Tests Derivation from Model Based Formal Specifications

Christophe Meudec
CPM Department
Institute of Technology Carlow*
Carlow, Eire

Abstract

Software testing consumes a large percentage of total software development costs. Yet, it is still usually performed manually in a non rigorous fashion. In this work we suggest how state of the art practices in the area of testing can be applied to the systematic generation of tests from model-based formal specifications.

1 Introduction

Software testing is maybe the least well defined activity in the entire software development life cycle, its value is even sometimes contested on the basis of the now famous Dijkstra's comment [11]:

> Program testing can be used to show the presence of bugs, but never to show their absence!

This argument is still valid in that even if a program validates the best tests one can devise it does not imply the correctness of the system under consideration with respect to its specification, only formal proving can achieve that aim. Against this background it should be kept in mind that formal proving of program correctness is, at least when not completely automated, an error prone activity and that therefore, at least to inspire confidence into the program delivered, software testing is also necessary even after a proof of the program has been established. [29] is an early note on why testing is still necessary after correctness proof construction.

Testing a program in its environment of use also means that the compiler, the operating system and the run time environment are all taken into account while establishing the reliability of the software delivered. Furthermore, one cannot ignore the prevalence of testing in everyday software development over formal development. And instead of dismissing it out of hand [22], accommodating it within the current formal framework advocated by many software engineers and academics may be more productive [20].

In doing so we may also remove some of the myths around both testing and formal development [17, 6]. Formality should not be an end to itself but faced with such a mass of ad-hoc techniques which constitute the testing phase one cannot believe that incorporating formality [4] in the testing process would not at least clarify the situation and allow further progress.

However, even the aim of software testing can lead to misunderstandings—i.e. is it to find as many errors as possible or to inspire confidence in the program under test? Here we will concentrate on defect testing as defined below.

- the aim of defect testing is to find as many errors as possible in the program under test (using "special-values testing" [18]).
- the aim of operational testing is to increase the reliability of the program under test (using tests that mimics the "operational distribution" of typical usage [18]).

*Most of the work described here was carried out at the Queen's University of Belfast while the author was working towards a Ph.D. [23]

However we should always keep in mind the shortcomings of defect testing as highlighted in [15]: primarily it may not always be the most straightforward method for increasing the reliability of the software under test.

The testing activity is increasingly being automated. However, the crucial area of tests generation is still largely performed manually and, according to Ould [26], is the most important aspect of software testing requiring automation. Because manual testing demands fastidiousness, and hence is often performed non-rigorously, the need for automated tests generating tools in the software engineering community is strong.

But automatic test generation is still in its infancy: few automatic tests generators have actually been implemented and far less, if any, are in use everyday [19]. Most automatic test generators use white box techniques to derive the tests [10]. White box techniques use the source code as their basis for tests derivation. Black box techniques, where some sort of program specification is used for the derivation process, have not received the same amount of consideration. This is certainly due to the absence of intuitive methods for the derivation of tests from specifications. The main exception to this remark are test selection methods based on deterministic Finite State Machines where many useful theoretical results exists [9, 27, 16]. Without extensions however, FSMs have a limited modelling ability: only the control aspect of systems can be specified. Some formal notations that extend somewhat the usefulness of FSMs as a general specification technique are: LOTOS [5], Estelle [8] and SDL [3]. However those notations are not as expressive as model-based formal specification languages such as Z and VDM-SL.

These preliminary considerations form the rationale to our undertaking of trying to systematically generate tests from VDM-SL specifications.

Previous work in this area include the commendable work of J. Dick and A. Faivre [12] for their implementation of an automatic test generator prototype from a subset of VDM-SL. Although the VDM-SL subset used is very limited (e.g. no quantified expressions) their work forms much of the basis for other work in this area. In particular [21] re-applies their work, without extending it, to Z. Stocks and Carrington [28] present a formal framework for tests derivation from Z specifications where, unfortunately, automation is not addressed.

2 Adapting Testing Practices

Generating tests using black box techniques is usually performed manually from a natural language specification using a partitioning strategy [2, 7]. Partitioning strategies divide the input domain of the program under test into equivalence classes whose elements are somehow the same. That is, it should be sufficient to select one member of a particular equivalence class to represent the entire sub-domain—every sub-domain should be homogeneous in respect to a fail-pass criterion based on the correct behaviour of the program. An equivalence class is therefore a nonempty subset of the input domain of a program. In software testing, members of a partition are often informally called classes rather than equivalence classes.

However, tests adequacy criteria—criteria stating when enough testing has been performed by executing and checking the outcome of the tests—are not as straightforward to formulate as in white box testing where coverage measures of the underlying source code tested are often used as approximations. The British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST)'s recently released standard for software component testing [7] suggests the following testing coverage measure for partitioning techniques:

(number of identified equivalence class tested / number of identified equivalence class) * 100

Since it is left to the practitioner to identify equivalence classes this measure is wholly inappropriate as an indication of test set adequacy.

Further, the general problem of determining a priori which set of tests among several has a greater likelihood of detecting defects in the program under test has no practicable solution. We therefore have to resort to a subjective approach and define an adequate test set as follows:

> an adequate test set is a test set that has been manually derived using state of the art testing principles or that has been systematically generated following such state of the art principles.

Hence, we will have two ways of checking the degree of adequacy of a test set: by comparing it to a manually derived test set or by ensuring it has been generated following established test generation techniques. We must remember however that the adequacy of test sets is subjective and a matter of ongoing theoretical research and debate [31, 15].

The size of the test sets generated is also an important practical concern and techniques to reduce the size of the test sets while respecting the chosen criteria have to be devised. This will be discussed in section 4.

More about partitioning as a testing theory can be found in [18, 30].

3 Partitioning VDM-SL Expressions

As suggested, it may be possible to partition a specification into equivalence classes. These equivalence classes should, in theory, be homogeneous in respect to the likelihood of finding an error in the behaviour of the system under test, thus allowing a single random sample of each class to be then taken to obtain a test set or possibly test cases.

This was demonstrated by Dick and Faivre in [12] where straightforward partitioning rules for logical expressions and VDM-SL control expressions were presented. However, no attempts were made at justifying the technique nor the rules themselves. Further, many important aspects of VDM-SL were not considered (such as quantified expressions and the Logic of Partial Functions). Dick and Faivre placed more emphasis on implementing a prototype tool than on exploring fundamental issues.

3.1 Justification: an Attempt

To apply the partitioning theory to VDM-SL expressions, it must be mitigated, by our ignorance as to what characterize homogeneous equivalence classes in terms of intrinsic formal features and by the technical difficulties which may arise from attempting to extract such classes from a formal specification.

However, advantage can be taken of the intrinsic mathematical nature of formal specifications to generate partitions in the mathematical sense rather than using the more fuzzy notion of homogeneous classes in respect to the likelihood of finding an error in the system under test.

Bearing this in mind, one is left to consider ad hoc techniques based on the structure of the specification. The homogeneity of the classes which may thus be generated will therefore be highly subjective. Indeed, in the partitioning rules introduced below few formal arguments towards any kind of justification will be given since they emanate from ad hoc practices—some indeed borrowed from structural testing. No claim can be made concerning their universality or adequacy and, instead, we are left to assume that the mathematical classes are homogeneous in terms of likelihood of finding an error in the behaviour of the system under test.

Although we have ruled out applying partitioning techniques to VDM-SL statements [23], implicit VDM-SL specifications can still be said to have a control structure, in forms of if and $cases$ expressions, resembling imperative programs. The control structure of specifications is usually far more limited than in computer source programs—i.e. the number of paths in a source program would usually be far greater than the number of control classes in its specification counterpart: this can be thought to be due to the greater expressiveness of mathematics over imperative programming language constructs and to the concern in specifications to express the $what$ rather than the how. However, the logical structure of specifications—in the form of pre and post conditions where logical operators are widely used—can also be taken advantage of in a similar way as control structures in attempting to generate equivalence classes.

For example an expression of the form $A \vee B$ can be validated in two-valued logic in three ways: A is true and B is false, A is true and B is true, A is false and B is true. These three conditions represent a logical decomposition of the original expression in the same way as: $if\ A\ then\ B\ else\ C$ is decomposed into A is true and B is true, A is false and C is true.

We therefore propose, on grounds of similarity alone, to extend common structural partitioning techniques to logical expressions for formal specifications in the hope of improving the homogeneity of the classes generated.

3.2 Notation

A set of VDM-SL predicates $P = \{e_1, \ldots, e_n\}$ where $n \geq 1$ is a partition of a satisfiable VDM-SL predicate E if and only if:

$$(\forall i \in \{1, \ldots, n\} \cdot consistent(e_i)) \wedge (e_1 \oplus \ldots \oplus e_n \Leftrightarrow E)$$

where $consistent$ is a predicate provided by a solver which takes a VDM-SL predicate as argument and returns $True$ if it is satisfiable and $False$ otherwise; $\oplus$ denotes the exclusive or Boolean operator so that only one of the e_i is satisfied at any one time. The predicates e_is are called equivalence classes, or simply classes, of the partition.

In addition, the empty set, denoted by $\emptyset$, is considered to represent the partition of all unsatisfiable expressions (denoted by $\perp$).

The partitioning function P has signature: $VDMexpression \rightarrow Partition$. To combine partitions we define the full combination operator $\times$ as:

$$Partition \times Partition \rightarrow Partition$$
$$P_1 \times P_2 = \{c_1 \wedge c_2 | c_1 \in P_1, c_2 \in P_2\}$$

We can therefore note the partition of $A \vee B$ as:

$$P(A \vee B) = \bigcup \left\{ \begin{array}{c} P(A \wedge \neg B) \\ P(A \wedge B) \\ P(\neg A \wedge B) \end{array} \right\}$$

where the braces denote a set of partitions to which the distributed union operator $\bigcup$ is applied to obtain a partition.

3.3 Partitioning Rules

Below are partitioning rules in two-valued Boolean logic:

$P(\perp) = \emptyset$

$P(A) = \{A\}$ when other rules do not apply

$P(\neg A) = \{\neg A\}$ when other rules do not apply

$$P(A \vee B) = \bigcup \left\{ \begin{array}{c} P(A \wedge \neg B) \\ P(A \wedge B) \\ P(\neg A \wedge B) \end{array} \right\}$$

$$P(A \wedge B) = P(A) \times P(B)$$

$$P(A \Rightarrow B) = \bigcup \left\{ \begin{array}{c} P(\neg A \wedge \neg B) \\ P(\neg A \wedge B) \\ P(A \wedge B) \end{array} \right\}$$

$$P(A \Leftrightarrow B) = \bigcup \left\{ \begin{array}{c} P(A \wedge B) \\ P(\neg A \wedge \neg B) \end{array} \right\}$$

$$P(A = B) = P(A \Leftrightarrow B)$$

$$P(A \neq B) = \bigcup \left\{ \begin{array}{c} P(A \wedge \neg B) \\ P(\neg A \wedge B) \end{array} \right\}$$

Using this scheme it is possible to systematically generate intermediate partitions (i.e. in which unsatisfiable equivalence classes may remain) from simple VDM-SL expressions. For example the intermediate partition of the expression: $E = (((x = 0 \vee y = 0) \Rightarrow x + y = 0) \wedge D = no)$ is:

$$P(E) = \left\{ \begin{array}{l} x \neq 0 \wedge y \neq 0 \wedge x + y \neq 0 \wedge D = no \\ x \neq 0 \wedge y \neq 0 \wedge x + y = 0 \wedge D = no \\ x = 0 \wedge y \neq 0 \wedge x + y = 0 \wedge D = no \\ x = 0 \wedge y = 0 \wedge x + y = 0 \wedge D = no \\ x \neq 0 \wedge y = 0 \wedge x + y = 0 \wedge D = no \end{array} \right\}$$

To obtain the final partition the classes are individually checked for satisfiability to obtain:

$$P(E) = \left\{ \begin{array}{l} x \neq 0 \wedge y \neq 0 \wedge x + y \neq 0 \wedge D = no \\ x \neq 0 \wedge y \neq 0 \wedge x + y = 0 \wedge D = no \\ x = 0 \wedge y = 0 \wedge x + y = 0 \wedge D = no \end{array} \right\}$$

We can present the equivalence classes of partitions in a representation easily readable by a human; we will say that we simplify the predicates. These simplifications are not necessary and are performed for clarity only.

Thus: $P(((x = 0 \vee y = 0) \Rightarrow x + y = 0) \wedge D = no) =$

$$\left\{ \begin{array}{l} x \neq 0 \wedge y \neq 0 \wedge x \neq -y \wedge D = no \\ x \neq 0 \wedge x = -y \wedge D = no \\ x = 0 \wedge y = 0 \wedge D = no \end{array} \right\}$$

We can extend our partitioning rules to deal with VDM-SL control expressions. A simple if expression of the form: $if\ B\ then\ E_1\ else\ E_2$ can be partitioned into:

$$\bigcup \left\{ \begin{array}{c} P(B \wedge E_1) \\ P(\neg B \wedge E_2) \end{array} \right\}$$

Attempting to deal with $Cases$ expressions is problematic since pattern matching can behave non deterministically and introduce looseness in expressions. Although loose expressions can be treated in a similar fashion to deterministic expressions as far as the test derivation process is concerned they introduce problems in terms of redundant tests and with the checking of test results [23] (that is the provision of an oracle based on the formal specification).

We must also consider that VDM-SL, unlike Z, is based on a three valued logic. We cannot ignore the effect that LPF has on some expressions because the partitions generated could then be incomplete. Consider for example the following expression: $if\ x/y = 0\ then\ M\ else\ N$ which once partitioned in two valued logic yields: $x/y = 0 \wedge M \oplus x/y \neq 0 \wedge N$. Let $y = 0$ in the original expression. According to the semantics of the $/$ operator, $x/0$ is undefined, thus $x/0 = 0$ is also undefined. Further, according to the semantics of VDM-SL for if expressions, N should then be satisfied. But, in our partition both equivalence classes evaluate to undefined, whenever $y = 0$, i.e. $\perp$ because they are not satisfiable. Therefore the test input $y = 0$ can never be generated, even randomly, from the classes since it does not satisfy any of the equivalence classes.

This incompleteness can have dire consequences. For example consider the expression:

$y = 0 \wedge if\ x/y = 0\ then\ M\ else\ N$

which yields an empty partition and thus cannot be tested by this technique.

Further, in specifications where LPF does have an effect on the semantics, and whether testing an implementation or a specification, it is highly desirable to test such effects since they are rarely obvious when reading the specification—unless comments are used to highlight, for the benefit of the reader, instances where undefined expressions are to be found and are intended—and may therefore carry the risk to be overlooked or misinterpreted.

We therefore feel compelled to test such effects. For example the if rule becomes in its simplest form:

$$P(if\ B\ then\ E_1\ else\ E_2) = \bigcup \left\{ \begin{array}{c} P(B \wedge E_1) \\ P(\neg B \wedge E_2) \\ P(B* \wedge E_2) \end{array} \right\}$$

where $*$ denotes undefinedness (i.e. for $B*$ to be satisfied B must be undefined).
The undefined or rule becomes:

$$P((B_1 \vee B_2)*) = \bigcup \left\{ \begin{array}{c} P(\neg B_1) \times P(B_2*) \\ P(B_1*) \times P(\neg B_2) \\ P(B_1*) \times P(B_2*) \end{array} \right\}$$

and the or rule becomes:

$$P(B_1 \vee B_2) = \bigcup \left\{ \begin{array}{c} P(B_1) \times P(\neg B_2) \\ P(B_1) \times P(B_2) \\ P(\neg B_1) \times P(B_2) \\ P(B_1) \times P(B_2*) \\ P(B_1*) \times P(B_2) \end{array} \right\}$$

The other partitioning rules are modified according to VDM-SL three valued logic in a similar way.

LPF does introduce extra testing requirements which we feel must be taken into account unless incompleteness and lower test set quality are accepted. Although slightly more complex, this process can still be automated as for each operator its definition input domain is well defined by the VDM-SL semantics. For example, the divide operator is undefined if one of its operands is not a number type or if the denominator equals 0. The principle holds for map application, for example, where in $Map(E)$, E must be an element of the domain of the map Map.

From a tractability point of view, the behaviour of LPF specifications induces many more equivalence classes in specifications. But most of these will be inconsistent if common specification writing style is assumed. If one assumes that the specification correctly models the intended behaviour of the system and some kind of annotation is used to indicate places where LPF does induce a different behaviour of the specification then one could restrict the extended partitioning rules to those instances. This would greatly reduce the number of expressions that need to be checked for satisfiability. However, if such annotations are absent, too unreliable, or if the aim is to test the specification itself [14] then one should be left in no doubts as to the necessity of the extra effort required.

3.4 Refinements

The approach described so far is heavily influenced by the work of Dick and Faivre [12]: we have added a discussion of the impact of LPF, looseness and presented a formalism suitable for expressing partitioning rules. But, as Stocks et al. recently remarked, with an implicit reference to Dick and Faivre's work, in [28]:

> A standard approach in specification-based testing is to reduce the specification to disjunctive normal form and choose inputs satisfying the preconditions of each disjunct. This tends to be too simplistic because model-based specifications are generally quite flat, and because specification languages have powerful operators built into the notation which hide the complexity of the input domain from a disjunctive normal form transformation.

I.e. the semantics of operators such as $\cup, \cap, \geq, \triangleleft$ must also be used to generate adequate test sets. Note that the explicit reduction to disjunctive normal form mentioned above has been shown to be unnecessary [23].

A simple way to illustrate the problem is to consider the expression $x \geq 6$ where x is a natural number. Currently this expression is not partitioned and if it was representing an equivalence class on its own, a single test would be generated through random sampling: say $x = 1997$.

However, in the case of $x = 6 \vee x > 6$ three equivalence classes are generated, and after consistency checking and random sampling, two tests are generated: $x = 6$ and say $x = 1997$. This obviously gives greater coverage of the semantics of the expression. Since the two expressions are semantically equivalent we feel this partition should be generated directly from $x \geq 6$.

The flatness of many specifications caused, for example, by large conjunctions of quantified expressions, also raises concern as it can potentially lead to the generation of a partition consisting of a single equivalence class representing the entire specification.

One could decide, to resolve this problem, to randomly sample each equivalence class a given number of times. This however would be admitting defeat, as it would constitute the ultimate recognition that our classes are non homogeneous. Furthermore, coverage would be difficult to achieve as to randomly generate $x = 6$ from $x \geq 6$ would take an unknown but usually very large number of tries.

Better, would be to try to refine our partition using the information encapsulated in VDM-SL complex operators and other constructs.

We can propose a series of partitioning rules such as: $P(x \geq 6) = \left\{ \begin{array}{l} x = 6 \\ x > 6 \end{array} \right\}$ and $P(x \neq y) = \left\{ \begin{array}{l} x = y + 1 \\ x > y + 1 \\ x + 1 = y \\ x + 1 < y \end{array} \right\}$

A justification of which is attempted in [23]. All operator can be considered including set and map operators.

Dealing with quantified expressions has not been considered before in tests generation techniques from model based specifications. It is an important area since high level specifications are likely to use quantifiers heavily. Taking into account quantified expressions reveals some important problems. For example, while one can devise a partitioning rule for, say, existential quantified expressions in LPF logic such as: $P(\exists x \in e_1 \cdot exp_2) =$

$$\bigcup \left\{ \begin{array}{l} P(e_1 \neq \{\}) \times P(\forall x \in e_1 \cdot exp_2) \\ \{\exists x \in e_1 \cdot \neg exp_2 \oplus exp_2 *\} \times \left\{ \begin{array}{c} \exists x \in e_1 \cdot c_1 \\ \forall x \in e_1 \cdot \neg c_1 \end{array} \right\} \times \ldots \times \left\{ \begin{array}{c} \exists x \in e_1 \cdot c_n \\ \forall x \in e_1 \cdot \neg c_n \end{array} \right\} \end{array} \right\}$$

where $P(exp_2) = \{c_1, \ldots, c_n\}$, proving that the set generated is an actual partition of the original expression is problematic when taking into account the possibility of loose expressions. For example the following does not hold in VDM-SL because of potential looseness:

$$\exists! x \in \{a, b, c\} \cdot f(x) \equiv \exists x \in \{a, b, c\} \cdot (f(x) \wedge \forall y \in \{a, b, c\} \cdot f(y) \Rightarrow (y = x))$$

In fact loose expressions can lead to surprising and non-intuitive results and while, after some efforts, one can evaluate complex expressions, the proof of theorems is, again, of a higher degree of complexity. Consider for example the VDM-SL standard's example [13]:

$$\begin{array}{l} unique = \exists! l_1 \frown l_2 \in (let\ s \in \{\{[4], [2, 5], 7\}, \{[2], [3], [7, 4]\}\}\ in\ s) \cdot \\ \quad let\ x \in \{2, 4\}\ in\ x \in (elems\ l_1 \cup elems\ l_2) \end{array}$$

This expression can be shown, as done in the standard [13], to evaluate to $unique = true$ (i.e. the expression is not globally loose, although internally looseness is present at every level), however when the mirror $iota$ expression (an iota expression evaluates to the unique binding which makes the unique existential expression true) is evaluated, it yields four different values.

While it can be argued that such complex examples will never find their way into real specifications, we cannot foresee how one would identify them automatically nor is it easy to define a, not too restrictive, safe VDM-SL subset which allows the use of all pattern forms when no looseness arises (e.g. the complex expression above could be allowed on the grounds that it does not introduce looseness on a global level). It could also be argued that to allow the possibility of such degree of complexity in specifications hinders the development of formal methods support tools and may deter some from using formal methods.

3.5 A Combinatorial Explosion

Using our approach as described so far leads to an unjustifiably large number of equivalence classes according to testing practices. Consider for example that the partitioning of the expression: $x \neq 6 \vee y \neq 0$ where x and y are defined integers (i.e. LPF plays no role in the semantics of this expression) leads to 24 equivalence classes and thus 24 tests :

$$\left\{ \begin{array}{cccc} x < 5 \wedge y = 0, & x = 5 \wedge y = 0, & x = 7 \wedge y = 0, & x > 7 \wedge y = 0, \\ x < 5 \wedge y < -1, & x = 5 \wedge y < -1, & x = 7 \wedge y < -1, & x > 7 \wedge y < -1, \\ x < 5 \wedge y = -1, & x = 5 \wedge y = -1, & x = 7 \wedge y = -1, & x > 7 \wedge y = -1, \\ x < 5 \wedge y = 1, & x = 5 \wedge y = 1, & x = 7 \wedge y = 1, & x > 7 \wedge y = 1, \\ x < 5 \wedge y > 1, & x = 5 \wedge y > 1, & x = 7 \wedge y > 1, & x > 7 \wedge y > 1, \\ x = 6 \wedge y < -1, & x = 6 \wedge y = -1, & x = 6 \wedge y = 1, & x = 6 \wedge y > 1 \end{array} \right\}$$

arising from the $\neq$ operator partitioning rule.

In practice, and according to Beizer [2], the sub-domains attached to x and y in this case need not be systematically combined because they are orthogonal. Orthogonality and domain testing are discussed in [2] but the discussion is limited to one or two variables. Further, there is no attempt at justifying the practice of limiting the combination of sub-domains in these circumstances.

The theoretical work of Weyuker [30] and Zhu [31] does not give any indications as how to reduce the number of classes generated without impairing adequacy, nor does it shed light as to why the ad hoc practice of Beizer is justifiable with respect to the likelihood of finding an error in the system under test.

Nevertheless, Beizer's indication that in some circumstances sub-domains need not be systematically combined is of relevance here. If some criteria relevant to formal expressions can be found that indicate when systematic partitioning is necessary and when a minimal cover of domains is sufficient then we could reduce the number of classes generated without impairing the quality of the test set produced.

4 Sensible Partitioning

The reason why many of the classes generated above seem redundant when the subsequent test set is examined as a whole, and individual test justification difficult, is due to the independence of the variables x and y.

In the expression analyzed, x and y are independent, i.e. the value of one of the variables never affects—does not constrain—the value of the other. Therefore the sub-domains attached to x (as defined by the predicates involving x such as $x < 5$, $x = 5$) are only functions of x, and similarly for y. Intuitively, then, there is no need to exhaustively combine the sub-domains generated for each variable from the separate expressions which constrain them. In fact, the independence of sub-domains implies the orthogonality notion discussed by Beizer [2] in relation to software testing.

Thus with respect to our convention for expressions involving the $\neq$ operator, the following set of classes:

$$\left\{ \begin{array}{c} x < 5 \wedge y = 0 \\ x = 5 \wedge y = -1 \\ x = 7 \wedge y = 1 \\ x > 7 \wedge y > 1 \\ x = 6 \wedge y < -1 \end{array} \right\}$$

would be sufficient to test the expression $x \neq 6 \wedge y \neq 0$ as each of the sub-domains generated in the original partition is represented in this set of classes. There are many similar sets of classes. From our point of view we deem these potential test sets as equivalent: their probability of finding an error in the system under test is equal. This is similar to considering a redundant test to be as likely to reveal an error in the implementation as a random test thus, the tests we deem redundant must be eliminated from our final test set.

The notion of variable dependency in formal specifications can be defined in various manners. Stricter definitions entail the derivation of more tests that looser definitions. We suggest that this could be a transparent way of controlling the size of the test sets generated.

In [23] we justify this approach of restricting the combination of classes using probabilities. Further we define the notion of minimal set of classes to be:

A set of classes is minimal if:

- it is composed of classes from the final partition (i.e. the partition obtained using systematic partitioning)
- it covers all dependent sub-domains from the final partition
- there does not exist a smaller set of classes for the final partition such that the two properties above are valid.

It is often the case that for a given final partition there are many minimal set of classes candidates. Any such set of classes can be chosen for sampling to lead to a minimal set of test in the sense defined above.

In [23] we detail a systematic technique for the generation of such a minimal set of test as defined. We have also generated tests for the Triangle Problem [24] and are satisfied of their adequacy as discussed in [23]. The specification used is from North [25] as detailed in the appendix.

5 Shortcomings and Conclusion

While satisfied that we have made progress in the area of tests derivation from model based specifications we must also raise some of our main concerns.

Firstly, while our technique can be described as being systematic in the sense that two human beings would derive identical test sets (identical in sense previously defined) if following the technique correctly, it is far from being automatable due to the complexity of VDM-SL. In particular the inconsistencies detection capabilities required, even for small examples, may well be far beyond what may ever be achieved. This problem is inherent to dealing with any high level mathematical notation.

The complexity of VDM-SL also implies that tests set generation following our technique is impractical without powerful tool support: too many complex equivalence classes need to be checked for satisfiability. The complexity is in part implied by potential, or actual, LPF and loose behaviour. Without tool support the derivation of tests is likely to be tedious and error prone even on small scale examples such as the Triangle Problem.

We make the following suggestions to ease the heavy implementation efforts that would be involved in building a prototype test generator based on our technique:

- To study the particular constraint solving requirements of an eventual tool, an automated oracle from a high level formal notation such as VDM-SL could be developed first. Traditionally in testing a human oracle provides the expected result of a test run. This expected result is then simply compared with the actual result for validation purposes. Using a formal specification however the result would not be computed (as this may not always be possible using non executable specifications) but instead the test input and actual test results would be fed to the specification for instantiation and simplification. The goal is then to detect whether the resulting, partly instantiated, formal specification can be satisfied.

  This automated oracle would on its own be valuable and entail the kind of constraint solving facilities necessary during tests generation. It would also establish to what extend our systematic technique could be automated.

- Our approach is flexible, many rules can be simplified in the first instance. For example, LPF behaviour could be broadly by-passed by not generating specific tests for its validation (or by using annotations to indicate where LPF behaviour is actually intended). Also for basic operators, simpler rules than those we have used could be employed (e.g. for $\neq$).

- Integration of an automatic tests generator in IFAD's toolbox [14] could also be envisaged. The toolbox only manipulates an executable subset of VDM-SL which would simplify our tests generation technique greatly. Also, the toolbox already has a test coverage analysis tool but the ability to generate tests automatically is lacking. Furthermore, the addition of another useful tool would be of benefit to the toolbox and the specification manipulation facilities of the toolbox could reduce the amount of effort required when compared with a stand alone tests generator.

- A simpler formal notation could be used. For example B [1] has a simpler semantics than VDM-SL or Z: it operates on two-valued logic, non-deterministic behaviour must be made explicit. B has also a rapidly growing range of tool support in which our approach could be integrated.

We are convinced that an implementation of our technique would help the development of rigorous software testing techniques and have a positive impact on the use of formal software engineering methods.

6 Acknowledgement

This work was funded by a EU Human Capital and Mobility grant (number ERBCHB CT93 0328).

A Appendix: The Triangle Problem

An informal specification of the Triangle Problem [24] could be:

> The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

North's VDM-SL specification of the Triangle Problem [25], given below, is not intuitive nor straightforward and, in fact, has been devised to set a challenge to tests generation techniques.

Id.	Test Input	Oracle	Id.	Test Input	Oracle
1	$[]$	Invalid	20	$[100, 3, 5, 10]$	Invalid
2	1.42	Invalid	21	$[0, 0, 0, 0, 0, 0, 0]$	Invalid
3	$*$	Invalid	22	$[1, 5, 10, 9, 20, 45]$	Invalid
4	$[0, 70, 30, M]$	Invalid	23	$[1, 5, 10, 50, 8]$	Invalid
5	$[1, 1, 1]$	Equilateral	24	$[0, 0, 0]$	Invalid
6	$[42, 42, 42]$	Equilateral	25	$[42, 20, 62]$	Invalid
7	$[10, 4, 10]$	Isosceles	26	$[20, 23, 100]$	Invalid
8	$[5, 5, 9]$	Isosceles	27	$[-10]$	Invalid
9	$[10, 7, 8]$	Scalene	28	$[53.95, -78.9]$	Invalid
10	$[5, 10, 14]$	Scalene	29	$['H', 3]$	Invalid
11	$[5, 10, 8, 22]$	Invalid	30	$['A', 'Z', 'E', 'R']$	Invalid
12	$[2, 20, 5, 42, 41, 5]$	Invalid	31	$[*, 'D', *, 'J']$	Invalid
13	$[7, 10, 45, 20, 5, 8, 94]$	Invalid	32	$[4.56, 9, -3.14, 2.3, 13, 10.4]$	Invalid
14	$[0]$	Invalid	33	$[-4, 10, -3, 6, 42, 10]$	Invalid
15	$[50]$	Invalid	34	$[*, *, 9]$	Invalid
16	$[23, 23]$	Invalid			
17	$[10, 5]$	Invalid			
18	$[0, 0, 0, 0]$	Invalid			
19	$[3, 13, 4, 20]$	Invalid			

Table 1: Test Cases for the Triangle Problem

In North's specification, a triangle is specified as a sequence of 3 natural numbers, representing the lengths of the sides of the triangle, where, for the triangle to be valid, the double of each side's length must be less than the perimeter of the triangle. This is encapsulated in the invariant of the $Triangle$ type.

$$
\begin{array}{l}
Triangle\_type = SCALENE \mid ISOSCELES \mid EQUILATERAL \mid INVALID \\
Triangle = \mathbb{N}* \\
\quad inv\ Triangle(sides) == \\
\quad\quad len\ sides = 3 \wedge \\
\quad\quad let\ perim = sum(sides)\ in\ \forall i \in elems\ sides \cdot 2 * i < perim
\end{array}
$$

$$
\begin{array}{l}
sum : \mathbb{N}* \to \mathbb{N} \\
sum(seq) == \\
\quad if\ seq = [] \\
\quad\quad then\ 0 \\
\quad else\ hd\ seq + sum(tl\ seq)
\end{array}
$$

$$
\begin{array}{l}
variety\ :\ Triangle \rightarrow Triangle\_type \\
variety(sides) == \\
\quad cases\ card(elems\ sides)\ of \\
\quad\quad 1 \rightarrow EQUILATERAL \\
\quad\quad 2 \rightarrow ISOSCELES \\
\quad\quad 3 \rightarrow SCALENE \\
\quad end
\end{array}
$$

$$
\begin{array}{l}
classify\ :\ \mathbb{N}* \rightarrow Triangle\_type \\
classify(sides) == \\
\quad if\ is\_Triangle(sides) \\
\quad\quad then\ variety(sides) \\
\quad else\ INVALID
\end{array}
$$

Our systematically generated tests for the Triangle Problem as specified are given in Table 1. The actual test derivation process is detailed in [23].

References

[1] J.-R. Abrial. *B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0 521 49619 5.

[2] B. Beizer. *Software Testing Techniques.* International Thomson Computer Press, 2nd edition, 1990. ISBN 1850328803.

[3] F. Belina and D. Hogrefe. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems*, 16, 1989.

[4] J. Bicarregui, J. Dick, B. Matthews, and E. Woods. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29:53–78, 1997.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[6] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, pages 34–41, July 1995.

[7] British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), Version 4, April 97. *Standard for Software Component Testing*.

[8] S. Budkowski and P.Dembinski. An introduction to estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1), 1987.

[9] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[10] P.D. Coward. Symbolic execution systems—a review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[11] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*, volume 8 of *APIC Studies in Data Processing*. Academic Press, 1972.

[12] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93 Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag, 1993.

[13] Draft International Standard ISO/IEC JTC1/SC22/WG19 N-20. *Information Technology Programming Languages—VDM-SL*, November 1993.

[14] R. Elmstrøm, P.G. Larsen, and P.B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specification. *ACM SIGPLAN Notices*, 29(9):77–81, 1994. http://www.ifad.dk/.

[15] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Choosing a testing method to deliver reliability. In *Proceedings of the 19th International Conference on Software Engineering*, pages 68–78. ACM Press, May 1997.

[16] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[17] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[18] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[19] R. Hamlet. Special section on software testing. *Communications of the ACM*, 31(6):662–667, June 1988.

[20] C.A.R. Hoare. How did software get so reliable without proof? In *FME'96: Industrial Benefits and Advances in Formal Methods*, pages 1–17, 1996.

[21] H.M. Hörcher. Improving software tests using Z specifications. In J.P. Bowen and M.G. Hinchey, editors, *ZUM'95: 9th International Conference of Z Users*, LNCS. Springer, 1995.

[22] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International(UK), second edition edition, 1990.

[23] C. Meudec. *Automatic Test Cases Generation from Formal Specifications*. PhD thesis, Faculty of Science, The Queen's University of Belfast, 1998.

[24] G.J. Myers. *The Art of Software Testing*. Wiley-Interscience Publication, 1979.

[25] N.D. North. Automatic test generation for the triangle problem. Technical Report DITC 161/90, NPL, February 1990.

[26] M.A. Ould. Testing—a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, March 1991.

[27] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[28] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[29] A.S. Tanenbaum. In defense of program testing or correctness proofs considered harmful. *ACM SIGPLAN Notices*, 11(5):64–68, May 1976.

[30] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[31] H. Zhu and P.A.V. Hall. Test data adequacy measurement. *Software Engineering Journal*, pages 21–29, January 1993.