Appendix A
Java Source Code of the Qune-McCluskey Algorithms

// global variables:
int NUMINPUTS; // the number of inputs
int NUMTERMS; // the number of minterms

/*
* irredundant removes duplicate terms from the parameter "terms"
*/
void irredundant( Vector terms ) {
    for (int term = terms.size() - 1; term > 0; term--) {
        for (int check = term - 1; check >= 0; check--) {
            boolean duplicate = true;
            for (int literalIndex = 0; literalIndex < NUMINPUTS; literalIndex++) {
                if ( ((int[]) terms.elementAt(term))[literalIndex] != ((int[]) terms.elementAt(check))[literalIndex] ) {
                    // a literal is different: term and check are not duplicates
                    duplicate = false;
                    break;
                }
            }
            if ( duplicate ) {
                terms.removeElementAt(term);
                break;
            }
        }
    }
}

/*
The first Quine-McCluskey algorithm:
* data is a vector containing the first level
* of terms (the minterms). The structure of
* data is a 3-dimensional vector/array combination
*/
* where the first "tier" is a vector of terms.
* Each term in the first "tier" holds an array of literals.
* QM1 returns a vector of terms containing arrays of literals.
*/

Vector QM1( Vector data ) {
    // initialize variables:
    // initialize variables
    int diff = -1; /* the index of the literal that the terms being compared have a difference at.
    * diff is -1 if there is no difference.
    */
    boolean termIsPrime = false; // is the term "term" (loop index) prime?
    int nextLevelTermCount; /* the number of terms in the next "level" of data - 1.
    * corresponds to the index of the last term in the next level.
    */
    Vector primes = new Vector(); // a vector containing all prime implicants. It will be returned at the end.

    /* repeat for each level:
    * start on the first level to create the second level, then
    * go to the second level to create the third level, and so on.
    * when there is no more reduction possible, break.
    */
    for (int level = 0; level <= NUMINPUTS; level++) {
        // initialize variables:
        nextLevelTermCount = -1;
        irredundant( (Vector) data.elementAt(level) ); // remove duplicates from the list of terms
        data.addElement(new Vector()); // initialize the next level
        Hashtable nonprimes = new Hashtable(); /* a hashtable containing the indeces of the terms
        * that are not prime. It will be used to find
        * the terms that are prime.
        */

        // 2 nested loops to compare a term term with a term compterm to see if they can be combined
        for (int term = 0; term < ((Vector) data.elementAt(level)).size() - 1; term++) {
            for (int compterm = term + 1; compterm < ((Vector) data.elementAt(level)).size(); compterm++) {
                // check literals in term and compterm to ultimately see if the terms can be combined
                for (int literal = 0; literal < NUMINPUTS; literal++) {

```
if ((int[]) ((Vector) data.elementAt(level)).elementAt(term))[literal] != ((int[]) ((Vector) data.elementAt(level)).elementAt(compterm))[literal] ) {
    // there is a difference between literals
    if (diff == -1) { // there was no previous difference
        diff = literal;
    } else { // there was a previous difference
        diff = -1;
        break; // there is more than one difference: get out of this comparison
    }
}

    // end of literal for loop
if (diff != -1) { // term and compterm can be combined
    nonprimes.put( new Integer(term), new Integer(term) ); // term is not prime
    nonprimes.put( new Integer(compterm), new Integer(term) ); // compterm is not prime
    // initialize a space to put the combined term in the next level
    ((Vector) data.elementAt(level + 1)).addElement(new int[NUMINPUTS]);
    nextLevelTermCount++; // we are adding a term to the next level
    System.arraycopy( (int[]) ((Vector) data.elementAt(level)).elementAt(term), 0,
                     (int[]) ((Vector) data.elementAt(level + 1)).elementAt(nextLevelTermCount), 0,
                     NUMINPUTS);
    // the different literal is a don't care:
    ((int[]) ((Vector) data.elementAt(level + 1)).elementAt(nextLevelTermCount))[diff] = 2;
    diff = -1; // initialize diff for the next comparison
}

    // end of compterm loop
}

    // end of term loop

// check to see if there were any primes:
for ( int i = 0; i < ((Vector) data.elementAt(level)).size(); i++ ) {
    if (!nonprimes.containsKey(new Integer(i))) {
        // the i_th term of this level is prime
        primes.addElement(new int[NUMINPUTS]);
        System.arraycopy( (int[]) ((Vector) data.elementAt(level)).elementAt(i), 0,
                         (int[]) primes.elementAt( primes.size()-1 ), 0, NUMINPUTS);
    }
}
if ( nonprimes.size() == 0 ) {
    // there was no reduction in this level
    break;
}
// end of level loop
return primes;

/*
The second Quine-McCluskey algorithm
minterms is a vector containing the minterms.
Each minterm is an array of 0 to NUMINPUTS - 1 literals.
implicants is a vector containing the implicants.
Each implicant is an array of 0 to NUMINPUTS - 1 literals.
*/
Vector QM2(Vector minterms, Vector implicants) {
    if (implicants.size() > 1) {
        /* Create a 2D array "checkMap" containing the checks in the table.
The first index is the minterm (column) and the second is the implicant (row)
*/
        boolean checkMap[][] = new boolean[minterms.size()][implicants.size()];
        // put "checks" into checkMap: checks have a value of true
        for (int implicant = 0; implicant < implicants.size(); implicant++) {
            for (int minterm = 0; minterm < minterms.size(); minterm++) {
                boolean impliesMinterm = true; // does the implicant imply the minterm? Init. to true.
                for (int literal = 0; literal < NUMINPUTS; literal++) {
                    if ( !(((int[])implicants.elementAt(implicant))[literal]
                           == ((int[])minterms.elementAt(minterm))[literal])
                        || ((int[])implicants.elementAt(implicant))[literal] == 2 ) ) {
                        // implicant does not imply minterm
                        impliesMinterm = false;
                    }
                }
                checkMap[minterm][implicant] = impliesMinterm;
            }
        }
        // next level
    }
}
break; // We know the minterm is not implied: get out of the comparison.
}
} // We know the minterm is not implied: get out of the comparison.

checkMap[minterm][implicant] = impliesMinterm; // put the check (or no check) into checkMap

Vector essentialPrimes = new Vector(); // essential primes: to be returned
boolean[] mintermsDone = new boolean[minterms.size()]; // what minterms are "circled"
// initialize mintermsDone to all false
for (int initMintermsDone = 0; initMintermsDone < minterms.size(); initMintermsDone++) {
    mintermsDone[initMintermsDone] = false;
}

// Checking columns and taking rows (implicants):
while (!allTermsDone(mintermsDone)) {
    // do columns in order of how many checks they have (least first)
    // find the col. with the least number of checks:
    int minChecksCol = 0;
    int minChecksVal = implicants.size() + 1;
    for (int col = 0; col < minterms.size(); col++) {
        int numChecksInCol = 0;
        if (!mintermsDone[col]) { // don't check any minterms that are circled
            for (int row = 0; row < implicants.size(); row++) {
                if (checkMap[col][row]) {
                    // there is a check. Add 1 to the number of checks in col.
                    numChecksInCol++;
                }
            }
        }
        if (numChecksInCol < minChecksVal) { // There are fewer checks in this column than the current choice.
            minChecksVal = numChecksInCol;
            minChecksCol = col;
        }
    }
    // find row with most additional checks from col minChecksCol
}
int maxChecksRow = 0;
int maxChecksVal = 0;
for (int row = 0; row < implicants.size(); row++) {
    if (checkMap[minChecksCol][row]) {
        // There is a check in this row that is in minChecksCol.
        // Count the number of additional checks in this row.
        int checksInRow = 0;
        for (int col = 0; col < minterms.size(); col++) {
            if (checkMap[col][row] && !mintermsDone[col]) {
                // There is a check, and its minterm is not circled.
                checksInRow++;
            }
        }
        if (checksInRow > maxChecksVal) {
            // There are more additional checks in this row than the current choice.
            maxChecksVal = checksInRow;
            maxChecksRow = row;
        }
    }
}
// Circle all minterms with checks in maxChecksRow:
for (int col = 0; col < minterms.size(); col++) {
    if (checkMap[col][maxChecksRow]) {
        mintermsDone[col] = true;
    }
}
// The implicant at maxChecksRow is essential. Take it:
essentialPrimes.addElement(new int[NUMINPUTS]);
System.arraycopy((int[]) implicants.elementAt(maxChecksRow), 0,
               (int[]) essentialPrimes.elementAt(essentialPrimes.size() - 1), 0, NUMINPUTS);
}
else {
    // There is only one implicant. No elimination is possible.
return implicants;
}

/*
 * allTermsDone checks to see if all the minterms in QM2 are done.
 */
boolean allTermsDone( boolean[] mintermsDoneArray ) {
    for (int ix = 0; ix < NUMTERMS; ix++) {
        if (mintermsDoneArray[ix] == false) {
            return false;
        }
    }
    return true;
}