1988

Automated knowledge acquisition for a computer hardware synthesis system

William P. Birmingham
Carnegie Mellon University

Daniel P. Siewiorek

Carnegie Mellon University.Engineering Design Research Center.

Automated Knowledge Acquisition for a Computer Hardware Synthesis System

by

William P. Birmingham and Daniel P. Siewiorek

EDRC 18-06-88

Automated Knowledge Acquisition for a Computer Hardware Synthesis System

William P. Birmingham and Daniel P. Siewiorek

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
412-268-6665
wpb@blee.demeter.cs.cmu.edu

13 May 1988

This work was funded by in part by National Science Foundation grant DMC-8405136 to the Demeter Project, and the Engineering Design Research Center, Carnegie Mellon University, an NSF engineering research center supported by grant CDR-8522616. funding agencies.

ABSTRACT

The MICON Synthesizer Version 1 (M1) is a rule-based system which produces a complete small computer design from a set of abstract specifications. The ability of M1 to produce designs depends on the encoding of large amounts of domain knowledge. An automated knowledge acquisition tool, CGEN, works symbiotically with M1 by gathering the knowledge required by M1. CGEN acquires knowledge about how to build and when to use various computer structures. This paper overviews the operation of CGEN by providing an example of the types of knowledge acquired and the mechanisms employed. A novel knowledge-intensive generalization scheme is presented. Generalization is a pragmatic necessity for knowledge acquisition in this domain. A series of experiments to test CGEN's capabilities are explained. A description of the architecture and knowledge-base of M1 is also provided.

1. Introduction

Knowledge-based systems have gained acceptance in computer-aided design systems for tasks ranging from diagnosis to hardware synthesis. Hardware synthesis, the creation of hardware from a set of high level specifications, is an area of success in the application of knowledge-based system concepts. The success of these systems depends not only on a good formulation of the synthesis problem, but also on the encoding of large amounts of domain knowledge [11]. In some domains, such as computer design, technology evolves so quickly that the knowledge-base of a synthesis tool is never complete. The knowledge-base must be continually enhanced to enable the synthesis system to exploit opportunities' afforded by new technology. This phenomenon has shifted the major effort of system development from building problem-solving architectures to developing knowledge-bases. Therefore, automated knowledge acquisition tools are critical to long-term success of knowledge-based synthesis systems.

The MICON Synthesizer Version 1 (M1)[4] is a rule-based system written in OPS\83[13] which produces a complete small computer design from a set of abstract specifications. During its design process, M1 performs two searchs. The first search concerns translating the specifications into physically realizable components. This search process is called the search for function. During the second search, a valid computer structure is formed from components selected in the search for function. The second search is called the search for structure. Both searchs occur in large spaces. However, domain knowledge is exploited by M1 to efficiently guide these searches.

An automated knowledge acquisition tool, CGEN (Code Generator) [5], works symbiotically with M1 by gathering the knowledge required by M1 to generate a design. CGEN acquires knowledge about how to build and when to use various computer structures[1]. The knowledge is available from hardware designers, domain experts, who are not familiar with the implementation of M1. Since a large number of structures are necessary to build a computer system, CGEN must provide an interface which is convenient to the domain experts. The interface allows designers to express computer structures with schematic drawings and to describe other types of design knowledge (e.g. constraints) without having to write complex programs.

This paper overviews the operation of CGEN by providing an example of the types of knowledge acquired and the mechanisms employed. A novel knowledge-intensive generalization scheme is discussed. Generalization is a pragmatic necessity for knowledge acquisition in this domain. A series of

[1] **Computer structures are legal configurations of hardware, such as the interconnection of a micro-processor to a memory array.**

experiments to test CGEN's capabilities are described. However, before the details of CGEN are given, a description of the architecture and knowledge-base of the performance program, M1, is provided. An understanding of M1 is necessary since its behavior is exploited by CGEN to guide the knowledge acquisition process.

2. The Performance Program

The performance program designs small computer systems. These systems are typically composed of the following sub-systems:

- single micro-processor
- static random access memory (SRAM) and read only memory (ROM)
- input/output (IO) devices
- miscellaneous support circuitry *(e.g.* address decoding logic, memory refresh logic)

The complexity of the designs produced by M1 is roughly equivalent to that of an IBM PC-AT.

The input to M1 is a set of high level specifications. The specifications describe global constraints on the design (area, power, and cost) and functional specifications for each sub-system. In Figure 2-1 an example of global constraints and a functional specification for a serial IO (SIO) device are given.

```
Input board  area upper bound:
Input power dissipation upper bound:
Input cost upper bound:

Number of SIO devices [1]:
SIO chip name [?]:
Baud rate for SIO port [300]:
RS232 compatible port [y]:
Port size [25 pin]:
Port type [DCE]:
```

Figure 2-1: Example specifications for global constraints and the SIO sub-system.

The primitive functions of the M1 architecture consist of search along two axes: a search for function and a search for structure. During the search for function a part implementing the function of a more abstract part must be found. Once the part is found, the search for a computer structure to support it must be performed.

The search for function occurs along the functional hierarchy. The hierarchy shows how to transform functionally abstract parts into successively more detailed parts until a physically realizable part is found. The functional hierarchy is created by abstracting the function of parts found in the domain. The higher a device is in the hierarchy, the more abstract is its function; similarly, the lower the device is in the hierarchy the better defined is its function. Devices at the lowest point in the hierarchy are referred to as *physical parts,* all others are called *abstract parts.* The use of some form of functional hierarchy is common in design systems which resolve functional specifications. For example, the VEXED system [12] employs a similar tactic for integrated circuit design. A simplified hierarchy is shown in Figure 2-2.

The search for function occurs for a given part, the parent, and the parent's children. Consider, for example, the hierarchy in Figure 2-2 where a search for function might begin with the 68XX_PROCJ). A decision is made between which child, the 6809 or the 6800, to choose. Details of the selection process are described later.

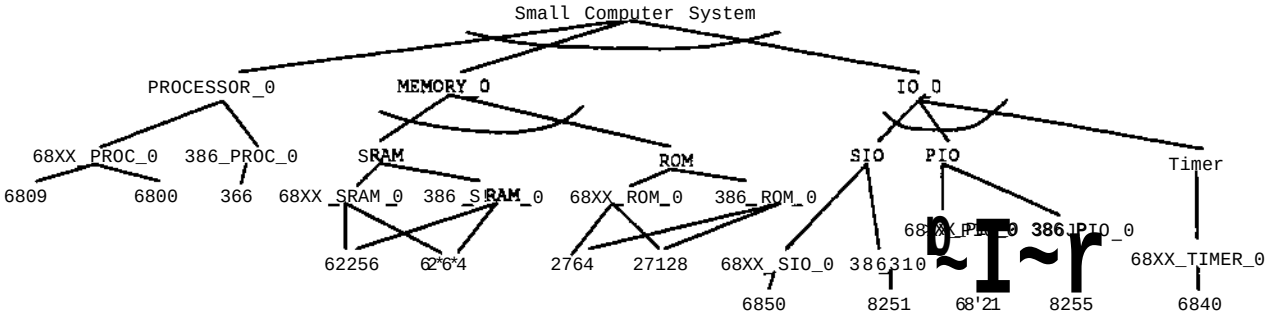

Figure 2-2: Simplified computer functional hierarchy.

Each part in the functional hierarchy has an associated part model. The part model contains, amongst-other things, definitions for characteristics and specifications. Characteristics are attribute-value pairs which describe properties of a part. Example classes of properties include: physical, electrical, mechanical, thermal, and physical dimensions. Specifications are also attribute-value pairs which provide values for properties of less abstract parts into which a given part will be mapped, i.e. values for parts lying below a given part in the functional hierarchy. Specifications, therefore, are excluded from models of physical parts.

Both the functional hierarchy and the part models reside in a relational database and are fixed before either M1 or CGEN is executed. The database is accessed by M1 and CGEN to retrieve part models.

Once M1 has successfully completed the search for function for a given set of parts, the next phase in the design process is to search for a structure, or mapping, which allows the newly instantiated part to be configured into the evolving design. Templates provide the mapping. An example template is given in Figure 2-3, showing the mapping from 68XX_SIO_0 abstract part into a 6850 physical part. Figure 2-4 shows the *68XX_SIO_0* abstract part - notice the part has a specific set of signals (represented as pins). These signals, in the context of a full design, are connected to other components. The objective of the mapping is to replace the *68XX__SIO_0* with a *6850* and associated parts. Figure 2-3 shows how the mapping is accomplished.-The box surrounding the *6850* is the *68XX_SIO_0.* The signals connecting to this boundary are more detailed implementations of the 68XX_SIO_0 signals. The other components in the template, the *CLOCK_GENERATOR_0, RS232JP0RTJ),* and *\NTR_BUSJRESOLVER_0* support the operation of the *6850.*

Often, there are multiple techniques for a part to implement its parent's function. Many templates exist in such situations, where each template is associated with an. implementation technique. For example, the template in Figure 2-3 maps the 68XX__SIO_0 into the 6850 with an RS-232 external port (as indicated by the part RS232_PORT_0 in the template). An almost identical template could be used for mapping the 68XX_SIO_0 into the 6850 with an RS-422 external port; the only change necessary is

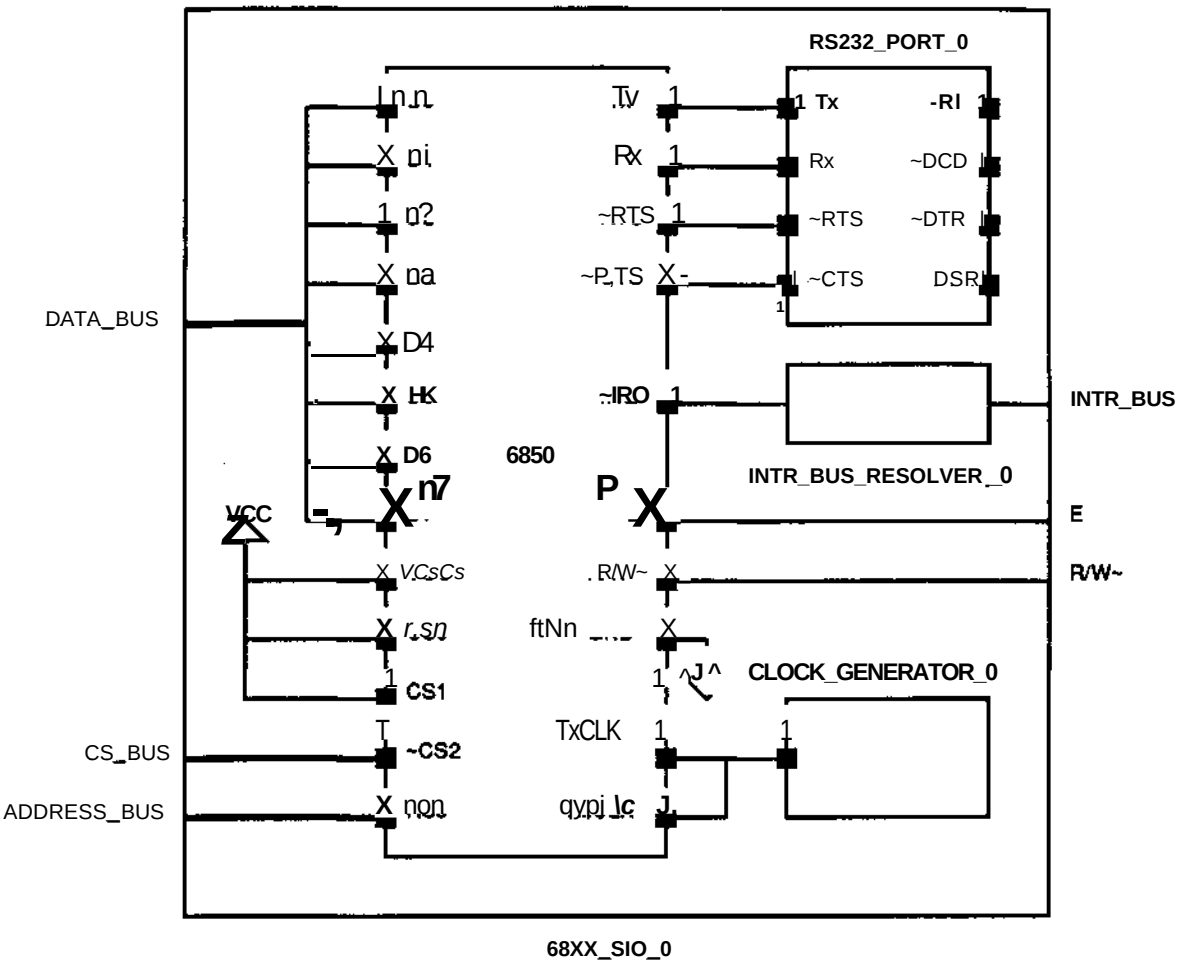

Figure 2-3: Template for mapping a 68XX_SIO_0 into a 6850.

replacing the original RS232_PORT_0 part with an RS422_PORT_0 part. Since M1 does not know a template's constituent parts.Jemplates must be marked with pre-conditions to identify when it should be applied. Furthermore, the pre-conditions must be unique across all templates so that no ambiguity exists regarding when a template should be selected. The pre-conditions associated with each template are based upon the *design state.* The design state consists of all reports (a set of variables containing constraint information) and the parts which are involved in the search for function and their corresponding specifications and characteristics. The design state for a parent part P_r and child part Pj is:

$$\textit{Design State} = R_t + (C_r + S_r) + \sum_{i}^{\textit{All Template Parts}} (C_t + S_i)$$

Where C_r and S_r are the characteristics and specification of P^ and, C; and S_y are the characteristics and

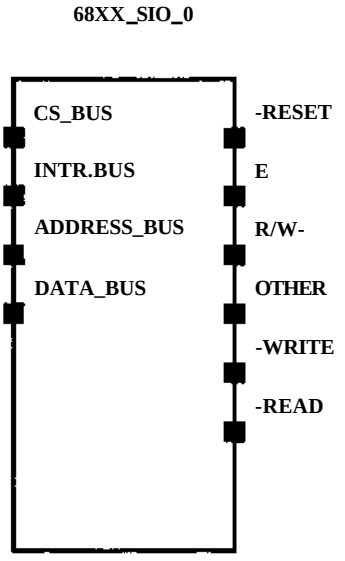

Figure 2-4: The 68XX_SIO_0 abstract part.

specification of P_i. The design state is unique. By associating a template with a design state, unique pre-conditions can be ensured.

The final component of the M1 architecture is the *design cycle.* The design cycle is the technique for using the functional hierarchy and templates to perform design. The design cycle is applied to each abstract part. By iteratively applying the design cycle to all abstract parts, they are refined into physical parts. There are the following five steps [3, 4] in the design cycle:

Specification (d_{spec}):
values for the specifications of a part are generated.

Selection (d_{selec}^: the parts lying below the part to be mapped on the functional hierarchy become *candidates.* The characteristic portion of the part model for each candidate is compared to the specifications generated in step d_{spec}. The part which matches most closely is chosen.

Part Expansion (d_{casc}): _
cascadable parts, memory for example, are built out to the proper width.

Structure Design ($d_{template}$):
a template is chosen and asserted.

Calculation (d^: various calculations associated with the template are performed. The calculations consist of updating the values of constraints and generating design information. All the information calculated is stored in a report.

Steps d_{spec} and d_{select} implement the search for function. The search for structure is realized by design cycle steps: d ^ , $d_{template}$, and d_{calc}.

A design cycle begins when a part's specifications are filled, with the specifications acting as a set of

pre-conditions. Therefore, a design cycle for an abstract part can begin when its specifications are complete. Spawning of design cycles occurs through a linking of d^ of one design cycle to d_{spec} of other design cycles. During d_{calc} new design information is generated in the form of reports which are then used to complete specifications for some set of abstract parts (which could have been introduced by any template). These parts then begin their own design cycles, causing other parts[1] specifications to be completed. This action continues until only physical parts exist in the design state.

3. The M1 Knowledge-base

M1's knowledge-base is divided into a set of partitions. Each partition supports either an architectural function or a step in the design cycle. The knowledge partitions are:

SpecificationKnowledge(k_{spec}):
this partition supports design step d_{spec} by providing values (either by calculation or query) to the specifications of parts in the current design cycle.

SelectionKnowledge(k_{select}):
the design step d_{select} is supported by this partition, providing constraints for comparing candidate part characteristics to specifications.

Part Expansion Knowledge (k^^:
this partition supports design step d_{casc}, describing how to build cascadable structures . There are two basic cascadable structures: matrices and trees.

StructuralKnowledge(k_{templ}):
design step $d_{template}$ is realized with this partition through the application of templates.

Evaluation Knowledge (k^:
support for design step d_{calc} is provided by k_{calc}. The means for evaluating design constraints and creating new design information is specified here.

Architecture and Support Knowledge (k^^:
a set of procedures and rules used for implementing the conflict resolution strategy and controlling the execution of design cycles resides in this partition. The remainder of this partition is devoted to a wide variety of functions to support input/output, mathematical functions, etc.

Knowledge contained in partitions k^,^, k_{spec}, k^p,^, and k_{calc} are collectively referred to as *design knowledge.*

A summary of information about the partitions is provided in Table 3-1. Notice that the amount of knowledge in partitions k^^ and k^^ are fixed over the lifetime of the system. Partition k_{casc} experiences no growth because the structures are general. MVs problem-solving technique does not change over time, therefoce-partition k^^ will not grow.

The total knowledge base, K_T, is the summation of the knowledge partitions:

$$K_T = k_{spec} + k_{select} + k_{cosc} + k_{template} + k_{calc} + k_{arch} \quad (1)$$

The design cycle imposes dependencies on the partitions. Whenever k_{templ} is modified, k_{spec} and k^ must be updated to ensure design cycles occur for each part introduced by the template. A *super-partition,* $\hat{K}_j$, is defined as:

$$\hat{K}_j = k_{spec} + k_{template} + k_{calc} \quad \hat{}$$

Equation 1 can be re-cast as:

Growth Rates and Knowledge Types			
Partition	Procedural	Declarative	Growth Rate
**spec	-	Yes	Rapid
^select	-	Yes	Rapid
^casc	Yes	-	No Growth
template	--	Yes	Rapid
k_{calc}	--	Yes	Rapid
k_{arch}	Yes	Yes	No Growth

Table 3-1: Summary of growth and representation knowledge.

$$K_T^S K_{elec} \quad {}_{asc} + k_{arch} \tag{3}$$

CGEN is designed to acquire K_T^S and $kg_{e,ect}$. The remainder of this paper concentrates on the acquisition of K_J^S, with particular attention given to $k_{template}$.

4. Example of K? Knowledge Acquisition

When M1 can not find a template for mapping a set of parts, it informs the user that a chunk of design knowledge is necessary. For example, if M1 requires a template for mapping the 68XX_SIOJ) into the 6850, but one does not exist in its knowledge-base, the following message is generated:

```
Template not found for interfacing the 6850 to 68XX_SIO_0
```

Along with this message, M1 produces a file containing design state information which is used to form a set of pre-conditions for the template to be acquired. An example design state is given in Figure 4-1 corresponding to the template in Figure 2-3. Pre-conditions PC1 through PC7 are specifications for the 68XX_SIO__0, and pre-conditions PC8 through PC12 are global constraints.

As an aside, if the template in Figure 2-3 were modified to create a new template compatible with an RS-422 external interface, pre-condition PC7 in Figure 4-1 would be:

```
(SPECIFICATION DRIVER_TYPE = RS-422) ;
```

This change would allow M1 to determine precisely when each 68XX_SIO_0 to 6850 template should be used.

After M1 produces a design state file, CGEN can be invoked once the domain expert has produced a schematic[2], as shown in Figure 2-3, and some other files (described later in this section). This set of knowledge constitutes a training session. The results of the training session are a set of rules, one of which is the $k_{tBmp|atB}$ rule in Figure 4-2. The left hand side (LHS) of the rule provides a generalized version of the pre-conditions from Figure 4-1. The rule's right hand side (RHS) contains a version of wire list M1 can read.

[2]CGEN actually uses a wire list generated by the drawing editor from the schematic. A wire list describes parts and their interconnections.

```
PCI:  (SPECIFICATION SIO_LINES = 1);
PC2:  (SPECIFICATION PROCESSOR = 6809);
PC3:  (SPECIFICATION TX_BAUD = 1200);
PC4:  (SPECIFICATION RX_BAUD = 1200);
PC5:  (SPECIFICATION TXJTYPE = ASYNC);
PC6:  (SPECIFICATION RXJTYPE • ASYNC);
PC7:  (SPECIFICATION DRIVERJTYPE = RS-232);
PC8:  (REPORT CLOCK_SPEED • 1000000);
PC9:  (REPORT IO_ACCESS_TIME • 500);
PC10:  (REPORT REMAINING_POWER = 10000);
PC11:  (REPORT REMAINING_BOARD_AREA = 500);
PC12:  (REPORT REMAININGJBOARD_COST = 80000);
```

Figure 4-1: Example pre-conditions for template 68XX_SIO__0 to 6850 derived from the design state.

```
RULE  instantiate_SIO_template
{
(GOAL name = assert_template)
(SPECIFICATION SIO_LINES = 1);
(SPECIFICATION PROCESSOR =
        (6800 OR 6809));
&A (CHAR MAXJ3AUD_RATE) ;
(SPECIFICATION TX_BAUD < &A);
(SPECIFICATION RX_BAUD < &A);
(SPECIFICATION DRIVER_TYPE = RS-232);
(SPECIFICATION TX_TYPE = ASYNC) ;
(SPECIFICATION RX__TYPE = ASYNC) ;
(REPORT IO_ACCESS_TIME > 300);
(REPORT REMAINING_POWER > 200) ;
(REPORT REMAINING_BOARD_AREA > 500);
(REPORT REMAINING_BOARD_COST > 120);
—>
make  (tag name = g8483943)
; The wire list is abbreviated for the sake of conciseness.  Normally
; all parts and connections in the schematic are listed here.
; getjpart retrieves a part model from the DB.
get_jpart (RS232_DRIVER)
getjpart (BAUD_RATE_GENERATOR)
...
; connect^net creates a connection between the parts and pins in the
; parameter list.^
connect_net(6850~, DO, SIOJ), DATA_BUS)
connect_net(6850, TxCLK, BAUD^RATE^GENERATOR, OUT)
...
}
```

Figure 4-2: Final rule for template shown in Figure 2-3.

As specified by KJ[S], values for the specification of all abstract parts introduced by the template must be acquired. In addition, any calculations needed to generate global design information must be culled from the domain expert. To create specifications for a part, the domain expert refers to the part's model and

provides a means for generating its value. Specifications, calculations, and generalization knowledge (described in Section 5) are input via *methods.* An example method for the specification PORTJTYPE for part RS232_PORT_0 is shown in Figure 4-3. A example method for specification CLOCK_FREQUENCY for part CLOCK_GENERATORJ) is given in Figure 4-4. CGEN ensures that all specifications for all parts have an associated method. The domain expert is requested to provide any missing methods.

```
calculation_type: specification
calculation: PORT_TYPE = MALE;
```

Figure 4-3: Example of domain expert input, a method, for specification PORT_TYPE.

```
calculation^ype: specification
calculation: CLOCK_FREQUENCY = TXJBAUD * 16;
```

Figure 4-4: Example of domain input, a method, for specification CLOCK_FREQUENCY. TX_RATE is specification of the abstract part 68XXJ5IOJ).

In addition to methods for specifications, the domain expert also supplies a method for calculating S!O_ADDRESS_SPACE[3]. The calculation is given in Figure 4-5.

```
calculation_type: calculation
calculation: SIO_ADDRESS_SPACE = 4 ;
```

Figure 4-5: A method for calculating SIO__ADDRESS_SPACE.

5. Generalization

During the creation of a $k_{temp(ate}$ rule structural knowledge is joined with a set of pre-conditions. The initial formulation of pre-conditions coincides exactly with the design state, yielding the tightest set of invocation conditions for describing when the new $k_{temp|ate}$ knowledge can be applied. This definition is necessary to comply with the M1 architecture's requirement each $k_{temp|ate}$ rule have a unique set of pre-conditions.

Often the pre-conditions derived from the design state are overly constraining, preventing the template from being applied irvother design states where it is perfectly valid. There are three causes of overly constraining pre-conditions:

Equality. equality is used as the test on a design state variable.

Constants: constants derived from the design state are used for comparison.

Irrelevant pre-conditions:

[3] **SIO_ADDRESS_SPACE is used by another computer sub-system for address decoding.**

pre-conditions which are superfluous to the proper definition of the design state are included.

The key to the generalization process is to identify those pre-conditions which exhibit one of these characteristics and implement an appropriate fix. Fixes consist of:

One- or two-sided constraints:
equality test is removed and a single or double-sided interval is used to test a variable.

Variablesubstitution:
variables are used whenever possible.

Delete pre-condition:
superfluous pre-conditions are removed.

Care must be exercised in the application of fixes to prevent over-generalization, resulting in the intersection of several templates' pre-conditions.

The knowledge needed for generalization, kgeneraiizo' explains how design variables interact and the proper range of such variables. The $k_{generalize}$ partition is actually composed of two separate bodies of knowledge:

$$k_{generalize} = k_{generalize\_rule} + k_{generalize\_method}$$

Where each body of knowledge is defined as:

$k_{generaiize\_metho}$ a description given by the domain expert indicating, for each training case, the variables to relax and their boundary conditions.

$k_{generaiize\_ruie}$ a set of generalization rules, residing in CGEN's knowledge-base, which capture constraint intervals that are applicable to all training cases.

The generalize knowledge resides with CGEN; it is not part of M1's knowledge-base.

Methods are also used to describe $k_{generalize\_method}$ knowledge to CGEN. The methods for relaxing some of the pre-conditions in Figure 4-1 are given in Figure 5-1. The TX_BAUD and RX_BAUD pre-conditions in Figure 5-1 are set less than or equal to the maximum baud rate (MAX_BAUDERATE)[4] the 6850 can sustain. The IO_ACCESS_TIME is bounded by the access time of the slowest instance of a 6850; a value supplied by the domain expert. Finally, the pre-condition CLOCK_SPEED is removed by the domain expert, because it is subsumed by IO_ACCESS_JIME.

Figure 5-2 contains an example set of $k_{generaljze\_mle}$ rules. These rules represent knowledge about general system level constraints. For example, the rule *relax__REMAINING_BOARD_AREA* sums the area consumed by each part asserted by the template *(areajconsumed* is a characteristic in every part model). The sum is used as Jhe lower for the constraint *REMAINING_BOARD_AREA.*

The generalize ruie-knowledge partition is generated by the *CGEN* knowledge engineer. The generalize method knowledge for each training case is examined. Any set of commonly occurring methods are encoded into CGEN rules, thus eliminating the need for these methods to be used by the domain expert in the future. Ideally, after enough training cases are seen, a *complete* set of kgeneraiize_ruie rules will exist in CGEN, drastically reducing the number of domain-expert generated methods.

[4]MAX_BAUD_RATE is a characteristic of the 6850.

```
method: TX Baud rate max. value
calculation:
 TX_BAUD <= MAX_BAUD_RATE*;

method: RX Baud rate max. value
calculation:
 RX_BAUD <= MAX_BAUD_RATE*;

method: relax IO_ACCESS_TIME
calculation:
  IO_ACCESS_TIME > 300;

method: Delete CLOCK_SPEED
calculation:
  CLOCK_SPEED = DELETE;

*- MAX_BAUD_RATE is a characteristic of the 6850.
```

Figure 5-1: methods for relaxing/deleting pre-conditions.

6. The CGEN Architecture

There are two components to CGEN's architecture, the acquisition cycle and rule generation. Two types of acquisition cycles exist, based on the type of knowledge being captured. The cycles are: acquisition of K_y^S and acquisition of k_{select}. The acquisition of k_{select} requires a subset of the functions needed for K_j^S and will not be discussed (for more information see Birmingham [5]). Rule generation involves the creation of code from CGEN's internal representation.

The function of the acquisition cycle is to guide the overall problem solving activity of CGEN during a training session. The acquisition cycles set up expectations about the types of knowledge being acquired during a training session. For example, when a $k_{temp|atB}$ rule is being acquired, CGEN ensures that ks_{pec} knowledge is supplied for all parts asserted in the template and that k_{calc} equations are given. The acquisition of these partitions is consistent with the definition of K_j^S.

In conjunction, the acquisition cycle also determines the types of analysis performed on the incoming knowledge. Two types of analysis are performed:

Error/completeness checking: the correctness and completeness of knowledge acquired during a session is cheCked.The types of checks used depend on the type of knowledge acquired.

Generalization: as mentioned previously, the pre-conditions associated with a template are generalized.

CGEN, through the acquisition cycle, encodes knowledge of the problem-solving method employed by M1. Implicit in the definition of K_j^S is the relationship of the design cycle (M1 's problem-solving paradigm) to the knowledge-base. Since the acquisition cycle is based on K_j^S, it drives CGEN to cull from the domain expert the knowledge necessary to maintain consistency in M1's knowledge-base. This ensures consistent action of the design cycle. Without exploiting knowledge of M1's behavior, CGEN would be incapable of verifying the completeness of incoming knowledge.

```
rule 68XXJbus_compatibility
{
...
(SPECIFICATION PROCESSOR = 6809)
-->
(PROCESSOR = 6809 OR 6800;)
}

rule relax_REMAINING_POWER
{
...
(REPORT REMAINING_POWER)
-->
(REMAINING_POWER > total power consumed by template parts;)
}

rule relax_!REMAINING_BOARD_COST
{
...
(REPORT REMAINING_BOARD_COST) ;
-->
(REMAINING_BOARD_COST > total cost consumed by template parts;)
}

{
rule relax_REMAINING_BOARD_AREA
...
(REPORT REMAININGJ3OARD_AREA) ;
-->
(REMAINING_BOARD_AREA > total area consumed by template parts;)
}
```

Figure 5-2: CGEN rules, kgeneraiize__ruie[for] taxing pre-conditions.

The acquisition cycle's definition is also influenced by the nature of the domain knowledge, i.e. the need for generalizing the pre-conditions for $k_{temp|ate}$ rules. The behavior of M1 would not be impaired if the generalization process were not applied; however, a burden would be placed on the domain expert to generate a larger number of training cases. For pragmatic considerations, the generalization process is necessary.

During the rule generation process, CGEN's intermediate representation of the acquired knowledge is translated into a set of rules executable by M1. M1 was designed so that all rules in a partition have the same canonical format. The canonical form breaks the LHS and RHS of a rule into two parts: context-specific and instance-specific. All rules in a partition share the same context-specific LHS and RHS. This ensures all rules in a partition are invoked uniformly and have similar actions. The instance-specific portion is generated for each rule based on information gleaned from the training case. For example, for template [rule] instances the template pre-conditions form the LHS instance-specific portion of the rule, and the RHS instance-specific portion is created from the wire list. After a training session is complete, the CGEN generated rules are compiled into M1's knowledge-base.

The simplified acquisition cycle for k_p^s is given below:

1. Read methods, wire list, design state, and create intermediate format
2. Identify pre-conditions for $k_{tempJaterul0}$
3. Apply $k_{generalize}$:
    a. Identify overly-constraining pre-conditions
    b. Relax/delete those pre-conditions
4. Instantiate k ^ p , ^ ^ :
    a. Generate LHS from pre-conditions
    b. Generate RHS from wire list
5. Generate k_{cajc} rules
6. Generate $\vec{K}$^^ rules

7. Experimentation

CGEN was subjected to a set of experiments to accomplish the following:

1. test CGEN's ability to capture design knowledge
2. test CGEN's ability to generate working code for M1
3. gain insight into the complexity of the small computer design domain

The experiments were designed to simulate as closely as possible the conditions under which CGEN was intended to perform. A version of M1 with a minimal knowledge-base was used as the performance system. At the beginning of the experimental process, the M1 knowledge-base contained only:

$$K_1 = k_{casc} + k_{arch}$$

Therefore, M1 started with no design knowledge. All knowledge was acquired through training sessions with CGEN.

M1 was taught to design with the following micro-processor families:

- Motorola 6809
- Motorola 680008
- Motorola 68010
- Intel 80386

A design for each processor was obtained from either published reports or industrial affiliates. All training cases for the physical components were derived from this set of designs. While all the designs were from actual applications, none stretched either the function or performance of any of the processor families.

A group of four designers, *i.e.* domain experts, were used to teach M1. None the domain experts, except one of the authors[5], were familiar with AI programming techniques, the OPS\83 language, or the implementation details of CGEN and M1. The designers received training in M1 and CGEN design philosophy and tool usage roughly equivalent to a two-day course. The domain experts did not write any

[5] Birmingham entered the 6809 family knowledge.

OPS83 code or modify the rules created by CGEN[6].

After the designers were familiar with their micro-processor families, the knowledge acquisition process began. The following steps formed the process:

1. part models were developed for all abstract and physical parts
2. data for all part models were entered into the central database
3. templates were drawn for all physical and abstract parts used
4. all methods were prepared
5. knowledge acquisition sessions were conducted using CGEN in conjunction with M1

7.1. Part Data

The size of the database in terms of part models is given in Table 7-1. Table 7-2 includes the average number of characteristics, specifications (applicable to abstract parts only), and pins per part model.

Part Model Totals	
Abstract Parts	167
Physical Parts	215
Total Parts	382

Table 7-1: Total part models.

Part Model Data		
	Average	Std. Dev.
Specifications/model (avg)	4	3
Characteristics/model (avg)	4	5
Pins/model (avg)	15	18

Table 7-2: Summary of part model data.

7.2. Rule Data

The number of training cases and rules for each design are given in Table 7-3. Table 7-4 shows the breakdown of rules by partition.

When the knowledge acqyisition process began, knowledge partitions k^^ and k_{arch} were fixed. The k_{casc} partition allowed construction of memory arrays and replication of IO devices; no other structures were necessary. IO replication occurs when the M1 user requests multiple copies of a single function; for example, two SIO devices. In this case, M1 creates as many instances of the function as the user requests. There was no growth in either k_{casc} or k_{arch} during the entire experimentation period, as expected. Table 7-5 shows the average growth rates for all partitions in units of rules per training case.

[6] **A call to a routine to generate connections for a certain type of chip was added by the author to one template in each micro-processor family.**

Training Cases and Rules Per Design		
Design	Cases	**Rules**
M6809	85	**343**
M68008	38	**173**
M68010	**19**	**147**
I80386	**79**	**256**
TOTAL	**221**	**919**

Table 7-3: Number of training cases and rules per design.

Rules per Partition		
Partition	Rules	**% of Total**
k_{spec}	552	**60**
^select	**8**	**.1**
template	**226**	**25**
^alc	**133**	**15**
TOTAL	**919**	**100**

Table 7-4: Number of rules per partition.

Average Growth Rate per Partition		
Partition	**Growth Rate**	Std. Dev.
^spec	**2.5**	**4.8**
^{k}select	**.04**	**0.3**
^{k}casc	**0**	**0**
^template	**1.0**	**0.2**
^{k}calc	**0.6**	**1.8**
^{k}arch	**0**	**0**
Number of cases	**221**	**...**

Table 7-5: Average growth rate per partition in rules/training case.

The complexity of the LHS of template rules is given in Table 7-6. The first entry in the table is the average size of the design state (or number of template pre-conditions generated by M1 at the beginning of a training session), followed by the average number of pre-conditions which survive to become pre-conditions for a k^p,^ rule . The number of pre-conditions removed by the domain expert is given next. This is followed by both the number and percentage of pre-conditions generalized of those not deleted. Note the high percentage, *77%,* of pre-conditions which are relaxed, verfying that a generalization technique is necessary.

Table *7-7* gives the total number of $k_{generaljze\_mle}$ rules in CGEN's knowledge-base. The average

number of these rules applied during a training session is also given. Finally, the percentage of pre-conditions automatically relaxed and deleted with respect to the data given in Table 7-6 is shown.

The percentage of relaxations and generalizations made by $k_{generaljze\ mle}$ indicates the coverage of these rules with respect to all the generalization knowledge required for a Training session. Note that the number of training cases utilizing the generalization rules (given by the number of cases entry in Table 7-7) is less than the total number of training cases in Table 7-3. The generalization rules were added to CGEN's knowledge base late in the experimental process.

LHS Data (avg)	
Design State Size (pre-conditions)	35
Pre-conditions/rule	13
Pre-conditions removed/rule	22
Pre-conditions relaxed/rule	10
% Pre-conditions relaxed/rule	77%
Number of cases	221

Table 7-6: LHS data averages.

CGEN Rule Data (avg)	
Number CGEN Rules	44
Avg Number Rules Applied	28
% Changes Made	88%
Number of Cases	117

Table 7-7: CGEN generalization rule data (averages).

The RHS data for k^p,^ rules is summarized in Table 7-8. The table shows the number of wires and parts asserted per rule. The large number of wires relative to number of parts is explained by three factors. First, many templates are comprised of wires only, implementing a *switchbox* function. Second, many templates (especially for the 80386) require a large number of wires to connect the parts in the template. Consider showing the connections for an 80386 which has 32 address and 32 data lines. Finally, the templates tend to have local information which does not require a large number of support parts. Locality of knowledge indicates the incremental nature of template knowledge. Templates divide the structural design problem into relatively small chunks.

RHS DATA (avg)	
Number Wires/rule	11
Number Parts/rule	1.1
Number of cases	221

Table 7-8: RHS data averages.

One of the assumptions underlying the M1 architecture is the orthogonality of pre-conditions for templates. To test this assumption, M1 was instrumented to write to a file every time more than one template occurred in the conflict set. Every run of M1, including all training runs, contributed to the file. Examination of the file verified that the conflict set never contained more than one template rule.

7.3. M1 Designs

From the training given, M1 is able to create an interesting variety of designs. Since knowledge is acquired concerning parts individually, unique combinations of parts can be configured into a design. The ultimate limitation on the size of a design comes from: the size of the micro-processor's memory and IO spaces, and physical constraints (power, area, cost).

Once training was completed, experiments were run using M1 to generate an example set of designs for all the micro-processors families. The specifications used for each design are shown in Table 7-9. These designs represent only a few points in a large space consisting of a wide variety of designs M1 is capable of producing.

Design Specifications for Major Functions				
Design	Amount SRAM	Amount ROM	Amount PIO	Amount TIMER
1	4 KBytes	1 KByte	1	0
2	4 KBytes	1 KByte	0	1
3	58 KBytes	4 KBytes	1	1
4	58 KBytes	4 KBytes	2	2

Table 7-9: Input specification summary for M1 design tests.

7.4. Design Construction

The design M1 produced for the 68008 using specification set 3 from Table 7-9 was built and is running. Analysis of the design shows:

- The design works at expected clock rates.
- The number of parts used is no greater than those generated by hand.

The favorable comparison with hand design in terms of part count is to be expected since the design was built almost entirely from very large scale integrated (VLSI) circuit components.

The construction of a working design validates M1's knowledge-base, and thereby validates CGEN's knowledge acquisition capability.

8. Discussion

CGEN's capabilities are derived from its understanding of the performance program's problem-solving method. Other successful knowledge acquisition tools have similarly exploited such knowledge. MOLE [6] provides an example of a tool that understands the types of knowledge in the domain in order to drive the development of heuristic classification system. The KNACK system [7] is used to create performance programs to evaluate electro-mechanical designs. KNACK explicitly builds a model of its domain. This model is used to gather additional knowledge from a domain expert.

SALT [9,10] is a knowledge acquisition tool for developing constructive systems. Embodied within SALT is a model of its performance program's problem-solving technique. During the knowledge acquisition process, SALT constructs a representation of the knowledge it has acquired. This representation, and the model of the problem-solving technique, are used to drive interactions with the domain expert. CGEN is similar to SALT in several ways. Both systems acquire knowledge for

constructive (synthesis) system and both systems have embedded models of their performance program's problem-solving paradigms. However, there are some differences, the most obvious being that each acquires and produces knowledge of very different types and in different domains. In SALT, the domain expert takes the initiative during knowledge acquisition, although he is guided by the tool; CGEN's knowledge acquisition sessions are initiated by the performance program which knows when its missing knowledge. Finally, CGEN utilizes a generalization technique to fully exploit knowledge gathered during a session, SALT does not require such a mechanism.

CGEN's generalization scheme is knowledge-intensive, requiring detailed knowledge for specific constraints. A more general approach is to encode a domain theory in the knowledge acquisition tool. The domain theory allows the tool to prove the correctness of knowledge being gathered. Based on the developed proof, the tool can then generalize the acquired knowledge. This process is called explanation-based generalization (EBG) [8]. LEAP [8] utilized this approach to acquire VLSI design knowledge. The drawback to EBG is the requirement of a formalized domain theory. In many synthesis applications, such as computer system design, a comprehensive and complete theory is not available.

Another traditional approach to generalization is induction, where a system is shown many examples of a concept. Examples of such systems are given by Arciszewski [1] and Bareiss [2]. Automatic induction techniques are not applicable for CGEN because each training case represents a unique concept. Thus, the requisite number of training cases are not available.

9. Summary

CGEN has demonstrated an effective mechanism for capturing knowledge from domain experts. The ability of domain experts unfamiliar with either OPS83 or the implementation of the M1 to develop working knowledge-bases shows the tool has a good representation of the domain. CGEN has also been shown to be effective for capturing the knowledge necessary to actually produce designs, thus proving its ability to encode such knowledge appropriately for the synthesis system.

The need for a generalization mechanism in the system has been illustrated. Without such a mechanism, it is doubtful that the amount of knowledge necessary to design could be captured in a reasonable time. The generalization scheme has been effective in domains which are ill-structured and have a low number of training cases.

There are several limitations on CGEN, which are summarized here. CGEN is tightly coupled to the M1 architecture. Only declarative knowledge can be captured; procedural knowledge must be hand-coded. CGEN's ability to reject incorrect knowledge is limited since it does not have a complete model of the domain knowledge~(sacTi as that used in EBG systems). Therefore, CGEN lacks a reference for verifying the acquired knowledge is correct.

CGEN is presently in use acquiring knowledge of new micro-processor families. The CGEN-M1 system is being deployed to several IC and computer manufacturers for p-testing in actual engineering design environments.

10. Acknowledgements

Anurag Gupta developed M1 and contributed greatly to the development of CGEN. Tom Mitchell provided helpful insight during the the course of the work described here. The authors thank Paul Birkel, Larry Eshelman, and Georg Klinker also provided many useful comments and suggestions regarding this paper and CGEN. The authors thank them all for their effort.

References

[I] T. Arciszewski, M. Mustafa, W. Ziarko.
A methodology of design knowledge acquisition for use in learning expert systems.
International Journal of Man-Machine Studies 26, January, 1987.

[2] E.R. Bareiss, B.W. Porter, C.C. Wier.
Protos: An Exemplar-Based Learning Apprentice.
In *Proceedings from the 2nd AAAI Knowledge Acquisition for Knowledge-based Systems Workshop.* AAAI, 1987.

[3] William P. Birmingham, Daniel P. Siewiorek.
Single Board Computer Synthesis.
Technical Report EDRC-18-02-87, Engineering Design Center, Carnegie Mellon University, 1987.

[4] William P. Birmingham, Audrey Brennan, Anurag P. Gupta, Daniel P. Siewiorek.
MICON: A Single Board Computer Synthesis Tool.
IEEE Circuits and Devices Magazine, January, 1988.

[5] William P. Birmingham.
Automated Knowledge Acquisition for a Hierarchical Synthesis System.
PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, 1988.

[6] L Eshelman, D. Ehret, J. McDermott, M. Tan.
MOLE: a tenacious knowledge-acquisition tool.
International Journal of Man-Machine Studies 26, January, 1987.

[7] G. Klinker, J. Bentolila, S. Genetet, M. Grimes, J. McDermott.
KNACK- Report-driven knowledge acquisition.
International Journal of Man-Machine Studies 26, January, 1987.

[8] T.M. Mitchell, S. Mahadevan, L. Steinberg.
Leap: A Learning Apprentice for VLSI Design.
In *Proceedings of IJCAI-85.* Morgan Kaufmann Publishers, 1985.

[9] S. Marcus, J. McDermott, T. Wang.
Knowledge Acquisition for Constructive Systems.
In *Proceedings of IJCAI-85.* Morgan Kaufmann Publishers, 1985.

[10] S. Marcus, J. McDermott.
SALT: A Knowledge Acquisition Tool for Propose-and-Revise Systems.
Technical Report CMU-CS-86-170, Carnegie Mellon University Deparment of Computer Science, 1986.

[II] J. McDermott.
Domain Knowledge and the Design Process.
In *18th Design Automation Conference.* IEEE Computer Society, 1981.

[12] T.M. Mitchell, L.I. Steinberg, J.S. Shulman.
A Knowledge-based Approach to Design.
IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-7(5), September, 1985.

[13] Production Systems Technology Incorporated.
OPS/83 User's Manual and Report Version 2.2
1986.