Plexify Project Reflection

When designing hardware, the traditional method is to build a truth table of all possible input combinations and their corresponding output for each of the output variables present in the system. Then, to get the equivalent simplified Boolean expressions for each output variable, you need to create a Karnaugh map for each individual variable. However, by the nature of needing to account for every possible combination of inputs, the total height of the truth table that one would have to write by hand would double for every additional input variable that was added to the system. Additionally, the size of every Karnaugh map that one would have to draw by hand would double. Because of this, any system that is larger than maybe six inputs becomes incredibly difficult to design. Therefore, I have wanted to build a way to automate this process for a long time.

However, translating this process into a computer-understandable set of instructions was very challenging. Because of this, I developed a different way that the program could perform these operations that I called "Plexify". It started when I was running analysis on a system that has three inputs—a, b, and c—and I had lined up all the input combinations that lead to the desired output "d" evaluating to be true. The cases I was analyzing can be found in the figure below.

a	b	c
0	0	0
0	0	1
0	1	0
1	1	0

If one were to do the traditional method of using a Karnaugh map, the resulting Boolean expression would be $d \equiv (\neg a \wedge \neg b) \vee (b \wedge \neg c)$. However, I then noticed that I could simplify these cases by identifying the cases that are the same with the exception of one bit that is inverted. We will refer to this as the "same except" rule for the rest of this document. In this scenario, the 1st and 2nd case are this way and the 3rd and 4th case are this way. I could then combine them by replacing the bit that differentiates them with a "don't care" bit as this would account for both cases. This means that the previous set of cases can be simplified into the set of cases found in the figure below.

a	b	c
0	0	x
x	1	0

This then results in the same Boolean expression that was gleaned from the Karnaugh map method assuming we are reading these cases as a sum of products expression: $d \equiv (\neg a \wedge \neg b) \vee (b \wedge \neg c)$. This is the algorithm that the first version of Plexify used. Below is the pseudocode of that algorithm.

```
REDUCE(elements): // where elements is the set of cases for a given output variable
    let q be some boolean value
    do:
        q ≔ false;
        for i ∈ [0, elements.length):
            for j ∈ [i + 1, elements.length):
                if elements[i ] is the same as elements[j ] with the exception of bit #k :
                    q ≔ true;
                    elements[i ][k ] ≔ “x”;
                    remove elements[j];
    while q = true;
    return;
```

However, I quickly found out that this did not always work. For example, let's say that the cases that defined "d" were the ones found below.

a	b	c
0	0	1
0	1	1
1	0	1
1	1	0
1	1	1

By using Karnaugh maps, we should get the expression, $d \equiv (a \wedge b) \vee c$. If we used the method mentioned above, in the first iteration, we would merge cases 1 and 2, then 3 and 5. This would leave us with the cases below.

a	b	c
0	x	1
1	x	1
1	1	0

Then, in the second iteration of the algorithm, we could merge cases 1 and 2 to get the cases below.

a	b	c
x	x	1
1	1	0

However, the expression we glean from this would be $d \equiv c \vee (a \wedge b \wedge \neg c)$. And this is not the most simplified version of the equation. This method also assumes that all the cases being dealt with are integral to the overall expression, meaning that we cannot include any cases where the desired output is a "don't care". Because of this, I created a new algorithm that exists in the most current version of Plexify.

Instead of just merging 2 cases that match according to the “same except” rule, we first split the cases into 4 lists. One list contains the cases that result in the desired output being 1 or 0, depending on whether you are looking for a sum of products or product of sums respectively. We will call these the main cases. Another list contains the cases that result in the desired output being a “don’t care”. We will call these the don’t-care cases. Then the last 2 lists will start empty. We will call these lists list 1 and list 2 respectively. Then we will iterate through the main cases and don’t-care cases. If we find any pairs of cases that match according to the “same except” rule, we will create a brand-new case that is equivalent to the result you would get from merging the matched cases. If the 2 matched cases are from the main cases, then we select list 1. Otherwise, we select list 2. We then check to see if this new case is redundant with the cases that are already in the selected list. We verify this by iterating over all the cases currently in the selected list. If a case is found that is equivalent to or accounts for the case that we are trying to insert, we do not insert it. For example, if we are trying to insert the case “00x” and the selected list already contains that case or it contains “0xx”, we do not insert it because “0xx” accounts for “00x”. If no cases like these are found, then we check to see if there are any cases in the selected list that are accounted for by the case we are inserting. If there are, we remove those cases. Then, we insert the case into the selected list.

Once all the cases in both the main cases and don’t-care cases have been iterated through, we will then perform the algorithm again on list 1 and list 2 as the main cases and don’t-care cases respectively. We will do this recursively until no more cases can be generated. Then, at the top of the recursive stack, we will attach the contents of list 1 and 2 onto the main cases and don’t-care cases respectively. Then, we will iterate through the main cases and don’t care cases again and if any redundancies are found (i.e. one case contains another case or 2 cases are the same) then we will remove the redundancy. This is then repeated back down the recursive stack such that when the recursive call returns, lists 1 and 2 will only contain the most simplified cases. Once we reach the bottom of the recursive stack and all the redundant cases have been removed from the main cases and the don’t-care cases, take the main cases in their final form and derive the simplified Boolean expression from that. Below is the pseudocode for this algorithm:

```
TRY_INSERT(case, list):
    for i ∈ [0, list.length):
        if list[i ] = case ∨ list[i ] accounts for case:
            return;
    for i ∈ [0, list.length):
        if case accounts for list[i ] ∧ case ≠ list[i ]:
            remove list[i ]
    list.append(case);
    return;
REDUCE(mainCases, dcCases):
    let list1[];
```

```
let list2[];
let result = false;
for i ∈ [0, mainCases.length + dcCases.length):
    for j ∈ [i + 1, mainCases.length + dcCases.length):
        if i < mainCases.length:
            if j < mainCases.length:
                if mainCases[i ] matches mainCases[j ] except bit #k :
                    q ≔ copy of mainCases[i ];
                    q[k ] ≔ “x”;
                    result ≔ true;
                    TRY_INSERT(q, mainCases);
            else:
                if mainCases[i ] matches dcCases[j ] except bit #k :
                    q ≔ copy of mainCases[i ];
                    q[k ] ≔ “x”;
                    result ≔ true;
                    TRY_INSERT(q, dcCases);
        else:
            if j < mainCases.length:
                if dcCases[i ] matches mainCases[j ] except bit #k :
                    q ≔ copy of mainCases[i ];
                    q[k ] ≔ “x”;
                    result ≔ true;
                    TRY_INSERT(q, dcCases);
            else:
                if dcCases[i ] matches dcCases[j ] except bit #k:
                    q ≔ copy of mainCases[i ];
                    q[k ] ≔ “x”;
                    result ≔ true;
                    TRY_INSERT(q, dcCases);
if result = true:
    REDUCE(list1, list2);
    while list1 is not empty:
        mainCases.prepend(list1.pop());
    while list2 is not empty:
        dcCases.prepend(list2.pop());
for i ∈ [0, mainCases.length + dcCases.length):
    for j ∈ [i + 1, mainCases.length + dcCases.length):
        if i < mainCases.length:
            if j < mainCases.length:
                if mainCases[i ] accounts for mainCases[j] ∨ mainCases[i] = mainCases[j]:
                    remove mainCases[j];
```

```
                else if mainCases[j] accounts for mainCases[i]:
                    remove mainCases[i];
                    insert mainCases[j] at index i;
            else:
                if mainCases[i] accounts for dcCases[j] ∨ mainCases[i] = dcCases[j]:
                    remove dcCases[j];
                else if dcCases[j] accounts for mainCases[i]:
                    remove mainCases[i];
                    insert dcCases[j] into mainCases at index i;
        else:
            if j < mainCases.length:
                if dcCases[i] accounts for mainCases[j] ∨ dcCases[i] = mainCases[j]:
                    remove mainCases[j];
                else if mainCases[j] accounts for dcCases[i]:
                    remove dcCases[i];
                    insert mainCases[j] into dcCases at index i;
            else:
                if dcCases[i] accounts for dcCases[j] ∨ dcCases[i] = dcCases[j]:
                    remove dcCases[j];
                else if dcCases[j] accounts for dcCases[i]:
                    remove dcCases[i];
                    insert dcCases[j] at index i;
    return;
```

To see this algorithm in action, we can look at the cases below for the variable "d" where we are trying to find the simplified sum of products expression.

a	b	c	d
0	0	1	1
0	1	1	1
1	0	1	x
1	1	0	1
1	1	1	x

In this scenario, if we are to take the don't care bits into account and use the Karnaugh map method, we should get the same Boolean expression that we got in the last example: $d = (a \wedge b) \vee c$. To start, we will break these cases up into their 2 constituent groups—main cases and don't-care cases—as well as create the 2 extra empty lists.

main cases

a	b	c
0	0	1
0	1	1
1	1	0

a	b	c

don't-care case

1	0	1
1	1	1

list 1

a	b	c

a	b	c

list 2

We then find all pairs of cases that match according to the "same except" rule in the main and don't-care cases and place them in list 1 or 2.

001 = 011 except bit 2 → 001 and 011 in main → place 0x1 in list 1
110 = 111 except bit 3 → 110 in main and 111 in don't-care → place 11x in list 2
101 = 111 except bit 2 → 101 and 111 in don't-care → place 1x1 in list 2

Below are the resulting tables:

a	b	c
0	0	1
0	1	1
1	1	0

main cases

a	b	c
1	0	1
1	1	1

don't-care case

a	b	c
0	x	1

list 1

a	b	c
1	1	x
1	x	1

list 2

We then perform the same process on list 1 and list 2 as our main cases and don't care cases respectively. We begin by creating 2 more empty lists we will call list 1' and list 2'.

a	b	c
0	0	1
0	1	1
1	1	0

main cases

a	b	c
1	0	1
1	1	1

don't-care case

a	b	c
0	x	1

list 1

a	b	c
1	1	x
1	x	1

list 2

a	b	c

list 1'

a	b	c

list 2'

We then find all pairs of cases that match according to the "same except" rule in list 1 and 2 and place them in list 1' or list 2'.

0x1 = 1x1 except bit 1 → 0x1 in list 1 and 1x1 in list 2 → place xx1 in list 2'

Below are the resulting tables:

a	b	c
0	0	1
0	1	1
1	1	0

main cases

a	b	c
1	0	1
1	1	1

don't-care case

a	b	c
0	x	1

list 1

a	b	c
1	1	x
1	x	1

list 2

a	b	c

list 1'

a	b	c
x	x	1

list 2'

We then iterate through all the cases in list 1' and list 2' and we find no pairs of cases that match according to the "same except" rule. This means we stop the recursion. We then prepend any simplified cases we got from list 1' and list 2' back onto list 1 and list 2 respectively.

a	b	c
0	0	1
0	1	1
1	1	0

main cases

a	b	c
1	0	1
1	1	1

don't-care case

a	b	c
0	x	1

list 1

a	b	c
x	x	1
1	1	x
1	x	1

list 2

We then remove all redundant cases from list 1 and list 2.

0x1 is accounted for by xx1 → 0x1 in list 1 and xx1 in list 2 → replace 0x1 with xx1
xx1 accounts for 1x1 → xx1 in list 1 and 1x1 in list 2 → remove 1x1

Below are the resulting tables:

a	b	c
0	0	1
0	1	1
1	1	0

main cases

a	b	c
1	0	1
1	1	1

don't-care case

a	b	c
x	x	1

list 1

a	b	c
1	1	x

list 2

We then prepend the simplified cases from list 1 and list 2 to back onto the main cases and don't-care cases respectively.

a	b	c
x	x	1
0	0	1
0	1	1
1	1	0

main cases

a	b	c
1	1	x
1	0	1
1	1	1

don't-care cases

We then remove all redundant cases from both the main cases and the don't-care cases.

xx1 accounts for 001 → xx1 and 001 in main → remove 001
xx1 accounts for 011 → xx1 and 011 in main → remove 011
xx1 accounts for 101 → xx1 in main and 101 in don't-care → remove 101
xx1 accounts for 111 → xx1 in main, 111 in don't-care → remove 111
110 accounted for by 11x → 110 in main and 11x in don't-care → replace 110 with 11x

Below are the resulting tables:

a	b	c
1	1	x
x	x	1

main cases

a	b	c

don't-care cases

We now finally take the main cases as our final simplified form. If we write out these cases as our solution, then we get $d = (a \wedge b) \vee c$. Which we confirmed earlier is the correct solution.

My initial goal in creating this program was to make hardware design easier when dealing with systems with a large number of inputs and outputs. However, I eventually hope to apply this system to other areas of computer engineering. I have built in a lot of the baseline software to expand the project to be able to use ternary and exclusive or operators to simplify the logic of a Boolean expression as well as manipulate the expression in other ways at the user's discretion. Eventually this could be used to simplify large logic trees in code as well as more complex decision making done by software.