Automatically generating information from a Z specification to support the Classification Tree Method

R. M. Hierons[1], M. Harman[1], and H. Singh[2]

[1] Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK
[2] DaimlerChrysler AG, Research Information and Communication Software Technology Research Lab (RIC/SM), Alt-Moabit 96a, D-10559 Berlin, Germany

Abstract. The Classification Tree Method provides a flexible basis for systematic testing. Traditionally the generation of a classification tree has been entirely manual. This paper introduces a new approach that extracts predicates from the Z specification of an operation and builds a Classification Tree from these predicates. It thus shows how the generation of a Classification Tree may be semi-automated on the basis of a Z specification. The paper also defines the notion of the test context of a predicate that determines when the value of this predicate is relevant. The test context is used to reduce the number of tests produced from the Classification Tree.
Keywords: Classification tree method, test automation, formal methods, Z notation.

1 Introduction

Formal Methods and Software Testing are two important approaches that aim to improve software quality. While these approaches were seen as rivals, they are now widely seen as complementary [6, 13]. Importantly the presence of a formal specification allows the automatic or semi-automatic generation of information that may assist testing. The presence of a formal specification may help reduce the cost of testing while making testing more systematic.

The Classification Tree Method [2, 3, 7, 17] involves choosing a set of *classification aspects*, each of which represents factors that the tester believes to be important. The tester then chooses a set of ranges of values, called *classes*, for each classification aspect. The set of classes corresponding to a classification aspect represent a partition of the values the classification aspect may take. Typically, all values within a class are considered, by the tester, to be equivalent. The classification aspects and classes are visually represented as a tree. Abstract test cases are produced from this classification tree, each abstract test case being defined by a set of classes. An abstract test case thus represents a constraint on the test input and equivalently a set of test cases. Concrete test cases are produced from each abstract test case. By allowing the tester to choose ways in which the classes are combined, the Classification Tree Method allows the tester to apply their expertise and domain knowledge. A further advantage of the Classification Tree Method is that it is supported by a tool [7]. The Classification Tree Method and previous work on testing from a Z specification will be briefly reviewed in Section 2.

A number of authors have considered the problem of automating test generation on the basis of a specification written in a model based notation such as Z or VDM [1, 4, 5, 8, 9, 15, 18]. One of the earliest, and most important, approaches to test automation is the *DNF method* [5]. Here the specification is rewritten to Disjunctive Normal Form and a partition of the input domain is formed from the preconditions of the disjuncts. Test cases are drawn from each subdomain of the partition.

While the DNF approach has many advantages, it may lead to a combinatorial explosion. Where the application of the DNF approach is impractical, an alternative is to produce a classification tree from the specification and combine classes from this tree to form abstract test cases [17]. While the use of all combinations of classes might reproduce the combinatorial explosion found in the DNF approach, the tester may choose to use only some of the combinations. This paper extends the work of [17] to show how information, that supports the application of the Classification Tree Method, may be automatically derived from a Z specification.

This paper makes the following contributions. First, Section 3 shows how classification aspects and classes may be automatically generated from the Z specification of an operation in order to assist the production of the classification tree. The analysis described in Section 3 is semantic in nature and typically will be expensive to apply. However, Section 4 introduces an efficient but approximate syntactic analysis. Section 5 shows how the test effort may be reduced by choosing appropriate abstract test cases. Specifically, information contained in the specification is used to automatically limit the combinations of classes used in testing. Finally, in Section 6 conclusions are drawn.

2 Preliminaries

Throughout this paper S will denote the body of a schema that specifies the operation under test. In order to simplify the exposition it will be assumed that S is in the form of a sequence of quantifiers followed by a quantifier free predicate S^* such that: S^* contains no logical operators other than $\vee$, $\wedge$ and $\neg$; and, in S^*, $\neg$ is only applied to atomic predicates. This form is called *Prenex Normal Form (PNF)*. Any predicate can be rewritten to PNF using standard rules. Note that the transformation to PNF may be preceded by the application of transformation steps. For example, $x \in \{y \in \mathbb{Z} \mid y > 0\}$ might be rewritten to $x > 0$.

2.1 Testing from a Z specification

This section will briefly describe the DNF approach [5] to generating tests from a Z or VDM specification of an operation.

The example in Figure 1 will be used as a running example throughout the paper. Here Q represents the rational numbers. This specification specifies a system that determines the cost of a rail ticket. The price depends upon the following factors: the start and destination of the journey; whether it is being booked more than a week in advance; whether the journey time is peak or off-peak; whether the customer has a railcard; and, if the person has a railcard, whether it is a student railcard or a pensioner's railcard. To simplify the specification it will be assumed that only the dates are considered when determining how far in advance the ticket is being booked. It will also be assumed that there are functions, $price\_peak$ and $price\_off\_p$, that give the basic cost of tickets during the peak and off-peak periods respectively. Railcards provide a discount as long as the journey is off-peak. The discount provided by booking more than a week in advance depends on whether the journey is peak or off-peak (10% if peak, 40% if off-peak).

Within this specification, RAILCARD denotes the types of railcard (STUDENT, OAP, and NONE), STATIONS denotes the set of stations, DATE denotes the set of dates and PERIOD denotes the travel period (peak and off-peak).

$$
\begin{array}{|l}
\hline
\textit{Price} \\
r? : RAILCARD \\
start?, end? : STATIONS \\
current\_date?, outward\_date? : DATE \\
travel\_period? : PERIOD \\
price! : Q \\
\hline
outward\_date? \geq current\_date? \\
(travel\_period? = peak \land \\
(outward\_date? - current\_date? > 7 \\
\land price! = 0.9 * price\_peak(start?, end?) \\
\lor outward\_date? - current\_date? \leq 7 \\
\land price! = price\_peak(start?, end?)) \\
) \lor \\
(travel\_period? = off\_peak \land \exists\, p1, p2 : Q \bullet \\
(r? = NONE \land p1 = 1 \lor r? = STUDENT \land \\
p1 = 0.9 \lor r? = OAP \land p1 = 0.7) \land \\
(outward\_date? - current\_date? > 7 \land p2 = 0.6 \lor \\
outward\_date? - current\_date? \leq 7 \land p2 = 1) \land \\
price! = p1 * p2 * price\_off\_p(start?, end?)) \\
\hline
\end{array}
$$

Fig. 1. The specification of *Price*

It is easy to confirm that *Price* may be rewritten to PNF by simply moving the quantifiers to the front.

The DNF approach considers the specification of an operation and generates a partition $P = \{D_1, \ldots, D_n\}$ of the input domain of that operation. The partition generated has the property that, according to the specification, all values in a subdomain should be treated in the same manner. Note that the input domain for an operation is formed from both the input received from the environment and the internal state space. Tests are produced from each subdomain.

Given a specification S of an operation op, the DNF approach initially involves rewriting S to DNF. Suppose this produces disjuncts $C_1, \ldots, C_m$. Then the precondition $\mathrm{pre}(C_i)$ of each C_i is found. This produces m predicates that, between them, partition the input domain. Initially, the $\mathrm{pre}(C_i)$ may not themselves partition the input domain: the $\mathrm{pre}(C_i)$ need not define disjoint sets. However, the partition may be formed from the $pre(C_i)$ [5, 9].

Consider, for example, *Price*. Rewriting the body of this to disjunctive normal form gives nine disjuncts, including the following:

$$\begin{aligned} outward\_date? \geq current\_date? \wedge travel\_period? = peak \wedge \\ outward\_date? - current\_date? > 7 \\ \wedge price! = 0.9 * price\_peak(start?, end?)) \end{aligned}$$

and

$$\begin{aligned} outward\_date? \geq current\_date? \wedge travel\_period? = off\_peak \wedge \\ \exists\, p1, p2 : Q \bullet r? = STUDENT \wedge p1 = 0.9 \wedge \\ outward\_date? - current\_date? \leq 7 \wedge p2 = 1 \\ \wedge price! = p1 * p2 * price\_off\_p(start?, end?) \end{aligned}$$

The above lead to the following conditions of subdomains of the partition:

$$\begin{aligned} outward\_date? \geq current\_date? \wedge \\ travel\_period? = peak \wedge \\ outward\_date? - current\_date? > 7 \end{aligned}$$

and

$$\begin{aligned} outward\_date? \geq current\_date? \wedge \\ travel\_period? = off\_peak \wedge \\ r? = STUDENT \wedge outward\_date? - current\_date? \leq 7 \end{aligned}$$

It has been noted [9] that it is not always necessary to rewrite the specification of an operation all the way to DNF. For example, suppose the specification of an operation is $(x? > 0 \vee y? > 0) \wedge r! = "ok"$. Here there is only one specified behaviour ($r! = "ok"$) with one precondition ($x? > 0 \vee y? > 0$). The DNF approach will split this, making it appear that there are two separate behaviours with separate preconditions. Thus, the DNF approach produces more subdomains than required and, as a consequence, may suggest more tests than necessary. This may reduce the set of problems for which the approach is practical. Another example of a case where the DNF approach splits the behaviour too far, is a specification of the form $x? > 0 \wedge (r! = "ok" \vee r! = "positive")$. The DNF approach will again produce two disjuncts but here the two have the same precondition.

An alternative approach, which tackles some of the problems described above, is based on estimating the role of predicates [9]. For example, in $x? > 0 \wedge (r! = "ok" \vee r! = "positive")$ it is clear that the two predicates $r! = "ok"$ and $r! = "positive"$ are not involved in defining any partition produced since they only refer to output. Thus, it is not necessary to rewrite the predicate $x? > 0 \wedge (r! = "ok" \vee r! = "positive")$.

While the approach of Ref. [9] overcomes the problems described above, it may still suffer from a combinatorial explosion. The approach outlined in this paper avoids the combinatorial explosion by not actually generating the partition. Instead, it identifies the predicates responsible for defining the partition and uses these to produce a classification tree. If all combinations of classes are chosen then the combinatorial explosion returns. However, using the approach introduced here, the tester may choose the combinations used in a way that avoids this.

2.2 The Classification Tree Method

The Classification Tree Method provides a flexible basis for test generation. The tester identifies a number of aspects, called *classification aspects*. Each aspect represents a property to be considered in testing. For example, for *Price*, the tester might identify the aspect: the period during which the journey will occur.

For each aspect a number of *classes*, which represent different values or sets of values for the aspect, are defined. For example, for the period in which the journey will occur, there might be two classes: peak and off-peak. If the number of days, between the booking and the date on which the outward journey is made, forms a classification aspect there might be two classes: the interval is greater than 7 days; and the interval is at most 7 days.

Typically, a class will represent some range of values for the classification aspect that are deemed to be treated in an equivalent manner. The classes must be disjoint and should be complete with respect to the classification aspect: the union of the classes should give the complete range of values. Where appropriate, classes might be included to represent values that are likely to find certain types of fault. For example, where there is a boundary between two ranges of interest, a class might represent values around this boundary. The addition of such classes allows the tester to specify the use of special values of interest (such as 0).

The tester defines an *abstract test case* by choosing a set of leaves from the classification tree. This set contains at most one class for each classification aspect. Each abstract test case is graphically represented by a horizontal line in which the classes defining the abstract test case are identified. Thus an abstract test case is effectively a set of constraints on the input and the internal state before the test. Given an abstract test case, the tester will subsequently find concrete values that satisfy these constraints. An abstract test case thus represents a set of concrete test cases, any one of which suffices. Where the classes are formally defined, an automated or semi-automated process might construct concrete test cases from abstract test cases.

As well as defining classification aspects and classes, the tester is able to include a precondition on the operation. This precondition forms part of the set of properties defining each abstract test case. Further, it is possible to add constraints that relate the classes. For example, it is possible to say that if one class c_1 has been chosen then some other class c_2, from a different classification aspect, must not be chosen.

The classification tree might be formed as a flat tree, by placing each classification aspect as a child of the root and, for each classification aspect, having each of its classes as a leaf of this node. Often, however, the tester places some structure on the classification tree. It is possible to place a classification aspect C below a class c. This means that the classification aspect C is only relevant where the class c has been chosen. Similarly, it means that if some class from C is chosen then the class c is also chosen.

Consider the classification tree in Figure 2. This classification tree is for the operation Price specified in Figure 1. The figure shows two abstract test cases. In the first, the travel period is off-peak, the person has a student railcard and is purchasing the ticket no more than a week in advance of the journey. In this classification tree the classification aspect *Railcard* being below the class *Yes* for *Off Peak* indicates that the classification aspect *Railcard* is only relevant when the travel period is *Off Peak*.

By allowing a range of classification aspects and classes to be chosen, the Classification Tree Method allows the tester to apply expert knowledge in both producing the classification tree and choosing abstract test cases. This allows the tester to concentrate effort on the parts of the system that are most likely to contain faults or are particularly critical. This is important where the number of classification aspects and classes does not allow the use of all combinations of classes.

Note that many systems will have an internal state and multiple operations. In such cases, once abstract test cases have been formed for each operation, it may be necessary to determine how to sequence these abstract test cases. This sequencing may be based on a finite state model of the system generated on the basis of the partitions [5, 8, 9, 11]. For information regarding the automatic

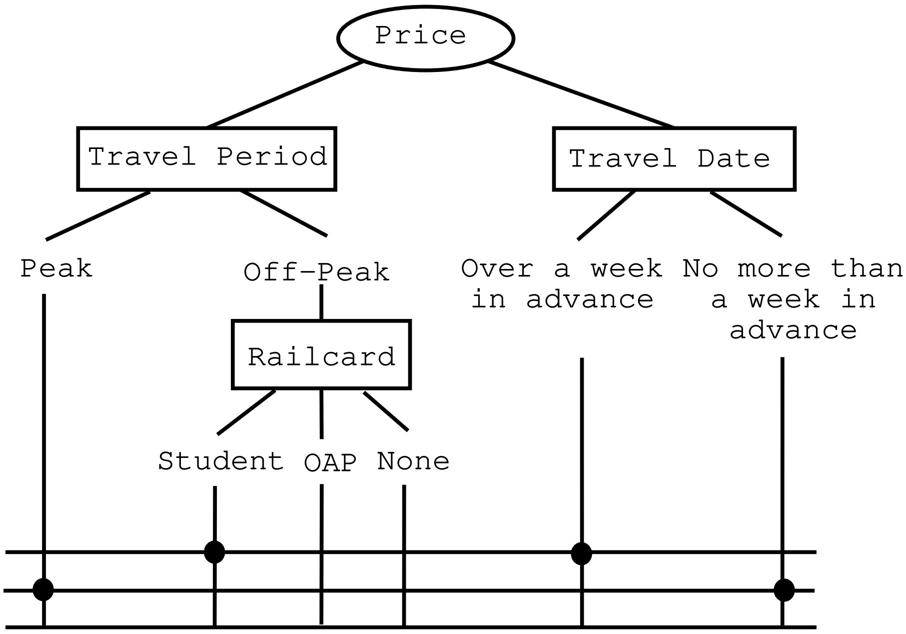

Fig. 2. A Classification Tree for Price

generation of test sequences from finite state models see, for example [10, 12, 14, 16].

3 Generating the Classification Tree

This section describes an approach to automatically generating elements of the classification tree based on the Z specification of an operation. Much of the analysis described is semantic in nature and is therefore unlikely to scale up to large examples. However, the material in this section motivates the syntactic approximation, introduced in Section 4, which allows the classification tree to be automatically developed in a more efficient way.

3.1 Generating information from the signature

The Z specification of an operation has a number of input variables and starting state variables to which the classification aspects should refer. Syntactic conventions allow these variables to be identified automatically. As described in Ref. [17], for each such variable, it is possible to generate a set of classes based on its type and expert knowledge.

Suppose an input variable or starting state variable x has an enumerated type T that contains only a few possible values. A class might be chosen for each of these values. Using this approach, if T is the *booleans* then x will have two classes: *True* and *False*.

Clearly this approach cannot be applied for infinite types and is of little value with a type that is finite but large. For such types, a number of standard partitions can be provided to the tester. Consider, for example, the type of natural numbers for which one commonly used partition is $\{\{0\}, \{1\}, \{2, \ldots\}\}$. Thus, given a variable x that represents a natural number, the tester can be offered a classification aspect with three classes defined by: $x = 0$, $x = 1$ and $x > 1$.

Given the specification S of an operation, it is possible to automatically identify the input and starting state variables from S and to provide a set of possible classifications for each of these. The tester might choose from this set of possibilities, basing their decision for a variable x on their knowledge of the role of x and the ways in which it is used. Section 4 describes how this information might be combined with information derived from the predicates.

3.2 Basing information on the predicates

This section shows how a classification tree may be based on the body of S. The analysis in this section will assume that the predicates in S are uniquely identified and thus, when considering a predicate p from S, it may be assumed that p occurs only once in S. Recall that it is assumed that S is in PNF.

The precondition of the operation may be found in the normal manner. This forms a condition for the classification tree and thus forms a condition for every abstract test case.

The problem now is to determine which predicates contribute to the precondition of some behaviour in S. Since these predicates are involved in determining which behaviour is applied, they will form the classification aspects of the classification tree. In order to determine which predicates do this, it is sufficient to rewrite S to DNF and consider the individual disjuncts, since each disjunct may be seen to form a specified behaviour. Thus, the following process may be applied in order to determine whether a predicate p of S contributes to the precondition of some behaviour:

1. Rewrite S to DNF, forming $c_1, \ldots, c_n$.
2. p contributes to the precondition of some behaviour if and only if there exists some $1 \leq i \leq n$ such that c_i has a different precondition when p is removed.

The predicates identified by the above process are those that define the partitioning of the DNF approach. However, in order to apply this approach it is necessary to define what it means to remove p from S. This is defined in Figure 3, in which $remove(p, c)$ denotes c with p removed.

Essentially, the function *remove* only alters the (single) occurrence of either p or $\neg p$ and replaces this by *true*. Note that this definition relies on S being in PNF: for some disjunct c, if c contains p or $\neg p$ there is no possibility of a term containing this being negated in c.

$$
\begin{aligned}
remove(p, q) &= q \text{ (atomic predicate } q \neq p)\\
remove(p, \neg q) &= \neg q \text{ (atomic predicate } q \neq p)\\
remove(p, p) &= true\\
remove(p, \neg p) &= true\\
remove(p, P \wedge Q) &= remove(p, P) \wedge remove(p, Q)\\
remove(p, P \vee Q) &= remove(p, P) \vee remove(p, Q)
\end{aligned}
$$

Fig. 3. The function *remove*

The classification tree may be constructed from the predicates that contribute to the precondition of a behaviour: these each have two classes, *True* and *False*.

Consider now the following, formed by rewriting the example to DNF and let this be denoted c.

$$
\begin{aligned}
outward\_date? \geq current\_date? \wedge\\
travel\_period? = peak \wedge\\
outward\_date? - current\_date? > 7\\
\wedge price! = 0.9 * price\_peak(start?, end?)
\end{aligned}
$$

$remove(travel\_period? = peak, c)$ and c have different preconditions and thus the predicate $travel\_period? = peak$ contributes to the precondition of a behaviour. In contrast, c and $remove(price! = 0.9 * price\_peak(start?, end?), c)$ have the same preconditions and thus $price! = 0.9 * price\_peak(start?, end?)$ does not contribute to the precondition of this behaviour.

The above process is likely to be highly inefficient: it combines the costs associated with rewriting to DNF, determining preconditions, and determining whether two predicates are equivalent. In fact, since the procedure involves determining whether c_i and $remove(p, c_i)$ have equivalent preconditions, the above procedure is not, in general, computable. We will now introduce a syntactic approximation that avoids these problems.

4 A syntactic approach to generating information from predicates

This section describes how classification aspects and classes may be derived automatically using a syntactic analysis of the predicates of a Z specification S of an operation. This process of deriving the classification aspects and classes is based on classifying the predicates on the basis of the variables they mention and the structure of S.

In the analysis described in this section it will be assumed that the predicates in the specification are uniquely identified. Thus, if two copies of a predicate are contained in S then we can distinguish between these copies.

Section 4.1 introduces a way of partitioning the predicates of S into input and output predicates. The input predicates will be used, both in forming and using the classification tree. Essentially, the input predicates are the predicates that define the partition formed by the DNF approach.

Section 4.2 shows how the input predicates are further classified as either partitioning predicates or defining predicates. The partitioning predicates will form classification aspects, each classification aspect having two classes: *True* and *False*.

4.1 Input and output predicates

Consider the Z specification given in Figure 1. It is possible to identify predicates that are involved in determining whether an input satisfies the precondition of the operation and which behaviour is applied when it does. These predicates are:

1. $outward\_date? \geq current\_date?$
2. $travel\_period? = peak$
3. $travel\_period? = off\_peak$
4. $outward\_date? - current\_date > 7$
5. $outward\_date? - current\_date \leq 7$
6. $r? = NONE$
7. $r? = STUDENT$
8. $r? = OAP$

By noting that this list contains some pairs of predicates, such that one is the negation of the other, this list may be further reduced to:

1. $outward\_date? \geq current\_date?$
2. $travel\_period? = peak$
3. $outward\_date? - current\_date? > 7$
4. $r? = NONE$
5. $r? = STUDENT$
6. $r? = OAP$

The problem now is to find a syntactic approach that will identify such predicates.

The predicates given above are responsible for defining the preconditions of the behaviours in the specification. Given this role, they mention only input variables (this specification has no state variables). For this reason they will be called input predicates. This can be extended to the case where a term mentions state variables, by defining a predicate to be an *input predicate* if each variable it mentions is either an input variable or a starting state variable [9].

Some predicates, such as $price! = price\_peak(start?, end?)$, are involved in defining an input/output behaviour. A predicate that is not an input predicate is an *output predicate* [9].

The process of classifying the predicates into sets of input and output predicates relies on one assumption: that no output predicate contributes to the precondition of any behaviour. This assumption might contribute to part of a definition of testability for Z specifications. Note that for a specification to fail to satisfy this it must describe a precondition in terms of output and final state values: something that is often possible, but which is likely to make the specification relatively difficult to understand. For example, if part of the precondition for an operation requires the state variable x to be non-negative, and the operation does not change the value of x, then this may be expressed by stating that the final value of x is non-negative ($x' = x \land x' \geq 0$). Where output predicates contribute to preconditions of behaviours, the impact on the approach outlined in this paper is simply that it may fail to identify some of the relevant predicates. However, the information returned should still be of value to the tester; it is sound but incomplete.

The classification of predicates into input and output predicates assumes the absence of locally quantified variables. However, even where there are locally quantified variables, the classification can be used to assist in the production of a classification tree: relevant predicates that do not contain locally quantified variables will be identified. This may be extended to predicates that mention locally quantified variables, by classifying the locally quantified variables as either input locally quantified variables or output locally quantified variables. This classification may be achieved by noting the predicates that mention each locally quantified variable and the variables mentioned in these predicates. For example, in *Price*, the locally quantified variables appear in predicates that mention output variables. Thus, the locally quantified variables in *Price* are all output locally quantified variables. The problem of classifying predicates containing locally quantified variables has been considered elsewhere [9] and will not be described further here for reasons of space.

The input predicates will now be further analysed.

4.2 Classifying input predicates

An input predicate may perform one of two roles: it might form part of the precondition of the operation as a whole or it might be involved in determining which of the behaviours is applied. For example, in *Price*, the input predicate $outward\_date? \geq current\_date?$ forms part of the precondition of the operation as a whole, while the input predicate $travel\_period? = peak$ is involved in determining which input/output relation is applied. An input predicate that forms part of the precondition of the whole operation will be called a *defining predicate*. The other input predicates, which determine which of the behaviours is applied, will be called *partitioning predicates*.

The partitioning predicates define the partition and thus are the predicates that will be used to form the classification tree. The defining predicates will contribute to a precondition on the operation and thus to every abstract test case. The problem now is to automatically determine which input predicates are defining predicates and which input predicates are partitioning predicates.

Recall that S is in PNF and that S^* denotes S with the quantifiers removed. In the following analysis it will be assumed that each input predicate appears only once in S^*. If an input predicate p does appear more than once, then copies of p may be treated as separate predicates.

Suppose $S^* = p \wedge Q$ for some input predicate p. Clearly, all specified behaviour must satisfy p and this suggests that p is a defining predicate. Suppose now that $S^* = (P_1 \vee P_2) \wedge P_3$ for predicates P_1, P_2, and P_3. The disjunction suggests that there are at least two possible behaviours with their own preconditions: $P_1 \wedge P_3$ and $P_2 \wedge P_3$ may be seen as separate behaviours. From this observation, it is possible to conclude that the input predicates from P_1 and P_2 are partitioning predicates and not defining predicates.

The above observation provides an initial way of classifying input predicates: an input predicate, p, is a defining predicate if the whole operation can be rewritten as $p \wedge P$ for some predicate P and otherwise it is a partitioning predicate. However, there is an exception: if the schema is of the form $(P_1 \vee P_2) \wedge P_3$ and P_1 and P_2 contain input predicates only then $P_1 \vee P_2$ forms part of the precondition for the whole operation and thus $P_1 \vee P_2$ is a (non-atomic) defining predicate. Thus, if P is a non-atomic predicate that contains only input predicate then P is a defining predicate in $P \wedge Q$. To see this consider the predicate $(x? \geq 0 \vee y? \geq 0) \wedge z! = x?$. This has precondition $x? \geq 0 \vee y? \geq 0$ and only one behaviour: the output $z!$ is equal to $x?$. To consider $x? \geq 0$ and $y? \geq 0$ as partitioning predicates would incorrectly suggest that there were two separate behaviours that need testing. Of course, the tester might still choose to make $x? \geq 0$ and $y? \geq 0$ form separate classification aspects.

From the above observations it is clear that the defining predicates need not be atomic. Let *out* denote a function that takes a predicate T and returns the (atomic) output predicates contained in T and *in* denote a function that takes a predicate T and returns the (atomic) input predicates contained in T. Further, let *atomic* denote a function that takes a predicate T and returns the set of atomic predicates contained in T and let $\overline{atomic}$ denote a function that takes a set $\overline{P}$ of predicates and returns the set of atomic predicates contained in the predicates in $\overline{P}$. The recursive function *def*, that returns the sets of defining predicates, is defined by the rules in Figure 4; these will now be explained. The function *part*, that returns the partitioning predicates, will be defined in terms of *def*.

The first two rules provide base cases and simply require that *def* applied to a constant returns no defining predicates. The third and fourth rules are also base cases, stating that $def(p)$ returns p if p is an atomic input predicate or the negation of an atomic input predicate. Essentially, p is a defining predicate in p if and only if p is an input predicate and $\neg p$ is a defining predicate in $\neg p$ if and only if p is an input predicate.

The fifth rule considers a predicate that is a conjunction of two predicates P and Q. Observe here that if P can be rewritten to the form $p \wedge P'$ then $P \wedge Q$ can be rewritten to the form $p \wedge R'$ (here $R' = P' \wedge Q$). Thus, if a predicate p is a defining predicate for either P or Q then it is a defining predicate for

$P \wedge Q$. Further, if $P \wedge Q$ can be rewritten to the form $p \wedge R'$ then either P can be rewritten to the form $p \wedge P'$ or Q can be rewritten to the form $p \wedge Q'$. Thus a predicate p is a defining predicate for $P \wedge Q$ if and only if it is either a defining predicate for P or it is a defining predicate for Q.

The final rule deals with predicates of the form $P \vee Q$ and essentially encapsulates two cases. If either P or Q contain one or more output predicates then P and Q represent separate sets of behaviours with their own preconditions. In this case, $P \vee Q$ will have no defining predicates. The second case is that $P \vee Q$ is, itself, a defining predicate if it contains only input predicates.

$$
\begin{aligned}
\mathit{def}(\mathit{true}) &= \varnothing \\
\mathit{def}(\mathit{false}) &= \varnothing \\
\mathit{def}(p) &= \text{ if } \mathit{out}(p) = \varnothing \text{ then } \{p\} \\
&\quad\; \text{else } \varnothing \text{ (} p \text{ is an atomic predicate)} \\
\mathit{def}(\neg p) &= \text{ if } \mathit{out}(p) = \varnothing \text{ then } \{\neg p\} \\
&\quad\; \text{else } \varnothing \text{ (} p \text{ is an atomic predicate)} \\
\mathit{def}(P \wedge Q) &= \mathit{def}(P) \cup \mathit{def}(Q) \\
\mathit{def}(P \vee Q) &= \text{ if } \mathit{out}(P \vee Q) = \varnothing \text{ then } \{P \vee Q\} \\
&\quad\; \text{else } \varnothing
\end{aligned}
$$

Fig. 4. A recursive definition of *def*

The set of defining predicates of S is $\mathit{def}(S^*)$. Since each input predicate appears only once and a predicate cannot be both a partitioning predicate and part of a defining predicate, we have that:

$$\mathit{part}(S) = \mathit{in}(S) \setminus \overline{\mathit{atomic}}(\mathit{def}(S^*)).$$

Each partitioning predicate will form a classification aspect with two classes: *True* and *False*.

It is possible to combine the classification aspects based on the predicates with those based on data type information to form a classification tree. The data type information should be compared with the defining predicates. For example, suppose that the following partition of Q is chosen for an input variable $a?$.

$$\{\{x : Q \mid x < 0\}, \{0\}, \{x : Q \mid x > 0\}\}$$

If there is a defining predicate $a? \geq 0$ then this reduces to:

$$\{\{0\}, \{x : Q \mid x > 0\}\}$$

.

4.3 Classifying the predicates of the example

Consider the application of *def* to *Price*. Applying the rule for conjunction gives the following term:

$$
\begin{aligned}
def(Price) = {} & def(outward\_date? \geq current\_date?) \cup \\
& def((travel\_period? = peak \wedge \\
& (outward\_date? - current\_date? > 7 \\
& \wedge price! = 0.9 * price\_peak(start?, end?) \\
& \vee outward\_date? - current\_date? \leq 7 \\
& \wedge price! = price\_peak(start?, end?)) \\
&) \vee \\
& (travel\_period? = off\_peak \wedge \exists\, p1, p2 : Q \bullet \\
& (r? = NONE \wedge p1 = 1 \vee r? = STUDENT \\
& \wedge p1 = 0.9 \vee r? = OAP \wedge p1 = 0.7) \wedge \\
& (outward\_date? - current\_date? > 7 \\
& \wedge p2 = 0.6 \vee \\
& outward\_date? - current\_date? \leq 7 \\
& \wedge p2 = 1) \wedge \\
& price! = p1 * p2 * price\_off\_p(start?, end?)))
\end{aligned}
$$

Applying the rule for disjunction and the rules for atomic predicates, this reduces to:

$$def(Price) = \{outward\_date? \geq current\_date?\}.$$

Thus *Price* has one defining predicate only: $outward\_date? \geq current\_date?$ and each test case is given the precondition $outward\_date? \geq current\_date?$. All other input predicates are partitioning predicates.

The following are thus the partitioning predicates of *Price*:

1. $travel\_period? = peak$
2. $travel\_period? = off\_peak$
3. $outward\_date? - current\_date? > 7$
4. $outward\_date? - current\_date? \leq 7$
5. $r? = NONE$
6. $r? = STUDENT$
7. $r? = OAP$

As the corresponding leaves of the classification tree have two associated values, *True* and *False*, this set may be reduced to the following:

1. $travel\_period? = peak$
2. $outward\_date? - current\_date? > 7$
3. $r? = NONE$
4. $r? = STUDENT$
5. $r? = OAP$

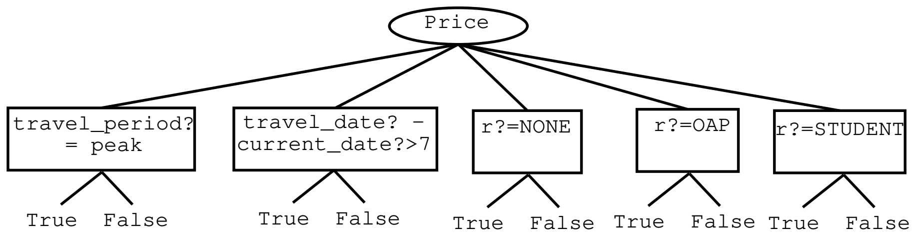

Fig. 5. Classification tree based on predicates only

The classification tree produced by this process is shown in Figure 5. The classification tree is flat: it has five nodes below the root, each leaf being capable of taking on one of the values *True* and *False*. Further, there are dependencies between these nodes. Specifically, if any of the three nodes that refer to $r?$ takes on the value *True* then the others must be *False*. It is thus desirable to add some structure to this classification tree. The tester might provide this structure. The automatic identification of situations in which the structure may be improved is a problem for future work. Note that while the example does not include state variables, the inclusion of these does not affect the overall procedure.

Given the classification tree, the tester may choose any combination of leaves. For example, they might choose an abstract test case defined by the classes: $travel\_period? = peak$ being *True*, $travel\_date? - current\_date? > 7$ being *True*, and $r? = NONE$ being *True*. This places the constraint $travel\_period? = peak \wedge travel\_date? - current\_date? > 7 \wedge r? = NONE$ on the input. The tester then has to find concrete input values that satisfy this constraint.

5 Test size reduction

This section will show how analysis based on the structure of S may be used to avoid certain combinations of classes. This may reduce the test effort by identifying abstract test cases that make little sense.

Suppose that the partitioning predicates p_1 and p_2 are conjoined in S, p_1 and p_2 appear only once in S and p_1 and p_2 have been chosen as classification aspects. Suppose further that, for an abstract test case being developed, the class *False* has been chosen for p_1. Since p_2 is conjoined with p_1, the value of p_2 is irrelevant. If the tester does not realise this, there are at least two potential consequences:

1. The tester chooses abstract test cases, with p_1 being *False*, that differ only in the value of p_2, believing these to test different behaviours.
2. In order to explore the impact of the value of p_2, the tester chooses abstract test cases that differ only in the value of p_2. However, in these abstract test cases, p_1 is *False*.

From the first case it is clear that there may be pairs of abstract test cases that test the same behaviour: the test contains unintended redundancy. This may reduce the efficiency of testing. From the second case it is clear that there may be a pair of abstract test cases, intended to test the impact of p_2, that cannot achieve this. This may reduce the effectiveness of testing. This section will consider an approach to automatically devising constraints in order to avoid the problems described above.

Ideally a constraint for p in S should give the conditions under which the value of p may affect the selection of a behaviour from S. This is when there is some disjunct c_i, formed by rewriting S to DNF, whose precondition differs from that of the predicate produced by replacing p in c_i with $\neg p$. Determining this condition is likely to be expensive. A further disadvantage is that the information returned by the above process might appear to bear little relationship to the predicates in the classification tree. Where this is the case, it does not help the tester to choose sensible combinations of classes.

A syntactic approximation, called the test context, will be introduced. The analysis will be computationally cheap to perform and the result of the analysis will be expressed in terms of the classes of the classification tree. The analysis will produce a necessary condition for the value of a predicate to be relevant. While the condition need not be sufficient, it is capable of warning the tester to avoid certain combinations of classes. Thus, the use of the test context should make testing more efficient and effective.

As before, it will be assumed that each partitioning predicate appears only once in S; where a partitioning predicate appears more than once the test context will be constructed for each separate occurrence and the disjunction of these taken.

Observe that the value of p is relevant if and only if, were S^* to be rewritten to DNF, one of the disjuncts containing either p or $\neg p$ contains no other term that takes on the value *false*. Thus the test context of p in S may be formed in the following way:

1. Rewrite S^* to DNF to form T_1.
2. Delete all disjuncts from T_1, that do not mention p, to form T_2.
3. Delete the references to p in T_2 to form T_3.
4. Delete the output predicates from T_3 to form T_4.
5. Return T_4 as the test context.

Although the above algorithm determines the test context of p in S, the first step may lead to a combinatorial explosion. Instead, a recursive definition will be given. Given S, let S' denote S with the output predicates removed and any quantifiers removed. Given partitioning predicate p from S, the test context of p in S will be denoted $tc(p, S')$. The function tc, which will be defined below, is more efficient than the above algorithm since it avoids the combinatorial explosion. tc has a precondition: when applied to (p, S'), the predicate p must appear exactly once in S'.

The test context is defined by the rules in Figure 6. Here, the first two rules form the base case: the value of p is always relevant to the values of both p and $\neg p$. The third rule considers the test context of p in $S' = P \wedge Q$. Suppose p appears in P (and thus does not appear in Q). The value of p appears in some disjunct, formed by rewriting S' to DNF, whose other predicates are all *true*, if and only if two conditions are satisfied:

1. P, when rewritten to DNF, has some disjunct that contains p and whose other predicates are *true*; and
2. Q, when rewritten to DNF, contains some disjunct whose predicates are all *true*.

The first of these conditions is equivalent to $tc(p, P)$ being *true*; the second is equivalent to Q being *true*.

The final rule deals with the case where $S' = P \vee Q$. Suppose that p is contained in P. Thus, p is in a disjunct, formed by rewriting S' to DNF, in which all other predicates are *true*, if and only if p is in a disjunct, formed by rewriting P to DNF, in which all other predicates are *true*. Thus, if p is contained in P then $tc(p, S') = tc(p, P)$ and otherwise $tc(p, S') = tc(p, Q)$.

$$
\begin{aligned}
tc(p, p) &= \mathit{true} \\
tc(p, \neg p) &= \mathit{true} \\
tc(p, P \wedge Q) &= \text{if } p \in \mathit{atomic}(P) \text{ then } tc(p, P) \wedge Q \\
& \quad \text{else } tc(p, Q) \wedge P \\
tc(p, P \vee Q) &= \text{if } p \in \mathit{atomic}(P) \text{ then } tc(p, P) \\
& \quad \text{else } tc(p, Q)
\end{aligned}
$$

Fig. 6. The test context

Consider now the application of the test context to the predicate $r? = OAP$ in *Price*. This is:

$$
\begin{aligned}
tc(r? = OAP, \mathit{outward\_date}? \geq \mathit{current\_date}? \wedge \\
(\mathit{travel\_period}? = \mathit{peak} \wedge \\
(\mathit{outward\_date}? - \mathit{current\_date}? > 7
\end{aligned}
$$

$$
\begin{aligned}
\vee outward\_date? - current\_date? \leq 7)) \vee \\
(travel\_period? = off\_peak \wedge \\
(r? = NONE \vee r? = STUDENT \vee r? = OAP) \wedge \\
(outward\_date? - current\_date? > 7 \vee \\
outward\_date? - current\_date? \leq 7)))
\end{aligned}
$$

Applying the rule for conjunction, this becomes:

$$
\begin{aligned}
outward\_date? \geq current\_date? \wedge \\
tc(r? = OAP, travel\_period? = peak \wedge \\
(outward\_date? - current\_date? > 7 \\
\vee outward\_date? - current\_date? \leq 7) \vee \\
(travel\_period? = off\_peak \wedge \\
(r? = NONE \vee r? = STUDENT \vee r? = OAP) \wedge \\
(outward\_date? - current\_date? > 7 \vee \\
outward\_date? - current\_date? \leq 7)))
\end{aligned}
$$

Applying the rule for disjunction, reduces this to:

$$
\begin{aligned}
outward\_date? \geq current\_date? \wedge \\
tc(r? = OAP, travel\_period? = off\_peak \wedge \\
(r? = NONE \vee r? = STUDENT \vee r? = OAP) \wedge \\
(outward\_date? - current\_date? > 7 \vee \\
outward\_date? - current\_date? \leq 7))
\end{aligned}
$$

Two more applications of the rule for conjunction reduces this to:

$$
\begin{aligned}
outward\_date? \geq current\_date? \wedge \\
travel\_period? = off\_peak \wedge \\
(outward\_date? - current\_date? > 7 \vee \\
outward\_date? - current\_date? \leq 7) \wedge \\
tc(r? = OAP, r? = NONE \vee r? = STUDENT \vee r? = OAP)
\end{aligned}
$$

Applying the rule for disjunction, followed by the rule that states $tc(p, p) = true$, gives:

$$
\begin{aligned}
outward\_date? \geq current\_date? \wedge \\
travel\_period? = off\_peak \wedge \\
(outward\_date? - current\_date? > 7 \vee \\
outward\_date? - current\_date? \leq 7)
\end{aligned}
$$

This reduces to:

$$outward\_date? \geq current\_date? \wedge travel\_period? = off\_peak$$

The first of these predicates is a defining predicate. Thus, if a class for the classification aspect $r? = OAP$ is chosen then the class *True* for the classification aspect $travel\_period? = off\_peak$ should also be chosen. This tells the tester that it is not necessary to have any abstract test cases that have the class *True* for $r? = OAP$ and the class *False* for $travel\_period? = off\_peak$.

6 Conclusions

This paper has introduced an approach that supports the Classification Tree Method. This approach partially automates the generation of a classification tree from a Z specification, providing a mechanism to avoid the combinatorial explosion inherent in previous approaches. The tester may further augment the tree constructed in order to exploit expert knowledge.

The approach described here uses two sources of information: the signature of a schema and the predicates contained within it. For each type of an input or state variable it is possible to devise a number of standard partitions. Thus, for example, with the integers one partition is: integers less than zero, zero, and integers greater than zero.

An algorithm has been given to identify predicates in the specification that are involved in either describing the precondition of the operation (defining predicates) or in determining which behaviour is applied (partitioning predicates). The defining predicates provide a precondition for each test case while the partitioning predicates are used to form nodes of the classification tree.

The paper has also defined the test context of a predicate. Given a partitioning predicate p in specification S, the test context of p in S identifies other choices that must be made in order for the value of p to be relevant. This allows the tester to avoid certain abstract test cases that make little sense, increasing test efficiency and effectiveness.

References

1. N. Amla and P.Ammann. Using Z specifications in category partition testing. In *COMPASS '92, Seventh Annual Conference on Computer Assurance*, pages 15–18, Gaithersburg, MD, USA, 1992.
2. T. Y. Chen and P. L. Poon. Construction of classification trees via the classification-hierarchy table. *Information and Software Technology*, 39:889–896, 1997.
3. T. Y. Chen and P. L. Poon. On the effectiveness of classification trees for test case construction. *Information and Software Technology*, 40:765–775, 1998.
4. J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Journal of Software Testing, Verification, and Reliability*, 9:27–50, 1999.

5. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93, First International Symposium on Formal Methods in Europe*, pages 268–284, Odense, Denmark, 19-23 April 1993. Springer-Verlag, Lecture Notes in Computer Science 670.
6. M. C. Gaudel. Testing can be formal too. In *TAPSOFT'95*, pages 82–96. Springer-Verlag, March 1995.
7. M. Grochtmann and K. Grimm. Classification trees for partition testing. *Journal of Software Testing, Verification and Reliability*, 3:63–82, 1993.
8. P. A. V. Hall and R. M. Hierons. Formal Methods and Testing. Technical Report 91/16, Computing Department, The Open University, August 1991.
9. R. M. Hierons. Testing from a Z specification. *Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.
10. R. M. Hierons and M. Harman. Testing comformance to a quasi-non-determinstic stream X-machine. *Formal Aspects of Computing*, 12:423–442, 2000.
11. R. M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using statecharts and Z. *Information and Software Technology*, 43:137–149, 2001.
12. R. M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Transactions on Computers*, 51:1111–1117, 2002.
13. C. A. R. Hoare. How did software get so reliable without proof? In *Proceedings of Formal Methods Europe, 96 (Lecture Notes in Computer Science 1051)*, pages 1–17. Springer-Verlag, 1996.
14. M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer-Verlag, 1998.
15. G. Laycock. Formal specification and testing: A case study. *Journal of Software Testing, Verification and Reliability*, 2:7–23, 1992.
16. D. Lee and M. Yannakakis. Principles and methods of testing finite-state machines. *Proceedings of the IEEE*, 84:1089–1123, 1996.
17. H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. In *First IEEE Conference on Formal Engineering Methods*, pages 81–90, Hiroshima, Japan, November 1997. IEEE Computer Society.
18. P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 2:777–793, 1996.