HAMLET - An Expression Compiler/Optimizer for the Implementation of Heuristics to Minimize Multiple-Valued Programmable Logic Arrays*

John M. Yurchak
Department of Computer Science
Naval Postgraduate School, Code CS/Yu
Monterey, CA 93943-5000

Jon T. Butler
Department of Elect. and Comp. Eng.
Naval Postgraduate School, Code EC/Bu
Monterey, CA 93943-5004

ABSTRACT

HAMLET is a CAD tool that translates a user specification of a multiple-valued expression into a layout of a multiple-valued programmable logic array (MVL-PLA) which realizes that expression. It is modular to accommodate future minimization heuristics and future MVL-PLA technologies. At present, it implements two heuristics, [2] and [8] and one MVL-PLA technology, current-mode CMOS [6]. Specifically, HAMLET accepts a sum-of-products expression from the user, applies a minimization heuristic, and then produces a PLA layout of a multiple-valued current-mode CMOS PLA.

Besides its design capabilities, HAMLET can also analyze heuristics. Random functions can be generated, heuristics applied, and statistics computed on the results. User-derived expressions can also be analyzed. In addition to the minimization heuristics [2] and [8], HAMLET can apply search strategies based on these heuristics, which, in the extreme, is exhaustive, producing true minimal forms. HAMLET is available to the public; instructions on how to obtain this program are in Appendix A. It is written in C and conforms to the UNIX command line format.

I. INTRODUCTION

The implementation of multiple-valued logic (MVL) circuits in VLSI has created a need for multiple-valued logic computer-aided design (MVL-CAD) tools. Programmable logic arrays (PLA's) are of special interest. Their design is regular, thus placing a lower demand on the tool's capabilities. Also, the technology for multiple-valued PLA's (MVL-PLA's) exists in CCD [4], current-mode CMOS [6], and voltage-mode CMOS [9]. Since the subject is so new, there is only one other MVL-CAD tool, for MVL-CCD PLA's [5].

This paper describes HAMLET (Heuristic Analyzer for Multiple-valued Logic Expression Translation), a tool that accepts an expression, applies minimization algorithms to the expression, and produces an MVL-PLA that realizes the minimized expression. The MVL-PLA layout [6] conforms to MOSSIS design rules and the output file circuits are in MAGIC format [10]. Unlike previous implementations of heuristics [1-3,5,8,12], which represent a function internally as a truth table, HAMLET represents the function as a sum-of-products expression. Thus, we avoid storage space limitations associated with truth tables of even moderately sized functions. This paper is intended to serve as an introduction to HAMLET. A manual, Yurchak and Butler [15], exists giving complete information on its use.

Although HAMLET is a CAD tool, it can also be used to analyze minimization heuristics. It does this by randomly generating expressions, applying the heuristics, and collecting the results. The use of random functions avoids bias that could unfairly favor one heuristic. On the other hand, especially chosen functions can be analyzed; this allows one to selectively investigate specific heuristic characteristics. HAMLET is designed to be easily modified. At present, it implements the Pomper and Armstrong [8] and Dueck and Miller [2] heuristics, as well as various search techniques derived from these. Both the basic heuristics and search algorithms can be modified. Indeed, we do this now as part of our research on improved minimization methods for MVL-PLA's.

While there has been little previous work on CAD tools for MVL circuits, there has been significantly more work on minimization algorithms. We know of three heuristic MVL sum-of-products minimization algorithms. Each uses the direct cover method, in which a minterm (assignment of values to all variables) is first determined and then an implicant is found that covers the minterm. Pomper and Armstrong [8] introduced in 1981 the first known direct cover method for MVL functions. It selects a minterm randomly and chooses the largest implicant covering the selected minterm. In 1986, Besslich [1]...
**HAMLET - An Expression Compiler/optimizer for the Implementation of \Heuristics to Minimize Multiple-Valued Programmable Logic Arrays**

**Abstract**

HAMLET is a CAD tool that translates a user specification of a multiple-valued expression into a layout of a multiple-valued programmable logic array (MVLPLA) which realizes that expression. It is modular to accommodate future minimization heuristics and future MVL-PLA technologies. At present, it implements two heuristics, [2] and [8] and one MVL-PLA technology current-mode CMOS [6]. Specifically, HAMLET accepts a sum-of-products expression from the user, applies a minimization heuristic, and then produces a PLA layout of a multiple-valued current-mode CMOS PLA. Besides its design capabilities, HAMLET can also analyze heuristics. Random functions can be generated heuristics applied, and statistics computed on the results. User-derived expressions can also be analyzed. In addition to the minimization heuristics [2] and [8], HAMLET can apply search strategies based on these heuristics which, in the extreme, is exhaustive, producing true minimal forms. HAMLET is available to the public instructions on how to obtain this program are in Appendix A. It is written in C and conforms to the UNIX command line format.
introduced another direct cover method that seeks to cover the most isolated minterms first. Like the Pomper and Armstrong [8] heuristic, it selects the largest implicant. In 1987, Dueck and Miller [2] introduced a direct cover method that also seeks the most isolated minterm first (by a method different than that of Besslich [1]), but chooses an implicant that tends to introduce the fewest discontinuities when subtracted from the function. Dueck [3] has modified the heuristic described in [2] obtaining improved performance on specific examples.

In addition to the Pomper and Armstrong and Dueck and Miller heuristics, HAMLET implements adaptations of these. Specifically, where these heuristics make one choice from several, HAMLET allows a search in which all choices are examined. The Besslich heuristic was not chosen because of speed considerations. Also, it relies on a truth table representation that is not compatible with the expression representation we chose to implement.

II. BACKGROUND

Let \( X = \{x_1, x_2, \ldots, x_n\} \) be a set of \( n \) variables, where \( x_i \) takes on values from \( R = \{0, 1, \ldots, r-1\} \). A function \( f(X) \) is a mapping \( f : R^n \rightarrow R \cup \{r\} \), where \( r \) is the don't care value. Specifically, \( f(X) \) is said to be an \( n \)-variable \( r \)-valued function. Fig. 1 shows an example of a 2-variable 4-valued function. A function value \( f(m) \) corresponding to a specific assignment of values \( m \) to variables in \( X \) is called a minterm iff \( 0 < f(m) < r \). For example, in Fig. 1 there are seven minterms with value 1, three with value 2, and one with value 3.

Functions realized by the PLA's described in [4,6,11] are composed of three functions,

1. literal: \( f(x_i) = x_i^a \), \( a = r-1 \) if \( a \leq x_i \leq b \), else \( = 0 \),
2. MIN: \( f(x_1, x_2) = \min(x_1, x_2) \), and
3. SUM: \( f(x_1, x_2) = x_1 + x_2 \) (minimum \( x_1 + x_2 \), \( r-1 \)), where \( x_i \) is viewed as an integer and + is integer addition. The SUM operation (+) is thus addition truncated to the highest logic value.

III. HEURISTIC MINIMIZATION ALGORITHMS

We have devised improved versions of existing heuristic methods for the minimization of expressions for implementation by MVL-PLA's. Existing heuristics, when given a choice, choose one option and never backtrack to determine if another choice would have resulted in an improved realization. Our improved versions of these heuristics allow some specified degree of search. For small expressions, exhaustive search can be applied in a reasonable time to produce a minimal expression, while, for large expressions, the search can be restricted to avoid the excessive computation time of exhaustive search.

A. THE HEURISTIC TEMPLATE

All known MVL minimization heuristics use the direct cover method. This is a two step process that...
chooses, for the given function, \( f(X) \),

1. a minterm \( m \) of \( f(X) \), and
2. an implicant \( I(X) \) of \( f(X) \) that covers \( m \).

Function \( f(X) - I(X) \) is formed and the two step process performed on it, until a final function is obtained consisting entirely of 0's and r's (don't cares). With logic values viewed as integers, the operation -- is ordinary integer subtraction except for the following cases. 1. If \( f(x) \) is a don't care, then so also is \( f(x) - I(x) \). Thus, a don't care value in the original function appears as a don't care in all subsequent functions. 2. If \( f(x) - I(x) \) is 0 or less and the given function is \( r - 1 \), then \( f(x) - I(x) \) is a don't care value. This is to accommodate the truncated sum operation when the sum of implicant values produce \( r - 1 \) or more. Since any \( r - 1 \) in the given function is potentially a sum that has been truncated, the algorithm tracks such values; otherwise, certain minimal solutions would be lost. For example, consider a 4-valued function whose minimal sum-of-products expression consists of two implicants with constant 2 that cross at some \( x \) where the function has value 3. Subtracting one of these implicants leaves a function with 2's except for a 1 at \( x \). To realize the minimal solution, the heuristic must now realize the resulting function with just one implicant. That is, it must "recognize" that the 1 was once a 3 and that a 2 can be subtracted from it.

We give here qualitative descriptions of these heuristics. Formal algorithmic descriptions appear in [12].

B. POMPER AND ARMSTRONG [8]

In this version of the direct cover method, minterm \( m \) is chosen randomly. Next, the implicant is chosen so that 1. the implicant value is equal to that of \( m \). 2. the implicant results in the most 0 values in \( f(X) - I(X) \), 3. among the set of all implicants from 2. the largest are selected, and 4. among the set of implicants from 3. is arbitrarily chosen. Consider, for example, the function in Fig. 1. Assume the 1-minterm at \( x_1x_2 = 2 \). is the randomly chosen minterm. Then, the implicant \( 1x_1x_2 \) is selected because it is uniquely the largest implicant that produces the most 0's when subtracted from the function.

C. DUECK AND MILLER [2]

In this version of the direct cover method, minterm \( m \) is chosen as the most isolated minterm. Specifically, for each minterm \( m \) with the smallest value \( f(m) \), the clustering factor \( CF(m) \) is computed and the minterm with the smallest \( CF(m) \) is chosen. To compute the clustering factor, one tallies, for each minterm, adjacent minterms with which \( m \) can be combined and the directions (variables) having at least one minterm with which \( m \) can be combined. Minterms that are isolated (i.e., surrounded by few other minterms) tend to have the lowest clustering factor, and are chosen first. Consider again the function in Fig. 1. There are seven minterms with the smallest \( f(m) \), those corresponding to 1's in the map. Among these, the minterm \( m \) with lowest \( CF(m) \) is \( x_1x_2 = 3 \). Here, the clustering factor is 4, while all other clustering factors are greater than 4. Note that \( x_1x_2 = 3 \) is the only 1-minterm that is adjacent to less than two other minterms.

For a selected minterm, an implicant is chosen that has the smallest \( rbc \), relative break count. That is, the relative break count is a measure of how many discontinuities are introduced into a function when the present implicant is subtracted. For example, there are four implicants covering the 1-minterm at \( x_1x_2 = 3 \). An implicant that leaves holes or break up a function tend to have a higher \( rbc \) than those that do not. This has the intuitive interpretation that the preferred implicants are those implicants whose subtraction leaves a function that is realized by as few remaining implicants.

D. HEURISTICS APPLIED WITH BACKTRACKING

The heuristics discussed above proceed from a given function to a function consisting entirely of 0's and don't cares. At each step, only one choice is made, even when there is more than one (equally good) choice. That is, in the case of ties, only one is chosen. In HAMLET, the user is given the option of exploring various choices. That is, at any point in the algorithm, a set of choices is recorded so that at a later time, when backtracking occurs, these alternative choices can be made. This option is implemented with a recursive program that searches a tree, in which nodes correspond to functions and arcs to implicants. The root node corresponds to the given function and all its children to functions derived by subtracting single implicants from the given function. The recursive program calls itself as it moves through the tree searching for the realization with fewest implicants. The depth to which the program goes corresponds to the number of implicants. Initially, the program searches to some maximum depth determined by the best known expression for the given function. As better realizations are found, this maximum depth becomes smaller, until the end when the expression with the fewest implicants has been determined. At each application of the recursive call, two expressions are considered, the current (probably incomplete) solution and the best obtained so far.

IV. MINIMIZATION USING HAMLET

A. THE STRUCTURE OF HAMLET

HAMLET is a family of utility programs. Written in the C programming language, it currently runs on the UNIX operating system, but should port easily to other
environments. The user controls the behavior of HAMLET by supplying command line options formatted according to standard UNIX conventions. Some of the programs that compose HAMLET are

- **mvlc**: An expression compiler and optimizer. Applies one or more heuristics to MVL expressions, reports heuristic performance and produces an MVL-PLA data file that is the input to the PLA layout generator, *mvll*.
- **mvlt**: A test expression generator. Produces sets of randomly generated expressions that conform to parameters supplied by the user.
- **nzvla**: A heuristic performance analyzer. Takes heuristic performance data from successive runs of *mvlc* and produces statistical data.
- **mvll**: A PLA layout generator. Accepts a data file supplied by *mvlc* and produces a layout of a current-mode CMOS PLA realizing the expression in its data file.

**B. mvlc - Expression Compiler.**

The most important of these programs is *mvlc*. Its 5700 lines of code correspond to about 85% of HAMLET. *mvlc* provides a user-interface similar to that of a typical high-level programming language compiler. The user creates an input file, using a text editor, consisting of MVL expressions. Fig. 2 shows the 2-variable 4-valued expression in (1) in the format suitable for *mvlc*.

![Figure 2. *mvlc* Format for the Expression in (1).](image)

```
4:2:
+1*X1(0,0)*X2(0,0)
+1*X1(1,1)*X2(0,1)
+1*X1(0,2)*X2(1,1)
+1*X1(0,3)*X2(1,1)
+1*X1(1,2)*X2(2,3)
+2*X1(0,0)*X2(2,2);
```

**Figure 2. *mvlc* Format for the Expression in (1).**

*mvlc* extracts semantic information from the input expression and stores it as a linked list. For example, the expression given in Fig. 2 is stored as shown in Fig. 3. This original input expression is called $E_{\text{Orig}}$, *mvlc* uses this structure as a basis for applying selected heuristics in an attempt to produce an equivalent structure (identical coverage) with fewer implicants. For example, when the value $f(m)$ of the expression is needed for some assignment of values $m$ to the variables $X$, the linked list of implicants is scanned (by subroutine EVAL) as the contribution of each to $f(m)$ is tallied. At the end, a correct value of $f(m)$ is achieved. In addition to reporting heuristic performance results, *mvlc* creates an output file representing an optimized MVL expression that is the input to *mvll*, the layout generator.

**One-Pass Method**

This is the default mode. Assuming a user-selected heuristic, $H$, the one-pass method proceeds as follows.

1. Copy $E_{\text{Orig}}$ to $E_{\text{Work}}$.
2. Initialize $E_{\text{Final}}$ to an empty expression (no implicants).
3. Repeat
   1. Apply $H$ to $E_{\text{Work}}$, producing an implicant, $I$;
   2. Subtract $I$ from $E_{\text{Work}}$;
   3. Add $I$ to $E_{\text{Final}}$;
4. Report the results and output $E_{\text{Final}}$;
5. Stop

**Figure 3. Internal Representation of the Expression in (1).**

Using command line options, the user selects a) one or more heuristics and b) a search method used by *mvlc*. There are two categories of search methods, one-pass and multi-pass.

**Multi-Pass Method**

The multi-pass method is a backtracking search. Treating the current state of the working expression, $E_{\text{Work}}$, as a node in an $n$-ary tree, each candidate implicant for that expression, when subtracted from $E_{\text{Work}}$, yields a daughter node corresponding to an expression with one less term. The search can be configured in many ways to allow subtle alterations in the performance of the selected heuristic. The application of the direct cover heuristic invokes two choices a) a minterm selection function $f_{\min}$ and b) an implicant selection function $f_{\imp}$.
Both are used within a recursive function, $f_{\text{Search}}$, whose algorithm is shown in Fig 5. The multi-pass method applies $f_{\text{Search}}$ in a depth-first search for the shortest path to coverage of the input expression. This algorithm is shown in Fig 6.

For the current state of $E_{\text{Work}}$

1. Apply $f_{\text{Min}}$
   1.1 Select one or more minterms and save them in a list, $L_{\text{Min}}$;

2. Apply $f_{\text{Imp}}$
   2.1 For each minterm in $L_{\text{Min}}$
      2.1.1 Select one or more implicants and save them in a list, $L_{\text{Imp}}$;
   3. For each implicant $I$ in $L_{\text{Imp}}$
      3.1 Push (save) the state of $E_{\text{Work}}$ and $E_{\text{Final}}$;
      3.2 Subtract $I$ from $E_{\text{Work}}$;
      3.3 Add $I$ to $E_{\text{Final}}$;
      3.4 If $E_{\text{Work}}$ is covered |
         3.4.1 Save $E_{\text{Final}}$;
         3.4.2 Return;
      3.5 Recursively apply $f_{\text{Search}}$;
      3.6 Pop (restore) the state of $E_{\text{Work}}$ and $E_{\text{Final}}$;
   }

Return

Figure 5. The Algorithm for $f_{\text{Search}}$.

For each input expression $E_{\text{Orig}}$

1. Copy $E_{\text{Input}}$ to $E_{\text{Work}}$;
2. Initialize $E_{\text{Final}}$ to an empty expression (no implicants);
3. Apply $f_{\text{Search}}$ to $E_{\text{Work}}$;
4. Report the results and output the saved $E_{\text{Final}}$;

Stop

Figure 6. The Algorithm for the Multi-Pass Method.

Example of Results from mvlc

To observe how mvlc applies a heuristic, consider once again the function in Fig. 1. This can be realized by an expression with six implicants, as shown in (1) and Fig. 1. Assume the user has established the input file with the six implicant expression shown in Fig. 2. Consider the application of Dueck and Miller [2] using the one-pass method. Fig. 7 shows the selection of the first implicant.

First, for all minterms of lowest logic value, the clustering factor is computed and the lowest is identified. Here, the 1-minterm at $x_1x_2 = 31$ is the only one with the lowest clustering factor of all minterms with lowest logic value. As can be seen in Fig. 7, it is the only 1-minterm that is adjacent to one other minterm; all other 1-minterms are adjacent to at least two other minterms.

Next, the implicant is selected. Fig. 7 shows the four implicants that cover the selected minterm, as well as the $rbc$ for each. There is exactly one with the lowest $rbc$. It is

$$1^*X1(1,3)^*X2(1,1)$$

This is subtracted from $E_{\text{Work}}$ and added to $E_{\text{Final}}$ as shown in Fig. 8. Here, $E_{\text{Work}}$ is the same as the $E_{\text{Work}}$ shown in Fig. 3 except for an additional implicant on the right. This corresponds to $-1^*X1(1,3)^*X2(1,1)$. This process is repeated until all minterms in $E_{\text{Work}}$ evaluate to 0 or don‘t care. In the end, $E_{\text{Final}}$ is a structure similar to $E_{\text{Orig}}$, except that its implicants are chosen by the heuristic. This structure can be used to generate an input file for the layout generator. Fig. 9 shows the expression realized by the one-pass Dueck and Miller [2] heuristic applied to the function in Fig. 1. As can be seen, only five implicants are needed, which represents a reduction of one implicant over the user-defined expression.

One limitation of this heuristic is that, for a given state of an expression, minterms with the lowest clustering factor and implicants with the lowest $rbc$ do not always yield the minimal expression. The multi-pass method corrects this by examining alternatives. To illustrate its flexibility, consider the running example expression. If the multi-pass method is applied to the expression in Fig. 2, and we require that for each partial expression, the three implicants of lowest $rbc$ are chosen (in comparison with the one-pass method, in which only one implicant is chosen), then a solution with four implicants is chosen, as shown in Fig. 10. It is interesting that this solution is not found in the one-pass method because, in choosing
Figure 8. $E_{Work}$ and $E_{Final}$ After the First Implicant is Chosen.

1. "For which of 1000 randomly generated expressions does Dueck and Miller do better than Pomper and Armstrong?"

2. "For how many expressions in a set of 2000 random expressions does Dueck and Miller yield greater than 10% more implicants than is in the original expression?"

3. "What is the mean performance of some search option as compared to Pomper and Armstrong?"

In many cases, such problems are solved automatically by mvlc in conjunction with other programs in the HAMLET family. For example, we use mvlt to generate large sets of random expressions to obtain statistical data on heuristic performance. This tool creates ordinary text files that can be read and edited by the user if desired. mvla automatically executes mvlc on sets of expressions, comparing the performance of various heuristic options are graphing the results on a Postscript printer. As an example, we can use mvla to automatically generate data for a graph of the relationship between, say, the number of terms per expression and the time to find a solution for a given heuristic.

C. mvlt - Test Expression Generator

mvlt is a program that generates a set of random expressions for use by mvlc. The expressions generated by mvlt use a random number generator that generates

1. a nonzero coefficient $p$ from $r - 1$ possible coefficients with uniform distribution and
2. a set of $n$ intervals $(a_i, b_i)$, where $a_i \leq b_i$, from the set of all $\binom{r}{2} + r$ possible intervals with uniform distribution, where $n$ is the number of variables.

One implicant is formed with a structure as follows:

$$p \cdot x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}$$

If more than one implicant is requested, then a similar process is repeated for each. The generator makes one further restriction on the set of implicants. Whenever an
implicant is generated, a check is made to see if an implicant was generated earlier identical to the present one except perhaps in the constant. If so, the present one is discarded and a new implicant is generated. \( mvlt \), however, does not check whether a currently generated expression was generated earlier. This would increase considerably the execution time and only preclude a typically unlikely event.

D. \textit{mvla} - Statistical Data Analyzer

An important use of HAMLET is in analyzing the behavior of heuristics. This is done in \textit{mvla}, which uses random expressions generated by \textit{mvlt} and minimized by \textit{mvlc} to produce various graphs and histograms of heuristic performance. Included in the operations performed by \textit{mvla} are

1. average value of number of implicants used over the ensemble,
2. percent of the minimized expressions that have fewer, the same, and more implicants than the number of implicants in the given function set,
3. total number of implicants used by the minimized expressions divided by the total number of implicants in the given set of expressions
4. average number of implicants used in the minimized expressions where the minimized expression had fewer implicants than the given expression.

An especially useful feature is the automatic generation of histograms from the data generated. For example, the application of a heuristic on say 1000 functions with say 6 implicants produces some number of minimized functions with 6, 5, etc. implicants. When \textit{mvla} is completed, a plot is printed showing the number of functions for each number of implicants in the form of a histogram. Similarly, plots can be automatically generated of some behavior like the average number of implicants versus radix or the time of computation versus the number of variables. An example of the output produced by \textit{mvla} is shown in Fig. 11. Here \textit{mvlt} was asked to generate 1000 random 4-valued 2-variable functions each with six implicants. Both the one-pass (top histogram) and multi-pass (bottom histogram) versions of Dueck and Miller were applied to this set. The line just below each histogram shows statistics associated with the plot above. Starting from left to right, the value to the right of the \(<\) sign shows the fraction of expressions improved by the heuristic (85% and 94%), the value to the right of the \(=\) sign shows the fraction of expressions where the heuristic produced exactly the same number of implicants as the randomly generated function (12% and 6%), and the value to the right of the \(<\) sign shows the fraction expressions where the heuristic did not do as well as the randomly generated function (3% and 0%). These figures show clearly the improvement achieved by the search technique associated with the multi-pass method. The value to the right of \(P_m\) shows the performance, the fraction of the total number of implicants in the ensemble used by the heuristic (69% and 65%). That is, over all 1000 expressions generated, the one-pass method Dueck and Miller heuristic used 69% of the 6 · 1000 required in the realization of set of unminimized randomly generated expressions. The multi-pass method used 4% fewer implicants or 65%. The standard deviation appears to the right of the performance figure (0.23 and 0.20). Next, is the fraction of implicants used by the best (B) expression and the worst (W) expression. These are 17% and 17% for the best and 133% and 11% for the worst. That is, for the one-pass method, out of the 1000 randomly generated expressions, the expressions requiring the least and most implicants in the expressions after application of the heuristic required 17% (or 1) and 133% (or 8) of the implicants in the given expression (6). At the extreme right is the time required by \textit{mvla} complete the analysis. This shows that for the one-pass method, 0.01 seconds per expression were required, and, for the multi-pass method, 0.32 seconds per expression were required. Thus, the time required rises significantly, because of the search done in the multi-pass method. There is a variability in the time required by various expressions. That is, \textit{mvla} prints out a
number indicating which expression it is currently minimizing. The times to execute various expressions differ greatly, especially in the multi-pass method; some proceed quickly, while others much longer. The rightmost figure shows the variance in time required for minimization. It is small in the one-pass method (0.00) and large in the multi-pass method (0.44). The black box in the abscissa shows the number of implicants in each of the randomly generated functions, 6 in the case of this example. Just above each histogram is a date/time stamp and the mvlc command that created the histogram.

E. **mvll - PLA Layout Generator**

**mvll** produces the layout of a current-mode CMOS PLA that realizes the given input expression. The layout conforms to the conventions of Berkeley’s Magic program [10]. Thus, a Manhattan scalable CMOS design is produced that satisfies the Mead/Conway lambda design rules. There are no options; the design is produced directly from the input. Fig. 12 shows the layout of a current-mode CMOS PLA produced by **mvll** that realizes the minimal expression of the running example (Fig. 10). Here, the two inputs (x00 and x01) enter from the left, while the four product terms are laid out horizontally and are summed at the bottom to form the output (f00).

VII. CONCLUDING REMARKS

**HAMLET** is a CAD tool for multiple-valued logic expressions. It accepts a user-specified sum-of-products expression, attempts to find a smaller expression, and then produces the layout of a PLA that realizes the expression. In addition, **HAMLET** is an analysis tool. For example, it can generate random functions, apply chosen minimization heuristics, and compile statistics. This allows us to compare heuristics. Yang and Wang [14] have used **HAMLET** to develop new heuristics for MVL-PLA minimization. An operation manual, Yurchak and Butler [15], exists showing the complete set of options available in **HAMLET**. Appendix A shows how **HAMLET** can be obtained over the ARPANET.

**HAMLET** has been designed to be easily modified; for example, new heuristics are easily added, as well as layout generators for technologies other than current-mode CMOS. A significant part of the effort was devoted to developing a program that would have a long lifetime. Our expectation is that, as more experience is gained with MVL-PLA’s, they will become a predominant part of MVL circuit design. Since we have only a limited basis on which to choose heuristic minimization algorithms, we view this as a productive research area. **HAMLET** is highly structured, so that, with simple well-documented procedures, one can easily generate this companion software. For example, the EVAL function, a subroutine that accepts an assignment of values to variables and produces a function value, can be easily changed to a fast table lookup program when significantly larger memories become available, allowing a truth table lookup. At this time, EVAL scans the linked list for the function value on the first call, stores the value (when storage is available), then simply returns this value on all subsequent calls.

VII. ACKNOWLEDGMENTS

Most of **mvll**, the VLSI layout generator, was developed by Y.-H. Ko [6] as part of his M.S. thesis at the Naval Postgraduate School. A preliminary version of the heuristic minimization part of **HAMLET** appeared in the M.S. thesis of H.-S. Lee [7]. The authors express their gratitude for their contribution to this work. Also, the authors appreciate the comments of the referees, which led to improvements in this paper.

![Figure 12](image-url)
REFERENCES


APPENDIX A. HOW TO TRANSFER HAMLET TO YOUR ACCOUNT

HAMLET is public domain software. No warranties are made regarding its operation. If you have an ARPANET connection, you can obtain HAMLET by logging into your account, moving to the directory where you will be using HAMLET, and applying the following procedure.

1. % ftp cs.nps.navy.mil
   Invoke the file transfer program, connecting to the VAX-11 in the Department of Computer Science at the Naval Postgraduate School. If this succeeds, you will see the login prompt.

2. Login as username "anonymous", password (your own name)

3. > cd pub
   Change the current directory to the public domain directory.

4. > binary

5. > get mv1.tar.Z
   Transfer the set of mv1 programs to your directory on your home account.

6. > bye
   Exit the Naval Postgraduate School's system.

7. % uncompress mv1.tar.Z
   Convert the files to standard uncompressed format.

8. % tar xvf mv1.tar
   Extract the files.

9. % more README
   Read the latest changes and instructions.

10. % make all