Capturing Designer Expertise:
The CGEN System

by

William P. Birmingham, Daniel P. Siewiorek

EDRC18-08-89

Capturing Designer Expertise

The CGEN System[1]

William P. Birmingham[2]
Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, Michigan 48109
313-936-1590
wpb@crim.eecs.umich.edu

Daniel P. Siewiorek
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

November 4, 1988

[1]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.

[2]Principal author

Abstract

Knowledge-based systems are becoming pervasive in the computer-aided design area. For these systems to achieve satisfactory levels of performance large amounts of knowledge are necessary. However, the acquisition of knowledge is a difficult and tedious task. Automated knowledge-acquisition tools (AKAT) provide capabilities for quickly building and maintaining knowledge-bases. This paper describes the CGEN AKAT, which allows hardware designers, unfamiliar with artificial intelligence programming techniques, to deposit their expertise into a synthesis tool's knowledge-base. A set of experiments which tested CGEN's capabilities are presented. The experiments show that with CGEN hardware designers can produce high quality knowledge-bases.

1 Introduction

Knowledge-based systems (KBS) have gained wide acceptance in computer-aided design (CAD) systems for tasks ranging from diagnosis to hardware synthesis. It appears that the popularity of KBS will continue to increase as new applications are found. The appeal of these systems is the promise of reasonable solutions to problems which have defied automation in the past usually due to inadequate mathematically-based formulations. Knowledge-based techniques provide a means for creating entirely new symbolic problem formulations for a particular task-domain. Problem-solving techniques specific to a task-domain can then be employed.

KBS are composed of two major elements: the problem-solver and the knowledge-base. The problem-solver contains the heuristic methods used to solve a particular class of problems, for example the design cycle in the MICON system [1,8]. The knowledge-base contains the specific knowledge necessary for the problem-solver to find a solution. Much research has been devoted to the development of problem-solving techniques (for examples see Gajski[7] and Mitchell[15]), with little attention paid to the structure and engineering of the knowledge itself (an exception is Mitchell[13]).

The ultimate success of KBS, however, depends not only on a good problem-solver, but also on the encoding of large amounts of domain knowledge[12]. This is because the problem-solving techniques employed in KBS are not inherently powerful. The problem-solver draws much of its ability from its store of knowledge. If the knowledge-base contains errors, or is incomplete, the problem-solver will generate incorrect answers. Stated differently, a KBS will only be as good as its knowledge-base. Therefore, the development of a high quality knowledge-base sufficiently broad enough to cover all situations the problem-solver will encounter is essential.

CAD applications, in particular hardware synthesis, require substantial knowledge-bases for several reasons. Hardware design requires substantial expertise, usually accumulated over many years. This expertise tends to be centered around particular components and systems of components. For example, the synthesis of micro-processor based computers requires knowledge specific to the components of systems (e.g. the micro-processor and memory chips). In addition, knowledge of how to connect subsystems to form a *system architecture*[3] is required. Thus, as the number of components and system architectures grows, so does the knowledge-base. Further compounding the situation is the rapid evolution of electronic technology. A knowledge-base must be continually updated to enable the KBS to exploit opportunities afforded by new technology. It is important to recognize that these concerns are common to other CAD domains where KBS are employed, not just synthesis.

To cope with the sheer volume of knowledge which must be captured and represented within a knowledge-base, two actions must take place. First, some form of assistance must be provided to help construct a knowledge-base. Typically, automated knowledge-acquisition tools (AKAT) are used. These tools interface a domain expert to a KBS's knowledge-base without requiring the expert to write application code. The second action is to design the knowledge-base to support automated knowledge-acquisition. In particular, the knowledge-base must be clearly defined with respect to the *types* of knowledge used by the problem-solver and how the knowledge is *used* when developing a solution.

CGEN (Code GENerator)[5], a part of the MICON system[4], is an automated knowledge acquisition tool for the MICON Version 1 (M1) synthesis KBS[8,2]. M1 is a rule-based system written in OPS/83[9] which produces a complete small computer design from a set of abstract specifications. CGEN acquires knowledge about how to build and when to use various computer structures— the domain knowledge required by M1— from hardware designers unfamiliar with KBS programming techniques. As such, CGEN provides a user-friendly interface for

[3]For example, a high speed system will require a cache not needed for a lower performance system. The addition of a cache will impact the bus structure of the computer.

Figure 1: Tradition knowledge engineering process.

Figure 2: Automated knowledge engineering process.

designers, allowing them to express computer structures with schematic drawings and to describe other types of design knowledge (e.g. constraints) without having to write complex programs.

Following this introduction, a brief review of the knowledge acquisition process and related AKATs is provided. A discussion of Mi's problem-solving approach and knowledge-base are given, followed by an explanation of CGEN's design and implementation. Finally, a set of experiments testing CGEN's capabilities are presented.

2 Review of the Knowledge-Acquisition Process

The traditional knowledge-acquisition process is illustrated in Figure 1. The responsibility of the knowledge engineer (KE) is to query the expert, culling the knowledge necessary to enhance or debug the program under development, called the performance program. The gathered knowledge is re-formulated and encoded into a format (rules for example) which the KBS can utilize, becoming part of its knowledge-base.

The knowledge engineer can be replaced by a program as shown in Figure 2. Since it interacts directly with a domain expert, the knowledge acquisition tool must hide the implementation details of the KBS, as would a KE. The languages used to implement a KBS are difficult to work with; a challenge facing AKAT builders is defining the proper domain expert interface. This problem is analogical to developing high level language compilers to hide the details of assembly language! 14].

Knowledge acquisition tools are synergistic with their associated performance programs. In general, AKATs exploit knowledge about the performance program and, possibly, the domain to assist the acquisition of knowledge. Minimally, these tools must generate code which can be executed properly by the KBS, therefore, knowledge of the representations used in the performance program is necessary. At a deeper level, an AKAT utilizes knowledge about the problem-solving behavior of the performance program to check the correctness and completeness of inputs captured from the domain expert.

CGEN's capabilities are derived from its understanding of Mi's problem-solving method. Other successful knowledge acquisition tools have similarly exploited such knowledge. The KNACK system[10] 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. Another example system is SALT[11], an AKAT used to build configuration (synthesis) systems. SALT was used initially for designing elevators and is now being applied to other domains. As with CGEN, SALT embodies 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 further interactions with the domain expert

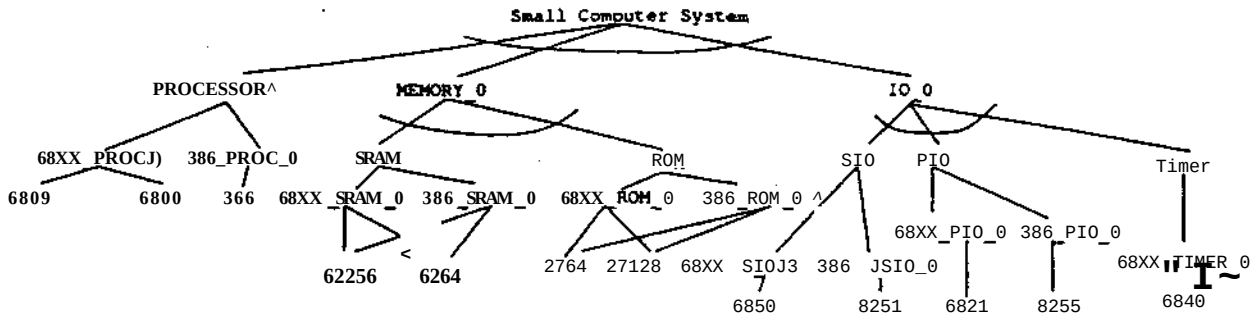

Figure 3: Simplified computer funciional hierarchy.

3 The Ml System

This section overviews Mi's knowledge-base and problem-solving approach. Understanding Ml is vital to understanding CGEN. A complete description of the problem-solving approach is provided by Gupta[8],

3.1 Mi's Problem-solving Approach

Systems designed by Ml are high performance, small computers roughly equivalent in complexity to an engineering workstation. The input to Ml is a set of high level specifications. The specifications describe global constraints on the design (area, power, and cost) and functionity of subsystems.

The primitive functions of the Ml 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 structure into which it can be interconnected 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. A simplified hierarchy is shown in Figure 3.

The search for function occurs for a given part, the parent, and the parent's children. Consider, for example, the hierarchy in Figure 3 where a search for function might begin with the 68XX_PROC_0, an abstract part. A decision is made between which child to choose, either the 6809 or the 6800, both being physical parts.

Each part in the functional hierarchy has an associated part model. The part model contains, among 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 and thermal. 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 an external relational database that is accessed by Ml and CGEN via a standard query language.

Once Ml 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 4, showing the mapping from the 68XX_SIO_0 into a 6850. The objective of the mapping is to replace the *68XXSIOJ),* shown in Figure 5, with a *6850* and its associated supporting parts. Figure 4 shows how the replacement is accomplished. The box surrounding the *6850* is the *68XXJSIO0.* The signals connecting to this boundary are more detailed implementations of the 68XX_SIO_0 signals. The other components in the template, the *CLOCK_GENERATOR0, RS232PORTJ),* and *INTR_BUSJtESOLVER_0* support the operation of the *6850.*

Often multiple techniques exist for a part to implement its parent's function. In such situations a template is associated with each implementation technique. For example, the template in Figure 4 uses an RS-232 external port. An almost identical template could be used for an RS-422 external port by replacing the original RS_232_PORT_0 part with an RS_422_PORT_0 part

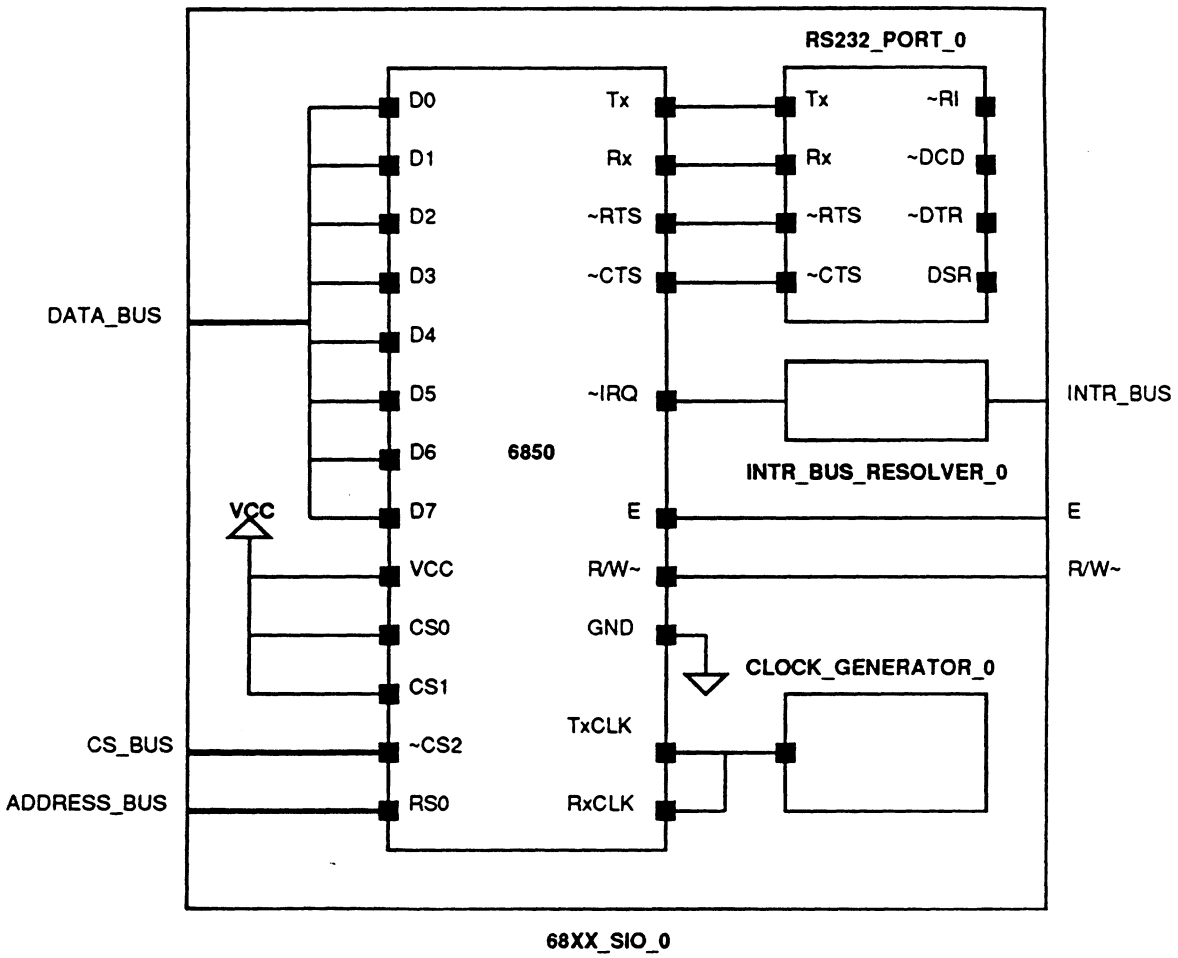

Figure 4: Template for mapping a 68XX_SIO_0 into a 6850.

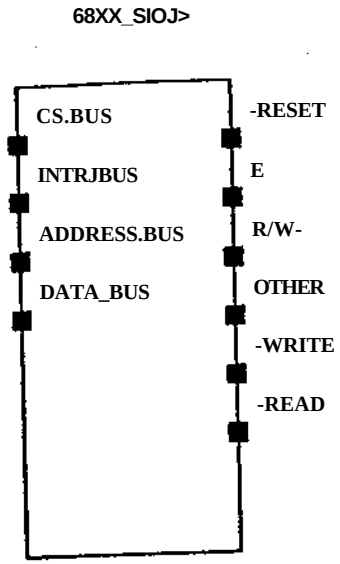

Figure 5: The 68XX_SIO_0 abstract part

Pre-conditions associated with a template determine when it should be applied. The pre-conditions are 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), the parts which are involved in the search for function, and their corresponding specifications and characteristics.

The final component of the Ml architecture is the *design cycle.* The following steps(6,2] compose the basic design cycle:

Specification (d^): values for the specifications of a part are generated.

Selection (d^u):* 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 ($d_{\epsilon mc}y$. cascadablejKrts (e.g. memory) are built out to the proper width.

Structure Design (d^^^^ujz a template is chosen and asserted.

Calculation (dc^u): various calculations associated with the template are performed, including: updating the values of constraints and generating design information.

Steps *dsptc* and $d^{\wedge}{}_{Uc}t$ implement the search for function; the search for structure is realized by design cycle steps: *dcascy* $d_{ump}uu<$, and *d^^.*

A design cycle begins only when a part's specifications are complete. Spawning of new design cycles occun through a linking of *dcau* of one cycle to *d*^* of another cycle. During 4c«dc new information is generated in the form of reports which are then used to complete specifications for some *set* of abstract parts introduced by templates into the design state. These parts then begin their own design cycles, causing other parts* specifications to be completed. This action continues until only physical parts exist in the design state.

```
RULE instantiate__SIO_template
{
(GOAL name = assert^template)/
(SPECIFICATION SIO_LINES = 1);
(SPECIFICATION PROCESSOR -
(6800 OR 6809));
&A (CHAR MAX_BAUD_RATE);
(SPECIFICATION TX_BAUD < &A);
(SPECIFICATION RX_BAUD < &A);
(SPECIFICATION DRIVERJTYPE - RS-232);
(SPECIFICATION TXJTYPE - ASYNC);
(SPECIFICATION RXJTYPE = ASYNC);
(REPORT IO_ACCESS_TIME > 300);
(REPORT REMAINING_J>OWER > 200);
(REPORT REMAINING_BOARD_AREA > 500);
(REPORT REMAINING__BOARD_COST > 120);
— >
make (label name = g8483943);
; The wire list is abbreviated for the sake of conciseness.  Normally
; all parts and connections in the schematic are listed here.
; get_jpart retrieves a part model from the DB.
get_part(RS232_DRIVER);
get_part(BAUD_RATE_GENERATOR);
...
; connect_net creates a connection between the parts and pins in the
; parameter list.
connect__net(6850, DO, SIO_0, DATA_BUS);
connect__net(6850, TxCLK, BAUD_RATE_GENERATOR, OUT);
...
}
```

Figure 7: Final **rule** for template shown in Figure 4.

```
description:  RS-232 PORT_TYPE specification method
calculation_type: specification
calculation: PORTJTYPE = MALE;

description: CLOCKJ3ENERATOR FREQUENCY specification method
calculation_type: specification
calculation: CLOCK_FREQUENCY = TX_BAUD * 16;
```

Figure 8: Example specification methods.

4.2 Generalization

The initial formulation of a template's pre-conditions coincides exactly with the design slate, yielding the tightest set of invocation conditions. This is necessary to comply with Mi's requirement that each $k_{ump}i_{au}$ rule have a unique set of pre-conditions. Often the pre-conditions are overly constraining, preventing the template from being applied in other design states where it is perfectly valid. There are three causes for this:

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

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

Irrelevant Pre-conditions Used: superfluous pre-conditions are used.

The generalization process is used to relax overly constraining pre-conditions by applying 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.

Variable substitution: 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, $k_{g€H£ra}u_{u}$, explains how design variables interact and specifics the proper range of such variables. The $k_{geMra}uu$ partition is composed of two separate sets of knowledge:

$$\hat{}_{generalize} = *^{t}_{generalize\text{-}method} + \&^{'}_{generalize\text{-}rule} \quad (4)$$

where each set is defined as:

$k_{generalize}$*-method** a description given by the domain expert indicating, for each training case, the variables to relax and their new boundary conditions.

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

The $k_{g€M}raUi€$ knowledge resides with CGEN; it is not part of Mi's knowledge-base.

Domain expert supplied methods are used to describe $k_{gmura}iug\text{-}method$ knowledge. The methods for relaxing some of the pre-conditions in Figure 6 are given in Figure 9. The TXJBAUD and RX_BAUD pre-conditions in Figure 9 are set less than or equal to a variable representing the maximum baud rate (MAX_BAUDJRATE) of the 6850[5]. The pre-condition CLOCK_SPEED is removed, because it is subsumed by IO_ACCESS_TIME.

Figure 10 contains an example set of $k_{g€9ura}\backslash xte\hat{}nde$ rules. These rules represent knowledge about general system level constraints. For example, the rule *relax_REMAININGJ2OARDj\REA* sums the area consumed by each part in the template[6], which is used as the lower bound for the constraint *REMAININGJ3OARDj\REA.*

The $k_{gen4}raiiu\text{-}mu$ knowledge partition is generated by the *CGEN* knowledge engineer. Any set of commonly occurring methods are encoded into CGEN rules.

5 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: acquisition of Alf and acquisition of $k_{se}u_{ct}$. Rule generation involves the creation of code from CGEN's internal representation.

[5]MAX_BAUD_RATE is a characteristic of the 6850.

[6]The characteristic *area_consumed* is in every part model

```
description: TX Baud rate max value
calculation__type: generalization
calculation:
 TX_BAUD <= MAX_BAUD_RATE;

description: RX Baud rate max. value
calculation:  RX__BAUD <= MAX_BAUTD_RATE;

method: Delete CLOCK_SPEED
calculation_type: generalization
calculation:  CLOCK_SPEED - DELETE;
```

Figure 9: Methods for relaxing or deleting pre-conditions.

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

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

Figure 10: Example *k^waitie-nde* for relaxing pre-conditions.

```
rule RULE_NAME
  <context-specific LHS>
  <instance-specific LHS>
— >
  <context-specific RHS>
  <instance-specific RHS>
```

Figure 11: General Ml rule form.

The function of the acquisition cycle is to guide the overall problem solving activity of CGEN during a training session. The acquisition cycle determines the types of analysis performed on the incoming knowledge. Two types of analysis are performed:

Error/completeness checking: **(applied during both cycles) the correctness and completeness of knowledge acquired during a session is checked.**

Generalization: **(applied only during *Kj*) the previously described generalization scheme is applied.**

Implicit in the definition of ATE is the relationship of the design cycle (Mi's problem-solving paradigm) to the knowledge-base. Since the acquisition cycle is based on $K\$_t$ it drives CGEN to cull from the domain expert the knowledge necessary to maintain consistency in Mi's knowledge-base. This ensures consistent design cycle action. Without exploiting knowledge of Mi's behavior, CGEN would be incapable of verifying the completeness of incoming knowledge.

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_{Umplau} rules. The behavior of Ml would not be impaired if the generalization process were not applied; however, the domain expert would need to supply a larger number of training cases.

During the rule generation process, CGEN's intermediate representation of the acquired knowledge is translated into a set of rules executable by Ml. Ml 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, as shown in Figure 11. The LHS context-specific portion describes, among other things, the design cycle step a rule belongs to. The RHS context-specific portion performs various data maintenance tasks. All rules in a partition share the same context-specific LHS and RHS, ensuring all rules in a partition are invoked uniformly and have similar actions[7]. The instance-specific portion is generated for each rule based on information gleaned from the training case. For example, for k_{Ump}\au rule instances, the design state pre-conditions form the LHS instance-specific portion of the rule, and the RHS instance-specific portion is created from the wire list.

An important property of the representation of templates and *canonical* form for rules is that new knowledge can be added without destructively interfering with existing knowledge. In domains where the knowledge-base will grow larger over an extended period this property is essential, especially if many different experts will be adding to the system.

The simplified acquisition cycle for *K%* is given below:

1. **Read inputs (wire list, methods, design state)**
2. **Identify pre-conditions for k_{Umplau}**
3. **Apply $k_{generaUze}$:**
   (a) **Identify overly-constraining pre-conditions**
   (b) **Relax or delete those pre-conditions identified**
4. **Instantiate k_{te}*mpiau'*-

[7] **This eliminates many of the annoying side-effects inherent in rule-based programming.**

(a) Generate LHS from pre-conditions from methods

(b) Generate RHS from wire list from methods

5. Generate $k_{ca}i_c$ rules

6. Generate $k_{sp\epsilon C}$ rules

6 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 Ml

A group of four designers, *i.e.* domain experts, used CGEN exclusively to teach Ml how to design. None of the domain experts, except one of the authors[8], was familiar with AI programming techniques, the OPS/83 language, or the implementation details of CGEN and M1. The designers received training in Ml and CGEN design philosophy and tool usage equivalent to a two-day course. During the knowledge acquisition process, the domain experts did not write any OPS/83 code or modify the rules created by CGEN[9].

At the beginning of the experimental process, the Ml knowledge-base contained only:

$$Kl = kcasc + karch \tag{5}$$

Therefore, Ml started with no design knowledge.

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

- Motorola 6809
- Motorola 68008
- 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.

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. Training cases were then developed for each part. Each training case consisted of the following steps:

   (a) Templates were drawn using a schematic drawing package.

   (b) The set of *methods* was prepared.

   (c) Knowledge acquisition sessions were conducted using CGEN.

6.1 Part Data

The size of the database in terms of part models is given in Table 2. Table 3 provides the average number of characteristics, specifications, and pins per part model.

[8] Birmingham entered the 6809 family knowledge.

[9] Four rules were hand-edited by Birmingham to add a special RHS action, which is no longer needed.

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

Table 2: Summary of part data.

Part Model Data		
	Average	Std. Dev.
Specification/model	4	3
Characteristics/model	4	5
Pins/model	15	18

Table 3: Summary of part model data.

6.2 Rule Data

The number of training cases and rules for each design are given in Table 4, with the breakdown of rules by partition shown in Table 5. The knowledge-base is nearly three times as large as the original MICON system[3], and significantly larger than most reported data for design systems. Considering that it took roughly four man-months to build the knowledge-base, CGEN assisted experts can produce about 11 fully debugged rules per day. This represents an order of magnitude increase in productivity over hand-coding.

Partitions k_{casc} and k_{arCh} were fixed when the knowledge acquisition process began. The k_{casc} partition allowed construction of memory arrays and replication of IO devices; no other structures were necessary. 10 replication occurs when the Ml user requests multiple copies of a single function; for example, two SIO devices. In this case, Ml creates as many instances of the function as needed. There was no growth in either k_{CQsc} or k_{arch} during the entire experimentation period, as expected. Table 6 shows the average growth rates for all partitions in units of rules per training case.

Training Cases and Rules Per Design		
Design	Cases	Rules
M6809	86	343
M68008	38	173
M68010	19	147
I80386	79	256
TOTAL	221	919

Table 4: Number of training cases and rules per design.

Rules per Partition		
Partition	**Rules**	**% of Total**
Kpec	552	60
Kelect	8	.1
*Kemplat**	226	25
Kale	133	15
TOTAL	919	100

Table 5: Number of rules per partition.

Average Growth Rate		
Partition	**Growth Rate**	**Std. Dev.**
k_{spec}	2.5	4.8
KeUct	.04	.03
Jocose	0	0
kumglate	1.0	0.2
Kale	0.6	1.8
Kirch	0	0

Table 6: Average growth rate per partition in rules/training case.

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: LHS data averages.

RHS Data (avg)	
Number Wires/Rule	11
Number Parts/Rule	1.1
Number of cases	221

Table 8: RHS data averages.

The complexity of the LHS of template rules is illustrated in Table 7. The first entry in the table is the average size of the design state or number of template pre-conditions generated by Ml at the beginning of a training session, followed by the average number of pre-conditions surviving to become part of a k_{Umpiau} rule . The number of pre-conditions removed by the domain expert is given next. This is followed by both the number, and then percentage of pre-conditions generalized (of those not deleted). Note the high percentage, 77%, of pre-conditions relaxed, verifying that a generalization technique is necessary.

The RHS data for k_{Umplau} rules are summarized in Table 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 two factors. First, many templates (especially for the 80386) require a large number of wires to connect the parts in the template[10]. Second, the templates tend to have local information[8] 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 sub-problem^.

An assumption underlying the Ml architecture is the orthogonality of template pre-conditions. To confirm this assumption, Ml was instrumented to write to a file every time more than one template was eligible for selection. Every Ml run contributed to the file. Examination of the file verified that in all cases, no more than one template rule was eligible for execution.

6.3 Design Generation and Construction

Once training was completed, experiments were run using Ml to generate an example set of designs for all the micro-processors families, thereby verifying correctness of the knowledge acquired by CGEN. The specifications used for each design are shown in Table 9. Ml successfully produced these designs[11].

[10] Consider that an 80386 has 32 address and 32 data lines.

[11] These designs represent a few points in the large space of designs Ml is capable of producing.

8 Acknowledgments

Anurag Gupta implemented Ml and contributed greatly to the development of CGEN. Tom Mitchell provided many helpful insights.

References

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

[2] William P. Birmingham, Audrey Brennan, Anurag P. Gupta, and Daniel P. Siewiorek. Micon: a single board computer synthesis tool. ***IEEE Circuits and Devices Magazine,*** **January 1988.**

[3] William P. Birmingham and Daniel P. Siewiorek. Micon: a knowledge-based single board computer designer. In ***Proceedings of the 21st Design Automation Conference,*** **IEEE and ACM-SIGDA, IEEE Computer Society, 1984.**

[4] W.P. Birmingham, A.P. Gupta, and D.P. Siewiorek. The micon system for computer design. In ***Submitted to The 26th Design Automation Conference,*** **IEEE Computer Society, 1989.**

[5] W.P. Birmingham and D.P. Siewiorek. Automated knowledge acquisition for a computer hardware synthesis system. In ***Proceedings from the 3rd AAAI Knowledge Acquisition for Knowledge-based Systems Workshop,*** **AAAI, 1988.**

[6] W.P. Birmingham and D.P. Siewiorek. Single board computer synthesis. In Michael D. Rychener, editor, ***Expert Systems for Engineering Design,*** **Academic Press, 1988.**

[7] F.D. Brewer and D.D. Gajski. An expert-system paradigm for design. In ***23rd Design Automation Conference Proceedings,*** **IEEE Computer Society, 1986.**

[8] A.P. Gupta and D.P. Siewiorek. A hierarchical problem-solving architecture for synthesis. In ***Submitted to The 26th Design Automation Conference,*** **IEEE Computer Society, 1989.**

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

[10] G. Klinker, J. Bentolila, S. Genetet, M. Grimes, and J. McDermott. Knack- report-driven knowledge acquisition. ***International Journal of Man-Machine Studies,*** **26, January 1987.**

[11] S. Marcus and 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.**

[12] J. McDermott Domain knowledge and the design process. In ***18th Design Automation Conference,*** **IEEE Computer Society, 1981.**

[13] T.M. Mitchell, S. Mahadevan, and L. Steinberg. Leap: a learning apprentice for vlsi design. In ***Proceedings ofIJCAI-85,*** **Morgan Kaufmann Publishers, 1985.**

[14] E. Soloway, J. Bachant, and K. Jensen. Assessing the maintainability of xcon-rime; coping with the problems of a very large rule-base. In ***Proceedings of AAAI-87,*** **Morgan Kaufmann Publishers, 1987.**

[15] L.I. Steinberg and T.M. Mitchell. A knowledge-based approach to vlsi cad. In ***Proceedings of the 21st Design Automation Conference,*** **IEEE and ACM-SIGDA, IEEE Computer Society, 1984.**

Design Specifications				
Design	Amount SRAM	Amount ROM	Amount P1O	Amount Timer
1	4 KBytes	1 KByte	1	0
2	4 KBytes	1 KByte	0	1
3	58 KBytes	4 KByte	1	1
4	58 KBytes	4 KByte	2	2

Table 9: Input specification summary for Ml design tests.

The design Ml produced for the 68008 using specification set 3 from Table 9 was built and is working. Analysis of the design shows the following:

- The design works at expected clock rates.
- The number of parts used is the same as a hand-generated design.

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.

6.4 Other Experience

CGEN, as part of the MICON system, has been used extensively outside its original development and testing group. As described in a companion paper [4], CGEN has been used in a number of industrial-university workshops. The workshop attendees included hardware designers, who independently verified the experimental results by using CGEN to capture a small amount of design knowledge (roughly 10 rules). Additionally, CGEN is being evaluated at several industrial sites.

CGEN is also being used in an education setting, as part of a micro-processor interfacing class taught at the University of Michigan. Students use CGEN to build an Intel 8086 family knowledge-base, which they subsequently exercise with Ml to generate designs.

The total number of rules CGEN has acquired in experimental, educational, and industrial settings combined is over 1100.

7 Summary

CGEN has demonstrated an effective mechanism for capturing knowledge from domain experts. The ability of domain experts unfamiliar with either OPS/83 or the implementation of the Ml 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 produce designs, thus proving its ability to encode such knowledge appropriately for Ml.

There are several limitations on CGEN. CGEN is tightly coupled to the Ml architecture. Only declarative knowledge is 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. Therefore, CGEN lacks an independent reference for verifying that the acquired knowledge is correct.

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