCE (2 choices);
- **Port size**: 3, 5, or 9 lines (3 choices);
- **Connector type**: Male or female (2 choices);
- **Connector size**: 9 or 25 pin (2 choices).

Thus the actual design space for the 68XX SIO port covers

\[ 2 \times 5 \times 3 \times 4 \times 2 \times 3 \times 2 \times 2 = 2880 \]

possible designs. Similar analysis of the other subsystems can be done to show that the actual design space covered by M1 is much larger (approximately 10³⁰) than that apparent from the table.

The designs were verified by inspecting the schematic manually generated from the M1-generated netlist. In some cases, errors were found in the designs. These errors were traced back to errors in templates input to CGEN, which then propagated to M1's knowledge base; these templates were corrected and new rules were generated, finally resulting in error-free designs.

The designs produced by M1 compare favorably to manually produced designs. They work at expected clock rates, and use the same number of parts as a human designer, except in the rare cases where M1's local part selection causes M1 to make a globally sub-optimal choice. In these cases, M1 would be rerun and coerced by the user to select the better chip. This is acceptable, as M1's strong point is its rapid-prototyping capability—producing comparable results in a smaller fraction of time than it would take for manual design.

Three of the M1-generated designs have been fabricated.

- The 68008-based design in specification Set 4 from Table III resulted in a two-layer 5-in. × 5-in. printed circuit board, and is running at the designed clock rate of 8 MHz.
- The 20-MHz 80386-based design in specification Set 12 from Table III was fabricated, and resulted in a six-layer 10-in × 13-in printed circuit board. The motherboard design works and is interfaced to other AT boards (additional memory, disk-controller, etc.). This workstation is comparable to an 80386-based machine on the market, and underscores M1's capability of effectively handling large, sophisticated designs.
- The 80188-based design in specification Set 16 from Table III was fabricated, and resulted in a two-layer 8-in × 4-in, printed circuit board. The board drives a custom CGA display interface board and is the computational engine in wearable portable computer that aids navigation through architectural blueprints.

5.3. Tradeoff Capabilities

M1 attempts to meet both the board-cost and board-area constraints. In the selection step of the design cycle, the weighted objective function makes an implicit cost-area tradeoff.

An experiment illustrating M1's cost-area tradeoff capability in selecting from among three ROM chips was conducted [9]. M1 always selected the chip with least cost and area expense. In cases where different chips resulted in minimum cost and area, M1 correctly biased the selection toward the chip with minimum area (cost) if the design was running out of area (cost) resource. While this experiment illustrates how M1's part-selection function works, it also brings out a limitation of M1. M1 implicitly
### TABLE III
**PARTIAL SPECIFICATIONS OF THE DESIGNS CREATED BY M1**

<table>
<thead>
<tr>
<th>Design Number</th>
<th>Processor</th>
<th>Memory</th>
<th>IO</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processor</td>
<td>Memory</td>
<td></td>
<td>Physical Part-Count</td>
</tr>
<tr>
<td>1</td>
<td>M68008</td>
<td>SRAM: 32KB, ROM: 4KB</td>
<td>PIO: 1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>M68008</td>
<td>SRAM: 60KB, ROM: 4KB</td>
<td>PIO: 1</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>M68008</td>
<td>SRAM: 60KB, ROM: 4KB</td>
<td>SIO: 1</td>
<td>28</td>
</tr>
<tr>
<td>4(^\d)</td>
<td>M68008</td>
<td>SRAM: 60KB, ROM: 4KB</td>
<td>PIO: 1</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>M68008</td>
<td>SRAM: 60KB, ROM: 4KB</td>
<td>SIO: 1</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>M6809</td>
<td>SRAM: 32KB, ROM: 4KB</td>
<td>PIO: 1</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>M6809</td>
<td>SRAM: 32KB, ROM: 4KB</td>
<td>SIO: 1</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>M6809</td>
<td>SRAM: 32KB, ROM: 4KB</td>
<td>PIO: 1</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>M6809</td>
<td>SRAM: 60KB, ROM: 4KB</td>
<td>SIO: 1</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>M6809</td>
<td>SRAM: 60KB, ROM: 4KB</td>
<td>PIO: 2</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>180386</td>
<td>SRAM: 60KB, ROM: 8KB</td>
<td>SIO: 1</td>
<td>79</td>
</tr>
<tr>
<td>12(^\d)</td>
<td>180386</td>
<td>SRAM: 1 MB, ROM: 256KB</td>
<td>SIO: 1</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>180387</td>
<td>Cache: 32KB</td>
<td>Keybd. Ctrl.: 1 Bus Interface: AT</td>
<td>—</td>
</tr>
<tr>
<td>13(^\d)</td>
<td>1803865X</td>
<td>DRAM: 1 MB, ROM: 16K</td>
<td>PIO, SIO Controllers: Video, Keyboard, Hard Disk Bus Interface: AT</td>
<td>252</td>
</tr>
<tr>
<td>14(^\d)</td>
<td>18086</td>
<td>SRAM: 1 MB, ROM: 4KB</td>
<td>PIO: 1, SIO: 1</td>
<td>33</td>
</tr>
<tr>
<td>15</td>
<td>M6802</td>
<td>SRAM: 8KB, EPROM: 47KB, EEPROM: 8KB</td>
<td>SIO: 1, PIO: 6</td>
<td>26</td>
</tr>
<tr>
<td>16(^\d)</td>
<td>180188</td>
<td>SRAM: 8KB, ROM: 512KB</td>
<td>Bus Interface: AT</td>
<td>32</td>
</tr>
</tbody>
</table>

\(^1\)On a MicroVax II, UNIX 4.3BSD.
\(^\d\)Actual run-time plus time spent accessing shared database over the network. Database server runs on another MicroVax II, UNIX 4.3BSD.
\(^\d\)Some data missing because experiment was run at an industrial affiliate's lab.
\(^\d\)On a 386-based Intel 301, System V.
\(^\d\)On a DEC 3100, Unix V4.2.
\(^\d\)The unusually larger value for these is due to the sharing of the database with other programs that happened to be running at the same time.

\(^\d\)Actually fabricated.

assumes that the memory array is homogeneous (i.e., made with identical chips). This reduces the design-space size, but in some cases, M1 may not be able to obtain the optimal result.

Another capability of M1 is maximizing performance while satisfying the global constraint on cost and area. For example, M1 will add wait-states to peripherals instead of using costlier chips to meet a global cost constraint. In another experiment, M1 was given a choice of physical parts with different costs and access times (as shown in Fig. 9) for the SIO peripheral. A minimum clock speed of 2 MHz was specified, resulting in an access-time constraint of 210 ns on the SIO device. With a loose cost constraint, M1 used a MC68850 as the SIO device. As the cost constraint was made tighter, the MC68850 became too costly, and M1 added one wait-state on the peripheral (in the Failure-Handling step); now M1 switched to the HD6850P since it was the cheapest.

5.4. Productivity Enhancement

M1, together with other tools in the MICON system, is targeted toward increasing the productivity of the user and the domain expert or designer. While it may be conceptually obvious that the hierarchical organization of design knowledge would increase design reuse and automation
would reduce design time, in this section we provide some anecdotal evidence\textsuperscript{10} to substantiate these claims.

An interesting case study of using MICON was conducted with two hardware designers from an industrial affiliate. These designers went through the complete process of learning about MICON, using the tools to add knowledge about a new design to M1's knowledge base, and then used M1 to produce a new design. New parts in the design included a Motorola 6802 microprocessor, new SIO and PIO chips, and a keyboard controller. The designers were trained in using the tools in two days, added the new design knowledge in the next two days,\textsuperscript{11} and subsequently used M1 to generate a complete new design in an hour.\textsuperscript{12} From M1's perspective, the following points are noteworthy.

- Since the 6802 was a 68XX-family microprocessor, substantial leverage was derived from reusing the knowledge previously entered for the 6809 microprocessor of the same family. Existing designs for generic parts such as RS-232 ports also reduced the number of tasks for designers.
- M1 could generate the new design within an hour. It was estimated that the designers would need two weeks to produce a design of the same complexity from scratch. Thus, for the design space covered by its knowledge base, M1 significantly reduces design time, thereby increasing user productivity by two orders of magnitude. Since it took almost a week to add the knowledge into M1, designer productivity is doubled.

For a more accurate comparison, the effort in building M1's knowledge base must be amortized over all the de-

\textsuperscript{10}A more concrete analysis is not possible, since for both an evolving M1 and a designer who goes through several projects, one cannot objectively measure the design effort for a subset of the designs. For example, a designer may spend time learning about a new part, but this time must be amortized over all the designs using this part that he produces over his lifetime. Of course, this is difficult to quantify.

\textsuperscript{11}A member of the project occasionally provided guidance.

\textsuperscript{12}Actually, two runs of M1 were required; the netlist generated by the first run had some errors, due to incorrect design knowledge provided by a template, which were fixed within two hours.

VI. RELATED WORK

Behavioral approaches were shown to be unsuitable for the system-level synthesis problem in Section II. The type of design performed by M1 has been referred to as configuration [14], where a set of components are assembled to meet input specifications. While other configuration systems differ in the problem-solving approach used, they all assume that subproblem interactions are known before run time. This allows these systems to use prestored control knowledge, usually in the form of a plan, or set of plans.

As stated earlier, subproblem interactions can not be predicted when creating computer systems. Thus, M1 extends the models assumed by these systems by determining interactions and scheduling design subproblems at run time. This adds an important degree of flexibility, which is essential for M1 to perform its task. Furthermore, the elimination of centralized-control knowledge facilitates the knowledge-acquisition task, which is essential to the viability of knowledge-based-design tools.

In addition, most configuration systems reason only with function, or structure, but not both. For example, these systems assume a fixed architecture, or a relatively small set of architectures. M1 explicitly reasons about the proper structure for each part of the design, and must choose between a large number of them. This reasoning capability significantly extends the range of designs that can be produced by M1, and overcomes a fundamental limitation noted in prior work [3].

VII. SUMMARY AND CONCLUSIONS

A model to support synthesis of digital systems at higher abstraction levels has been presented. M1, a knowledge-based system founded on the model, can design single-board computers from high-level specifications using a library of parts. Experiments with M1 have demonstrated the model to be effective for both organizing and utilizing design knowledge. The tool has generated hundreds of designs, three of which were built, and has been shown to increase designer productivity by a factor of at least two and user productivity by at least two orders of magnitude. M1 has been successfully integrated with an automated knowledge-acquisition tool. This combination of tools has allowed hardware designers unfamiliar with AI or M1's implementation to develop a large knowledge base. M1's knowledge base currently contains...
extensive expertise to design with five microprocessor families.

M1 has significantly extended the capabilities of configuration design tools in two ways. First, dynamic calculation of subproblem execution order allows the tool to track technology change, and facilities knowledge acquisition. Second, simultaneous reasoning about function and structure enables M1 to generate structures that meet the specifications of each design, instead of relying on a canonical one.

Extrapolating the trend toward higher abstraction levels for digital design, the system-level synthesis task will become increasingly important. The model presented in this paper could form a basis for building knowledge-based design tools to address these tasks.

ACKNOWLEDGMENT

The authors would like to thank all members of the MICON project, especially those who trained M1 and were instrumental in building its large knowledge base. Ajay Daga has contributed greatly to this work by identifying key improvements to the design cycle.

REFERENCES


Anurag P. Gupta (S'87) received the Bachelor of Technology degree in electrical engineering from the Indian Institute of Technology, Delhi, in 1986 and the M.S. degree in electrical engineering from Carnegie Mellon University, Pittsburgh, PA, in 1988. He is currently working toward the Ph.D. degree in the Department of Computer Engineering at Carnegie Mellon University.

His research interests include CAD for synthesis of digital systems, timing verification, knowledge-based design systems, and computer architecture.

Mr. Gupta is a member of the Association of Computing Machinery.

William P. Birmingham (S'80-M'83) received the B.S. degree in electrical engineering in 1982, the M.S. degree in computer engineering in 1983, and the Ph.D. degree in electrical engineering in 1985, all from Carnegie Mellon University. He is currently an assistant professor in the Electrical Engineering and Computer Science Department at the University of Michigan, where he is a member of the Advanced Computer Architecture Laboratory and the Artificial Intelligence (AI) Laboratory. At the University of Michigan, he is pursuing research in the areas of CAD and AI. Birmingham has spent several years in industry developing CAD tools. While at Carnegie Mellon, Birmingham participated in the development of advanced Computer-Aided Design (CAD) tools and the design of several integrated circuits. He has worked at Mentor Graphics, United Technologies Microelectronics Center, and Carnegie Group Incorporated. Birmingham was a founder of Tri- meter Technologies, where he developed AI-based CAD tools.

Dr. Birmingham is a member of the Association of Computing Machinery and Sigma Xi. He is also a National Science Foundation Presidential Young Investigator.

Daniel S. Siewiorek (S'67-M'72-SM'79-F'81) received the B.S. degree in electrical engineering from the University of Michigan, Ann Arbor, in 1968, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, in 1969 and 1972, respectively.

He is a professor in the School of Computer Science and the CIT Department of Electrical and Computer Engineering at Carnegie Mellon University, where he helped to initiate and guide the CMU project that culminated in an operational 50-processor multiprocessor system. He has been a key contributor to the reliability and maintainability of over two dozen commercial systems. His current research interests include computer architecture, reliability modeling, fault-tolerant computing, modular design, and design automation. He has conducted research and served as a consultant to several commercial and government organizations, while serving on six technology advisory committees. He has also written textbooks in the areas of parallel processing, computer architecture, reliable computing, and design automation, and over 275 papers. He is currently Director of CMU's Engineering Design Research Center Design for Manufacturing Laboratory.

Dr. Siewiorek is a member of the Association of Computing Machinery, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.