Combinatorial Completion by Rule Definition with Interactive Value Colouring[*]

Michael Breen

2005-8

Abstract

The combinatorial completion problem arises where one wishes to define a set of rules which collectively address all possible combinations of circumstances, as, for example, in a decision table. After some rules have been defined but, e.g., millions of cases remain to be addressed, how can the specifier best be helped to complete the remaining rules so that they cover every possible scenario? A novel technique is described based on providing interactive feedback to the user during rule definition. This can be used with non-tabular as well as tabular rules. Where previously quality would have depended on essentially sample-based approaches like testing, this technique makes it easy to adopt a rigorously complete approach to considering large numbers of possibilities.

1 The Combinatorial Completion Problem

Table 1. shows, in the form of a decision table, a set of rules specifying an activity to try depending on the outdoor conditions. Every possibility to consider corresponds to some combination of values of the variables `Weather`, `Temp` (temperature), and `Wind`. Rule A, for instance, indicates windsurfing to be a suitable activity in four different combinations, including the combination in which `Temp` is `warm`, `Weather` is `fine`, and `Wind` is `breeze`. The symbol '*' means "don't care"; this is equivalent to a list of all possible values of the corresponding variable.

[*]A shorter version of this article appeared in ACM SIGSOFT Software Engineering Notes (March 2005).

Table 1: An example of a decision table.

Rule	Temp	Weather	Wind	Action
A	hot warm	fine rain	breeze	windsurf
B	cold warm	fine rain	calm	hillwalk
C	hot	fine	calm	sunbathe
D	*	*	gale	read
E	*	snow	breeze calm	ski

Whether in some kind of decision table or otherwise, a frequent objective in defining a set of rules is to cover every possibility, that is, for every possible combination of variable values, to have at least one rule which specifies an outcome for that case. In fact, Table 1 is not complete: there are some cases not covered by any rule. Though this is not generally obvious by inspection, it is a property easily checked in software.

However, the problem of assisting a human being to complete such a set of rules is much less straightforward. This problem, herein called the combinatorial completion problem, has been described and addressed in various ways previously.

2 Previous Approaches

The most obvious solution is simply to have the software list all the uncovered combinations. This is the approach taken by some decision table tools and indeed it can work reasonably well where there are not many combinations to consider. For example, Table 2 contains the combinations not covered by any rule in Table 1.

Table 2: Cases not covered in Table 1.

Temp	Weather	Wind
cold	fine	breeze
cold	fog	breeze
cold	fog	calm
cold	rain	breeze
hot	fog	breeze
hot	fog	calm
hot	rain	calm
warm	fog	breeze
warm	fog	calm

However, if there are thousands of possible combinations then the obvious solution is not very helpful: in theory, the user could add an action or outcome to each combination listed in order to create a new rule for each – but at the cost of having to define thousands of rules. To create rules like those in Table 1, the user would need to sift through the many individual combinations and find those with identical outcomes which also may be grouped together to be covered by a single rule in the conjunctive normal form (CNF) of the table.

An improvement over listing individual combinations is to have a software tool itself arrange the unconsidered combinations into groups. This is the approach taken in a tool called SAST [1]; the result is a list of CNF conditions, somewhat like the rows of Table 1 without the last column. Ideally, the user would then merely need to add an outcome to each condition to order to complete the set of rules. However, there is no reason to suppose that the desired outcome will be the same for all of the combinations in each group - since the program can have no knowledge of the outcomes until the rules are completed. For example, had the tool output the same condition as rule B in Table 1, we might well create a new rule from it by adding the action "hillwalk"; however, we might instead decide that hillwalking is a good idea in, say, three of the four cases covered but is not desirable when it's both cold and raining. Thus, we must break up the condition provided by the tool across a number of separate rules and we end up having to define a more numerous and fragmented set of rules than might otherwise have been arranged.

Of course, one way of managing this problem is to re-organize the fragmented conditions into an equivalent smaller set of conditions suitable to be made into rules. Unfortunately, this becomes quite difficult in non-trivial cases, where there are many rules or many variables. Imagine just two variables, the first with possible values {A, B, C} and the second, {P, Q, R} and consider the following conditions in abbreviated form:

(A, PQ)

(BC, P)

(AB, R)

(C, QR)

– where, e.g., (A, PQ) means that the first variable has the value A and the second is either P or Q. Provided the outcomes in the appropriate cases are the same, two rules would be sufficient to cover the same set of combinations:

(AC, PQR)

(B, PR)

However, it is not immediately apparent that the cases covered by the two sets of rules are the same, even for this very limited example. In this case, the simplest way to find the more concise arrangement is to draw a grid with the possible values of the variables on the axes; but this isn't possible if there are, say, ten variables.

A variation on the above approach would be merely to fragment the conditions output by the program as necessary, add the outcomes to form rules, and then let software combine rules with the same outcome as best it can. In fact, this task is computationally as well as manually difficult. However, the problem from a user's point of view is that this doesn't avoid the labour of splitting up the conditions and defining a larger set of rules in the first place.

A variation on the idea of presenting pre-formed conditions to the user is described by Seagle and Duchessi [2]. The difference in their approach is that the conditions output by the program, while still covering all the unconsidered combinations and no others, are not required to be mutually exclusive. This means that the program is less restricted in deriving conditions and so the user is more likely to be able to find a condition covering a reasonable number of cases for which the outcome is the same. Each time the user forms a new rule, the list of incomplete rules is updated so that the final set of rules are still mutually exclusive.

However, the user may need to look through many pre-formed conditions before finding one to which an outcome can be added, especially if there are many possible outcomes and many uncovered combinations. Also, because the incomplete rules generated either restrict each variable to a single value or allow it to be "don't care," it is less useful with non-boolean variables – since no condition in which a variable is restricted to some proper, non-unitary subset of its values will ever be available for selection (which would exclude all of the conditions listed in the two alternative sets from the example above): any potential rule with such a condition will therefore still be replaced by many separate rules. This difficulty is not easily fixed: to let the program to produce conditions containing proper subsets of two or more values would lead to an exponential increase in the number of conditions presented to the user, only very few of which one would expect to be able to form into rules.

The reports [3] and [4], describe a completely different approach to the problem:[1] the computer performs "structural analysis" on a decision table, the results of which give the user an indication of how to complete the table. For example, single-variable structural analysis might show that, in every rule, a given boolean variable appears only either restricted to the value "true" or else unrestricted ("don't care"). A little thought confirms that, provided the combinations covered by the rules are mutually exclusive, this implies that the table is incomplete and that it can be completed either by adding rules in which that variable is restricted to the value "false" or by changing to "don't care" those which cover only the combinations where it is "true." Structural analysis over more than one variable produces more complex results. An advantage of this approach is that the user has the choice of modifying existing rules where this will suffice rather than simply adding new ones. However, the user must still manually consider the structural analysis results in conjunction with the existing rules, a task which increases in difficulty as the rules already defined increase in complexity and number.

3 Factors to Consider

A few simple and related observations are relevant to an alternative approach to the combinatorial completion problem. First, in general, the number of rules required to cover all combinations reduces when one allows rules with the same outcome to overlap, that is, to cover some of the same combinations. For example, if the outcome is the same for the combinations marked with an 'X' in Figure 1 then two rules with the non-mutually exclusive conditions (AB, PQ) and (BC, QR) are sufficient to cover them; otherwise, three rules are needed. Note that we are assuming, for now, that rule conditions may be expressed only in CNF: clearly, a single rule would suffice if its condition were expressed as a general formula.

In certain contexts, such as hardware design, this

[1]The solution described in the two reports is essentially the same, though, confusingly, the convention in one is to apply a logical inversion not present in the other.

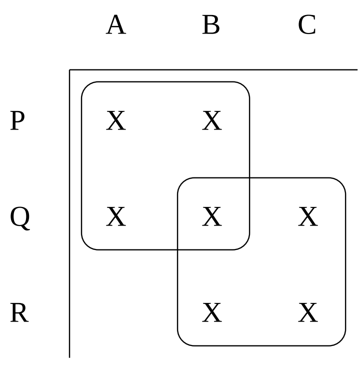


Figure 1: Minimizing number of rules by allowing overlaps.

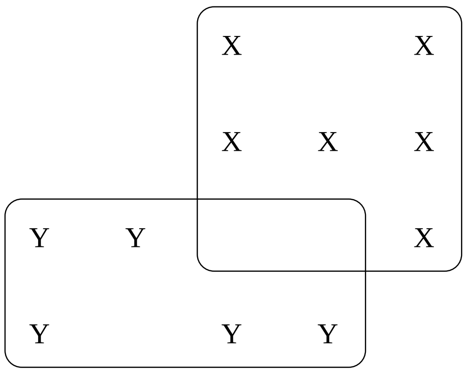


Figure 2: Rules with different outcomes overlapping in impossible cases.

property is often exploited but in others, including most uses of decision tables, it is usual for all rules to be mutually exclusive. Exclusivity does have the merit of ensuring that, to change the outcome for a specific case, one never needs to change more than one rule. However, if a change affects several combinations then it may involve changing several rules anyway – conceivably, even more than one would need to change had the rules been allowed to overlap. Further, as will be seen below, it can be an advantage to allow overlaps at least during the process of rule definition, even if one intends ultimately to make all the rules mutually exclusive.

The second point concerns impossible cases. If certain combinations of values are impossible, then it shouldn't matter if some rules specify a particular outcome for these cases. For example, Rule D in Table 1 covers combinations in which `Temp` is `hot` and `Weather` is `snow`, which we might define to be impossible; if this were not permitted then Rule D would need to be replaced with two separate rules for what to do in a gale, one of which would cover the single combination (`cold`, `snow`, `gale`). In fact, rules with different outcomes may even be allowed to overlap, provided the combinations covered by both are impossible. Such a situation is illustrated in Figure 2, where rules for outcomes X and Y overlap in the impossible cases represented by blanks.

Allowing rules to include impossible combinations is especially useful when many combinations are impossible. This may arise, for example, with variables representing different aspects of a finite state machine subject to many invariants. In such a case, though there may be an enormous number of possible combinations (i.e., possible states) these may be far outnumbered by those which are impossible. Allowing impossible combinations to pepper those covered by the rules then means much fewer rules are necessary.

Note that, for a software tool to allow rules to include impossible combinations without reporting an inconsistency, the impossible combinations cannot be regarded merely as cases with a different outcome from the others. Thus, this approach is incompatible with that sometimes seen used with decision tables, where rules (with a special "impossible" outcome) are included for impossible cases to avoid any incompleteness error (and thereby confirm that all the possible cases have been covered too); an approach which also suffers from the disadvantage of mixing up rules which are qualitively different, with some stating an assumed or static property and others a requirement or a dynamic property. If such an approach is combined with a convention of not allowing even rules with the same outcome to overlap then the number of rules required may multiply enormously. In the worst case, one is also dealing with a relatively sparse space of possible combinations so that the special "impossible" rules far outnumber those for the possible cases.

Taking the above into account, the approach that follows is designed for a situation where any cases which are impossible are defined as such, separately from the rules. This may be done in any way, for example, as a set of formulas defining the invariants of a finite state machine model, or using a "constraint table" (the method supported by Statestep, the same tool which implements the interface described below); exactly how the impossible cases, if any, are defined is not of further concern here.

The approach does not, of course, prevent one from specifying the impossible combinations using special rules instead. Similarly, it is required neither that rules be allowed to overlap nor that there be impossible combinations; it is merely that the relative strengths of the approach are greatest when these properties are exploited. Finally, as we will see later, though the interface is designed for rules with conditions expressed in CNF, this also is not an essential requirement; indeed, the tool which implements the solution below supports rules with conditions expressed as formulas.

4 A New Approach

To restate the problem, we seek a way to help a person to complete a set of rules which collectively define an outcome in every possible combination of some set of circumstances. We are not concerned with rule, or term, minimization per se, but rather with a user interface and a heuristic approach wherein reducing the number of rules is part of the way in which the effort required of the user is minimized.

The nub of the problem is that the assistance any software tool can give a user is limited by the fact that it does not know the outcomes of the rules yet to be defined. To address this, the approach is based on providing an interactive dialogue between user and computer *during* rule definition, so that the user's knowledge of the desired outcomes converges with a suitable set of missing cases to form each new rule.

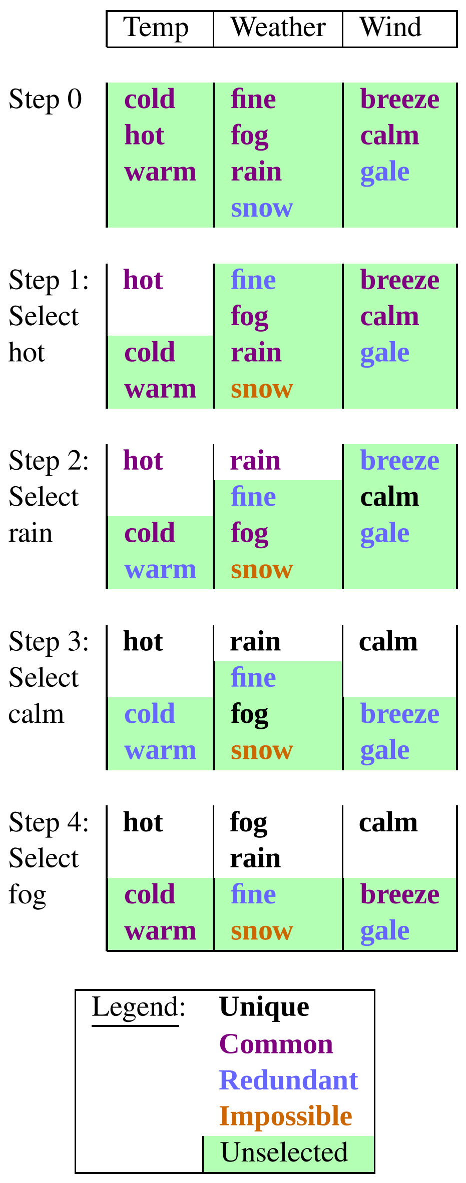
This is most easily introduced by way of example. Assume we have defined the set of rules in Table 1. We now wish to continue adding rules until all possible combinations of `Temp` , `Weather` and `Wind` are covered by some rule. Assume that we have specified that certain combinations of variable values are impossible. In particular, suppose we have specified that `Weather` cannot be `snow` unless `Temp` is `cold` .

For the following description, the sequence of interactions we might go through in choosing the condition of the new rule is shown, with the help of colour coding, in Figure 3. For convenience, Figure 4 conveys the same information but without the use of colour.

At Step 0, we have not yet begun to specify a condition for our new rule. However, the program we are using displays the values we might select in the rule's CNF condition together with summary information about each. In this case, it tells us that the values `snow` and `gale` are *redundant*, that is, there is no point in selecting either value as all the combinations in which they appear are already covered by some rule. For example, every possible combination in which `Weather` is `snow` is covered by Rules D and E of Table 1.

All the other values are *common*, that is, there are combinations in which they occur that are not covered by any rule – but these values also occur in combinations for which rules do exist. For example, the value `fine` for `Weather` is common: when it's fine, Rule A tells us what to do if it's there's a breeze and it's warm; but no activity is specified for when it's breezy and cold.

Step 1: We begin by (perhaps arbitrarily) selecting `hot` . Now, in classifying the values of `Weather` and `Wind` , the program is considering only those uncovered combinations in which `Temp` is `hot` . Thus, `snow` has become *impossible*. Also



	Temp	Weather	Wind
Step 0	**cold**	**fine**	**breeze**
	hot	**fog**	**calm**
	warm	**rain**	**gale**
		snow	
Step 1:	**hot**	**fine**	**breeze**
Select		**fog**	**calm**
hot	**cold**	**rain**	**gale**
	warm	**snow**	
Step 2:	**hot**	**rain**	**breeze**
Select		**fine**	**calm**
rain	**cold**	**fog**	**gale**
	warm	**snow**	
Step 3:	**hot**	**rain**	**calm**
Select		**fine**	
calm	**cold**	**fog**	**breeze**
	warm	**snow**	**gale**
Step 4:	**hot**	**fog**	**calm**
Select		**rain**	
fog	**cold**	**fine**	**breeze**
	warm	**snow**	**gale**

Legend:	
	Unique
	Common
	Redundant
	Impossible
	🟩 Unselected

Figure 3: Choosing a new rule condition.

	Temp	Weather	Wind
Step 0	*(cold)*	*(fine)*	*(breeze)*
	(hot)	*(fog)*	*(calm)*
	(warm)	*(rain)*	*[gale]*
		[snow]	
Step 1:	**(hot)**	*[fine]*	*(breeze)*
Select		*(fog)*	*(calm)*
hot	*(cold)*	*(rain)*	*[gale]*
	(warm)	*{snow}*	
Step 2:	**(hot)**	**(rain)**	*[breeze]*
Select		*[fine]*	*calm*
rain	*(cold)*	*(fog)*	*[gale]*
	[warm]	*{snow}*	
Step 3:	**hot**	**rain**	**calm**
Select		*[fine]*	
calm	*[cold]*	*fog*	*[breeze]*
	[warm]	*{snow}*	*[gale]*
Step 4:	**hot**	**fog**	**calm**
Select		**rain**	
fog	*(cold)*	*[fine]*	*(breeze)*
	(warm)	*{snow}*	*[gale]*

Legend:	
	Unique
	(Common)
	[Redundant]
	{**Impossible**}
	Unselected

Figure 4: Non-colour version of Figure 3.

`fine` has become redundant: there is no combination in which `Temp` is `hot` and `Weather` is `fine` that is not covered by Rules A, C, or D.

Step 2: Although our example is very small, let's imagine we've already defined so many rules that it's difficult to keep track of them. Seeing the remaining value classifications, we therefore form the intention of selecting `rain` and `breeze` for the remaining parts of the condition and specifying windsurfing as the activity. However, when we select `rain` for `Weather`, the value `breeze` becomes redundant: the combination `hot`, `rain`, and `breeze` is already covered by Rule A. Further, the value `calm` is now *unique*: if this value is selected then the new rule will not cover any combination in which `Wind` is `calm` that's already been covered.

Step 3: We could at this point change the values selected for either `Temp` or `Weather`. However, the easiest thing is to change tack: Instead of a windsurfing rule, we decide to define one for when to read a book and continue by selecting `calm`. By now, all the remaining unselected values are either redundant or impossible – except for `fog`.

Step 4: Since reading a book is also appropriate for when it's hot, foggy, and calm, we add `fog` to the selected values for `Weather`. Some of the other values again become common, for example, `breeze` – since there's no rule for the combination `hot`, `fog`, and `breeze` (which, arguably, should be impossible). However, there's no point in selecting it – not because the other combination it would cause the rule to cover (`hot`, `rain`, `breeze`) is already covered by an existing rule, but because the activity for that case is not to read a book, but to go windsurfing. The same consideration applies in respect of the other unselected values which are common; we have therefore finished defining the condition for our new rule.

The preceding example includes an instance of each of the four possible classifications of variable values that are employed in the approach; Figure 5 shows more formally how any value in a rule, whether selected in a CNF condition or not, may be classified as unique, common, redundant, or impossible. Generally, neither redundant nor impossible values will be selected but the distinction between the two is useful; for example, one may wonder why a value is impossible, which may lead to a mistake being discovered in the definition of the impossible combinations.

4.1 Tolerating Overlaps

Earlier, we saw why it was convenient to allow rules to overlap. It was also claimed that, even when one prefers to make all rules mutually exclusive, it is useful to allow new rules to overlap existing ones and to defer dealing with the overlaps until all the rules have been entered.

To see this, suppose we call the rule defined in the previous example Rule F (that is, with the condition shown at the end of Figures 3 and 4) and we add another rule, also with the outcome "read", called Rule G. The condition for Rule G is displayed by the program as shown in Figure 6. Note that we have included common as well as unique values among those selected in the rule condition: our tactic is to include these freely as long as all the combinations covered by the rule have the same outcome; the alternative to doing this is to define more than one rule at this point to cover the same set of combinations.

Having defined Rule G in this way, we may later return to review the rules, at which point we find that Rule F is now displayed as shown in Figure 7: the value `fog` has become redundant in Rule F because of the overlap with Rule G. If Rule F is now changed to exclude `fog`, that is, if only `rain` is selected for `Weather`, the overlap is eliminated and the values in each rule become unique. The point to remember is, had we not allowed Rule G to contain common as well as unique values, we would have ended up defining more than these two rules to cover the same set of combinations.

Of course, the above example worked out very neatly partly because it is necessarily restricted in size and consequently relatively simple. In gen-

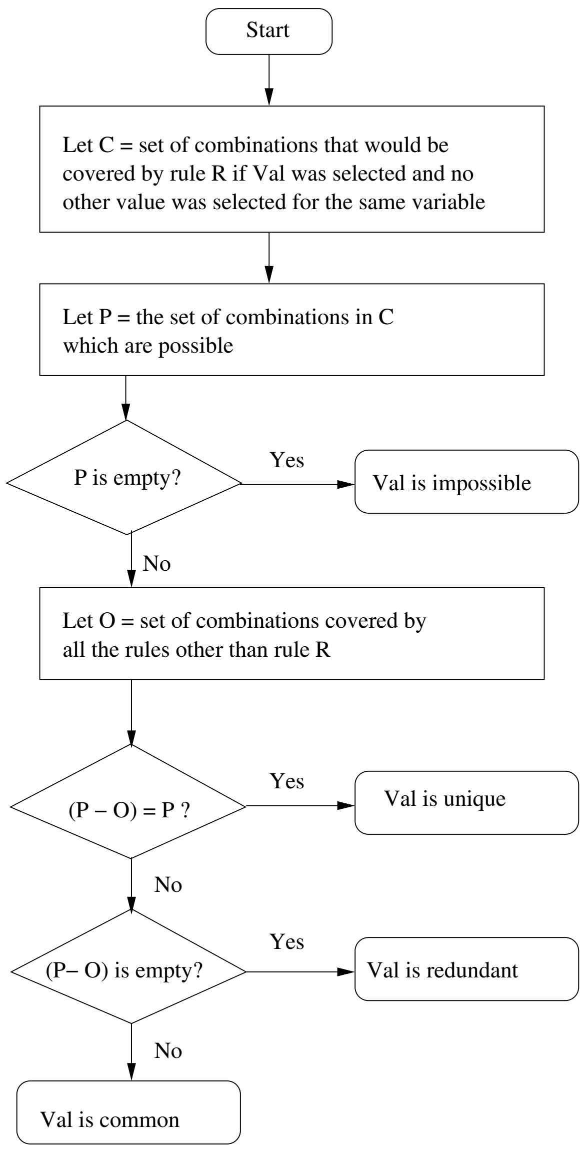

Start
Let C = set of combinations that would be covered by rule R if Val was selected and no other value was selected for the same variable
Let P = the set of combinations in C which are possible
P is empty?
Yes
Val is impossible
No
Let O = set of combinations covered by all the rules other than rule R
(P – O) = P ?
Yes
Val is unique
No
(P– O) is empty?
Yes
Val is redundant
No
Val is common

Figure 5: Classification of a variable value (Val) for the CNF condition of a rule (R) in a set of rules.

Temp	Weather	Wind
cold	**fog**	**breeze**
hot	**fine**	**calm**
warm	**rain**	
	snow	**gale**

(Non-colour version:)

Temp	Weather	Wind
cold	**(fog)**	**breeze**
(hot)	*(fine)*	**(calm)**
warm	*(rain)*	
	[snow]	*[gale]*

Figure 6: Condition for Rule G as displayed.

Temp	Weather	Wind
hot	**fog**	**calm**
	rain	
cold	**fine**	**breeze**
warm	**snow**	**gale**

(Non-colour version:)

Temp	Weather	Wind
(hot)	**[fog]**	**(calm)**
	rain	
[cold]	*[fine]*	*[breeze]*
[warm]	*{snow}*	*[gale]*

Figure 7: Condition for Rule F as displayed after Rule G has been defined.

eral, even after removing the newly redundant values when the rules had been completed, further changes would still be needed to remove all the overlaps, involving the break-up of some rules into “smaller” rules. However, once again, the argument for not removing overlaps after all the rules have been defined is a fairly strong one, especially given that a tool can distinguish between overlaps, where the rules involved specify the same outcome, and conflicts, where the outcomes are different.

4.2 Non-CNF Rule Conditions

In principle, the approach may be used even where rule conditions are not expressed in conjunctive normal form. The tool supporting the approach allows each rule condition to be specified either in tabular form or as a formula – or a combination of the two, wherein the overall condition is the logical conjunction of the table and formula elements.

Consider Figure 8 as an example of a condition for a rule that might be added to those in Table 1: the condition is expressed here only by a formula; though none of the values have been selected, they are still classified with respect to this rule and the others according to the same scheme. The values `breeze` and `gale` for `Wind` are impossible simply because they are excluded by the formula; that `snow` is impossible is derived as before from the fact that this is inconsistent with “Temp = hot” and (obviously) with the second part of the disjunction also. Note that the value `fine` is redundant: thus, we can see immediately that there is no point in retaining the “OR Weather = fine” part of the formula.

Given that the information displayed is essentially a slice of data in CNF which to, a varying extent, will be incongruent with the structure of formulas used to define rule conditions, the approach can be expected to be less effective when used in this context. However, what is displayed can still be quite helpful. Also, this facility means that one may use CNF in most cases, keeping the freedom to use a formula in any situation where CNF would be unsuitable and inefficient.

Temp	Weather	Wind
cold	**fine**	**breeze**
hot	**fog**	**calm**
warm	**rain**	**gale**
	snow	
(Temp = hot OR Weather = fine) AND Wind = calm		

(Non-colour version:)

[cold]	*[fine]*	*{breeze}*
(hot)	*fog*	*(calm)*
[warm]	*rain*	*{gale}*
	{snow}	
(Temp = hot OR Weather = fine) AND Wind = calm		

Figure 8: Classification of values for a condition given as a formula.

5 Implementation

The example above is necessarily somewhat artificial. While it is easy to theorize about an approach to combinatorial completion problem, the only proper way to evaluate them is to test it out on realistic problems and develop an intuition for its operation in practice. The approach described here has been implemented in a tool called Statestep, which is available from the author. Although it is primarily designed to support a specification method based on the creation of a finite state machine model [5], Statestep may also be used for ordinary decision tables.

The non-polynomial computational complexity of the underlying analysis which must be performed – and the fact that this must be carried out at speeds allowing interactive updating of colours as a user a edits a rule – may be expected to restrict the size of models to which the approach can be applied in practice. However, when using symbolic algorithms on a reasonably modern PC, experience to date suggests that a desire to restrict the variables to a number that will fit across the width of a printed page is more likely to be the limiting factor. A detailed discussion of algorithm and performance issues is however beyond the scope of this article and so is deferred to a separate account.

6 Conclusion

This article introduces a powerful new way of dealing with the combinatorial completion problem, one which allows interactive navigation of very large sets of combinations. It differs from the closest comparable techniques in that feedback is provided during rule definition and continually updated; further, the user generally needs to consider only a fixed, limited volume of information rather than a long list of existing rules or uncovered combinations. Thus, one can easily adopt a systematic and exhaustive approach to considering possible scenarios, even in cases where such an idea might typically be dismissed as infeasible.

References

[1] "SAST User Manual," ATC-NY (previously Odyssey Research Associates), Cornell Business & Technology Park, 33 Thornwood Drive, Suite 500, Ithaca, NY 14850-1250, U.S.A., Tech. Rep. Version 0.2, 1993.

[2] J. P. Seagle and P. Duchessi, "Acquiring expert rules with the aid of decision tables," *European Journal of Operational Research*, vol. 84, pp. 150–162, 1995.

[3] D. N. Hoover and Z. Chen, "Tbell: A mathematical tool for analyzing decision tables," NASA Langley, Hampton, Virginia, Tech. Rep. Contractor Report 195027, 1994.

[4] D. Hoover and Z. Chen, "Tablewise, a decision table tool," in *Proc. of the 10th Annual Conference on Computer Assurance (COMPASS '95)*. Gaithersburg, MD: IEEE, June 1995.

[5] M. Breen, "Experience of using a lightweight formal specification method for a commercial embedded system product line," *Requirements Engineering Journal*, vol. 10, pp. 161–172, 2005.