Automatic Generation of Software Test Cases From Formal Specifications

A Thesis Submitted in Fulfilment of the Requirements
for the Degree of Doctor of Philosophy
in the Faculty of Science
of The Queen's University of Belfast

by

Christophe Meudec

May 1998

Abstract

Software testing consumes a large percentage of total software development costs. Yet, it is still usually performed manually in a non rigorous fashion. While techniques, and limited automatic support, for the generation of test data from the actual code of the system under test have been well researched, test cases generation from a high level specification of the intended behaviour of the system being developed has hardly been addressed. In this thesis we present a rationale for using tests derived from high level formal specifications and then set to find an efficient technique for the generation of adequate test sets from specifications written in our study language, VDM-SL.

In this work, we formalise the traditional high level partitioning technique used in a previously researched test cases generator prototype, and extend it to take the semantics of VDM-SL fully into account. We then discuss, and illustrate, the shortcomings of the technique as used, which results in too few tests being generated and potentially large sections of a specification not employed by the test generation process.

Another strand of research, based on test generation from Z predicates, is examined and extended using our formalism to include quantified expressions and other, more complex, constructs of formal specification languages. We then show that this more refined technique complements the previous work and that their combination should allow the generation of adequate test sets from formal specifications.

Using our formalism, we illustrate that to synthesise pragmatically the two techniques, one must find heuristics for the detection of redundant test cases. We present our central heuristic, justified using probabilities, based on the independence of some divisions of the input domain of the system under test, which allows the contraction of test sets without impairing their likelihood of revealing an error in the system under test. Finally, we propose a technique for the efficient generation of adequate test cases.

Acknowledgements

I wish to thank Dr. Ivor Spence, my supervisor, for his help and patience during the course of this work.

I would also like to thank Mr. John Clark and Prof. John McDermid of the university of York for allowing me to pursue my work while being employed on one of their projects.

This work was partially funded in its first year by the SERC (grant number 92313295). It was generously funded by a EU Human Capital and Mobility grant (number ERBCHB CT93 0328) for two years. Prof. Maurice Clint and the department of computer science at Queen's at large, must be thanked for their efforts in obtaining this latest grant.

Finally, I am indebted to my wife, Tracey, and my son Mikaël for putting up with me during difficult times and encouraging me all the way.

To the memory of my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Using formal specifications to derive software tests may seem an odd idea. Many would think that, if a formal specification has been derived from informal requirements it was surely to allow proving of the program code, in which case proper testing could be side stepped.

It may seem paradoxical to use the product of a formal development process to help with, what must still constitute, one of the most informal and controversial phases in the software life cycle.

In this thesis we aim at removing this paradoxical element and demonstrate how automatic generation of software test cases from formal specifications complements existing software testing techniques as well as showing how it can be performed.

1.1 Software Testing: a Fuzzy State of Affairs

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 [1]:

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

Even the aim of software testing is not firmly established as many variations of its purpose can be found in the literature not to mention the industry—e.g.

is it to find as many errors as possible (using "special-values testing" [2]) or to inspire confidence in the program under test (using tests that mimics the "operational distribution of typical usage" [2])?. These confusions are mainly due to a lack of agreement on the place of testing in the software development life cycle and to the absence of a complete theory of testing.

There are various arguments against software testing. The principal one, sometimes considered as ultimate, is 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. [3] 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, accommodating it within the current formal framework advocated by many software engineers and academics may be more productive.

In doing so we may also remove some of the myths around both testing and formal development [4, 5]. 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 in the testing process would not at least clarify the situation and allow further progress.

1.2 A Gap to be Filled

As outlined, software testing taken at large is a fuzzy subject (see also [6]). For this reason the very existence of automated tests generating tools can be surprising. The testing activity is increasingly being automated. However, the crucial area of tests generation is still largely performed manually and, according to Ould [7], 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. Automated testing tools are valuable—even without a sound theoretical base—as they provide theoretical researchers in testing with much needed experience and data as well as systematising basic testing, [8, 9, 10] detail this view.

However after reviewing the tests generating tools in existence and the ad hoc techniques they originate from, one cannot fail to notice the gap, in terms of research interest and actual use, between structural tests generating tools—which derive tests from the code of the program under test—and functional tests generating tools—which derive tests from a specification. The latter usually require a significant amount of manual work prior to the automatic tests generation. This is due, in part, to having to write test scripts in non-standard notations.

In this thesis we will show how deriving tests from standard, high level, formal specifications can be performed. We will examine the work of North [11], review the technique presented in [12] in light of recent work by Dick and Faivre [13, 14] and Stocks and Carrington [15, 16] and finally present an original, automatic technique for test cases generation from formal specifications. As a side issue we will also try to show that this work may be of benefit to software testing as well as to formal development techniques.

1.3 Outline of Thesis

In Chapter 2 we conduct an extensive review of Automatic Tests Generators—or ATGs—and present their theoretical underpinnings. This, although not essential to the comprehension of the main results of this work, has been included to show the continuity and the necessity of the work presented within the testing process taken as a whole. It will also, it is hoped, provide a rationale for work undertaken.

Chapter 3 elucidates many related areas of concern when testing from formal specifications. It also introduces VDM-SL as our study language. North's case study [11] into the feasibility of testing from formal specifications is discussed and its example, the Triangle Problem, revisited. Some general remarks about the activity as a whole are then put to the reader.

Partitioning, the thrust of functional testing, is reviewed in Chapter 4. We build a new formalism for the systematic partitioning of VDM-SL specifications. We discuss in depth and synthesise the results of Dick and Faivre and, of Stocks and Carrington. Small examples are used throughout to illustrate the discussion.

In Chapter 5, the systematic partitioning of VDM-SL is reviewed and heuristics to reduce the number of tests generated without impairing the overall quality of the test sets generated are introduced. We also discuss several techniques which we have investigated for the automatic construction of reduced test sets. Test sets for several small examples of VDM-SL specifications are generated.

In Chapter 6, we discuss looseness and recursion with a view to integrate them in our approach. To conclude this chapter, a test set for the Triangle Problem as introduced in Chapter 3 is generated using our technique. This is followed by a discussion of the degree of adequacy achieved by this test set.

Finally, in Chapter 7, we outline the contributions made by our work and present several suggestions to improve the quality of the test sets generated using our technique.

Chapter 2

Previous Work in Automatic Tests Generation

In this chapter we introduce the many aspects of automated tests generation by examining theoretical results as well as existent testing tools.

We will first introduce some preliminary considerations which need to be addressed before examining automatic tests generation.

The following section introduces the theoretical background of general purpose automatic tests generators (ATGs). Categorising ATGs is not an easy task as many use mixed strategies. Nevertheless it is roughly agreed that ATGs can be categorised into random, structural and functional classes: this is the partition adopted here. The theoretical flaws and advantages of each strategy are presented. A complementary, i.e. more formal, approach to this introduction, can be found in [17]. Specific testing areas for which general purpose strategies are usually not considered suitable are also reviewed.

Some implementations of the general purpose strategies described in the theory section are introduced and discussed at length in the experience section. Although the presentation does not pretend to be exhaustive—that would be presumptuous—the examination of the most representative tools for each class of ATGs is sufficient to allow us to understand the many problems that still hinder their everyday use. Specific areas' testing tools are also presented.

We conclude with some suggestions for an improved ATG. More specifically the emphasis is on features that should make the tool widely acceptable to the *software engineering community.*

2.1 Preliminaries

A number of books and articles [18, 19, 20, 21, 22, 23, 24] introduce the testing process and diverse tests generation techniques. While Myers' book [18] is a classic book that shaped many of the current ideas on software testing, it needs to be supplemented with one of subsequent effort by Beizer [19], or Hetzel [22] for example, to get a more up to date picture of software testing. Gelperin and Hetzel in [8] describe the evolution of views on the testing activity from its origin to the present time.

The British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST) has recently released a standard for component testing [25]. Component, or unit, testing is concerned with the testing of individual software items for which a separate specification is available. The standard prescribes characteristics of the test process and describes a number of techniques for test case design and measurement. Many of the automatic tests generators we will see use these techniques. A glossary of terms used in software testing has also been produced [26]. Together, these documents represent a remarkable improvement on the situation prevailing when the standardisation process started in 1989 when no standard was available as to how the testing of components should be performed.

Note that we will also in this work concentrate on component testing rather than on integration or system testing (testing performed when the components are put together to form a coherent system). Component testing has traditionally received the most attention from tool developers.

Also as is usually the case, we will limit the scope of our discussion of automatic software testing to dynamic testing. Dynamic testing requires the execution of the program under test with data, as opposed to static testing. Static testing (see [27] for a practical, established approach) includes program proving and anomaly analysis—also called data flow analysis.

Finally, and to conclude these preliminaries, we should discuss the primary aim of the testing process. As hinted in the introduction of this work the primary aim of software testing is not always clearly defined. We give the two main variations:

- the aim of defect testing is to find as many errors as possible in the program under test (using "special-values testing" [2]).
- 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 [2]).

It is only very recently that theoreticians have clearly separated the two practices [28]. In their article, "Choosing a Testing Method to Deliver Reliability" [28], Frankl, Hamlet, Littlewood and Strigini have clearly shown that the two testing practices above are distinct. In particular, they have highlighted the main shortcoming of defect testing: informally, defect testing may not lead to reliable software if the inputs that precipitate any of the remaining erroneous behaviour in the software under test occur often in the operational distribution of typical usage of that software. Thus detecting as many defects as possible in the software under test may not be the most efficient way of increasing the reliability of the software delivered.

There are however problems with operational testing. For example, a highly reliable piece of software may still be unacceptable if the remaining defects have catastrophic consequences. For example, a software package which contains an error that does not occur very often, but which causes the formatting of the hard drive of a personal computer would be unacceptable. This highlights the problems of reliability measures: they do not take into account the seriousness of the errors. Further, errors which may be easily revealed using defect testing techniques may have a very low likelihood of detection using operational testing. This is simply because the inputs leading to the errors may not appear often in the distribution of typical usage.

Finally, and the main reason as to why we will in the remainder of this work concentrate on defect testing is that, it is difficult, using operational testing, to find the tests that should be applied to the software under test. This is due to the difficulty of knowing in advance the input distribution of typical usage. In particular, no automatic tool can discover this distribution of typical usage.

Hence, in this work our objective will always be to reveal as many defects as possible in the software under test. However we should always keep in mind the shortcomings of defect testing as highlighted in [28]: primarily it may not always be the most straightforward method for increasing the reliability of the software under test.

2.2 Theory

In this section we will briefly examine the theoretical background of general purpose tests generators. By general purpose we mean generators able to test programs for which specific strategies of testing do not seem necessary. Testing areas for which it is usually deemed unsuitable to use general purpose tests generators include user interfaces, object oriented systems, real-time systems and concurrent systems. Those testing areas are examined last. Testing of knowledge based systems is more concerned with the validation of the knowledge stored in the data base than with the validation of the implementation of the system and is therefore not examined here.

Complementary explanations can be found in [29] or more consistently in [23]. The strategies presented here have all been at least partially automated.

2.2.1 Overview

The ultimate goal for ATGs is to perfectly implement testing strategies that verify criteria stating when enough testing has been performed. These criteria are formally called test data adequacy criteria. Several teams are currently working on deriving such criteria [30, 31, 32].

Zhu and Hall's [32] research seems particularly promising because of its generality but their criteria might prove to be too formal to be taken into account in the design of ATGs. Their work also leads to the notion that adequacy criteria should not be stop rules (i.e. the test sets are either good or bad) but functions from specifications, programs and subsets of data into a degree of adequacy over the real interval $[0, 1]$. We will retain this definition as it allows us to discuss the relative adequacy of various test sets derived from the same problem.

To date however, no definite practical criteria for test adequacy have been found. Therefore, testing strategies—needless to say ATGs—try to reach an imperfect goal. This problem—of not having a sound theoretical base—puts in doubt the efficiency of any testing strategy and consequently any ATG. For this reason, at least, further research in the test data adequacy area is needed.

So, as testing strategies cannot be evaluated against formal criteria it is tempting to try to compare them through the use of mathematical models.

Unfortunately, it has been found difficult [2, 33, 34] to construct mathematical models for the different testing strategies available—even for random testing—that could be used to assess their performance. The human factors involved in the software development process are partially to blame as they are, indeed, difficult to formalise. It is hazardous to construct such models because one cannot argue that an error will not be revealed by a given input: it partially depends on the team that developed the program and not entirely on the supposed ill foundation of the input chosen. The models developed so far are consequently grossly imperfect and unstable in that minor variations in their parameters—those parameters are set subjectively—result in huge differences in terms of efficiency for the strategy they are supposed to model.

Further research in this area is obviously needed if a theoretical means to evaluate the degree of adequacy of the test sets generated by a particular strategy is to be found. Intuition and loose arguments are still the most common means of evaluation of testing strategies despite the availability of mutation analysis techniques.

Mutation analysis can be valuable for evaluating the efficiency of a test set in finding errors in a program. This method is not a universal solution for comparing testing strategies—as the mathematical models aim to be—as the evaluation is only based on the test sets generated for a finite number of programs. [23, 35] present this technique. Mutation analysis uses the program to evaluate the adequacy of the test data. By performing minor modifications—mutations—in the program under test many mutants of the program are created. To evaluate the adequacy of a particular test set, it is sufficient to execute the set of mutants against the test set and count the number of mutants detected. This process requires many executions of every mutant created and is therefore extremely time consuming. Choi and Mathur [36] use MIMD architecture to perform mutation analysis in the hope of reducing the execution time. The same objective has been pursued by Krauser et al. using SIMD architecture [37]. However mutation analysis is intrinsically demanding on computational power.

Various testing strategies have been compared using mutation analysis for example in [38]. This small study, based on only one program, turned in favour of constraint-based testing—see the part devoted to structural testing. However, this result should not be surprising as the goal of constraint-based testing is precisely to score highly on the mutation adequacy criteria. Mutation analysis is not, and should not be considered as, the ultimate decision technique for ruling which testing strategy is the most effective at generating high performance test sets. We could even argue on the significance of the results of mutation analysis for strategies other than structural strategies.

Nevertheless mutation analysis is a valuable technique especially because, as seen, no theoretical result at the moment allows the human tester to evaluate the efficiency of an automatically—or for that matter, manually—generated test set. More experimental results, using tests produced for more than one program, would be useful to theorists as well as to designers of ATGs.

Use of coverage analysis methods—for example to measure the proportion of the program under test that has been executed by a given test set—to evaluate the adequacy of the test generated is common. These methods are even more controversial than mutation analysis as the significance of the results is not fully understood but have the advantage of being easily automated and of having low overheads.

In [39] is described a benchmark framework for the evaluation of software testing techniques. In effect, faced with such low level theoretical results and no agreed experimental means to evaluate testing technique efficiency one is left to consider establishing a rigorous repository of correct and incorrect software along side test sets generated from different techniques. This huge undertaking would be invaluable to practitioners and academics alike since it would allow, for the first time, a classification of software techniques based on empirical scientific studies to emerge. It is hoped that it would also firmly anchor software testing within software engineering.

2.2.2 Random Testing

The random testing strategy generates a test set by randomly selecting inputs from the input domain description of the program under test. This form of random testing is purely automatic.

A more refined form of random testing is to use a probable operational input distribution to generate the test set as discussed in the preliminaries of this chapter. A probable operational input distribution, or typical usage, is however difficult to obtain prior to software delivery and even then difficult to agree upon: automation for the actual generation of tests is difficult to envisage under this scheme.

Random testing is intuitively the poorest strategy of selection of inputs to test a program. Early books on software testing [18]do not discuss it at all other than to dismiss it.

A study by Hamlet [2] on the efficiency of random testing tends to dismiss this intuitive view. It follows a report from Ntafos [33]. Partition testing strategies—see below—have been examined in comparison to random testing. In [34] Weyuker reviews the results of Hamlet and clarifies them. Where an experimental approach was used by Hamlet and Ntafos, Weyuker uses an analytic approach. Her results are more in tune with the intuitive feeling that random testing is of poor value.

In the absence of a real consensus about the efficiency of random testing one is obliged to accept that at least in particular areas—especially where other strategies are too complex to implement—random testing is an acceptable strategy.

As seen, a recent article by Frankl et al. [28] somewhat clarifies this debate by focussing on operational testing (rather than pure random testing) and demonstrating how it may well be the most appropriate testing technique for achieving high reliability. This clarifies how random testing can sometimes be seen as helping to gain confidence in the program under test [40].

2.2.3 Structural Testing

Structural testing strategies generate a test set by analysing the code of the program under test.

ATGs based on a structural approach are primarily pathwise generators. Here we are only considering pathwise generators for programs in an imperative, sequential language as they are the most common. Specific pathwise generators are described in section 2.3.4 which is devoted to specific areas of testing. Pathwise generators may, or may not, use symbolic execution as described in the experience section. The ideal criteria should ensure that all possible paths the program under test could go through are traversed. Because of the astronomical number of possible paths and the vast number of infeasible paths this strategy is impractical. Therefore, weaker criteria have to be adopted such as: all statements, all branches or all linear code sequence and jumps in the program under test should be executed at least once.

There are many other similar criteria. In order to fulfill one or several of them, one should generate all possible paths in the program by examining its code and then generate test inputs in accordance with the criteria chosen. However, the problems raised by the vast number of potential paths and by the checking of their feasibility still hinder these strategies.

A radically different strategy for generating tests based on mutation analysis has been developed by R.A. Demillo and A.J. Offutt [41, 42]. This strategy—constraint-based testing or CBT—uses algebraic constraints to describe test cases designed to find particular kind of errors in the program under test. Mutations are first introduced in the program under test (in a similar fashion to mutation analysis) while algebraic constraints specifying the error introduced in the program are generated. By solving these constraints, tests are created which will detect, by construction, the errors introduced in the program. The original program is then tested using the tests generated.

This strategy, although interesting, has not received sufficient attention yet:

it needs to mature and to spread to other researchers to be fully judged. It is also very computationally intensive despite the works described in [36, 37]. Woodward in [43] describes the various variants of mutation testing and introduces most of the current research in this area.

Along with random testing, structural strategies do not provide any prospect for an automated oracle—that is, means to determine the correctness of the test results obtained. Generators based on a black-box strategy generally provide a means to implement oracles.

2.2.4 Functional Testing

ATGs using black box strategies, of which random test generators are a degenerate form, do not require the code source to be examined for generating tests. Instead tests are derived solely from a specification—modelled from the system requirements—of the intended behaviour of the program or from test scripts describing loosely which tests should be generated. They do not take into account the internal structure of the program, indeed functional testing strategies are implementation language independent.

ATGs using test scripts, e.g. [44], are considerably simpler to devise than ATGs using specifications since the tests requirements are explicitly stated. However, test scripts have to be manually derived using a specific notation and are only useful for the generation of test sets. In comparison, specifications can be useful at every level of the software life cycle. ATGs using scripts, therefore, impose a significant extra burden on the human tester when compared with ATGs using specifications.

We therefore limit our review to ATGs using a specification of the behaviour of the program under test as other tools do not in fact automate the entire tests generation process.

The specification can take several forms. It can be a highly abstract mathematical expression—e.g. in VDM-SL, or in an algebraic or axiomatic form—a Finite State Machine (FSM) etc. How the specification of the program is handled to obtain a test set is closely related to the style of the specification. For highly abstract mathematical specifications, for example, a form of equivalence partitioning is usually applied.

Many test selection methods based on deterministic FSMs have been developed: it is indeed the area of testing where the theory is the most developed and many useful theoretical results exists. We cite the W-method [45], the Unique-Input-Output (UIO) method [46] and the Wp (or partial W method) method [47]. These methods are all based on identification of a number of fault models. One way to define fault models is by specifying a set of implementations with erroneous behaviour (mutants) that should be detected by a test suite. Based on the set of mutants considered, the methods allow derivation of test suites that will completely identify (or kill) all the mutants. Typically the mutation types considered for FSMs are output faults (the output of a transition is wrong) or transfer faults (the next state of a transition is wrong). Without extensions however, FSMs have a limited modelling ability: only the control aspect of systems can be specified. Some formal notations that extend the usefulness of FSMs as a specification technique are: LOTOS [48], Estelle [49] and SDL [50].

Another area of current research is focused on partitioning strategies for high level mathematical notations. Partitioning strategies divide the input domain of the program under test into sub-domains whose points are somehow the same. That is, it should be sufficient to select one member of a particular sub-domain 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.

However, test data adequacy criteria are not as evident to find as for structural testing. Test set size is also an important concern and techniques to reduce the size of the test sets generated while respecting the chosen criteria have to be devised. More about the theory of partitioning can be found in [2, 34].

The mathematical definition of a partition adapted to software testing is given below.

Definition 2.1 *Partition, Equivalence Classes*

Given the definition domain D of a predicate, a partition of D is a set of definition domain subsets, formally called equivalence classes, that are nonempty, disjoint and whose union is D itself.

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.

In general, tests generated using black box techniques will need to be transformed before their execution is possible: from an abstract test specification representation to the particular representation used in the implementation language of the program under test. This is not the case for tests generated using white box techniques.

But, as the program behaviour is often—always when using a formal specification of the behaviour of the program, e.g. under the form of a graph representation, an algebraic specification, etc. —entirely contained in the specification, it is the best strategy for providing the expected output, that is the generation of test cases. It might be impossible to obtain the expected result of a computation from model-based specification languages. This is primarily because these languages are often non executable as they allow implicit as well as loose specifications to be constructed. However, as we will see in the next chapter, it is possible with such specifications to ensure that the result computed is consistent with the specification: this is sufficient for our purpose.

Furthermore, because black box testing is code independent, it allows the tester to use the same tool whatever language to implement the program to test has been used. Other advantages of black box testing are described in [51, 52].

2.2.5 Specific Areas

The testing strategies described above are principally applicable to programs in an imperative, sequential, language. Testing strategies for these programs have been, historically, investigated first as their testing requirements are better understood. However, research is also being carried out on specific areas of software testing. Arbitrarily we have gathered under the heading specific areas the testing of user interfaces, object oriented systems, concurrent systems and real-time systems. Although testing strategies for these systems often share a large part of their contents with mainstream testing strategies they usually pose specific problems. It is those specific problems we will examine now.

User interface systems are difficult to formally specify—although research is active in this area—and structural testing is intuitively inefficient for these systems. Therefore the testing strategies used for these systems are mainly random and statistical strategies.

It should be noted that structural testing strategies, being language dependent, are sometimes tied to so called specific testing areas. For example, to construct a tool to white box test C++ programs it is necessary to identify the features inherent to object oriented systems—i.e. inheritance and message passing.

For functional testing, however, it is theoretically sufficient to obtain a specification of the program under test to apply one of the mainstream functional strategies. For example, object oriented systems can be tested using functional testing: whether the specification takes into account the specific programming style of object oriented systems is theoretically irrelevant. Similarly, the functional testing of concurrent systems can be performed using specifications that do not specify the concurrent aspect of the system.

There are however specialist formal notations for object oriented systems (see [53] for example which is an extension of Z, or [54] for an effort based on VDM-SL) or for concurrent systems (CSP [55], VVSL [56]). However these notations have not, to the author's knowledge, been examined from a tests generation point of view.

When considering real-time systems one is confronted with non-functional requirements such as the constraining of the worst case response time—WCRT. While static analysis techniques can be applied rigorously to deal with the inherent concurrency of most of those systems [57], they rely on an accurate worst case execution time—WCET—of the sequential code. An automatic testing technique and tool for determining the WCET of sequential code are currently being developed at the University of York, UK.

As seen, most of the specific areas of testing reviewed above are only in their infancy. Nevertheless the volume of research in these areas is growing rapidly offering hopes for future development.

2.2.6 Theory: A Conclusion

Every testing strategy has its own advantages and flaws. For example, if white box testing cannot detect missing functions, black box testing cannot detect unused code, etc.

It is now accepted by most that using combined strategies is the best way to achieve high quality testing and that no existent strategy is sufficient on its own: a variety of strategies should be applied when constructing a test set.

Even random testing can be considered attractive because it is a testing strategy more suitable for inspiring confidence in the program under test than any other strategy [2, 40, 28]. Testing performed with random input values is more reassuring for the human tester than testing carried out with special values; as in the boundary analysis strategy [18]—which is a refinement of partitioning strategies—for example. This notion of confidence in the program tested is important if ATGs are to be accepted and trusted.

It should be pointed out that if automation of the test set generation process is a valuable objective, it should nevertheless also encompass the generation of the expected output for a given test input—or at least a means to ensure the consistency of the computed result. Only from the specification of a program can an automated oracle be devised. See [58] for a detailed example.

2.3 Experience

This section briefly presents existent ATGs. The emphasis is on the weaknesses of the systems. Details of the implementation and of the strategy used for each system are not described here.

While automatic software testing is an old idea few ATGs have actually been implemented. Far less, if any, are in use everyday [10]. A summary of the principal reasons for this poor use is presented in the concluding section. Some qualities that an ATG should exhibit to be acceptable by the software engineering community are also presented. The sketch of a new system having some of the qualities cited is presented.

2.3.1 Random ATGs

Random ATGs: An Introduction

Random ATGs are rarely cited in the literature reflecting that little applied research has been carried out in this area. The lack of intuitive attractiveness of the method is probably why random testing is more an area of theoretical research than of applied. But, because of their intrinsic simplicity and the huge need for automatic testing tools in certain areas, it is reasonable to suppose that many have been implemented in the industry on a case by case basis to test particular products.

To the author's knowledge there is no general ATG reported in the literature. Such generators would need a precise description of a program's input domain as source of knowledge to generate a purely random test set. The test set could be generated following a chosen specific distribution—operational distribution for example—to reflect the future everyday use of the product. The more often an input is likely to be entered by a user, the higher is the probability that it will be chosen by the generator. This kind of information is not easy to obtain prior to the product release after which, of course, testing is of little value.

Random ATGs: Description

In some areas, random ATGs seem to encounter a wider use than in others. In the area of compiler testing, they are appreciated for their relative simplicity of implementation.

Bazzichi and Spadafora in [59] describe an ATG that randomly test Pascal compilers. A syntactical description of the source language is needed to generate compilable programs. The formalism utilised is a context-free parametric grammar that permits testing of the context sensitive aspect of the programming language. Writing a context-free parametric grammar for a given language is, as described by the authors, certainly not trivial. Thus, the process of automation relies heavily on the tester ability to write a correct description of the language.

Another compiler test generator is a PL/1 test generator developed at IBM and described in [60]. This random generator can be tuned to test only known weak areas of the compiler. A system of weights is introduced to direct the generator to those statements which should be selected more often. The programs generated are executable.

This generator is very specific to the language processed by the compiler: large parts would have to be rewritten to test compilers for another language.

[60] introduces two other random ATGs for two specific systems: a graphics package and a sort/merge program. These random tests generators have a very limited scope of application. This is in accordance with their authors' beliefs that a tests generator should be specific rather than general purpose. These generators have been developed following principles for the design of random ATGs. However these principles are not described. Whatever principles are applied for developing random ATGs the cost of development is still higher than using a ready to use general random ATG.

Random ATGs: A Conclusion

Random test generators are easier to construct than their functional and pathwise counterparts. The method of selection for the input is random so a random number generator is all that is needed at the heart of the selection process. This simplicity is attractive. Doubts about the potentiality of a randomly chosen test set to find errors in a software are high. So, it seems that random tests generators are not trusted enough. Many are developed only because it is the easiest method of test set selection to implement.

In the absence of a simple technique to test particularly difficult software, e.g. compilers, some testers have chosen a random approach. They have not done so because randomly chosen sets of inputs have been proved to be a good testing strategy for the area concerned.

Because of the intuitive poor value of the theory underpinning the random testing strategy, the tester is tempted to use a random tests generator heavily on a given program ending up with a large number of outputs to check. The need for an automatic oracle is therefore more acute for random testing than for any other methods.

General tools—using high level notations to specify the input domain of the system under test—in place of case by case tools as developed at present could widen the use of random test generators. Whether this would be a good thing for testing in general is, surprisingly, still an open theoretical question [2, 34, 40].

2.3.2 Structural ATGs

Structural ATGs: An Introduction

Test generators based on a white-box strategy or structural ATGs, as they are sometimes called, are undoubtedly those, among ATGs, which have encountered the most success and the widest use in the industry. After random strategies structural strategies are the most intuitive means to test a program. It is also an area where automation is well advanced.

There is a wide range of white box strategies. Most structural ATGs are pathwise generators with the exception of generators using Constraints Based Testing which will be examined last. [18, 20, 21, 22] are introductions to pathwise generators: they only differ by the means by which they overcome the path revelation problem and by the criteria they apply to generate tests.

Symbolic execution is the most widely used means of generating path traversal conditions.

Structural ATGs: Description

In [61] Coward gives a good review of systems using symbolic execution to generate the different paths that a program can traverse. We illustrate on a small example how symbolic execution can help for the generation of tests.

Given the following fragment of a Pascal-like program, where `x` and `y` are input variables and `z` an output variable.

```
if x > y then x := x - y;
         else x := y - x;
z := x*2;
```

A symbolic executor would for every control path in the program express the output variables in terms of the input variables and constants. A path traversal condition is also generated in terms of the input variables and constants for every path: an input will exercise the path if this condition is satisfied.

On our example above there are two paths:

1. Path condition: $x > y$; Result: $z = (x - y) * 2$

2. Path condition: $x \leq y$: Result: $z = (y - x) * 2$

A test set to exercise, for example, every path in this program can be generated by finding, for every path traversal condition, inputs which satisfy the condition. In our example the tests: `x = 5, y = 2` and `x = 10, y = 20` will exercise the 2 paths in the program above.

However, several problems occur when using symbolic execution:

- the evaluation of loops.

- module calls.

- array references.

- the feasibility of a path.

They are illustrated in [61].

In [62], techniques used in Casegen to try to overcome some of these problems are described.

In [61], three criteria are given for a testing system to be said to use symbolic execution, they are:

- it produces a path condition for each path traversed.

- it determines whether a path condition is feasible.

- for each output variable it produces an expression in terms of input variables and constants.

Six systems described in the literature pass Coward's criteria, they are: EFFIGY [63], SELECT [64], ATTEST [65], CASEGEN [62], IPS [66], Fortran Testbed [67, 68].

Although these systems make use of symbolic execution the level of automation they achieve is quite uneven.

A system which only uses symbolic execution to generate the set of paths a program can traverse but which does not satisfy Coward's criteria can be said to be unsuitable for automatic generation of test inputs.

Only Casegen, is described here.

The Casegen [62] system consists of four components:

- a Fortran source code processor.
- a path generator.
- a path constraint generator.
- a solving constraint system.

The path generator uses the information produced by the Fortran source code processor—i.e. a flow graph, a symbol table and a representation of the source code—to generate a set of paths to cover all branches. The path constraint generator then produces a path condition for each path. The solving constraint system generates values to satisfy a particular condition. The strength of Casegen is probably its full coverage of the Fortran language. Nevertheless, it does not seem very reliable as in [62] `(26,7,7)` were the values generated for an isosceles triangle. This does not inspire great confidence in the value of the test generated by Casegen.

SYM-BOL [69], developed by Coward, is an attempt to improve symbolic execution techniques to avoid these problems. It is also an attempt to generalise pathwise generators—which are primarily intended for numerical software written in Fortran—to commercial systems written in Cobol. SYM-BOL uses a path condition which is a list of constraints representing the feasibility of a path. Every common problem encountered by pathwise generators—as presented above—are discussed in Coward's article and the strategy adopted for SYM-BOL is described. This tool represents a new wave of interest in pathwise generators and should be closely examined by structural ATGs designers. Although the full Cobol language is not accepted by the tool, SYM-BOL offers good hopes for future developments.

Korel [70] describes an automatic test data generator based on the actual execution of the program under test, dynamic data flow analysis and function minimisation and is therefore not related to the symbolic execution approach. This generator therefore, does not encounter the problems associated with symbolic execution. The basic operations of Korel's generator are: program control flow graph construction, path selection, and test generation. All steps can be automated. While Korel's approach is not problem free—limited ability to detect infeasible path, simplicity of the language for the program under test (subset of Pascal)—it requires further research to be fully judged. Korel in [70] presents several interesting possible areas of further research for his technique.

SMOTL [71] is another system of this kind. It conducts its path analysis while maintaining maximum and minimum values for each variable.

Constraint-based testing—CBT—has been implemented [38] for Fortran 77. However, this is such a new testing strategy that it is difficult to evaluate it. One of the problems with this method is the difficulty of eliminating redundant or ineffective tests.

Structural ATGs: A Conclusion

Pathwise ATGs are promising software development tools even if after nearly two decades of research in this field several technical problems still hinder their general use. These problems lie more in the generation of the possible paths, as illustrated by the symbolic execution approach, than in the actual selection of a test set that satisfies given criteria. CBT is a very promising strategy, however to make it more efficient would require the implementation of an oracle for which the method does not cater.

2.3.3 Functional ATGs

Functional ATGs: An Introduction

Automatic test generators using black-box strategies, extract the information they need to generate tests from the specification of the program or from a test specification. They differ mainly by the type of specification they use and by their sampling method.

Most functional ATGs oblige the tester to write a testing oriented specification. Although these oriented specifications could theoretically be used as a program's specification on their own, this is rarely done. These specifications are too constraining for the general purpose of program specification and sometimes too difficult to write.

Functional ATGs: Description

The statistical approach where particular requirements to be tested are identified first and then filtered through statistical considerations concerning operational use and risk factors has been investigated [72]. Unfortunately this method offers no room for complete automation as the statistical values assigned to every requirement to be tested are largely subjective. Because of theoretical results [28] operational testing deserves further attention.

The AGENT system [73] uses a function diagram, which is an extension of a cause effect graph [18], to generate test cases. A cause effect graph is a diagrammatic notation for expressing logical statements such as *if A and B then C*, *if A or B then C*, *if not A then B* etc.

A function diagram is composed of a state transition diagram and of a set of Boolean functions which are specified using a cause effect graph or a decision table. The tests automatically generated by AGENT satisfy fairly natural criteria defined upon the function diagram:

- they validate input and output conditions in all states.
- they pass all transitions at least once.

Other graph models have been considered, such as automata [47] and Petri nets [74], to generate test sets employing graph coverage criteria.

While these systems have the advantage of having well-defined criteria for test set selection, they all suffer from the difficulties one might be faced with when attempting to write the required appropriate function graph for a complex algorithm. Due to this difficulty it is hard to see these systems being widely used in the future. The generation of the expected output is generally straightforward due to the low level of abstraction of the graphs.

Tsai et al. [75] describe a system that automatically generates test cases from a relational algebra query. Relational algebra was chosen because it is often used in database and in data processing applications.

The efficiency of the system is compared to random testing. It seems efficient and reliable although further investigations would be needed to fully assess its real performance in finding errors. The small scope of relational algebra as a specification language is the weakest point of this system and limits this system to database testing.

FSM-based testing has received a lot of attention in relation to the testing of communication protocols. Communication protocols are rules that dictate how different components within a distributed system communicate. FSMs with extensions are often used for protocol specifications.

The rigorousness required when testing protocols is high and most ISO protocols are tested using standardised test purposes scripts.

We mention some results from the area of tests generation from LOTOS specifications. LOTOS [48] is a formal description technique particularly suited to protocol specifications. We mention in particular the early work of Pitt and Freestone [76] which considered laws for constructing tests from LOTOS specifications adapted from ad-hoc manual practices. However, the LOTOS subset considered was small (it excluded LOTOS expressions containing data parameters) and no automatic tool was constructed.

An automatic tool for the derivation of test cases for LOTOS expressions with data parameters has been constructed by Li, Higashino and Taniguchi [77]. The kind of expressions allowed are restricted so as to allow automatic decidability using simple linear integer programming. Their strategy is based on the transformation of a LOTOS specification into a transition tree representing its behaviour up to a static level of depth. From this description, test cases can be generated and unexecutable transitions, deadlock as well as nondeterministic branches can be detected. Their work however does not seem to have been widely reviewed by other researchers—maybe because of its late English translation—despite its promises.

For completness we also mention results by Ashford [78] which, using Prolog as specification language, allow automatic testing for a subset of class IV protocols.

Jalote [79] describes the SITE system that is able to generate test cases from the axiomatic specification of an abstract data type. The oracle is provided by an automatically generated implementation which reflects the behavioural properties of the specification. The test set is generated from the syntactic part of the specifications. All the valid expressions that produce instances of the abstract data type, up to a given maximum level of depth, are generated. Tests are then generated by applying the different behaviour operations on these instances.

This approach has several distinctive problems, its limited scope of application, the difficulty one can encounter in writing axiomatic specifications for complex data types, the lack of theory underpinning the method of test selection and the large number of tests generated. Nevertheless this system, because of its high level of automation, is interesting. Unfortunately it is difficult to see any enlargement of the scope of application for systems like SITE. Axiomatic specifications cannot be a general means to specify complex algorithms from various areas of software development and oracles provided by automatic generation of an implementation from a specification are still confined to simple problems or highly restricted areas and so cannot be generalised.

In [52] is formally described a method to construct test cases from formal specifications. This theory is applied to algebraic specifications in a system which constructs test cases. Despite the case study in [80] this approach is again limited by the difficulty of writing algebraic specifications for general purpose programs. However, the technique described is formal enough to hope for further interesting developments, maybe in the domain of model based specification languages.

Hoffman and Strooper [44] introduces some tools and techniques for writing scripts in Prolog that automatically test modules implemented in the C language. As the selection method for generating test sets is not automated—i.e. left to the human tester—one cannot consider this set of tools as forming an ATG.

Ostrand's approach [81]—category partitioning—does not pretend to be directly automated even if automation of the method can be envisaged as has been investigated by Laycock [82]. As remarked by Laycock the notion of category in Ostrand's approach is too informal to hope for an immediate implementation, therefore, this method is not discussed here.

We mention the work of Chang, Richardson et al. where in [83] a tentative method for test conditions generation from ADL assertions is presented.

Stocks and Carrington [15, 16] present a formal framework for test generation from Z specifications. Unfortunately automation is not discussed, but some of their results will be reviewed later.

Recent work from Dick and Faivre [13, 14] is promising as a tool has been built which generates test inputs from VDM specifications. This work will be discussed at great length later.

Functional ATGs: A Conclusion

As seen, recent research has demonstrated that black box testing can be a practical technique. The difficulty in assessing the actual systems lies in judging, fairly, the level of automation achieved and, of course, the efficiency of the systems in finding errors in a program. It would be a great advance in black box testing tools to be able to process already established specification languages—i.e. used in other areas of software development other than testing. If languages such as VDM-SL [84] or Z [85, 86] could be processed then the burden of writing specific specifications for testing purposes would be lifted thus, greater automation would, in a way, be achieved.

2.3.4 Specific Areas

Softbridge, Inc. [87] has developed commercial tools for the testing of OS/2 and Windows applications graphical user interface. These are based on three main characteristics:

- they allow storage and manipulation of user sessions.
- a testing language is provided to test specific situations.

- basic facilities for helping in the validation of the results are provided. (such as result logs, and automatic comparison with a baseline case in which the application performed correctly)

Other tools, such as [88, 89], are based on the same facilities.

These tools facilitate the testing of interfaces by automating many repetitive tasks which would otherwise have to be performed by a human tester.

Some general considerations when testing object oriented systems are reviewed by Graham et al. in [90] and by Barbey et al. in [91]. Although tools exist, for example Cantata [92], to white box test object oriented systems they are based on ad hoc techniques: the specific problems posed by these systems have not yet been systematically studied.

It seems particularly difficult to generate specific test sets for real-time systems. Formally specifying these systems is notoriously difficult despite the development of real-time CSP. And, although structural testing strategies are being specifically developed for determining the worst case execution time of sequential code, research in this area is only just beginning. However, as the current practice is to randomly test the code, using very large test sets devised for the validation of the functional properties of the system, and measure the execution time, research in this area should now be ongoing. Hence, no specific strategy for automatically testing those systems is available at the moment. Glass [93] presents a survey of the problems associated with the testing of real-time systems.

The problems of structural testing for concurrent programs are examined in [94, 95]. Path analysis developed for sequential programs cannot be applied to concurrent programs as the reproducibility of the program execution is not ensured. Yang and Chung's approach [94] is to construct two graphs to model the execution of a concurrent program: a program flow graph and a program rendezvous graph. These graphs are then combined to obtain a concurrent path model. The authors, in their article, also discuss the problems of test path selection, test generation and test execution. Korel et al.'s approach [95] is roughly similar. Their starting point is the TESTGEN ATG for sequential programs which they modified to cater for concurrent programs. Both works are based on a subset of Ada. Research in this area is fairly recent—apart for some isolated early works from Brinch Hansen—and need to be pursued for the implementation of a general structural ATG for concurrent programs to be possible. AdaTEST [96] is a testing tool for Ada programs where the human tester must write test scripts in Ada, these test scripts are then analysed by the tool and tests are applied to the program under test. This tool is simply a help to the testing of Ada programs but cannot be considered as an ATG.

2.3.5 Experience: A Conclusion

As seen there are a wide variety of ATGs but they all suffer serious deficiencies. We can summarise the problems of ATGs by the following:

- their efficiency is difficult to assess.
- the level of automation achieved is generally not high enough.
- they are all based on a single strategy.
- their scope of application is usually limited.

These problems are briefly discussed below.

To evaluate the efficiency of ATGs, short of any theoretical means, mutation analysis is the only technique currently available. Unfortunately, the actual significance of the measures produced by this technique is not fully agreed and implementation problems remain. Unless a breakthrough in the test data adequacy area emerges it is difficult to imagine any improvement of the situation in the near future. Designers of ATGs can only aim to faithfully implement formally described criteria in the hope that mathematical models will be able to represent them more accurately.

The human tester has still, in the majority of cases, to intervene to help with the execution of ATGs. Efforts must be pursued in that area to lift the burden of basic testing from human testers.

Software testing environments should be created to gather, in a friendly manner, various consistent testing strategies. Using a multitude of tools in the testing process is tedious: a single testing environment would democratise the use of ATGs.

The scope of application for ATGs can be extended for each of the three mainstream strategies presented. Random ATGs could be made general instead of specific as at present. Structural ATGs should accept a complete language definition and not subsets as it is usually the case at present. Functional ATGs should use high level abstraction languages instead of specially developed mathematical notations.

2.4 A Conclusion on Existent ATGs

As we have seen, current ATGs have not fulfilled engineers' expectations. The lack of theory underpinning such tools is partially to blame for this deception. Nevertheless, the author believes that better automatic testing tools could be developed even in the frame of the current testing theory. [10] details this view.

To generalise the use of ATGs the problems described in the conclusive part of the experience section must be addressed. The construction of a testing environment depends on the availability of distinctive, trusted, testing strategies. Unfortunately, none of the three mainstream testing strategies has yet reached a stage where full automation is achieved and the strategy trusted. Therefore, one is limited to concentrate on one strategy while hoping for further improvements in others.

In the area of functional testing the development of stable specification languages such as Z [85, 86], and VDM-SL [84] offers a good platform for the implementation of a multitude of supportive tools. Further, their current standardisation [97] should re-enforce this development. Software testing tools should benefit from this standardisation. As detailed in the next chapter, North [11] showed how manual test generation from VDM-SL is feasible.

It would be a great advance if an efficient way to test software could be designed using such high level specification languages. A means to determine the consistency of the computed results could be provided from such specifications. Of course, test data are difficult to generate from functional strategies: this is why a combined approach should, in the future, be of great value.

Finally, we should bear in mind that, as stated in [17], human competence and ingenuity remain the sine qua non condition for the success of the testing process.

Chapter 3

Testing From Formal Specifications

In this chapter we examine the characteristics of generating test cases from formal specifications. We first re-state the benefits of such activity then introduce the two main established formal languages, namely VDM-SL and Z. Having briefly described VDM-SL—the case study language chosen—we examine our first specification example based on the well understood Triangle Problem. From this study we expand, and discuss in a general manner, the issues involved when performing testing using test cases generated from formal specifications. Finally we will sketch our general approach for automatic test cases generation.

3.1 Benefits of Testing From Formal Specifications

As we have seen in the previous chapter, testing from formal specifications, being a functional strategy, completes the common, structural, approach taken towards the testing of software systems. For completeness, we re-state here the benefits of performing functional testing along side structural testing:

- it has the potential to detect missing functionalities.
- it is language-independent.
- it may provide an automatic oracle.

In addition to these benefits we can note that, from a more global point of view, the process of testing software from formal specifications brings more cohesion to the software development life cycle and helps dismantle some long established paradoxes about formal methods in general [4, 5]. In particular, using formal specifications to derive software test cases is not at odds with the techniques of formally proving software correctness. This is particularly true since such proofs are rarely provided for the entire specification but are more generally derived for those parts of the specification which are highly safety critical. Even after formal proving, testing is usually required to acquire confidence in the system being developed. By remarking that the use of formal methods is not in opposition with testing, we weaken the perceived paradox arising from generating test cases from formal specifications. Also, testing from formal specifications brings a double edged benefit to the software development process: it makes testing more acceptable to proponents of formal software development and brings added value to the use of formal specifications whose ultimate role should not be seen as being formal proving of the entire system under development. Many myths are attached to the use of formal methods [4, 5]: testing from formal specifications should allay some of them.

A recent paper by Bicarregui et al. [98] introduces and illustrates the many areas where formal specification, and formality in software development in general, can be best exploited not just for proving the correctness of a system but also for testing, prototyping and requirements capture. The notion that by adding to the usefulness of formal specifications and highlighting the many benefits of formality in software development, formal methods will become more acceptable to a wider community is very much in our mind.

Testing from high level specifications is not a recent idea, in 1986 Hayes [99] showed how to derive tests from Z specifications. However, the technique presented was purely manual with no prospect for automation. Because of the size of typical specifications, automation is a necessary condition for the success of testing from formal specifications. In this it mirrors structural testing strategies where, although a non-negligible degree of automation has been achieved, the techniques are rarely complete hence their limited usage. Human testers should concentrate on error guessing and generally apply their expertise to non-basic testing; complete automation should allow this to take place.

Implicitly, so far, we have only discussed the testing of software systems. A more introvert view of tests generation from formal specifications is to test the specification itself. This view is notably expressed by the authors of the IFAD's toolbox [100, 101, 102, 103]. A variant can also be found in [99] where tests were derived to show conformance between refinements of the same specification. Proof obligations are provided in VDM to ensure the mathematical self-consistency of VDM-SL specifications. To prove these obligations automatic theorem provers can be used but, although progress has been made since Lindsay [104] surveyed the systems available for mechanical proving in the late 80s, most provers require interactions with a skilled user to construct the proofs required. A recent method for translating VDM-SL to the specification language of the theorem prover PVS is described in [105]. The difficulty in manually or mechanically proving the proof obligations had led to this idea of testing specifications for conformance. Also, testing could detect immediately inconsistencies in the underlying specification thus alleviating much subsequent wasted effort at formally proving the required obligations.

So we must keep in mind that tests derived from a formal specification have two potential usages: to test the specification itself and to test the corresponding system. In the rest of this thesis we place the emphasis on testing software systems rather than specifications but, where appropriate, we will mention the consequences of our findings for specification validation.

3.2 High-Level Specification Languages for Test Generation

As seen in the previous chapter, many attempts have been made at generating tests from formal specifications of different styles. These attempts only achieved a low level of automation or the difficulty in writing the specification is hindering the wider usage of the technique. This is particularly true for tests generation from algebraic specifications where the technique can only be applied to the field of applications suitable for algebraic specification languages which is, at the moment, limited to simple data types. Algebraic specification languages are still in their infancy and may yet bring improvements to the specification process of software systems in the future. However, as we recall that one of our objectives is to generate tests automatically from a versatile specification language and as automatically as possible, we are constrained not to consider algebraic specification languages at this stage. Also, we move away from test scripts specific notations and consider, instead, established, general purpose formal languages.

In [11] North reports on a feasibility study concerning automatic test generation from specifications. His aim was to find out which style of specification language is the most promising for automatic test generation.

His study is based on the Triangle Problem first introduced by Myers [18]. We will discuss the Triangle Problem in further details later. Suffice to say for now that the problem is about classifying a triangle as scalene, isosceles or equilateral given the length of its sides—an extremely simple problem but which illustrates well the difficulty of finding a suitable test set for a given problem, it is also a well documented example.

North took three representative formal specification techniques of different style, specified the Triangle Problem using each technique, showed how to derive the tests suggested by Myers for the Triangle Problem by hand, and finally tried to systematically generate the same set of tests from each specification.

North's choice of specification languages was:

- VDM-SL; a model-based language
- Miranda; a functional language
- Prolog; a logical language

We only report here his conclusions.

One desirable feature for a suitable specification language is that as much information as possible should be explicitly available to the test generator. For example the language should be typed, allow type invariants to be expressed for easy boundary values generation. It should also be abstract enough to do away with implementation details.

From these considerations Prolog was deemed unsuitable as too much re-work of the specification was necessary and many parts of the specification were only concerned with implementation details—e.g. implementation of sets. This should not surprise us since Prolog was hardly designed as a specification language in the first place. However it needed to be covered since some seem to use it for that purpose, thus confusing prototyping and specifying activities.

Miranda was deemed worth of further considerations. Although not quite abstract enough, it benefits from executable specifications which are an advantage when considering the problem of the oracle.

VDM-SL was deemed suitable thanks to its high level nature but doubts where raised about the ability of generating test cases because most VDM-SL specifications are not executable.

North's remarks that

> the main difference between a Miranda-like language and VDM is that of executability vs abstractness: the price of executability is loss of abstractness.

From North's findings we can now make a knowledgeable decision as to which style of specification we should base our efforts on. Clearly, test cases generation from a functional language is potentially easier than from a model-based language. The oracle problem is immediately solved in the functional language case by executing the specification with the generated test inputs. However, we must also take into account the effort of writing the specification and the degree of acceptance of the specification language used. Currently, model-based languages are the widespread choice for specifying software systems. They also have the widest tool support.

Since one of our aims is to reduce the effort required to test software without imposing too much of a burden on the developers by having to write a specification of the problem in a somewhat unusual format, we reduce our search for a suitable specification language to model-based languages and will try to overcome the problem of the oracle.

[106] describes the fundamentals of model-based specifications as well as introduces its two main representative notations VDM-SL and Z. Briefly, model-based

specification languages concentrate on the specification of abstract machines by specifying their states and the operations which can applied on them—remark that algebraic specification languages concentrate on specifying abstract data types.

Strictly speaking VDM is a formal method whose specification language is VDM-SL whereas Z is simply a notation based on set theory. What makes VDM a method is largely its refinement capabilities, that is the possibility to refine the specification in a verifiable manner from a high-level of abstractness down to near implementation level [84], and the proof obligations of well formed specifications. However, refinements to a programming language level are limited by the difficulty in proving the refinement steps correct [107]. Only by choosing a functional language as target language can this process be eased [108].

For our purpose we only need to discuss the specification language used in VDM. Further, we shall not discuss the differences at the verification level between Z and VDM, instead we shall concentrate on the syntactic and expressibility differences of the two notations. Verification comparisons can be found in [109].

As reported elsewhere [106, 109, 110, 111] most of Z and VDM-SL differences are superficial. However we can note that VDM-SL is a strictly typed language when compared with Z, and that Z schemas are an elegant way of constructing large specifications with maximum reuse of components. From a test case generation point of view it is very difficult to foresee which language is the more suitable and both are potential choices.

Since we need to choose a language to make our approach explicit, the facts that VDM-SL has a more formal structure than Z—because it distinguishes between states, types and operations and requires pre-conditions to be explicitly given for operations and functions—and that the notation and semantics of VDM-SL are more stable with less variants than Z, along with the availability of an agreed ASCII syntax suitable for automatic treatment at the time, made us choose VDM-SL as our study language for automatic test cases generation from formal specifications.

The standard of VDM-SL [112] provides a means to clarify aspects of it semantics. However, because of its size and complexity the standard should not be viewed as providing an introduction to the language, [113] is far more suited for this purpose. Only when in doubts about some aspects of the semantics will we refer to the standard. [97, 114] provide a short introduction to the VDM-SL standard in particular and the standards of non-executable specification languages in general.

Note that as a consequence of the many similarities between Z and VDM-SL our findings should be applicable to either notation. When this is not the case we shall explicitly point it out.

3.3 A Brief Introduction to VDM-SL

We now very briefly introduce VDM-SL, and point out its most notable characteristics for our purpose. A comprehensive introduction to VDM-SL can be found in [113].

VDM-SL, as a model-based specification language, provides a model of a system's state in terms of a collection of state variables. Each state variable models some aspects of the system. The states can be specified as mathematical objects using common basic types (such as integer, real number, Boolean etc.), sets, maps, sequences or records. Many powerful predefined operators based on these types are available in VDM-SL.

One low level difference between Z and VDM-SL is the use in the latter of a three valued logic system—*True*, *False*, *Undefined*—also known as the Logic of Partial Functions (LPF). It is briefly described in [109]. Undefined values arise in VDM-SL when an operator, a function or an operation is used outside its definition domain. For example, division by zero returns an undefined value. Conditional, Boolean and quantified expressions do not systematically spread undefinedness. For example the Boolean expression: $True \vee Undefined$ evaluates to *True*. On the other hand most other expressions propagate undefinedness, fo example the expression: $3 + Undefined$ evaluates to *Undefined*.

A specification written in VDM-SL would classically be composed of a state, operations on the state and functions. A basic means of modularisation is also available for large VDM-SL specifications, but re-use is not as developed as in Z.

As an example we give below a state representing a date:

$$
\begin{array}{l}
\textit{state Date of} \\
\quad year : \mathbb{N} \\
\quad month : \{1, ..., 12\} \\
\quad day : \{1, ..., 31\} \\
\quad inv\ mk\_Date(y, m, d) == \\
\qquad (m \in \{4, 6, 9, 11\} \Rightarrow d \leq 30) \wedge \\
\qquad (m = 2 \wedge is\_leap(y) \Rightarrow d \leq 29) \wedge \\
\qquad (m = 2 \wedge \neg is\_leap(y) \Rightarrow d \leq 28)
\end{array}
$$

The state above is a record composed of three fields: *year*, *month* and *day* each of a particular type. An invariant—a property that should hold throughout the specification—is also provided for *date*: it specifies that the date represented should be a valid date in the common sense of the term.

Invariant preserving operations can be provided to change a state for example to advance the current date to the next. Functions cannot modify a state.

The *is_leap* implicit function is given below:

$$
\begin{array}{l}
is\_leap(y : year)\ r : \mathbb{B} \\
post\ r \Leftrightarrow y\ mod\ 4 = 0 \wedge y\ mod\ 100 \neq 0 \vee y\ mod\ 400 = 0
\end{array}
$$

It can also be given explicitly as follows:

$$
\begin{array}{l}
is\_leap : year \rightarrow \mathbb{B} \\
is\_leap(y) == y\ mod\ 4 = 0 \wedge y\ mod\ 100 \neq 0 \vee y\ mod\ 400 = 0
\end{array}
$$

The result of an implicit function must validate the postcondition (introduced by the *post* keywork above). They allow a greater degree of flexibility than explicit functions where the result is defined by evaluating the result type expression, i.e. the result has to be constructed rather than constrained. All functions are side effect free (e.g. they cannot call operations).

Functions can be loose, that is their result is not completely defined. For example consider the parameterless function, taken from [113], which returns an even number as defined below.

$even()\ e : \mathbb{N}_1$

$post\ e\ div\ 2 = 0$

The result of this function is only constrained to return an even natural number, but no specific number is requested. Loose functions are interpreted as under-specified rather than nondeterministic as is the case with operations. This means that although any correct result is allowed the same result must always be returned for the same set of parameters. Loose operations are interpreted as nondeterministic, thus no assumptions can be made about which result from the set of possible solution is returned.

For example with the *even* function above, $even() = 2$ may be *true* or *false* but $even() = even()$ is always *true*. However, if *even* was an operation then $even() = 2$ may still be *true* or *false* but, $even() = even()$ can now evaluate to *false* as well.

As we shall see loose expressions, and loose specifications in general, can lead to surprising results, but looseness is a powerful concept allowing greater expressiveness of the language and is an integral part of VDM-SL. Looseness enables the respect of the principle of minimality: that the specification language must not force the user to be any more specific than he or she wishes.

An implicit operation to update a date `d` to the next day is given below:

$$
\begin{array}{l}
next\_day(d :\ date)\ next : date \\
post\ \ if\ \ d(day) < 28 \\
\qquad\qquad then\ \ next = \mu(d, day \rightarrow day + 1) \\
\qquad elseif\ \ d(day) = 28 \wedge d(month) = 2 \wedge is\_leap(d(year)) \vee \\
\qquad\qquad d(day) < 30 \vee \\
\qquad\qquad d(day) < 31 \wedge d(month) \in \{1, 3, 5, 7, 8, 10, 12\} \\
\qquad\quad then\ \ next = \mu(d, day \rightarrow day + 1) \\
\qquad elseif\ \ d(month) = 12 \\
\qquad\quad then\ \ next = \mu(d, day = 1, month = 1, year = year + 1) \\
\qquad else\ \ next = \mu(d, day = 1, month = month + 1)
\end{array}
$$

An explicit operation to calculate the number of days separating the current date state and an arbitrary future date:

$diff\_days \ : \ date \ \to \mathbb{N}$

$diff\_days(d)$

$\quad ext \ rd \ d\_st \ date ==$

$\quad (dcl \ diff \ : \ int := 0,$

$\quad\ temp \ : \ date := d\_st;$

$\quad\ while \ temp \neq d \ do$

$\quad\ (temp := next\_day(temp);$

$\quad\ \ diff := diff + 1;$

$\quad\ );$

$\quad\ return \ - diff$

$\quad)$

Explicit operations are specified using statements similar to programming language constructs (assignments for example are allowed). These statements are however just as precisely defined as the rest of VDM-SL. Proofs involving statements are less straightforward than proofs involving only VDM-SL expressions. However, explicit operations are likely to be closer to an implementation in a conventional programming language.

There is much more to VDM-SL, and VDM in general, than this small introduction suggests. VDM-SL is a large and complex language but is also one of the most versatile specification languages currently available. For more in-depth discussion of the language the reader can refer to [84, 107, 112, 113].

3.4 The Triangle Problem and Some Suitable Test Cases

We now give a small, but useful, specification example and illustrate which test cases would be suitable for testing an eventual implementation. This will allow us to clarify some aspects of testing from formal specifications

The Triangle Problem was first proposed by Myers [18] as follows:

> 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.

As remarked by North [11], the above informal specification fails to mention what the behaviour of the program should be when the three integers denote an invalid triangle. North completes the specification by adding that *INVALID* should be returned when the integers do not represent a valid triangle.

3.4.1 North's Test Cases

North's VDM-SL specification of the Triangle Problem is not intuitive but has been chosen to illustrate the many ways in which problems can be specified. In VDM-SL $\mathbb{N}*$ denotes a sequence (possibly empty) of natural numbers. Note that this specification is not loose.

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

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

$$
\begin{aligned}
&\mathit{variety} \ : \ \mathit{Triangle} \to \mathit{Triangle\_type} \\
&\mathit{variety}(\mathit{sides}) == \\
&\quad \mathit{cases}\ \mathit{card}(\mathit{elems}\ \mathit{sides})\ \mathit{of} \\
&\qquad 1 \to \mathit{EQUILATERAL} \\
&\qquad 2 \to \mathit{ISOSCELES} \\
&\qquad 3 \to \mathit{SCALENE} \\
&\quad \mathit{end}
\end{aligned}
$$

$classify \ : \mathbb{N}* \rightarrow Triangle\_type$

$classify(sides) ==$

$\quad if \ \ is\_Triangle(sides)$

$\qquad then \ \ variety(sides)$

$\quad else \ INVALID$

North presents [11] thirty six test cases suitable for the Triangle Problem. These test cases arise from North's interpretation of Myers' test requirements [18]. Table 3.1 reproduces North's test set for the Triangle Problem (where M denotes for the greatest natural number available).

Id.	Test Input	Oracle	Id.	Test Input	Oracle
1	$[0,0,0]$	Invalid	19	$[2,3]$	Invalid
2	$[0,1,1]$	Invalid	20	$[4,4,4,4]$	Invalid
3	$[1,0,1]$	Invalid	21	$[M,M,1]$	Isosceles
4	$[1,1,0]$	Invalid	22	$[M,M,M]$	Equilateral
5	$[3,1,2]$	Invalid	23	$[M+1,M-1,M]$	Scalene_or_Invalid
6	$[1,3,2]$	Invalid	24	$[1,1,1]$	Equilateral
7	$[2,1,3]$	Invalid	25	$[1,2,2]$	Isosceles
8	$[1,2,5]$	Invalid	26	$[2,1,2]$	Isosceles
9	$[5,2,1]$	Invalid	27	$[2,2,1]$	Isosceles
10	$[2,5,1]$	Invalid	28	$[3,2,2]$	Isosceles
11	$[5,1,1]$	Invalid	29	$[2,3,2]$	Isosceles
12	$[1,5,1]$	Invalid	30	$[2,2,3]$	Isosceles
13	$[1,1,5]$	Invalid	31	$[2,3,4]$	Scalene
14	$[1,2,-6]$	Invalid	32	$[3,2,4]$	Scalene
15	$[-2,-2,-2]$	Invalid	33	$[3,4,2]$	Scalene
16	$[2,2.3,2]$	Invalid	34	$[4,3,2]$	Scalene
17	$['A',2,3]$	Invalid	35	$[4,2,3]$	Scalene
18	$['A','A','A']$	Invalid	36	$[2,4,3]$	Scalene

Table 3.1: North's Test Cases for the Triangle Problem

In the rest of this work, we will consider this test set to be adequate for the Triangle Problem. We do so because it represents the state of the art in tests derivation using black box techniques. In general test data adequacy criteria (criteria stating when enough testing has been performed by executing and checking the tests) are subjective and as we have seen still a matter of theoretical research [32]. We now give our definition of an adequate test set.

Definition 3.1 *Adequate Test Set*

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.

We will therefore 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. In particular, since we are considering tests generation from formal specifications, the theory of partitioning outlined in section 2.2.4. will play a central role in our technique. We must remember however that the adequacy of test sets is subjective and a matter of ongoing research and debate [32, 28].

We now return to North's test set for the Triangle Problem.

Firstly the Triangle Problem illustrates well that even simple problems may require a large number of tests. Also note that twenty of the tests are for invalid triangles. In [11] North shows the rationale behind the tests generated.

We remark that according to the specification, the tests inputs not validating the requirement: *is_seq_nat* do not evaluate to *INVALID* but result in the specification being in an undefined state. The test $[M+1, M-1, M]$ does not represent a valid sequence of integers according to our assumption and therefore *Classify*$([M+1, M-1, M])$ evaluates to *Undefined* and not to *INVALID* or even *SCALENE* as suggested by North. We shall return to these matters later.

North uses some aspects of partitioning theory to derive his tests. A high level partition of the specification can be obtained by starting with *Classify* which immediately divides the inputs into valid and invalid equivalence classes as shown below.

$is\_Triangle(sides) \wedge classify(sides) == variety(sides);$

$\neg is\_Triangle(sides) \wedge classify(sides) == INVALID;$

This first step arises from the partition of the *if* ... *then* ... *else* ... expression according to the truth value of the condition $is\_Triangle(sides)$. We remark that each equivalence class is a VDM-SL Boolean expression. Note the omission of the higher level partition implied by North:

$$is\_seq\_nat(sides) \land result == classify(sides);$$
$$\neg is\_seq\_nat(sides) \land result == INVALID;$$

$variety(sides)$ can be partitioned further according to a simple *case* expression rule to obtain the following partition:

$$\begin{aligned}
&is_Triangle(sides) \land card(elems\ sides) = 1\\
&\qquad \land classify(sides) == EQUILATERAL;\\
&is_Triangle(sides) \land card(elems\ sides) = 2\\
&\qquad \land classify(sides) == ISOSCELES;\\
&is_Triangle(sides) \land card(elems\ sides) = 3\\
&\qquad \land classify(sides) == SCALENE;\\
&\neg is_Triangle(sides) \land classify(sides) == INVALID;
\end{aligned}$$

where the $\neg is\_Triangle(sides)$ and $is\_Triangle(sides)$ expressions can of course be partitioned as well.

We can now see that the specification is indeed not loose. This facilitates the verification of output results from the implementation under test since only a simple comparison is required to ensure the correctness of its behaviour for a particular test.

North's study outlines the way partitions can be constructed from VDM-SL specifications using a combination of symbolic execution and partitioning rules. The resulting expressions are VDM-SL predicates defining equivalence classes which must be satisfied by the tests generated. These predicates, or constraints, over the variables of the specification need to be checked for consistency to ensure that the equivalence classes they define are not empty. Some predicates, such as $len\ sides = 3 \land sides = []$ for example, are inconsistent and cannot therefore be part of a partition.

Great care should be taken when devising the partitioning rules so as to respect VDM-SL semantics. For example, according to the semantics of the

VDM-SL case expression, the following partition consisting of four equivalence class, should actually be generated from $variety(sides)$:

$$
\begin{aligned}
&is\_Triangle(sides) \wedge card(elems\ sides) = 1 \\
&\qquad\qquad \wedge classify(sides) == EQUILATERAL; \\
&is\_Triangle(sides) \wedge card(elems\ sides) = 2 \wedge card(elems\ sides) \neq 1 \\
&\qquad\qquad \wedge classify(sides) == ISOSCELES; \\
&is\_Triangle(sides) \wedge card(elems\ sides) = 3 \wedge card(elems\ sides) \neq 1 \\
&\qquad\qquad \wedge card(elems\ sides) \neq 2 \wedge classify(sides) == SCALENE; \\
&\neg is\_Triangle(sides) \wedge classify(sides) == Undefined;
\end{aligned}
$$

This is in general necessary to ensure that if two choice conditions are satisfiable in a VDM-SL case expression (as is allowed) the first one is the one taken. In the above this is not necessary as the alternatives are disjoint.

We now discuss the fundamental, underlying, assumptions taken by North.

3.4.2 The Invalid Input Domain Problem

When attempting to generate tests manually through equivalence partitioning from the Triangle Problem specification, North adopts the valid/invalid input distinction introduced by Myers [18].

According to Myers, the invalid input domain—here any input not representing a valid triangle—must be considered as a special case once equivalence classes have been identified. Instead of trying to cover as many equivalence classes as possible with the minimum number of tests as for valid classes, Myers suggests that invalid classes should be covered by individual test cases. The rationale given for this idea is reproduced below.

> The reason that invalid classes are covered by individual cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states 'enter book-type (HARDCOVER, SOFTCOVER, or LOOSE) and amount(1–9999),' the test case *XYZ 0* expressing two error conditions (invalid book type and amount) will probably not exercise the check for amount, since the program may say 'XYZ IS UNKNOWN BOOK TYPE' and

not bother examine the remainder of the input.

North adopts this view and is constrained to deal with invalid input—in which he incorporates invalid triangles—in a separate manner. We note that from the specification point of view that only non sequences of integers are invalid inputs. North artificially includes non valid triangles to those invalid inputs. In an automatic test cases generator this would require prior human analysis of the specification to decide which classes ought to be considered invalid.

Accordingly North identifies four invalid classes:

- *Type of input* : *not a sequence of natural numbers*
- *Length of input* : $< 3, > 3$
- *Value of sides* : []
- $\neg(\forall i \in \mathit{elems}\ \mathit{sides} \cdot 2 * i < \mathit{perim})$

In [12] we described a test cases generation technique taking into account this perceived difference between valid and invalid input domains. It led to the following components of, say, an operation specified in VDM-SL to be treated separately:

- the basic type and any invariant for each variable
- the pre-condition
- the exception handling part
- the post-condition

and to a test generation technique which was variable driven rather than the more common operator driven approach (i.e. a variable would first be chosen and depending on whether testing a valid or invalid input domain, other tests for input variables would be generated). Beside being more cumbersome than the Dick and Faivre technique of [14], we now feel that Myers' recommendation is somewhat misplaced in the new formal specification context.

In effect while we acknowledge that Myers' point seems strong, we judge his argument to be too artificial and subjective. In particular, there are no semantic differences between the following pieces of specification:

if $A \vee B$ *then* *CORRECT_INPUT*

if $A \vee B$ *then* *INCORRECT_INPUT*

In both cases an implementation may consistently ignore parts of the condition, thus an artificial differentiation in the treatment of the equivalent classes is semantically unjustifiable. However this also shows that we also need to address Myers' concerns for valid inputs as well as for invalid inputs.

Further, even Myers' solution is incomplete since it runs the risk of not detecting combinations of invalid features such as, in Myers' own example, *XYZ 0*. For example the following implementation would pass Myers' test cases while being incorrect:

```
if book not in {HARDCOVER, SOFTCOVER, LOOSE}
   and amount in [1...9999] then INVALID INPUT
elseif book in {HARDCOVER, SOFTCOVER, LOOSE}
       and amount not in [1...9999] then INVALID INPUT
else ... — valid inputs
```

We believe that those concerns can be addressed by specifying the behaviour of the system for invalid inputs and by strict adherence to the semantics of VDM-SL. Thus, in the same way that Myers' informal specification was incomplete—by failing to specify the program behaviour for invalid triangles and invalid inputs—North's formal specification fails to indicate what is the intended behaviour of the system for invalid inputs—i.e. non sequences of natural numbers.

From a semantic point of view, the behaviour of the system as specified by North and as defined by the VDM-SL semantics, becomes undefined when a non sequence of natural numbers is encountered. We are reluctant to generate tests for which the overall behaviour of the specification is undefined for two reasons.

Firstly, it is not clear what the behaviour of the implementation should then be. At the interface level we may assume that an error message will be generated and the execution of the program terminated. We are then left with the problem of deciding if the system has behaved correctly. However, under VDM-SL semantics the matching of *undefined* with any other expression is always undefined. We cannot thus decide automatically if the test has passed or failed. From a theoretical point of view, incorrect inputs should not leave any system in an undefined state, i.e. not only should an incorrect input be rejected but it should not have any non intended consequences. This should be specified in the specification of the system, hence at least at the user interface level, input validation should be part of the specification.

Secondly, it is not clear how to sample a class such as North's $\neg seq\_of\_nat$. For example, any basic types, sets, maps, records, in fact any type other than a sequence could first be selected, and then the same sampling principle would apply for the elements of the non-basic types chosen. Clearly, type negation requirements needs to be handled with care if very large, inefficient, test sets are to be avoided.

Therefore for the Triangle Problem as specified, we would not consider generating tests which are not sequences of natural numbers. If input validation is a requirement of the system it should appear explicitly in its specification. As mentioned, for interface level functionalities this is always the case, but robust systems and software libraries must also be concerned.

One way to include input validation into the Triangle Problem is to add:

$$is\_seq\_int : token \to \mathbb{B}$$

$$is\_seq\_int(sides) = is\_seq(sides) \wedge \forall x \in elems(sides) \cdot is\_nat(x)$$

with *Classify* modified as follows:

$$Classify : token \to Triangle\_type \mid INVALID\_INPUT$$

$$\begin{aligned} Classify(sides) == \; & if \;\; is\_seq\_nat(sides) \;\; then \;\; \ldots \;\; as \;\; before \;\; \ldots \\ & else \;\; INVALID\_INPUT \end{aligned}$$

or one could use VDM-SL error handlers, which allow for such circumstances to be specified more explicitly. Further, different error messages could be specified for classes of invalid inputs: this would increase the testability and facilitate debugging.

Hence, for the Triangle Problem, $INVALID\_INPUT$ will be raised whenever $\neg is\_seq\_nat(sides)$ is true or when $is\_seq\_nat(sides)$ is undefined. Using a simple partitioning rule for or expressions of the form $A \vee B$ such as they are partitioned into three classes:

$A \wedge \neg B$

$A \wedge B$

$\neg A \wedge B$

allows the generation of test cases for inputs within the invalid domain. This will be illustrated in more detail in the next chapter.

To conclude on this matter, we started from a position where invalid inputs were treated differently from valid inputs and then moved to a unique, consistent and complete approach to deal with all inputs. The approach we have described here is actually implicitly used in [14, 15]. We have also suggested a specification writing style which encourages robustness and increase testability. It could also help in error tracking in case of incorrect behaviour of the system under test.

3.4.3 Boundary Values

To complete the Triangle Problem specification, we must provide the ATG—as well as the developers—with the range over which, for example, natural numbers are defined since the implementation will always be restricted to a finite domain. Thus, the M notation used by North should actually be part of the specification:

$is\_nat : token \to \mathbb{B}$

$is\_nat(to) = is\text{–}\mathbb{N}(to) \wedge to \leq M$

where M is a constant set in accordance with the system requirements.

Generalising this approach to every data type—sets, records, sequences, maps etc. by fixing their maximal size—we can assume that after validation all input and output variables in a VDM-SL expression have a finite domain, including if desired, reals. This also means that many proof obligations will be simpler, since the finiteness of most sets in the specifications will be easier to establish (in VDM-SL all sets must be finite: this requirement actually gives rise to many proof obligations).

While these requirements may seem drastic they are necessary for two reasons. Firstly, we will often choose boundary values when sampling classes, as these are experimentally deemed to offer the best prospect for finding an error in the system under test. We therefore need a concrete value for those boundaries. Secondly, it is easy to see how nearly any system will fail to behave correctly in respect to its specification if the test is so large as to exceed the available finite memory of the underlying hardware. To avoid this possibility inputs and outputs must be constrained, within the specification, to finite domains. Finally we note that large values, say for the length of a valid sequence, need not be chosen, as our purpose is not to test memory usage. This could be used advantageously later for consistency checks and test generation proper, since then many quantified expressions will be over small sets which could be transformed into equivalent logical assertions.

3.4.4 Testability

Another point which becomes apparent from North's case study is that given a system specification consisting of operations and functions, an automatic test cases generator should be provided with the list of functions and operations for which tests are required. North implicitly assumes that only *Classify* should be explicitly tested. *Sum* and *Variety* are implicitly tested as called functions.

This is not necessarily always the best solution. In particular if robustness testing is required—because the specification is part of a library or to facilitate regression testing for example—one must test every function and operation explicitly. i.e. one must consider each function and operation as being able to be called with any inputs and not just with those which are actually passed.

For example, if the general purpose function *sum* in North's specification of the Triangle Problem is required to be tested for robustness then a test for sequences of different length than three must also be generated. However always performing explicit testing on all functions and operations not only leads to a larger test set (but see below) but may also pose problems when trying to actually perform the tests. In the Triangle Problem, it is very likely that *sum* would simply be implemented as `x + y + z` in the invariant part of the triangle type. Thus executing the test on *sum* would certainly become very difficult—due to the broken isomorphism between the specification and the implementation under test in respect to functions and operations structures, isomorphism which should not be assumed—and would certainly be meaningless. Thus *sum* should not be tested explicitly but implicitly only. This situation can also arise if the specified function, although being separately implemented, cannot be tested as a unit because no stub is available (a stub simulates the actions of the functions called within the tested function). We note that the specification would facilitate the coding of stubs.

This matter, of which functions should be explicitly tested, ought to be left to the human tester as it requires a knowledge of the implementation and of the overall test requirements of the system. We therefore suggest that a list of functions and operations to be tested explicitly should be passed to the automatic test cases generator.

3.5 Context of Testing from Formal Specifications

Before discussing further tests cases generation we must put into context the use and meaning of such test cases. Therefore in this section, we assume that a test set for a given function or operation is available and consider its significance as well as how to perform the tests.

3.5.1 Formal Specifications and Implementations

Firstly, we must take note of the relationships between formal specifications and their implementation counterpart. These are important, basic, considerations but are often not mentioned when discussing testing from formal specifications.

The first consideration is that if an error is detected using the oracle provided by the specification, it can only be a specification-wise error. In other words, we can only detect differences between the specified behaviour of the system under test, as modelled by the specification, and the actual behaviour.

We acknowledge that the specification may fail to model the original intended behaviour of the system. This can occur if the system requirements have not been properly captured or if those requirements have not been properly modelled. This leads to the corruption of the automatic oracle. Human oracles could be used but these are expensive and error prone, especially if the test set is anything but small.

Deciding if the specification models the intended behaviour of the system can only be done through validation of the specification—as opposed to verification, since intrinsically this step is not formally decidable. Animation tools can help in this process [100, 101, 103, 115] as well as, as mentioned, tests conducted on the specification itself. However this problem lies beyond the scope of the work presented here.

We must also assume that the specification is correct. In VDM-SL for example numerous proofs must be derived to ensure correctness and consistency of the specification. Without the assurance, or at least the assumption, that the specification is correct we cannot ensure the validity of the errors detected. Note that we do not require proof of the satisfiability of the specification since as long as the specification is syntactically correct test cases could be generated.

In brief, we assume that the specification models perfectly the intended behaviour of the system and that it is syntactically as well as semantically correct (i.e. proofs of the proof obligations have been constructed).

Another relationship between implementations and specifications which needs to be considered is that although we could suppose that only one implementation will be tested—and as a consequence only one algorithm—many specifications could be given to the tests generator. As we have seen with the Triangle Problem specification as given by North, specifications may not be intuitive and may even be cumbersome to deal with (consider that in VDM-SL, as remarked by Dawes [113], *if true then* 3 *else false* + *'ABC'* is a valid expression which could be simplified by simply writing 3). It is obvious that different specifications of the same problem will lead to different test sets. This begs the question as to which specification style is likely to increase the adequacy of the test set produced.

Although specifications refined towards a target implementation language may produce tests closer to test data (i.e. tests that may be less problematic to transform into a representation suitable for actual execution—see section 3.5.6. for more details), thus increasing the testability of the implementation, we do not foresee any theoretical argument for tests generated from such specifications to be more adequate than tests generated from higher level specifications. Indeed, a definite answer is hard to imagine even if one were to be provided with a very large amount of experimental results, only guidelines could be issued and the rationale only statistical.

3.5.2 Testing Explicit Operations: the Use of Statements

Explicit operations are used in the final stages of refinement and are a means to get closer to the target implementation language. Explicit operations are executable, a feature which is exploited fully in the IFAD's toolbox [100, 101]. This is achieved, as we have seen, by the use of statements which are very close to imperative language constructs, although benefit from a formal semantic definition. Proofs involving statements are far from easy, indeed statements are not considered in a recent guide concentrating on proofs in VDM [107]. Statements, because of their closeness to imperative implementation languages, are very different from ordinary VDM-SL expressions, for example, as we have already seen, there are assign, loop and return statements.

Choosing partitioning rules for the generation of a partition of an explicit operation is difficult due to the sequential style of such specifications. It seems that structural techniques ordinarily used to generate tests from programs source code would be more appropriate for explicit operations. This would still amount to black box testing as the tests are generated from the specification.

It is not clear if explicit operations would benefit from devising a new testing technique for generating tests from statements. As opposed to ordinary VDM-SL expressions, where test generation is far from being understood, one could apply, nearly directly, any technique from the range of white box testing techniques described in the previous chapter.

Because of these considerations, we choose to concentrate on test cases generation from VDM-SL specifications excluding explicit operations. More precisely, we exclude statements from our discussion as we believe that available white box testing techniques could be used for them. This needs, of course, to be experimented with and confirmed.

3.5.3 The Oracle Problem

We recall that non-executability of the specification language was deemed a difficulty for the generation of the expected result of a computation from its specification. Classically an oracle provides the expected result for each test: the detection of an error then only amounts to a simple comparison between the expected result provided by the oracle and the actual result if any (the system under test may well not terminate or even crash). Such classical approach is not feasible here since some specifications do not bind totally the outputs after instantiation of the inputs with the test input vector. This can arise in two situations: when the solver is unable to find the unique solution of a system of constraints and of course because of non-determinism.

In the first case, the solver cannot find the solution of a satisfiable set of constraints. Consider the implicit specification of a function returning the largest integer from a set of positive integers:

$max(S : \mathbb{N}\text{--}set) M : \mathbb{N}$
$post\ M \in S \wedge \forall n \in S \cdot n \leq M$

Here, as with all implicit specifications, the result is not constructed but constrained. If the solver is not general enough, it might fail to find the result of *max(*$\{1, 2, 3\}$*)* .

Although this particular case would actually be easy to deal with, there will always be cases where to find the expected result, only a generate and test approach is available which would lead to extremely long searches. On the other hand if the actual result from the test run is also provided then the oracle work is reduced to consistency checking.

e.g. to check if 3 is the correct result for *max(*$\{1, 2, 3\}$*)*, the oracle has to check the consistency of:

$3 \in \{1, 2, 3\} \wedge \forall n \in \{1, 2, 3\} \cdot n \leq 3$

However, the great advantage of moving away from a classical oracle is the possibility to deal with loose specifications. As a simple example of a loose function, consider:

$f(s : \mathbb{N})r : \mathbb{R}$

post $r > s$

Given *f(4)* as a test yields $r > 4$ as the oracle expression. When the test result is available the oracle has to instantiate r and decide the satisfiability of the Boolean expression constituting the oracle. The classical approach would become intractable since it leads to generating reals greater than four until the actual result equals one of them. Further, if the actual result is in fact incorrect, according to the specification (for example $r = 3.14$ for *f(4)*), the entire set of implementation representable reals greater than four will have to be generated until the oracle can safely declare the actual output as incorrect.

Thus, we remark that given test inputs and the actual output one can instantiate the specification, and obtain a Boolean expression which must be true for the results to be correct. In most cases the truth of the resulting Boolean expression will be straight forward to establish after simple simplifications—as we have seen the oracle for the Triangle Problem amounts to a simple comparison of the actual result with the uniquely identified expected result. Even for loose specifications we envisage that the satisfiability of the oracle expression will be relatively easy to establish with the use of current constraints solvers. If it is not the case then human intervention will be necessary. Constraints solving is discussed later in this chapter.

On finding an inconsistency between the test results and the oracle expression one is left with the difficult task of debugging the implementation. It is hard to see how an ATG from formal specifications could be of any direct help in this regard. But we can remark that having executed the entire test set, it could be useful to reveal common features amongst the failed results in terms of the test requirements exercised. Further, as the formal specification is available there can be no misunderstanding as to the intended behaviour of the implementation:

thus the validity of the error detected can be readily established.

To be complete, and as already mentioned, such an oracle could also be used along side any testing technique for which deciding the success or failure of a test cannot be automated, e.g. structural techniques, error guessing. However, this kind of oracle is unsuitable for validating a specification against the user's requirements (animation tools are helpful for this) or for verifying the results of specification testing (as in the IFAD toolbox [102, 103]).

3.5.4 Executing the Tests

Given a test involving a VDM-SL state how do we execute the implementation to validate its behaviour? This problem relates to the way states should be instantiated before calling the appropriate operation. There are two views on this problem.

To execute the tests generated one can try to instantiate the states indirectly by using an appropriate test sequence—a pure black box technique—or can instantiate directly the states—a technique we have coined as white tinted.

This difference arises because of the presence of systems states in VDM-SL specifications —at first approximation states in VDM-SL are related to global variables in a structured imperative language such as Pascal or C.

In the pure black box technique, system states are reached through test suites. Test suites are devised to reach the chosen system state as described by the test inputs. In the white tinted technique test inputs for the system states are initialised through direct access to the global variables of the program under test.

A method to sequence tests is described in [13]; it relies on a Finite State Automaton (FSA). We make two remarks on this approach.

Firstly the technique is hard to automate, indeed in the version described above the FSA is constructed by hand: an axiomatic approach would facilitate the construction of the FSA thus, algebraic languages may be more suited for the construction of test suites, see [79]. Secondly, and more importantly, some tests cannot be carried out using this method because some states may well be unreachable. Whether this is a good thing for specifications in general is discussed by Nicholl in [116].

The impossibility of testing for unreachable states using this technique implies that robustness testing is impossible, i.e. one cannot test the software under consideration for unexpected states within the system. If the system under tests is correct in respect to the specification and if the specification is valid then robustness testing would be a waste of time. But while we may assume the latter, assuming the former would be denying the worthiness of our efforts to find errors in the system in respect to its specification. We could also restrain analysable specifications to eliminate instances where some states are not fully reachable. This, however, would require prior manual analysis of the specification.

To eliminate unreachable states in formal specifications, the developer must strengthen the state invariants. As remarked in [116] this is highly desirable for specifications intended for reuse. Another incentive for the developer to remove unreachable states in the specification is that otherwise the code might be tested against states which were unexpected and for which the code does not work. Thus deriving tests for robustness is clearly advantageous as many errors—in the implementation as well as when testing the specification itself—could be revealed.

Hence, we do not assume that all states are reachable and that a means to directly instantiate the state of the system under test must be made available to the tester. This requires more manipulation of the system under test than the sequencing of tests but if formal specification based testing is used in conjunction with structural testing techniques then most of the work should have already been carried out under the form of test drivers and stubs. Furthermore, at the unit testing level very little manipulation would be required.

Also, if the system has been entirely specified and robustness testing is not a requirement, testing from the user interface level could eliminate the need for harness and stubs.

A direct means of state instantiation for unreachable state is a necessity but could well be replaced by test sequencing in cases where all states are reachable. However, as test sequencing has not yet been made automatic, total reliance on direct means should not be seen as a defect of our approach but rather as a temporary solution. Further research on test sequencing is needed, but we believe that the problem of test set generation proper needs to be solved first for tests sequencing to become one of the next focus of research for formal specification based testing.

3.5.5 Test Cases Generator and the Software Life Cycle

We consider the place of an automatic test cases generator in the development life cycle. Whether the specification of the system has been written prior to the development of the system or after—as can be the case in reverse engineering—has of course a huge impact on when tests should be generated and executed.

If the specification has been written prior to the system coding one can even envisage an approach where test inputs are given to the developers as an indication of where errors are likely to be made in the coding process. However, intuitively this does not seem to be a very good idea since, although, errors are then less likely to be committed for those particular test inputs, the errors present may be more difficult to detect because atypical.

With a traditional approach however, the scope of applications of a test generator is vast, especially when the specification is not limited to units but extends to the entire system. One can envisage to use it from unit testing through system testing up to regression testing.

[117] describes how semi-automatic testing techniques were incorporated in the life cycle of various projects. It shows how a global approach to validation must be adopted for large projects to achieve higher quality systems as well as reducing inefficiencies in the development process proper. It also offers insight into industrial systems validation practices. Alternatively [19] offers a global view and introduces ideas on the possible position of an automatic test cases generator in the life cycle.

To conclude on this subject, we believe that a test generator from specifications, as discussed in this work, could have wide ranging use in the entire life cycle of systems and not just at the unit level as is often assumed. Experience should re-enforce this view.

3.5.6 Data Type Transformations

If we test a system rather than the specification itself, we need to address the problem of how to transform the test inputs generated from the specification to test data suitable for initialising the system under test in order to execute it. Similarly test results must be transformed from an implementation form to a VDM-SL form to exercise the oracle. This problem emerges because high level—abstract—data types and implementation—concrete—data types may not have the same internal structure.

For example, while the VDM-SL integer type may certainly match the integer type of the target language of the system under test, it most certainly will not be the case for VDM-SL sets. Set data are usually represented using arrays.

We note that this problem is absent when testing at the system level, or through the interface, and that the more refined the specification is the less problematic one can expect the problem to be.

This problem is akin to the one encountered when verifying reifications between specifications where retrieval functions must be provided between concrete and abstract data types. This aspect is described in [84] by Jones. [107] addresses this problem from a more formal point of view.

To make explicit the need for data type transformation and to clarify the potential extra work required from the human tester, we illustrate the problem with the abstract set data type and the chosen concrete data type, array.

Consider the VDM-SL set input $\{3, 1, 2\}$ and its concrete representations: $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$, $[3, 2, 1]$. At the implementation level six representations are available for the same VDM-SL data. The set representation does not impose any order on its elements therefore, according to the specification, the behaviour of the system under test must be the same for each representation. This may not be the case and this requirement needs to be tested.

A test result of the form $[2, 3, 4]$ has a unique representation in a VDM-SL set: $\{1, 2, 3\}$.

Such transformations could be specified using VDM-SL. This could allow an ATG to be used to generate multiple test data for the same VDM-SL test input.

If those transformations have not been specified, the human tester must implement them. This could mask original errors—i.e. undetectable by performing the tests generated—present in the system under test if not performed adequately. More likely, an incorrect implementation of the transformations required, would create undue apparent inconsistencies in the system under test.

Thus, where the specification involves high-level abstract data types—such as sets and maps—we suggest that transformation functions be provided in the specification to test the system thoroughly for its intended behaviour, and reduce the amount of work imposed on the human tester as well as eliminate the risk of introducing undue errors.

A last point to mention is the ubiquity of floats as implemented in most programming languages. VDM-SL has a type real which is defined in the mathematical sense. Thus 1/3 is a real in VDM-SL but is not representable with floats. This suggests that the transformation between VDM-SL reals and the implementation language floats is not straight forward. This problem relates to precision testing: an area not yet considered from a formal specification point of view.

3.6 Instantiation and Consistency Checking

Implicitly, so far, we have mentioned the availability of a solver to perform consistency checking of the VDM-SL expressions generated during partitioning and as forming the core of the oracle. We will also need to sample the classes of the partition to obtain the set of tests.

Consider the following simple VDM-SL expression which could be part of a larger implicit operation specification:

$X = 0 \wedge \ \mathit{if}\ X = 0 \vee Y = 0\ \mathit{then}\ R = f(0)\ \mathit{else}\ R = f(1)$

As such, this expression could be replaced by the semantically identical expression: $X = 0 \wedge R = f(0)$. However, bad style specification writing cannot be ignored. Furthermore, the property that X must be null could be somewhere else in the specification or arise implicitly.

We can partition the original expression and, without consistency checking, obtain the following four equivalence classes:

$$X = 0 \wedge X = 0 \wedge Y \neq 0 \wedge R = f(0)$$
$$X = 0 \wedge X = 0 \wedge Y = 0 \wedge R = f(0)$$
$$X = 0 \wedge X \neq 0 \wedge Y = 0 \wedge R = f(0)$$
$$X = 0 \wedge X \neq 0 \wedge Y \neq 0 \wedge R = f(1)$$

(The first 3 classes aboves are obtained by partitioning $X = 0 \vee Y = 0$).

Two of the above classes are inconsistent thus when trying to sample them no instantiation can ever be found. This needs to be detected without relying on an exhaustive search which would be very inefficient and is not always possible to perform even with finite domain input and output variables. Note that if the entire partition is generated without consistency checking we may end up with a very large partition with most of its classes being inconsistent. Therefore, one can envisage checking the partition for consistency at each step of construction in order to avoid problems relating to the size of the partition generated. For the moment we present the techniques and tools available for such consistency checking as well as for generating a unique solution if required.

We now clarify our ideas by giving a series of definitions.

We recall the mathematical definition of a partition given in the previous chapter:

> Given the definition domain D of a predicate, a partition of D is a set of definition domain subsets, formally called equivalence classes, that are nonempty, disjoint and whose union is D itself.

We can express the equivalence classes using VDM-SL predicates which specify whether or not some value is part of the subset.

Definition 3.2 *Constraint*

A constraint can be represented as a predicate over domain variables.

Definition 3.3 *Domain Variable*

A domain variable is a pair $< x, D >$ where x is a variable (a symbol) and D is a finite set called the domain of the variable.

In VDM-SL all variables are domain variables [112, 113](i.e. their definition domain is finite). We can therefore view a VDM-SL predicate as a constraint, and vice versa, represent a constraint over VDM-SL variables using VDM-SL predicates. The terminology is also interchangeable.

Definition 3.4 *Satisfaction, Solution*

A set of variable instantiations satisfies a contraint if its associated predicate is true for such interpretation. Such a set of variable instantiations is called a solution.

Definition 3.5 *Consistency*

A predicate is consistent if its associated constraint is satisfiable.

Definition 3.6 *Constraint Satisfaction Problem*

A constraint satisfaction problem (CSP) is a set of constraints. A solution of a CSP must satisfy each constraint in the CSP.

These relations between VDM-SL predicates, constraints and sets of solutions justify our flexible use of the terminology. For example an equivalence class can be viewed as a set of values, a predicate or a constraint. It will also differentiate our problem from the more difficult problem of automatic theorem proving.

We can view an equivalence class expressed using VDM-SL predicates as a set of solutions for its associated CSP.

Definition 3.7 *Consistency Checking*

The consistency checking of a class is the process of determining the existence of a solution for its associated CSP.

If a solution exists the class is consistent, or satisfiable, otherwise it is inconsistent, or unsatisfiable, and actually denotes an empty set which cannot therefore be part of a partition.

Definition 3.8 *Sampling*

The sampling of a consistent class is the process of finding a solution, a sample, to its associated CSP.

Definition 3.9 *Solver*

A solver is an automatic tool that can determine the existence of solutions for CSPs and find a solution.

In the following sections we review the techniques available for implementing solvers over a given domain.

3.6.1 Constraint Satisfaction Problems

Constraint Satisfaction Problems (for an informal introduction to CSPs refer to [118]) are in general NP complete and a simple *generate and test* strategy, where a solution candidate is first generated then tested against the system of constraints for consistency, is not tractable. Constraint satisfaction problems have long been researched in artificial intelligence and many heuristics for efficient search techniques have been found. For example linear rational constraints (i.e. constraints expressed using linear arithmetic predicates only over rational numbers) can be solved using the well known simplex method [119].

Linear arithmetic predicates are built using the following relational operators: $>$, $\geq$, $<$, $\leq$, $=$ and $\neq$ over expressions composed of numbers, variables and the four basic arithmetic operators ($-$, $+$, $*$, $/$). The linearity property is satisfied if in a multiplication at most one of the operands is a variable, and if in a division the denominator is a number.

To implement the kind of solver we require one could implement these heuristics by writing a specialised program in a procedural language (such as C). But although the heuristics are readily available, this approach would still require a substantial amount of effort and the resulting program would be hard to maintain, modify and extend. Ideally we would like to concentrate on the *'what'* rather than the *'how'*, i.e. we are more interested in the problem of combining the heuristics to achieve our goal rather than in implementing the internal mechanism of each individual heuristic search technique.

The advantages of logic programming, mainly under the form of the Prolog programming language [120], over procedural programming have long been recognised: the *'what'* and the *'how'* are more easily separated since Prolog is based on first order predicate logic and has an in-built SLD-resolution computation mechanism [121]. However, Prolog's relative poor efficiency when compared to procedural languages like C has hindered its general acceptance.

For CSPs, however, Prolog is still the language of choice. Searches are facilitated by its in-built depth-first search procedure and its backtracking facilities. But even in this area Prolog suffers from a general lack of facilities to express complex relationships between objects (terms): the semantics of objects has to be explicitly coded into a term. This is the cause of the perceived poor mathematical handling capabilities of Prolog when compared with its other facilities: only instantiated mathematics can be dealt with readily. Further, the basic inbuilt depth-first strategy tends to lead to a *generate and test* approach to most problems: specialised heuristics must be implemented to prune the search space.

3.6.2 Constraint Logic Programming

Constraint Logic Programming (CLP), as introduced by Jaffar and Lassez [122], reviewed by Colmerauer [123] and discussed in [124], alleviates these short falls by providing richer data structures on which constraints can be expressed and by using constraint resolution mechanisms (also known as a decision procedure) to reduce the search space. When the decision procedure is incomplete—for example, for nonlinear arithmetic constraints—the problematic constraints are suspended—or delayed—until they become linear. Nonlinear arithmetic constraints can become linear whenever a variable is instantiated or bound. This can happen when other constraints are added to the system of constraints already considered or during labeling. The labeling mechanism further constrains the system of constraints according to some strategy, it can be viewed as a process to make assumptions about the system of constraints under consideration. It is a very powerful mechanism and it is used to awake delayed constraints or to generate a solution to an already known satisfiable system of constraints. The labeling strategies are critical to the overall efficiency of the solver. A discussion of constraint satisfaction using CLP can be found in [125].

For example, in Prolog the equality:

```
X*X + Y = 10
```

results in failure, since in Prolog equality only holds between syntactically identical terms and `X` is just a variable of no particular type. Using a CLP language however, it is possible to code the semantics of `X` and `Y` as being integer-like and constrain them such as `X*X + Y = 10` holds. The constraint resolution mechanism will detect the constraint as nonlinear and reduce it as follows:

`X = X, Y = Y`

`X*X + Y = 10` is delayed

i.e. the system of constraints is satisfiable (subject to consideration of the delayed constraints) and a simplified version is internally held. A labeling strategy must impose further constraints on either `X` or `Y` for the satisfiability of the system of constraints to be confirmed.

A not so efficient strategy would select `Y` to be sampled first. The sampling strategy would then instantiate `Y` with, say, 2 thus awaking and simplifying the delayed constraint to `X*X = 8`. This constraint would have to be delayed. The labeling strategy now attempts to instantiate `X` repeatedly without success (because failure actually occurs on the entire definition domain of `X`) which induces backtracking in the traditional, logic programming, manner. Eventually `Y` is instantiated to another value, say `1`, thus reducing the constraint store to:

`X = X, Y = 1`

`X*X = 9` is delayed

The labeling mechanism now attempts to awake the delayed constraint: this can only be achieved by instantiating `X` to some value. Eventually `X` will be instantiated with `-3` or `3` and the system of constraints will be declared satisfiable and the sample, say, `X = 3, Y = 1` will also be available. To succeed, this labeling strategy has generated thousands (if the initial domain of the variables is of that order) of futile assumptions.

A more efficient labeling strategy would recognise that `X` is the variable on which the linearity of the delayed constraint depends and attempt to constrain it first. Any value in the domain of `X` will make the delayed constraint linear thus allowing the constraint resolution mechanism of the underlying solver to detect satisfiability directly. E.g. `X = -5` will awake the delayed constraint and the solver would directly yield:

```
X = -5, Y = -15
```

This latest labeling strategy is obviously more suitable in this case as only one assumption is made before yielding a positive outcome.

We add that the decisions procedure employed in CLP languages are incrementable that is, given a system of constraints known to be satisfiable the addition of further constraints does not require the entire system of constraints to be rechecked. This is an advantage over theorem provers.

Clearly, CLP languages would be ideal for our purpose as their in-built resolution mechanism should remove most of the needed development effort and still offer the flexibility of logic programming. In fact they allow the rapid development of efficient dedicated solvers. They were first mentioned in relation to testing in 1986 by Choquet [126].

3.6.3 Constraint Logic Programming Languages

Work in CLP languages is quite recent and is progressing fast: there are now many CLP languages available. The CLP languages on offer differ mainly on the constraint domains they can handle and on the degree of completeness of their resolution mechanisms. Our basic requirements are for constraints over integers and real numbers (Boolean expressions can be dealt with appropriately, in our case, using the general logic programming scheme). As an example of a CLP language we cite ECL^{i}PS^e [127, 128] (ECRC Common Logic Programming System). ECL^{i}PS^e is a generic development system and has been designed to facilitate extensions (for example, ECL^{i}PS^e is written in itself and not hard coded in C as is usually the case for Prolog implementations). Many constraint solvers are available as libraries which allow wide ranging experimentations and can be used as examples for better comprehension of the general CLP scheme. The only reservation one could make about ECL^{i}PS^e would be in terms of efficiency when compared with more dedicated CLP languages (such as CLP($\Re$) [129], CHIP [130] or Prolog III [123]).

3.6.4 A Solver for VDM-SL Predicates?

Systems of constraints arising from VDM-SL specification would in general be NP complete to solve but the problem is simplified by the fact that all the variables have a finite domain (e.g. all sets in VDM-SL are finite). Also, we need to find only one solution satisfying the system of constraints amongst the set of possible solutions [131] to demonstrate satisfiability or generate tests.

The ATG presented in [13, 14] by Dick and Faivre is underpinned by a purpose built partition analysis tool—thus, without labeling capabilities—based on SEPIA Prolog. Since then, SEPIA has been integrated into ECL^{i}PS^e. Using a CLP language would simplify the implementation of such solvers and certainly make them more versatile. For example, in the solver developed by Dick and Faivre most of the 200 or so inference rules used for the simplification and consistency checking of the partition classes would be implicit in a CLP language. VDM-SL however is a complex language and set constraints must be handled and, although for instantiated sets constraint solvers do exist (e.g. [132]), set constraints involving arithmetic are still very much open to research.

Research in artificial intelligence is also focusing on adding constraints solving capabilities to theorem provers.

We should mention that recent attempts at generating tests from Z specifications using optimisation techniques such as genetic algorithms and simulated annealing are being made as described in [133]. Optimisation techniques for constraints solving are still very much under consideration and have yet to match conventional solvers in terms of efficiency. However, where conventional techniques fail optimisation techniques could be used. This is in fact an area of research for ECL^{i}PS^e II where optimisation techniques will be available to improve labeling should conventional methods fail.

However, we note that our technique will probably be limited by the performance of the underlying solver, i.e. features of VDM-SL may have to be eliminated to achieve full automation or human intervention will be required.

3.7 Conclusion

In this chapter, we have clarified many important aspects surrounding the generation of test cases from formal specifications. We have also discussed at length the testing activity proper in light of tests generated from VDM specifications.

Some choices have already been made concerning the approach taken—e.g. white tinted, no test generation for unspecified requirements—and their consequences examined.

Figure 3.1 represents a coarse design of our technique as discussed so far.

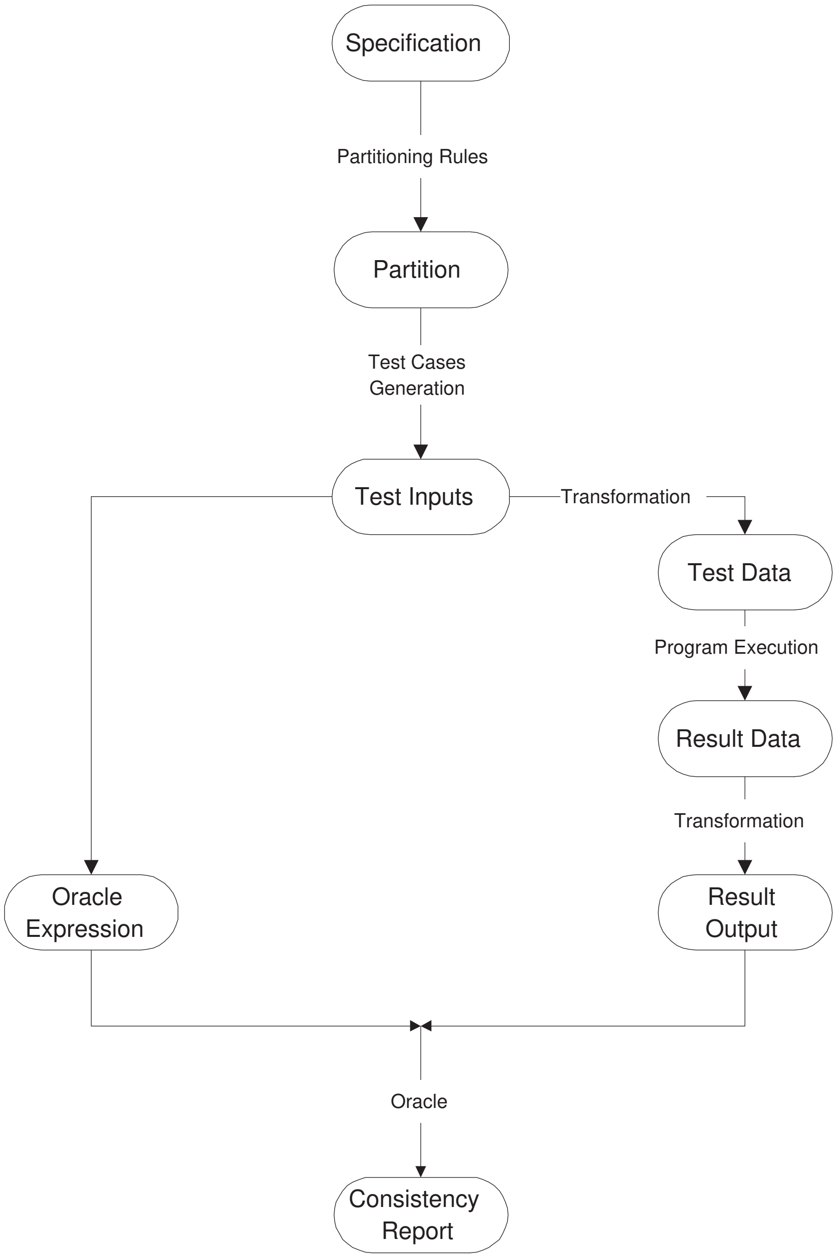

Figure 3.1: Overall Approach

We will now concentrate on partition generation and sampling techniques, and examine further oracle considerations.

Chapter 4

Systematic Partitioning

In this chapter, we first devise the basic partitioning technique necessary to the automatic processing of formal specifications with a view to generating test cases. Then, having illustrated the need for finer partitioning, we refine our technique to include lower level VDM-SL constructs. Using examples we show that the refinements introduced are valuable but problematic in relation to the number of test cases generated.

4.1 Partitioning VDM-SL Expressions

As suggested in the previous chapter, 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 [14] 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 semantic subtleties of VDM-SL were not considered. Dick and Faivre placed more emphasis on implementing a prototype tool than on exploring fundamental issues.

Below, we first attempt to justify the use of partitioning techniques for formal specifications and then present a flexible new formalism which will allow us to express partitioning rules. Using this formalism, we then derive partitioning rules for VDM-SL logical operators and control constructs taking into account the subtleties of the semantics of VDM-SL. We then extend our rules by taking into account VDM-SL three valued logic. Also, we show how to apply the technique in the context of VDM-SL as a language of expressions. We conclude this section by showing how the prototype built by Dick and Faivre could be improved to tackle larger specifications.

4.1.1 Justification: an Attempt

To apply the partitioning theory reviewed in the previous chapter, it must be mitigated, to be applied to formal specifications, by our ignorance as to what characterise homogeneous equivalent 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 homogenous 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. I.e. we assume an isomorphism between the mathematical classes and the theoretical classes of potentially misbehaving inputs. This isomorphism was actually implicit in our discussion of the theory of partitioning in chapter 2.

Although we have ruled out applying the partitioning technique to VDM-SL statements, implicit VDM-SL specifications can still be said to have a control structure, in forms of *if* and *cases* expressions, resembling imperative programs. Recalling how paths in structural testing can be elucidated and then individually tested as well as the testing efficiency claims attached to this technique we suggest that the same technique could be applied to VDM-SL specifications. 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 equivalent 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. Further by abuse of language and, for our purpose only, we use the same terminology in terms of paths.

Definition 4.1 *Path Expression*

We use the term path expressions for VDM-SL expressions which relate to the control structure of a specification (i.e. if, and cases expressions) or to logical expressions (e.g. expressions using the logical operators: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$)

Definition 4.2 *Path Partitioning Rule*

Path partitioning rules are rules allowing the generation of a partition from path expressions.

Definition 4.3 *Path*

A path is an equivalence class of a partition generated from a path expression using a path partitioning rule.

A path is infeasible if the class it represents is inconsistent. Further we shall denote by $\perp$ an inconsistent class.

4.1.2 Coarse Partitioning Rules

We now give the path partitioning rules in two-valued logic. The rules are similar to those of Dick [14] but are presented in a formal framework.

Logical Expressions

Logical expressions in VDM-SL are built from the common logical operators: negation, disjunction, conjunction, implication, equivalence, equality, inequality. The equivalence and equality operators are synonymous for Booleans while inequality stands for not equivalent. We shall first give the rules for negation, disjunction and conjunction, and then use these to build the others.

A conjunctive expression, $A \wedge B$, can only give rise to one class: there is only one possible instantiation (A is true and B is true). We note: $P(A \wedge B) = P(A) \times P(B)$, where P denotes the partitioning function and $\times$ the full combination operator.

Informally, we use the full combination operator to combine every class from one partition with every class from another.

Similarly, a negation expression, $\neg A$, where A is not a path expression gives rise to the class A is false. Thus $P(\neg A) = \neg A$.

As has been suggested, a disjunction of the form $A \vee B$, can be partitioned into three equivalence classes. This arises from the fact that only three distinct instantiations can be found for such expressions. Thus,

$$P(A \vee B) = P(A \wedge \neg B) \cup P(A \wedge B) \cup P(\neg A \wedge B)$$

Assuming that all three predicates are satisfiable, the set of VDM-SL predicates thus generated does indeed form a partition of $A \vee B$ because $(A \wedge \neg B) \oplus (A \wedge B) \oplus (\neg A \wedge B) \equiv A \vee B$ where $\oplus$ is the exclusive *or* operator.

We use this example to introduce and justify the notation used for partitions throughout the rest of this thesis.

Formally, 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:

$$(\forall i \in \{1, \ldots, n\} \cdot \mathit{consistent}(e_i)) \wedge (\bigoplus_{i=1}^{n} e_i \Leftrightarrow E)$$

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.

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

The partitioning function P has signature:

$\mathit{VDMexpression} \rightarrow \mathit{Partition}$.

The full combination operator $\times$ is defined by:

$\mathit{Partition} \times \mathit{Partition} \rightarrow \mathit{Partition}$

$P_1 \times P_2 = \{e_1 \wedge e_2 | e_1 \in P_1, e_2 \in P_2\}$

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

$$P(A \vee B) = \bigcup \begin{Bmatrix} P(A \wedge \neg B) \\ P(A \wedge B) \\ P(\neg A \wedge B) \end{Bmatrix}$$

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

Thus, $P((A \vee B) \wedge (C \vee D)) =$

$$\bigcup \begin{Bmatrix} P(A \wedge \neg B) \\ P(A \wedge B) \\ P(\neg A \wedge B) \end{Bmatrix} \times \bigcup \begin{Bmatrix} P(C \wedge \neg D) \\ P(C \wedge D) \\ P(\neg C \wedge D) \end{Bmatrix}$$

The full partition combination can be performed at this stage, if wished, to obtain:

$$\bigcup \left\{ \begin{matrix} P(A \wedge \neg B) \times P(C \wedge \neg D) \\ P(A \wedge \neg B) \times P(C \wedge D) \\ P(A \wedge \neg B) \times P(\neg C \wedge D) \\ P(A \wedge B) \times P(C \wedge \neg D) \\ P(A \wedge B) \times P(C \wedge D) \\ P(A \wedge B) \times P(\neg C \wedge D) \\ P(\neg A \wedge B) \times P(C \wedge \neg D) \\ P(\neg A \wedge B) \times P(C \wedge D) \\ P(\neg A \wedge B) \times P(\neg C \wedge D) \end{matrix} \right\}$$

and since $P(A \wedge B) = P(A) \times P(B)$:

$$\bigcup \left\{ \begin{matrix} P(A) \times P(\neg B) \times P(C) \times P(\neg D) \\ P(A) \times P(\neg B) \times P(C) \times P(D) \\ P(A) \times P(\neg B) \times P(\neg C) \times P(D) \\ P(A) \times P(B) \times P(C) \times P(\neg D) \\ P(A) \times P(B) \times P(C) \times P(D) \\ P(A) \times P(B) \times P(\neg C) \times P(D) \\ P(\neg A) \times P(B) \times P(C) \times P(\neg D) \\ P(\neg A) \times P(B) \times P(C) \times P(D) \\ P(\neg A) \times P(B) \times P(\neg C) \times P(D) \end{matrix} \right\}$$

which when A, B, C and D are not path expressions—that is $P(A) = \{A\}$ and $P(\neg A) = \{\neg A\}$—and by performing the $\times$ operation is identical to:

$$\bigcup \left\{ \begin{matrix} \{A \wedge \neg B \wedge C \wedge \neg D\} \\ \{A \wedge \neg B \wedge C \wedge D\} \\ \{A \wedge \neg B \wedge \neg C \wedge D\} \\ \{A \wedge B \wedge C \wedge \neg D\} \\ \{A \wedge B \wedge C \wedge D\} \\ \{A \wedge B \wedge \neg C \wedge D\} \\ \{\neg A \wedge B \wedge C \wedge \neg D\} \\ \{\neg A \wedge B \wedge C \wedge D\} \\ \{\neg A \wedge B \wedge \neg C \wedge D\} \end{matrix} \right\}$$

Finally, performing the distributed union we obtain:

$$\left\{\begin{matrix} A \wedge \neg B \wedge C \wedge \neg D \\ A \wedge \neg B \wedge C \wedge D \\ A \wedge \neg B \wedge \neg C \wedge D \\ A \wedge B \wedge C \wedge \neg D \\ A \wedge B \wedge C \wedge D \\ A \wedge B \wedge \neg C \wedge D \\ \neg A \wedge B \wedge C \wedge \neg D \\ \neg A \wedge B \wedge C \wedge D \\ \neg A \wedge B \wedge \neg C \wedge D \end{matrix}\right\}$$

We note that, because classes are always conjunctive expressions our notation could be changed to keep the notion of a partition as a set of sets: an equivalence class $\bigwedge_{i=1}^{n} e_i$ could be denoted by $\{e_i | i \in \{1 \ldots n\}\}$. However, this is has been avoided to retain a concise notation.

Given the *or*, *and* and *not* rules and using the common relationships between operators (such as $A \Rightarrow B$ is equivalent to $\neg A \vee B$), we can obtain partitioning rules for all VDM-SL logical operators:

$P(\perp) = \emptyset$

$P(A) = \{A\}$ when A is not a path expression

$P(\neg A) = \{\neg A\}$ when A is not a path expression

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

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

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

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

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

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

When in $P(\neg A)$, A is a path expression we develop $\neg A$ and use the other partitioning rules. For example $P(\neg(A \vee B)) = P(\neg A \wedge \neg B)$ and using the partitioning rule for conjunctive expressions it is equivalent to $P(\neg A) \times P(\neg B)$ and if A and B are not path expressions then we find $P(\neg(A \vee B)) = \{\neg A\} \times \{\neg B\}$ which yields $\{\neg A \wedge \neg B\}$

On a larger example, the sequence of partitioning operations for the expression: $E = (((x = 0 \vee y = 0) \Rightarrow x + y = 0) \wedge D = no)$ is as follows:
Using the $\wedge$ rule: $P(E) = P((x = 0 \vee y = 0) \Rightarrow x + y = 0) \times P(D = no)$
Using the $\Rightarrow$ rule:

$$P(E) = \bigcup \left\{ \begin{array}{c} P(\neg(x = 0 \vee y = 0) \wedge \neg(x + y = 0)) \\ P(\neg(x = 0 \vee y = 0) \wedge x + y = 0) \\ P((x = 0 \vee y = 0) \wedge x + y = 0) \end{array} \right\} \times \{D = no\}$$

Using the $\wedge$ rule:

$$P(E) = \bigcup \left\{ \begin{array}{c} P(\neg(x = 0 \vee y = 0)) \times P(\neg(x + y = 0)) \\ P(\neg(x = 0 \vee y = 0)) \times P(x + y = 0) \\ P(x = 0 \vee y = 0) \times P(x + y = 0) \end{array} \right\} \times \{D = no\}$$

Applying the $\neg$ operator:

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

Using the $\vee$ rule:

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

Using the $\wedge$ rule:

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

Applying the innermost $\times$ operators:

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

Applying the internal distributed union:

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

Applying the internal combination operator:

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

Applying the last distributed union:

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

Applying the remaining $\times$ operator:

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

To obtain the final partition the paths are individually checked for inconsistencies to obtain:

$$P(E) = \begin{Bmatrix} x \neq 0 \wedge y \neq 0 \wedge x + y \neq 0 \wedge D = no \\ x \neq 0 \wedge y \neq 0 \wedge x + y = 0 \wedge D = no \\ x = 0 \wedge y = 0 \wedge x + y = 0 \wedge D = no \end{Bmatrix}$$

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

$$\begin{Bmatrix} x \neq 0 \wedge y \neq 0 \wedge x \neq -y \wedge D = no \\ x \neq 0 \wedge x = -y \wedge D = no \\ x = 0 \wedge y = 0 \wedge D = no \end{Bmatrix}$$

From here on, we shall not give the details of the intermediary steps of partition derivation.

Control expressions

We now consider the composite VDM-SL expressions: *if* and *cases* expressions.

If expressions are of the form:

```
if expression =
   'if', expression, 'then', expression,
   {elseif expression clause},
   'else', expression;
elseif expression clause =
   'elseif', expression, 'then', expression;
```

A simple *if* expression of the form: *if* B *then* E_1 *else* E_2 can be partitioned into:

$$\bigcup \begin{Bmatrix} P(B \wedge E_1) \\ P(\neg B \wedge E_2) \end{Bmatrix}$$

More generally:

$if\ B_1\ then\ E_1$

$\quad elseif\ B_2\ then\ E_2$

$\quad \vdots$

$\quad elseif\ B_n\ then\ E_n$

$else\ E$

gives rise to the following partition:

$$\bigcup \left\{ \begin{array}{c} P(B_1 \wedge E_1) \\ P(\neg B_1 \wedge B_2 \wedge E_2) \\ \vdots \\ P(\neg B_1 \wedge \neg B_2 \wedge \cdots \wedge \neg B_{n-1} \wedge B_n \wedge E_n) \\ P(\neg B_1 \wedge \neg B_2 \wedge \cdots \wedge \neg B_n \wedge E) \end{array} \right\}$$

Cases expressions are of the form:

```
cases expression =
   'cases', expression, ':',
   expression choice list, [',', others expression choice], 'end';
expression choice list =
   expression choice, {',', expression choice};
expression choice =
   pattern list, '->', expression;
others expression choice =
   'others', '->', expression;
```

So that for the expression:

$cases\ E:$

$\quad Pt_{1,1}, \ldots, Pt_{1,n_1} \rightarrow E_1$

$\quad \vdots$

$\quad Pt_{n,1}, \ldots, Pt_{n,n_m} \rightarrow E_n$

$\quad others \rightarrow E$

end

the following partitioning rule is devised:

$$\bigcup \left\{ \begin{array}{l} P((bind(E, Pt_{1,1}) \vee ... \vee bind(E, Pt_{1,n_1})) \wedge E_1) \\ P(\neg bind(E, Pt_{1,1}) \wedge ... \wedge \neg bind(E, Pt_{1,n_1}) \wedge \\ \quad (bind(E, Pt_{2,1}) \vee ... \vee bind(E, Pt_{2,n_2})) \wedge E_2) \\ \vdots \\ P(\neg bind(E, Pt_{1,1}) \wedge ... \wedge \neg bind(E, Pt_{1,n_1}) \wedge \cdots \\ \quad \wedge \neg bind(E, Pt_{n-1,1}) \wedge ... \wedge \neg bind(E, Pt_{n-1,n_{n-1}}) \\ \quad \wedge (bind(E, Pt_{n,1}) \vee ... \vee bind(E, Pt_{n,n_m})) \wedge E_n) \\ P(\neg bind(E, Pt_{1,1}) \wedge ... \wedge \neg bind(E, Pt_{1,n_1}) \wedge \\ \quad \neg bind(E, Pt_{n,1}) \wedge ... \wedge \neg bind(E, Pt_{n,n_m}) \wedge E) \end{array} \right\}$$

Where *bind* is semantically equivalent to the VDM-SL binding function and $\neg bind(Pt, E)$ is equivalent to $\neg match(Pt, E)$.

Patterns are omnipresent throughout VDM-SL, they can be found in *cases*, quantified, *let* and comprehension expressions. The matching and binding process are described in [113], further considerations can be found in [107]. There are two forms of binds, set binds or type binds, and nine pattern forms. Some examples are given below:

- $x \in S$: a set bind with an identifier pattern
- $mk{-}T(x, \_) : T$: a type bind with a record pattern, an identifier pattern and a don't care pattern
- $\{m, n\} \in S$: a set bind with a set enumeration pattern
- $mk{-}(x, y) : \mathbb{N} \times \mathbb{N}$: a type bind with a tuple pattern
- $S \cup T \in \{\{1, 2, 3\}, \{4, 5, 6\}\}$: a set bind with a set union pattern
- $[x, y, z] \in \{[1, 2, 3], [4, 5, 6]\}$: a set bind with a sequence enumeration pattern
- $S_1 \curvearrowright S_2 \in \{[7, 8, 9]\}$: a set bind with a sequence concatenation pattern

The matching and subsequent binding processes have a very subtle semantics. Potentially, set enumeration, set union and sequence patterns can behave non deterministically and introduce looseness in expressions (expressions for which the evaluation process yields several possible outcomes). For example, the pattern $S \cup T$ can be matched against the set $\{1,2,3\}$ but this yields several bindings (e.g. $(S = \emptyset, T = \{1,2,3\}), (S = \{2\}, T = \{1,3\})$ etc.).

Loose expressions are notoriously difficult to reason about. We will return to them later in this thesis by considering ways to include them in our technique. For now, and to preserve the notation developed so far, we exclude in the set of VDM-SL constructs for which our partitioning rules apply bindings which introduce looseness. Note that we do not totally exclude the patterns mentioned above, for example, $[1] \frown tl \in \{[1,2,3],[4,5,6],[7,8,9]\}$ does not introduce looseness. However, recognition that such bindings are not loose may not be straightforward nor be feasible automatically.

We now return to the partitioning of *case* expressions by noting that it takes into account the non deterministic choice made when E matches more than one pattern within an expression choice.

To conclude, our formalism clearly shows how the partitioning rules presented so far can mechanically be combined. This could be performed using a dedicated parser. There are, however, two particular areas of difficulty to consider:

- the fact that VDM-SL is based on three valued logic: LPF (Logic of Partial Functions).
- path expressions within non path expressions e.g. $r = x + \mathit{if}\ B\ \mathit{then}\ y\ \mathit{else}\ z$.

4.1.3 Three Valued Logic: LPF and its Consequences

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: *if* $x/y = 0$ *then* M *else* N which once partitioned 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 undefined, further, still according to the semantics of VDM-SL for *if* expressions N should be satisfied. But, in our partition both expressions would then evaluate to undefined, 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 expressions.

This incompleteness can have dire consequences. Consider the piece of specification: $y = 0 \wedge \mathit{if}\ x/y = \mathit{0}\ \mathit{then}\ M\ \mathit{else}\ N$. This expression 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:

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

where $*$ denotes undefinedness (i.e. for $B*$ to be satisfied B must be undefined).

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

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

The other partitioning rules are modified according to VDM-SL three valued logic in a similar way. We give, for reference, in 4.1 the truth tables of the VDM-SL logical operators.

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.

B_1	B_2	$\neg B_1$	$B_1 \vee B_2$	$B_1 \wedge B_2$	$B_1 \Rightarrow B_2$	$B_1 \Leftrightarrow B_2$	$B_1 = B_2$	$B_1 \neq B_2$
T	T	F	T	T	T	T	T	F
T	F	F	T	F	F	F	F	T
T	*	F	T	*	*	*	*	*
F	T	T	T	F	T	F	F	T
F	F	T	F	F	T	T	T	F
F	*	T	*	F	T	*	*	*
*	T	*	T	*	T	*	*	*
*	F	*	*	F	*	*	*	*
*	*	*	*	*	*	*	*	*

Table 4.1: LPF Truth Table for the Logical Operators

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 paths in specifications. But most of these paths will be infeasible if common specification writing style is assumed. If one supposes 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 to check for consistency. However, if such annotations are absent, too unreliable, or if the object is to test the specification then one should be left in no doubts as to the necessity of the extra effort required.

4.1.4 Embedded Path Expressions

Definition 4.4 *Embedded Path Expression*

We call embedded path expressions, path expressions which are contained within a non-path expression.

For example, the path expression *if B then y else z* contained in $r = x +$

if B then y else z is embedded and cannot be partitioned directly using our rules as presented so far.

To leave expressions such as $r = x + if\ B\ then\ y\ else\ z$ intact during the partitioning process, thus testing them only once, would not achieve great coverage of their potential behaviour. The internal, or embedded, *if* expression should be subject to our partitioning rules.

We remark that this expression has the same semantics as $if\ B\ then\ r = x + y\ else\ r = x + z$ where the internal *if* expression has been brought to the fore.

Thus, we can now apply our previous *if* partitioning rules to obtain:

$$\bigcup \left\{ \begin{array}{c} P(B \wedge r = x + y) \\ P(\neg B \wedge r = x + z) \\ P(B* \wedge r = x + z) \end{array} \right\}$$

This is a refined partition which, in this case, and according to our ad hoc partitioning rules, should improve the homogeneity of the classes generated. Here the first expression can be thought of as a simple shorthand for the second expression.

However this technique, of bringing path expressions to the fore, is not always applicable. For example within the scope of quantified expressions: $\forall x \in S \cdot x > 0 \vee x \leq 0$ is not semantically equivalent to $(\forall x \in S \cdot x > 0) \vee (\forall x \in S \cdot x \leq 0)$. But $\exists x \in S \cdot x > 3 \vee x = 2$ is equivalent to $(\exists x \in S \cdot x > 3) \vee (\exists x \in S \cdot x = 2)$.

An expression of the form $\exists bind \cdot E_1 \vee E_2$ could be partitioned into:

$$\left\{ \begin{array}{c} (\exists bind \cdot E_1) \wedge (\forall bind \cdot \neg E_2) \\ (\exists bind \cdot E_1) \wedge (\exists bind \cdot E_2) \\ (\forall bind \cdot \neg E_1) \wedge (\exists bind \cdot E_2) \end{array} \right\}$$

using the equivalence: $\exists bind \cdot E_1 \vee E_2 \equiv (\exists bind \cdot E_1) \vee (\exists bind \cdot E_2)$.

However, this may be stretching the notion of paths in specifications too far. More importantly, something very subtle in the semantics of VDM-SL in respect to testing for misbehaviour may well be lost by this approach as many interesting test cases could not be elucidated. For example the following classes are not revealed: $(\forall bind \cdot E_1 \wedge \neg E_2)$ and $(\exists bind \cdot E_1) \wedge (\exists bind \cdot \neg E_1) \wedge (\forall bind \cdot \neg E_2)$.

This problem arises for other VDM-SL expressions containing embedded path expressions. In fact, whenever a local scope is introduced, bringing path expressions to fore is not directly feasible. VDM-SL constructs which introduce a local scope are: quantified expressions as well as sequence, set and map comprehension expressions.

Definition 4.5 *Scoping Expression*

We use the term scoping expressions to designate quantified expressions as well as sequence, set and map comprehension expressions.

Thus, we decide that path expressions should only be brought to the fore within non scoping expressions. This solves our first dilemma of having to specify constructs for which path expressions can be brought to the fore (e.g. an *or* expression in an existential quantified expression but not in a universal quantified expression). It also does not compromise the potential extensive behaviour coverage of scoping expressions.

Hence, the technique developed so far leaves VDM-SL constructs which introduce variable scoping intact within partitions. Scoping constructs will be considered later in this chapter.

4.1.5 Implementation Considerations

Having extended the basic partitioning rules of Dick and Faivre [14] we must ask ourselves if their tool implementation could withstand this generalisation.

Leaving consistency checking apart, for which we feel, as already discussed in the previous chapter, constraint programming offers the best potential, Dick and Faivre's tool is based on simple prolog generation rules such as $A \vee B \vdash (A \wedge B)\ OR\ (\neg A \wedge B)\ OR\ (A \wedge \neg B)$. We have replaced this notation and extended the *or* rule by:

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

which can just as easily be automatically generated as our rules are simple syntactic manipulations. Therefore, an implementation of a tool to generate our extended partitions is feasible.

Dick and Faivre's tool, as described in [13], could not process very large specifications as these could generate numerous very large expressions. We must address this aspect as our final partitions are potentially even larger due to our treatment of LPF and, as we shall see later, our partition refining technique.

Annotations, as proposed when reviewing the LPF behaviour of some constructs, could be used, and even extended to cover the non deterministic choices within *case* constructs, to reduce the size of the intermediate partitions as well as the final one.

Further, [13, 14] advocates reducing the VDM-SL specification into Disjunctive Normal Form (DNF) prior to using the partition generation rules. For example, the post condition of the *max* operation on two operands a and b of the form:

$$(max = a \vee max = b) \wedge max \geq a \wedge max \geq b$$

is first reduced into DNF in [14] (by distributing the $\vee$) to obtain:

$$(max = a \wedge max \geq a \wedge max \geq b) \vee (max = b \wedge max \geq a \wedge max \geq b)$$

on which the *or* generation rule is applied to get:

$$\begin{Bmatrix} max = a \wedge max = b \wedge max \geq a \wedge max \geq b \\ max = a \wedge max \neq b \wedge max \geq a \wedge max \geq b \\ max \neq a \wedge max = b \wedge max \geq a \wedge max \geq b \end{Bmatrix}$$

which is then simplified to:

$$\begin{Bmatrix} max = a \wedge max = b \\ max = a \wedge max > b \\ max = b \wedge max > a \end{Bmatrix}$$

DNF reduction can, as acknowledged by Dick and Faivre, generate very large intermediate expressions. Further, the *or* rule will always be performed on larger expressions than were present in the original expression—manual derivation using the *or* rule on the above would clearly show how this can lead to long manipulations of large expressions with numerous extra consistency checks. Studying the partitioning process or using our formalism we can clearly see that DNF reduction is superfluous.

Thus, we would partition the post condition of max directly as follows:

$$\bigcup \left\{ \bigcup \left\{ \begin{array}{c} \{max = a \wedge max \neq b\} \\ \{max = a \wedge max = b\} \\ \{max \neq a \wedge max = b\} \\ \{max = a \wedge (max = b)*\} \\ \{(max = a)* \wedge max = b\} \end{array} \right\} \times \{max \geq a \wedge max \geq b\} \right\}$$

which when processed is equal to

$$\left\{ \begin{array}{l} max = a \wedge max = b \\ max = a \wedge max > b \\ max = b \wedge max > a \end{array} \right\}$$

if $max = b$ and $max = a$ cannot be made undefined.

Hence, there is no need for explicit DNF reduction as it is an intrinsic part of the partitioning process: the classes generated are always conjunctions of non path expressions and, by construction, mutually exclusive and complete. This way of proceeding is considerably less computationally intensive.

In Dick and Faivre's tool the partition is generated using purely syntactic rules—our partitioning rules are also purely syntactic, i.e. only symbolic manipulations are required—from a DNF expression. Every path in the partition is then checked for individual satisfiability.

To conclude, our partitions, even if potentially larger than those generated by the Dick and Faivre's tool (because we take into account the LPF behaviour of specifications), could be readily generated. Hence the shortcomings of our technique do not lie so much with implementability problems but with the more fundamental issue of specification behaviour coverage.

4.2 Refinements

Our test generation technique has so far been based on the work of Dick and Faivre. After having extended their results we need to address the now acknowledged problem of simple logical partitioning: the classes generated seem coarse in comparison with manually generated classes and tests. Below we illustrate this problem and address it.

4.2.1 Is the Partitioning Too Coarse?

So far we have not presented an evaluation of our technique. The Triangle Problem introduced in the previous chapter will suffice to illustrate its inadequacy.

To evaluate our technique, we will compare North's test set [11] for the Triangle Problem, which we have deemed adequate, and Dick and Faivre's automatically generated tests [13]. We can do this because our technique as described so far is similar to Dick and Faivre's.

While North in [11] manually derived thirty-six test cases, on the recommendations of Myers [18], for the Triangle Problem, Dick and Faivre's tool generates only eight classes as illustrated in [13] for which a single random sample can be taken. The eight tests automatically generated and their North's equivalent according to Table 3.1 are listed in table 4.2.

Test Input	Oracle	North Ref.
$[77, 35, 50]$	Scalene	31
$[71, 71, 33]$	Isosceles	27
$[25, 14, 14]$	Isosceles	28
$[42, 39, 42]$	Isosceles	26
$[74, 74, 74]$	Equilateral	24
$[1, 1, 10, 36, 95]$	Invalid	20
$[9, 13, 26, 27]$	Invalid	20
$[]$	Invalid	20

Table 4.2: Dick and Faivre's Test Cases for the Triangle Problem

That the automatically generated tests are represented in North's test set is encouraging.

However, as we considered North's test set to be adequate for the Triangle Problem, the large number of missing tests is difficult to accommodate. For example, when North deems necessary to manually derive twenty tests to cover the `INVALID` outcome, only three are automatically generated by the current technique. Even those three tests can be said to test only one requirement—the sequence of integers must be of length three—although this is arguable. The other missing tests are due to the absence of permutations in the sequence of integers for isosceles and scalene triangles.

Further, boundary values are not covered by the tool. Although this omission is mentioned in [13] the issue is not discussed further. Even if boundary value generation was perfectly implemented, around twenty-four tests would still be missing.

Our extended partitioning technique would not substantially improve the quality of the coverage as LPF plays only a limited role in North's specification of the Triangle Problem.

One can only conclude that the classes generated are far from being homogenous, i.e. they are too coarse to allow elucidation of interesting tests through single random sampling. This was already hinted when trying to tackle VDM-SL scoping constructs for which our technique does not yet apply.

As Stocks et al. recently remarked, with an implicit reference to Dick and Faivre's work, in [16]:

> 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 $\cup, \cap, \geq, \lhd$ must also be used to generate adequate test sets.

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 non-path 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 scoping constructs. We now illustrate how this can be achieved.

4.2.2 Partitioning Expressions Using Non Logical Operators

As we have seen expression constructed from basic operators need to be partitioned. So for example, assuming that x is real: $P(x \geq 6) = \begin{Bmatrix} x = 6 \\ x > 6 \end{Bmatrix}$. Although this partition may seem obvious, any partition would theoretically be valid e.g.

$$P(x \geq 6) = \begin{Bmatrix} x \geq 6 \wedge x < 10 \\ x \geq 10 \wedge x < 20 \\ x \geq 20 \end{Bmatrix}.$$

The ultimate choice of partitioning rules must be based here on the informal notion of the likelihood of finding an error in the system under test. Again, few robust arguments can be given to justify our partition choices. However, we will attempt to do so whenever possible.

Consider $P(x > 6)$, a reasonable partition might simply be $\{x > 6\}$. However to test for boundary conditions, the partition : $\left\{\begin{matrix} x = 6 + \delta \\ x > 6 + \delta \end{matrix}\right\}$ where δ is the smallest increment for the type of x (e.g. for integers δ is 1 and for reals δ is the specified requested precision of real variables) is more suitable. For integers the test $x = 7$ certainly has potential to reveal errors whereas for real variables the above partition could be thought of as enclosing tests for precision testing.

So even for simple operators, the choice of a reasonable partitioning rule can be large and should ultimately rest with the degree of test quality requested. Some partitioning rules for expressions composed of basic VDM-SL operators are conventions (e.g. for $>$ and $\geq$). Other expressions can be given partitioning rules according to basic conventions and our partitioning technique.

For example, to obtain a justified partitioning rule for expressions of the form $x \neq y$ we can start by the basic equivalence: $x \neq y \equiv x > y \vee x < y$. Therefore, whenever x and y are assumed to be integers, $P(x \neq y) =$

$$\bigcup \left\{\begin{matrix} P(x > y) \times P(x \geq y) \\ P(x > y) \times P(x < y) \\ P(x \leq y) \times P(x < y) \end{matrix}\right\}$$

Thus, $P(x \neq y) =$

$$\bigcup \left\{ \begin{array}{l} \left\{ \begin{array}{l} x = y + 1 \wedge x = y \\ x > y + 1 \wedge x = y \\ x = y + 1 \wedge x > y \\ x > y + 1 \wedge x > y \end{array} \right\} \\ \left\{ \begin{array}{l} x = y + 1 \wedge x + 1 = y \\ x = y + 1 \wedge x + 1 < y \\ x > y + 1 \wedge x + 1 = y \\ x > y + 1 \wedge x + 1 < y \end{array} \right\} \\ \left\{ \begin{array}{l} x = y \wedge x + 1 = y \\ x = y \wedge x + 1 < y \\ x < y \wedge x + 1 = y \\ x < y \wedge x + 1 < y \end{array} \right\} \end{array} \right\}$$

so after simplifications $P(x \neq y) =$

$$\left\{ \begin{array}{l} x = y + 1 \\ x > y + 1 \\ x + 1 = y \\ x + 1 < y \end{array} \right\}$$

Stocks and Carrington [15, 16] present a framework for specification based testing using Z as a purpose language. They discuss test derivation, test oracle, regression testing and test suites. We are interested here, in their informal partitioning for expressions constructed from binary operators. We note that their work does not discuss automation and its coverage of tests derivation from formal specification is limited.

We give in Table 4.3 their informal domain division for expressions based on set union operator (the last sub-domain as given in [15] was rectified in [16], the correct version is given here) as well as the expected outcome.

How can these divisions be justified? First we should note that the 8 predicates form a partition of the expression $S \cup T = R$ (they are satisfiable, mutually exclusive and their disjunction is equivalent to $S \cup T = R$).

We can also give an informal justification in figure 4.1 by drawing all the Venn diagrams that can be obtained from an expression of the form $S \cup T$. We

Ref.	Domain Division
1	$S = \{\} \wedge T = \{\} \wedge R = \{\}$
2	$S = \{\} \wedge T \neq \{\} \wedge R = T$
3	$S \neq \{\} \wedge T = \{\} \wedge R = S$
4	$S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\} \wedge R = S \cup T$
5	$S \neq \{\} \wedge T \neq \{\} \wedge S \subset T \wedge R = T$
6	$S \neq \{\} \wedge T \neq \{\} \wedge T \subset S \wedge R = S$
7	$S \neq \{\} \wedge T \neq \{\} \wedge S = T \wedge R = S$
8	$S \neq \{\} \wedge T \neq \{\} \wedge S \cap T \neq \{\}$
	$\wedge \neg(S \subset T) \wedge \neg(T \subset S)$
	$\wedge S \neq T \wedge R = S \cup T$

Table 4.3: Stocks et al. Domain Division of $S \cup T = R$

give with each diagram its equivalent in the informal derivation.

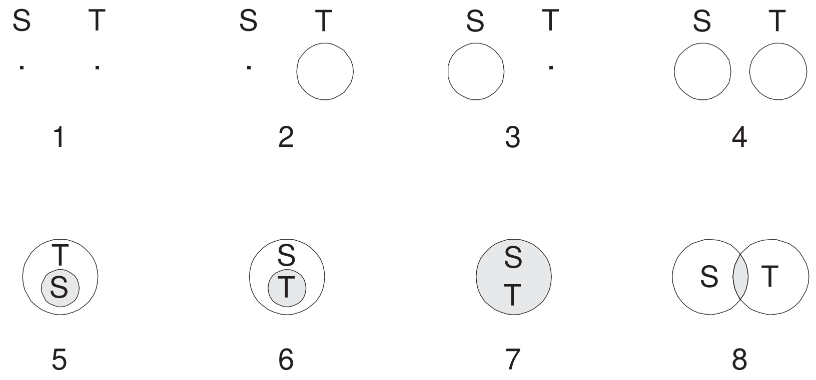

Figure 4.1: Venn Diagrams for the Partition of $S \cup T = R$

So we can see how expressions involving basic operators can be partitioned.

One of the problems not discussed by Stocks and Carrington is whether all expressions constructed from basic operators should be partitioned. If we allow the partitioning of expressions of the form $S \cup T = R$, for example, what about $x + y = r$? For example:

$$P(r = x + y) = \begin{Bmatrix} x = 0 \\ x > 0 \\ x < 0 \end{Bmatrix} \times \begin{Bmatrix} y = 0 \\ y > 0 \\ y < 0 \end{Bmatrix} \times \begin{Bmatrix} abs(x) = abs(y) \\ abs(x) > abs(y) \\ abs(x) < abs(y) \end{Bmatrix} \times \{r = x + y\}$$

This partition is again highly subjective and it may seem pointless to generate such a complex partition for one of the most simple expressions. However it could be argued that the equivalence classes containing the constraint $x = -y$ would be of value. Certainly for expressions containing more complex arithmetic operators such as *mod* and *rem* partitioning should be performed given the likelihood of confusion between the two.

Although we can keep our formalism for path expressions partitioning we change the terminology to differentiate the equivalence classes generated from non path expressions.

Definition 4.6 *Sub-domain*

A sub-domain is an equivalence class of a partition generated from a non path expression using a non path partitioning rule.

We conclude on the partitioning of basic expressions by noting that:

- many different partitions of the same expression can be generated.
- the basic choices can be informally justified using the likelihood of error detection (which is very subjective).
- the formalism, developed for coarse partitioning, can be re-used for non path expressions partitioning.

An area not discussed by Dick and Faivre nor by Stocks and Carrington is the partitioning of scoping expressions. Scoping expressions are omnipresent in formal languages and cannot certainly be left as such during partitioning otherwise many specifications would simply give rise to few, complex, classes, or even to a single class if the entire specification under consideration is, for example, a large quantified expression.

4.3 Partitioning Scoping Expressions

We recall that scoping expressions in VDM-SL take the forms of quantified expressions as well as sequence, set and map comprehension expressions. They are more difficult to manipulate because of the complexity of their semantics. We illustrate our approach by considering set comprehension expressions and quantified expressions and identify problematic areas concerning their semantics.

4.3.1 Quantified Expressions

We now show how quantified expressions can be partitioned into sub-domains. An existential quantified expression has the form $'\exists'$, `bindlist`, $'\cdot'$, `expression` and a universal quantified expression $'\forall'$, `bindlist`, $'\cdot'$, `expression`. For simplicity we only consider cases where the binding expression is of the form $x \in e_1$.

If there is no binding possible an existential quantified expression evaluates to *false* and an universal quantified expression evaluates to *true*. The use of LPF in quantified expressions also needs to be highlighted. Informally, an existential quantified expression is undefined if no binding evaluates to *true* and there is a binding which evaluates to $*$ (undefined). A universally quantified expression is undefined if no binding evaluates to *false* and all bindings not evaluating to *true* are undefined. Also, if the binding process is itself undefined, quantified expressions evaluate to undefined.

Formally, we propose $P((\exists x \in e_1 \cdot exp_2)*) =$

$$\bigcup \left\{ \begin{array}{l} P((e_1 \in x)*) \\ \{(\exists x \in e_1 \cdot exp_2)*\} \times P(\exists x \in e_1 \cdot exp_2*) \end{array} \right\}$$

and $P((\forall x \in e_1 \cdot exp_2)*) =$

$$\bigcup \left\{ \begin{array}{l} P((e_1 \in x)*) \\ \{(\forall x \in e_1 \cdot exp_2)*\} \times P(\exists x \in e_1 \cdot exp_2*) \end{array} \right\}$$

We now consider $P(\exists x \in e_1 \cdot exp_2)$.

Let $P(exp_2) = \{c_1, \ldots, c_n\}$ then we propose $P(\exists x \in e_1 \cdot exp_2) =$

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

This stems from $(\exists x \in e_1 \cdot exp_2) \Leftrightarrow ((\forall x \in e_1 \cdot exp_2) \oplus ((\exists x \in e_1 \cdot exp_2) \wedge (\exists x \in e_1 \cdot \neg exp_2 \oplus exp_2*)))$ and from $(\exists x \in e_1 \cdot exp_2) \Leftrightarrow (\exists x \in e_1 \cdot c_1 \vee \cdots \vee c_n) \Leftrightarrow ((\exists x \in e_1 \cdot c_1) \vee \cdots \vee (\exists x \in e_1 \cdot c_n))$.

Further we propose, $P(\forall x \in e_1 \cdot exp_2) =$

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

To obtain the above, we have used the *or* rule. If LPF behaviour is disabled then, of course, the rules become considerably easier to manipulate.

We can easily show that $P(\forall x \in e_1 \cdot exp_2)$ where $P(exp_2) = \{exp_2\}$ is equal to:

$$\bigcup \left\{ \begin{array}{l} P(e_1 = \{\}) \\ P(e_1 \neq \{\}) \times \{\forall x \in e_1 \cdot exp_2\} \end{array} \right\}$$

and that $P(\exists x \in e_1 \cdot exp_2)$ where $P(exp_2) = \{exp_2\}$ is equal to:

$$\bigcup \left\{ \begin{array}{l} P(e_1 \neq \{\}) \times \{\forall x \in e_1 \cdot exp_2\} \\ \{\exists x \in e_1 \cdot exp_2\} \times \{\exists x \in e_1 \cdot \neg \, \mathrm{exp}_2 \oplus exp_2 *\} \end{array} \right\}$$

This conforms with the intuition that for example when S is a non empty set of integers, $P(\forall x \in S \cdot x = 6) = \{\forall x \in S \cdot x = 6\}$.

The partitioning of quantified expressions can result in a large number of classes, however the partitions generated can be shown to be adequate.

For example, consider $P(\exists x \in e_1 \cdot A \vee B)$.

The partition of $A \vee B$ is: $\bigcup \left\{ \begin{array}{l} P(A) \times P(\neg B) \\ P(A) \times P(B) \\ P(\neg A) \times P(B) \end{array} \right\}$

in two valued logic. We limit this example to two valued logic (i.e. we assume $P(A*)$ and $P(B*)$ are inconsistent) for concision. Further we assume that A and B are not path expressions and that $P(A) = \{A\}$, $P(B) = \{B\}$. We will use the short form $\forall expression$ for $\forall x \in e_1 \cdot expression$ and similarly for the existential quantifier.

The partition of $P(\exists A \vee B)$, before the final simplification but after consistency checking, is equal to:

$$\left\{\begin{array}{l}
(\forall A \vee B) \wedge (\exists A \wedge \neg B) \wedge (\forall \neg A \vee \neg B) \wedge (\forall A \vee \neg B) \\
(\forall A \vee B) \wedge (\exists A \wedge \neg B) \wedge (\exists A \wedge B) \wedge (\forall A \vee \neg B) \\
(\forall A \vee B) \wedge (\forall \neg A \vee B) \wedge (\exists A \wedge B) \wedge (\forall A \vee \neg B) \\
(\forall A \vee B) \wedge (\exists A \wedge \neg B) \wedge (\forall \neg A \vee \neg B) \wedge (\exists \neg A \wedge B) \\
(\forall A \vee B) \wedge (\exists A \wedge \neg B) \wedge (\exists A \wedge B) \wedge (\exists \neg A \wedge B) \\
(\forall A \vee B) \wedge (\forall \neg A \vee B) \wedge (\exists A \wedge B) \wedge (\exists \neg A \wedge B) \\
(\forall A \vee B) \wedge (\forall \neg A \vee B) \wedge (\forall \neg A \vee \neg B) \wedge (\exists \neg A \wedge B) \\
(\exists \neg A \wedge \neg B) \wedge (\exists A \wedge \neg B) \wedge (\forall \neg A \vee \neg B) \wedge (\forall A \vee \neg B) \\
(\exists \neg A \wedge \neg B) \wedge (\exists A \wedge \neg B) \wedge (\exists A \wedge B) \wedge (\forall A \vee \neg B) \\
(\exists \neg A \wedge \neg B) \wedge (\forall \neg A \vee B) \wedge (\exists A \wedge B) \wedge (\forall A \vee \neg B) \\
(\exists \neg A \wedge \neg B) \wedge (\exists A \wedge \neg B) \wedge (\forall \neg A \vee \neg B) \wedge (\exists \neg A \wedge B) \\
(\exists \neg A \wedge \neg B) \wedge (\exists A \wedge \neg B) \wedge (\exists A \wedge B) \wedge (\exists \neg A \wedge B) \\
(\exists \neg A \wedge \neg B) \wedge (\forall \neg A \vee B) \wedge (\exists A \wedge B) \wedge (\exists \neg A \wedge B) \\
(\exists \neg A \wedge \neg B) \wedge (\forall \neg A \vee B) \wedge (\forall \neg A \vee \neg B) \wedge (\exists \neg A \wedge B)
\end{array}\right\}$$

We list in Table 4.4 the resulting classes after a final simplification.

Ref.	Domain Division
1	$\forall A \wedge \neg B$
2	$(\forall A) \wedge (\exists B) \wedge (\exists \neg B)$
3	$\forall A \wedge B$
4	$(\forall A \vee B) \wedge (\forall \neg A \vee \neg B) \wedge (\exists A) \wedge (\exists B)$
5	$(\forall A \vee B) \wedge (\exists A \wedge \neg B) \wedge (\exists A \wedge B) \wedge (\exists \neg A \wedge B)$
6	$(\forall B) \wedge (\exists A) \wedge (\exists \neg A)$
7	$\forall \neg A \wedge B$
8	$(\forall \neg B) \wedge (\exists A) \wedge (\exists \neg A)$
9	$(\forall A \vee \neg B) \wedge (\exists A \wedge \neg B) \wedge (\exists \neg A) \wedge (\exists B)$
10	$(\forall \neg A \vee B) \wedge (\exists A) \wedge (\forall A \vee \neg B) \wedge (\exists \neg A)$
11	$(\exists A) \wedge (\exists B) \wedge (\forall \neg A \vee \neg B) \wedge (\exists \neg A \wedge \neg B)$
12	$(\exists A \wedge \neg B) \wedge (\exists A \wedge B) \wedge (\exists \neg A \wedge B) \wedge (\exists \neg A \wedge \neg B)$
13	$(\forall \neg A \vee B) \wedge (\exists A) \wedge (\exists \neg A \wedge B) \wedge (\exists \neg B)$
14	$(\forall \neg A) \wedge (\exists B) \wedge (\exists \neg B)$

Table 4.4: Domain Division of $\exists binds \cdot A \vee B$

To illustrate the results we can use Venn diagrams by considering two sets A and B. A is the set of variable tuples for which the expression A evaluates to true and similarly for the set B. This is shown in Figure 4.2.

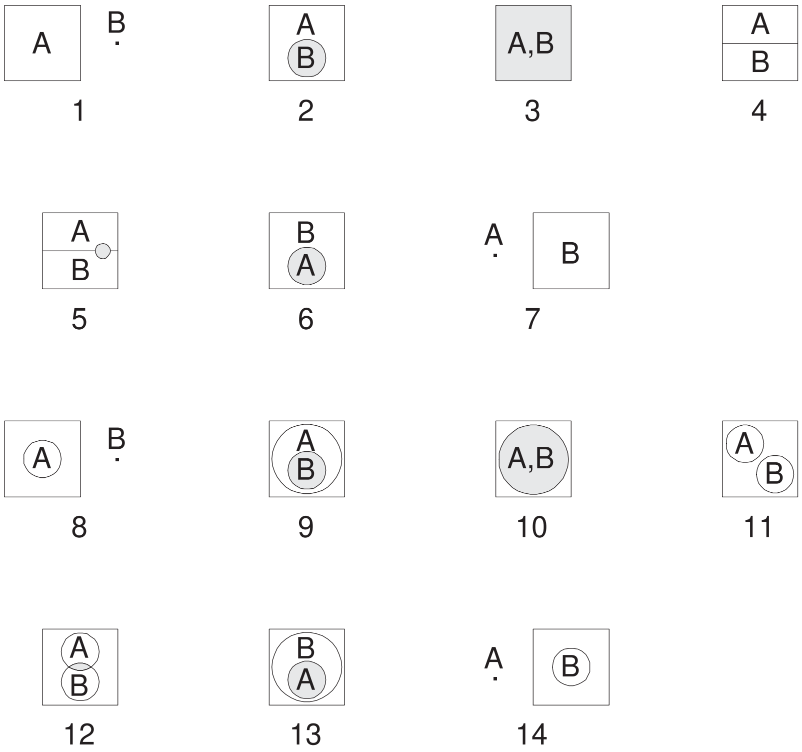

Figure 4.2: Venn Diagrams for the Partition of $\exists binds \cdot A \vee B$

The partitioning has been total, i.e. no further partitioning is feasible at this level.

To apply the quantified expressions rules, as presented, always leads to very long partition derivation even for basic quantified expressions. Unique quantified expressions are of the form $'\exists !'$, `bindlist`, $' \cdot '$, `expression` and their study offers an insight into the semantics of VDM-SL, particularly in relation to looseness. In classic mathematics equivalences of the form [113]:

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

always hold. Therefore, a partitioning rule for the unique quantifier could be derived from the existential and universal quantifiers rules.

However, loose expressions, as already seen when considering pattern matching, can lead to very complex expressions with several allowed behaviours. In these circumstances, it is not straightforward to establish the veracity of the above equivalence. The VDM-SL standard [112] provides an example of complex unique quantified expression:

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

This expression can be shown, as done in the standard [112], 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.

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. We cannot rely on the above equivalence in these circumstances until a proof has been established.

While it can be argued that such complex examples will never find their way into real specifications, we cannot envisage how to identify them automatically nor is it easy to define a 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 such degree of complexity in specifications hinders the development of formal methods support tools or deters some from using formal methods.

This discussion reinforces our decision to exclude patterns which may introduce looseness (i.e. set enumeration, set union and sequence patterns). As is clear from the above example, let be expressions of the form $let\ Pt \in S\ be\ st\ B\ in\ E$ or $let\ Pt \in S\ in\ E$ must also be excluded. Later in this thesis, we will return to loose expressions and attempt to accommodate them.

Under these restrictions the equivalence given for unique existential expressions holds and can be used to generate their partition.

4.3.2 Set Comprehension Expressions

Set comprehension expressions have the following form:

`'{', expression, '|', bind list, ['.', expression], '}'`

Our approach to partitioning into sub-domains set comprehension expressions can be used as a model for sequence and map comprehension expressions partitioning, since their semantics and syntax are similar.

Consider $SetComp = \{f(x)|x \in e_1 \cdot Q(x)\}$. Whenever this set is not empty this expression is equivalent to:

$$SetComp = S \wedge \forall x \in e_1 \cdot Q(x) \Rightarrow f(x) \in S$$

We therefore propose that $P(SetComp = \{f(x)|x \in e_1 \cdot Q(x)\})$ be equal to:

$$\bigcup \begin{Bmatrix} P(e_1 = \{\}) \times \{SetComp = \{\}\} \\ P(e_1 \neq \{\}) \times P(\forall x \in e_1 \cdot \neg Q(x)) \times \{SetComp = \{\}\} \\ P(e_1 \neq \{\}) \times \{\exists x \in e_1 \cdot Q(x)\} \times \\ \quad P(\forall x \in e_1 \cdot Q(x) \Rightarrow f(x) \in S) \times \{SetComp = S\} \end{Bmatrix}$$

For example consider the map comprehension $Over = \{a \mapsto m(a)|a \in dom(m)\backslash S\}$ where m is map and S a set. Note that here $Q(x)$ is absent and can be considered to be *true*.

We must first agree on a partition for the new operators *dom*, $\backslash$ and $\mapsto$.

$$P(p = e_1 \mapsto e_2) = P(tmp_1 = e_1) \times P(tmp_2 = e_2) \times \{p = tmp_1 \mapsto tmp_2\}$$

$$P(p = dom(e_1)) = \begin{Bmatrix} tmp_1 = \{\mapsto\} \wedge p = \{\} \\ tmp_1 \neq \{\mapsto\} \wedge p = dom(tmp_1) \end{Bmatrix} \times P(tmp_1 = e_1)$$

$$P(p = e_1\backslash e_2) = \begin{Bmatrix} tmp_1 = \{\} \wedge tmp_2 = \{\} \wedge p = \{\} \\ tmp_1 \neq \{\} \wedge tmp_2 = \{\} \wedge p = tmp_1 \\ tmp_1 = \{\} \wedge tmp_2 \neq \{\} \wedge p = \{\} \\ tmp_1 \neq \{\} \wedge tmp_2 \neq \{\} \wedge tmp_1 \cap tmp_2 = \{\} \wedge p = tmp_1 \\ tmp_1 \neq \{\} \wedge tmp_2 \neq \{\} \wedge tmp_1 \subset tmp_2 \wedge p = \{\} \\ tmp_1 \neq \{\} \wedge tmp_2 \neq \{\} \wedge tmp_2 \subset tmp_1 \wedge p = tmp_1\backslash tmp_2 \\ tmp_1 \neq \{\} \wedge tmp_2 \neq \{\} \wedge tmp_1 = tmp_2 \wedge p = \{\} \\ tmp_1 \neq \{\} \wedge tmp_2 \neq \{\} \wedge tmp_1 \cap tmp_2 \neq \{\} \wedge \\ \quad \neg(tmp_1 \subset tmp_2) \wedge \neg(tmp_2 \subset tmp_1) \wedge tmp_1 \neq tmp_2 \wedge \\ \quad p = tmp_1\backslash tmp_2 \end{Bmatrix}$$
$$\times P(tmp_1 = e_1) \times P(tmp_2 = e_2)$$

Note that the sub-domains of the set difference $\backslash$ are the same as the sub-domains of $\cup$.

We can now proceed with the partitioning of $Over$:

$P(Over = \{a \mapsto m(a) | a \in dom(m)\backslash S\}) =$

$$\bigcup \left\{ \begin{array}{l} P(dom(m)\backslash S = \{\}) \times \{Over = \{\mapsto\}\} \\ P(dom(m)\backslash S \neq \{\}) \times P(\forall a \in dom(m)\backslash S \cdot \neg true)\times \\ \quad \{Over = \{\mapsto\}\} \\ P(dom(m)\backslash S \neq \{\}) \times \{\exists a \in dom(m)\backslash S \cdot true\}\times \\ \quad P(\forall a \in dom(m)\backslash S \cdot true \Rightarrow a \mapsto m(a) \in T) \times \{Over = T\} \end{array} \right\}$$

The first partition expression concerns the case when the map comprehension is empty because $dom(m)\backslash S$ is empty. Therefore,

$P(dom(m)\backslash S = \{\}) \times \{Over = \{\mapsto\}\}$ is equal to:

$$\left\{ \begin{array}{c} Over = \{\mapsto\} \wedge m = \{\mapsto\} \wedge S = \{\} \\ Over = \{\mapsto\} \wedge m = \{\mapsto\} \wedge S \neq \{\} \\ Over = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge S \neq \{\} \wedge dom(m) \subset S \\ Over = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge S \neq \{\} \wedge dom(m) = S \end{array} \right\}$$

Further, because $\forall a \in dom(m)\backslash S \cdot \neg true$ is equivalent to $\perp$ (i.e. is inconsistent) whenever $dom(m)\backslash S \neq \{\}$, $P(\forall a \in dom(m)\backslash S \cdot \neg true) = \emptyset$, the second partition expression evaluates to $\emptyset$

We can now concentrate on the last partition expression. $\{\exists a \in dom(m)\backslash S \cdot true\}$ is equivalent in this context to $\{true\}$. We therefore need to consider:

$$P(dom(m)\backslash S \neq \{\}) \times P(\forall a \in dom(m)\backslash S \cdot a \mapsto m(a) \in T) \times \{Over = T\}$$

Because $P(a \mapsto m(a) \in T) = \{a \mapsto m(a) \in T\}$, $P(\forall a \in dom(m)\backslash S \cdot a \mapsto m(a) \in T)$ amounts to $\{\forall x \in dom(m)\backslash S \cdot a \mapsto m(a) \in T\}$ in this context. We are left with:

$$P(dom(m)\backslash S \neq \{\}) \times \{(\forall x \in dom(m)\backslash S \cdot a \mapsto m(a) \in T) \wedge Over = T\}$$

which when evaluated and simplified is equal to:

$$\left\{\begin{array}{l}Over = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge S = \{\} \\ Over = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge S \neq \{\} \wedge dom(m) \cap S = \{\} \\ Over = \{a \mapsto m(a) | a \in dom(m) \backslash S\} \wedge m \neq \{\mapsto\} \wedge S \neq \{\} \wedge S \subset dom(m) \\ Over = \{a \mapsto m(a) | a \in dom(m) \backslash S\} \wedge m \neq \{\mapsto\} \wedge S \neq \{\} \wedge \\ \quad \neg(dom(m) \subset S) \wedge \neg(S \subset dom(m)) \wedge dom(m) \neq S\end{array}\right\}$$

Similar generation can be performed with set and sequence comprehension expressions. We remark that most of the simplifications are straightforward but that derivations can be large.

We now illustrate how the coarse partitioning rules, corresponding to logical and control VDM-SL constructs, can be combined with the finer partitioning rules of VDM-SL scoping expressions and simple expressions built from operators.

4.4 A Direct Synthesis

In the previous two sections we have presented, discussed and extended two major results, the work of Dick and Faivre [13, 14] on coarse partitioning, and the work of Stock and Carrington [15, 16] on finer partitioning. Our main contributions consist of:

- an informal justification of coarse partitioning
- extended coarse partitioning rules
- showing the limitation of coarse partitioning
- showing how Dick and Faivre's prototype could be improved
- an informal justification of basic finer partitioning rules (e.g. for expressions based on the set union operator)
- finer partitioning rules for scoping expressions
- a formalism suitable for both activities, highlighting the mechanical nature of partition generation

The formalism developed allows us to directly synthesize coarse and finer partitioning. Below we examine the potential quality of the test classes thus generated.

4.4.1 Systematic Formal Partitioning

Having given formal partitioning rules for some basic VDM-SL constructs we will now apply them on a short example combining logical and mapping operators to illustrate the continuity of the partitioning process from coarse to finer partitioning.

Consider the expression, $dom(n) \lhd m = S \wedge Over = S \uplus n$ whose coarse partition is $P(dom(n) \lhd m = S) \times P(Over = S \uplus n)$. We decompose $dom(n) \lhd m = S$ into $dom(n) = D \wedge D \lhd m = S$, and using the definition of the domain restriction operator $\lhd$, $D \lhd m = \{a \mapsto m(a) | a \in dom(m) \backslash D\}$, for which we have already generated the partition in a previous example, we find that $P(dom(n) \lhd m = S)$ is equal to:

$$\left\{\begin{array}{l}
S = \{\mapsto\} \wedge m = \{\mapsto\} \wedge D = \{\} \\
S = \{\mapsto\} \wedge m = \{\mapsto\} \wedge D \neq \{\} \\
S = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge D \neq \{\} \wedge dom(m) \subset D \\
S = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge D \neq \{\} \wedge dom(m) = D \\
S = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge D = \{\} \\
S = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge D \neq \{\} \wedge dom(m) \cap D = \{\} \\
S = \{a \mapsto m(a) | a \in dom(m) \backslash D\} \wedge m \neq \{\mapsto\} \wedge D \neq \{\} \wedge D \subset dom(m) \\
S = \{a \mapsto m(a) | a \in dom(m) \backslash D\} \wedge m \neq \{\mapsto\} \wedge D \neq \{\} \wedge \\
\quad \neg(dom(m) \subset D) \wedge \neg(D \subset dom(m)) \wedge dom(m) \neq D
\end{array}\right\}$$

$$\times \left\{\begin{array}{l}
n = \{\mapsto\} \wedge D = \{\} \\
n \neq \{\mapsto\} \wedge D = dom(n)
\end{array}\right\}$$

which after combination, and consistency checking is equivalent to:

$$\left\{\begin{array}{l}
S = \{\mapsto\} \wedge m = \{\mapsto\} \wedge n = \{\mapsto\} \\
S = \{\mapsto\} \wedge m = \{\mapsto\} \wedge n \neq \{\mapsto\} \\
S = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge dom(m) \subset dom(n) \wedge n \neq \{\mapsto\} \\
S = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge dom(m) = dom(n) \wedge n \neq \{\mapsto\} \\
S = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge n = \{\mapsto\} \\
S = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge dom(m) \cap dom(n) = \{\} \wedge \\
\quad n \neq \{\mapsto\} \\
S = \{a \mapsto m(a) | a \in dom(m) \backslash dom(n)\} \wedge m \neq \{\mapsto\} \wedge dom(n) \subset dom(m) \wedge \\
\quad n \neq \{\mapsto\} \\
S = \{a \mapsto m(a) | a \in dom(m) \backslash dom(n)\} \wedge m \neq \{\mapsto\} \wedge \neg(dom(m) \subset dom(n)) \\
\quad \wedge \neg(dom(n) \subset dom(m)) \wedge dom(m) \neq dom(n) \wedge n \neq \{\mapsto\}
\end{array}\right\}$$

The partitioning of a map merge expression, such as $Over = S \uplus n$, is equivalent to: $P(dom(Over) = P(dom(S) \cup dom(n))) \times \{Over = S \uplus n\}$ so, we can re-use the set union partitioning rule given in table 4.3. Further, to simplify the derivation we note that in our example $dom(S) \cap dom(n) = \{\}$.

So, $P(Over = S \uplus n)$ is equal to:

$$\left\{\begin{array}{c}
dom(S) = \{\} \wedge dom(n) = \{\} \wedge Over = \{\mapsto\} \\
dom(S) = \{\} \wedge dom(n) \neq \{\} \wedge Over = n \\
dom(S) \neq \{\} \wedge dom(n) = \{\} \wedge Over = S \\
dom(S) \neq \{\} \wedge dom(n) \neq \{\} \wedge dom(S) \cap dom(n) = \{\} \wedge Over = S \uplus n
\end{array}\right\}$$

Therefore, $P(dom(n) \lhd m = S \wedge Over = S \uplus n)$ is equal to:

$$\left\{\begin{array}{l}
S = \{\mapsto\} \wedge m = \{\mapsto\} \wedge n = \{\mapsto\} \wedge Over = \{\mapsto\} \\
S = \{\mapsto\} \wedge m = \{\mapsto\} \wedge n \neq \{\mapsto\} \wedge Over = n \\
S = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge dom(m) \subset dom(n) \wedge n \neq \{\mapsto\} \wedge Over = n \\
S = \{\mapsto\} \wedge m \neq \{\mapsto\} \wedge dom(m) = dom(n) \wedge n \neq \{\mapsto\} \wedge Over = n \\
S = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge n = \{\mapsto\} \wedge Over = S \\
S = \{a \mapsto m(a) | a \in dom(m)\} \wedge m \neq \{\mapsto\} \wedge dom(m) \cap dom(n) = \{\} \wedge \\
\quad n \neq \{\mapsto\} \wedge Over = S \uplus n \\
S = \{a \mapsto m(a) | a \in dom(m) \backslash dom(n)\} \wedge m \neq \{\mapsto\} \wedge dom(n) \subset dom(m) \wedge \\
\quad n \neq \{\mapsto\} \wedge Over = S \uplus n \\
S = \{a \mapsto m(a) | a \in dom(m) \backslash dom(n)\} \wedge m \neq \{\mapsto\} \wedge \neg(dom(m) \subset dom(n)) \\
\ \wedge \neg(dom(n) \subset dom(m)) \wedge dom(m) \neq dom(n) \wedge n \neq \{\mapsto\} \wedge Over = S \uplus n
\end{array}\right\}$$

So we see that coarse partitioning and finer partitioning can indeed be combined to generate finer partitions.

The expression, $dom(n) \lhd m = S \wedge Over = S \uplus n$ is actually the definition of the map override operator, $m \dagger n = Over$, so that the partition generated above can be used directly as a justified partitioning rule for such expressions.

This example shows how from basic partitioning rules, more complex expressions can be given, systematically, a sound partitioning rule. This of course implies, that direct partitioning rules can be used for all VDM-SL constructs and that the decomposition process, as performed above, does not have to be repeated.

In [15], Stock and Carrington informally derive a partition for the similar map override operator in Z. The same decomposition process is used but informal basic partitioning rules are applied.

In their partition the equivalence class when $dom(m) \subset dom(n)$ does not appear explicitly. This difference arises because an informal partition for expressions based on the domain restriction operator $\lhd$ is used, instead of decomposing such an expression using a map comprehension expression for which we have developed a partitioning rule. As a consequence, it is very unlikely that the desirable test when $dom(m) \subset dom(n)$ will be generated from their partition. Indeed, if we believe the partition homogeneous, a single random sample should be taken from each equivalence class. Our partition covers more of the semantics of expressions based on the map override operator and the eventual test set would have an increased adequacy.

4.4.2 A Combinatorial Explosion

While in [15, 16] the process of informally partitioning basic expressions is presented, and coarse partitioning criticised, no attempt to reconcile coarse and finer partitioning is presented. As we have seen above, a direct synthesis using our formalism is appropriate for some expressions. For example, even if the partition for the † operator may seem large it can be justified informally in terms of likelihood of finding an error in the system under test (using Venn diagrams if necessary).

Our direct synthesis allows the systematic generation of compact partitions: we could be satisfied with this result and concentrate on implementation matters which are crucial if the technique is to be used. However we note that, for some expressions the number of classes generated explodes and the justification of each individual test becomes, to say the least, dubious.

For example the following expression: $x \geq 0 \wedge x \leq 10 \vee y \geq 0 \wedge y \leq 20$ (which could be part of the input validation of a specification for example) is partitioned (with LPF behaviour disabled) as follows:

$$\bigcup \left\{ \begin{array}{c} P(x \geq 0) \times P(x \leq 10) \times \bigcup \left\{ \begin{array}{c} P(y < 0) \\ P(y > 20) \end{array} \right\} \\ P(x \geq 0) \times P(x \leq 10) \times P(y \geq 0) \times P(y \leq 20) \\ \bigcup \left\{ \begin{array}{c} P(x < 0) \\ P(x > 10) \end{array} \right\} \times P(y \geq 0) \times P(y \leq 20) \end{array} \right\}$$

We generate sub-domains for the non-path expressions indiscriminately, to obtain the following set of classes (a total of 33 equivalence classes):

$$\left\{\begin{array}{c} x=0 \wedge y=-1, x>0 \wedge x<10 \wedge y=-1, x=10 \wedge y=-1 \\ x=0 \wedge y<-1, x>0 \wedge x<10 \wedge y<-1, x=10 \wedge y<-1 \\ x=0 \wedge y=21, x>0 \wedge x<10 \wedge y=21, x=10 \wedge y=21 \\ x=0 \wedge y>21, x>0 \wedge x<10 \wedge y>21, x=10 \wedge y>21 \\ x=0 \wedge y=0, x>1 \wedge x<10 \wedge y=0, x=10 \wedge y=0 \\ x=0 \wedge y>0 \wedge y<10, x>1 \wedge x<10 \wedge y>0 \wedge y<10 \\ x=10 \wedge y>0 \wedge y<10 \\ x=0 \wedge y=10, x>1 \wedge x<10 \wedge y=10, x=10 \wedge y=10 \\ x=-1 \wedge y=0, x=-1 \wedge y>0 \wedge y<20, x=-1 \wedge y=20 \\ x<-1 \wedge y=0, x<-1 \wedge y>0 \wedge y<20, x<-1 \wedge y=20 \\ x=11 \wedge y=0, x=11 \wedge y>0 \wedge y<20, x=11 \wedge y=20 \\ x>11 \wedge y=0, x>11 \wedge y>0 \wedge y<20, x>11 \wedge y=20 \end{array}\right\}$$

Clearly, the exhaustive partitioning would generate too many tests here when compared to normal testing practices.

To add to this example, we illustrate this explosion in the number of classes generated by considering the expression: $x \neq 6 \vee y \neq 0$ where x and y are integers (i.e. LPF plays no role in the semantics of this expression). The systematic partitioning is this expression leads to the following classes:

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

The partition generated from $P(x \neq 6) \times P(y \neq 0)$ seems to contain redundant equivalence classes, i.e. classes from which tests which are no better than random tests according to ad hoc testing practices would be generated. This is caused by the full combination operator which systematically combines all equivalence classes of a its operands. Using ad hoc testing principles would probably lead to the following partition:

$$P(x \neq 6 \vee y \neq 0) = \begin{Bmatrix} x < 5 \wedge y = 0 \\ x = 5 \wedge y = -1 \\ x = 7 \wedge y = 1 \\ x > 7 \wedge y > 1 \\ x = 6 \wedge y < -1 \end{Bmatrix}$$

(there are of course many such partitions as we will see in the next chapter)

This latest partition seems to encompass the previous partition. Of course, the $\neq$ partitioning convention could be relaxed if, in a given environment, it is deemed that its domain coverage does not have to be so complete. This would lead to a smaller test set. Here however, we do not have any knowledge about the particular system being tested so our partitioning conventions are difficult to simplify.

We therefore, need to modify not our conventions but the partitioning process itself in some circumstances to be able to generate adequate test sets according to state of the art testing practices.

If, in this chapter, we have pursued the development of a complete partitioning technique, we now acknowledge that sometimes the test sets generated are not adequate. The next chapter is devoted to identifying circumstances where our technique, as it is, is not suitable, and developing a justified and appropriate approach.

Chapter 5

Sensible Tests Generation

We will show that the systematic partitioning of VDM-SL specifications, as presented in the previous chapter, can lead to very large test sets which are difficult to reconcile with our aim of finding the maximum number of errors in the system under test using a minimum number of tests. This will lead us to find heuristics for the identification of redundant equivalence classes.

We will argue that the adoption of our heuristics only results in a small loss of coverage of the behaviour of a specification and that it leads to a justifiable compromise between the degree of coverage and the size of the test sets generated.

We will extend the formalism introduced so far to illustrate our arguments.

5.1 Controlling Partitioning

As we remarked in the previous chapter, the systematic partitioning of some expressions seems to generate more equivalence classes than is necessary to sample from a testing point of view. When partitioning is extended to non-path expressions—as in our systematic partitioning technique which uniquely reconciles coarse and finer partitioning to achieve complete and uniform coverage of specifications—a combinatorial explosion of classes is observed.

Our partitioning technique of specifications, as presented so far, is complete and uniform: we have simply extended the previous results in this area and integrated them in a unique procedure; we have blindly taken partitioning to its logical conclusion. Faced with the unjustifiable explosion of classes generated, we must now attempt to mitigate our results by re-examining their impact on the probability of revealing an error in the system under test in the hope of finding ways of reducing the number of tests generated without impairing the quality of the test sets produced.

We are reluctant to dismiss out of hand the partitioning theory as applied to software testing—not least because of the absence of alternatives—but realise that ways to identify redundant classes must be found to restore confidence in the practicability and worthiness of the approach. For, if very large test sets are systematically generated for the simplest of expressions and the mapping between individual tests and ad hoc justifications is broken, the value of our technique will, justifiably, be judged worthless and impracticable.

The explosion in the number of classes generated is exacerbated by our partitioning rules for non-path expressions. However, even if the partitioning of non-path expressions is disallowed (as is the case in the Dick and Faivre approach [13, 14]) a geometric growth in the number of classes generated can be observed for some specifications [98]. For example, the expression

$(x > 0 \vee x < 5) \wedge (y > 0 \vee y < 5)$

without partitioning of non-path expressions gives rise to the partition:

$$\left\{\begin{array}{l}
x > 0 \wedge x \geq 5 \wedge y > 0 \wedge y \geq 5 \\
x > 0 \wedge x \geq 5 \wedge y > 0 \wedge y < 5 \\
x > 0 \wedge x \geq 5 \wedge y \leq 0 \wedge y < 5 \\
x > 0 \wedge x < 5 \wedge y > 0 \wedge y \geq 5 \\
x > 0 \wedge x < 5 \wedge y > 0 \wedge y < 5 \\
x > 0 \wedge x < 5 \wedge y \leq 0 \wedge y < 5 \\
x \leq 0 \wedge x < 5 \wedge y > 0 \wedge y \geq 5 \\
x \leq 0 \wedge x < 5 \wedge y > 0 \wedge y < 5 \\
x \leq 0 \wedge x < 5 \wedge y \leq 0 \wedge y < 5
\end{array}\right\}$$

In practice, and according to Beizer [134], the sub-domains attached to x and y in this case need not be systematically combined because the boundaries are orthogonal. Orthogonality and domain testing are discussed in [134] 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.

Nevertheless, if combinations are limited the following set of classes can be obtained:

$$\left\{\begin{matrix} x > 0 \wedge x \geq 5 \wedge y > 0 \wedge y \geq 5 \\ x > 0 \wedge x < 5 \wedge y > 0 \wedge y < 5 \\ x \leq 0 \wedge x < 5 \wedge y \leq 0 \wedge y < 5 \end{matrix}\right\}$$

Sampling these would cover the original sub-domains of x and y in a minimal number of tests. We should note however that the set of classes above does not amount to a partition of the original expression $((x > 0 \vee x < 5) \wedge (y > 0 \vee y < 5))$ because their disjunction is not equivalent to it. Also the predicate above are classes selected from the original final partition.

The theoretical work of Weyuker [34] and Zhu [32] 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.

We now set to find such criteria.

5.1.1 Context Dependent Combination

We re-examine, from the previous chapter, the expression $x \neq 6 \vee y \neq 0$ and its corresponding partition:

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

arising from the $\neq$ operator partitioning rule.

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 analysed, 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 [134] 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.

Note that the set of classes above does not amount to a partition, in the mathematical sense of the definition **2.1**, of the original expression. We will still however call the elements of such sets as classes, in reference to their origin.

We can also reduce the number of classes for specifications with embedded function calls. If we suppose that a function f in the specification has been implemented independently as a subprogram then we can make two remarks:

- The behaviour of f as implemented needs to be covered only once: thus in case of several calls to f many of the classes generated can be eliminated.
- The dependence of sub-domains is local to f: e.g. in the function call $f(x, y)$, even if x and y are dependent in the calling context, this dependence is not implied when generating classes for f. Vice versa, if in f, x and y are dependent, this is not implied in the calling environment.

In general, a function in the specification cannot always be considered independent from the rest of the specification. For example in the Triangle Problem, *sum* is unlikely to be present at all in an implementation. Further, even if a function which fulfils the role of *sum* has been implemented, knowledge of the specification would surely have been taken into account to implement it as $x + y + z$, thus it cannot be said that *sum* in the implementation models *sum* in the specification: *sum* is not independent from the rest of the specification. Using knowledge of the implementation, or our judgement, as to which functions have been implemented independently can lead to large reductions in the number of classes. For example if the partitioning of f leads to n classes, then the expression $f(x) + f(y)$ can be covered in a minimal number of n classes.

This treatment of function calls has some important consequences for quantified expressions. We recall from the previous chapter that an expression of the form $\forall x \in S \cdot f(x)$ where $f(x) = x > 5$ with the convention that $P(x > 5) = \begin{Bmatrix} x = 6 \\ x > 6 \end{Bmatrix}$ would lead to a set of classes which can be denoted as:

$$\left\{\begin{array}{c} S = \{\} \\ S = \{6\} \\ S = \{x_1, \ldots, x_n\} \\ S = \{6, x'_1, \ldots, x'_m\} \end{array}\right\}$$

where the x_is and x'_is are greater than 6.

If f is independent from the rest of the implementation, then the sub-domains of $P(\forall x \in S \cdot f(x))$ can be entirely covered by the classes: $S = \{\}$ and $S = \{6, x'_1, \ldots, x'_m\}$.

We will illustrate further the consequences of context dependent combinations later but first, we justify this intuitive notion and identify the circumstances under which it can be applied.

5.1.2 Rationale For Context Dependent Combination

We first give our definition of variable dependence.

Definition 5.1 *Dependence*

Two variables are dependent within a specification if there exists a non-path expression where the two variables appear. This is mitigated for independently implemented functions as discussed in the previous section. The dependence relation is commutative and associative.

By extension we will also talk of the dependence of sub-domains. For clarity we also give the following definitions:

Definition 5.2 *Final Partition*

The final partition is the partition obtained at the end of our systematic partitioning process as described in chapter 4

Definition 5.3 *Cover*

A set of dependent sub-domains present in the make-up of one or several equivalence classes from the final partition is covered by a class, if that class is part of the final partition and the set of dependent sub-domain is present in the make-up of that class.

Definition 5.4 *Minimal Set of Classes*

A set of classes is minimal if:

- *it is composed of classes from the final partition*
- *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.

Definition 5.5 *Redundant Class*

A redundant class is an element of the final partition that is not an element of the minimal set of classes chosen.

We shall formalise some of these definitions later. But we first justify our use of minimal set of classes.

Given the conjunction of two independent predicates (representing two independent domains) which has been partitioned into sub-domains regrouped into equivalence classes forming the final partition, we will argue that a selection of classes aimed at covering the dependent sub-domains in a minimal set of classes has, once sampled, the same probability of revealing an error in the underlying implementation as the original set of tests generated from the final partition. We will use basic probability laws to demonstrate this.

We feel that this explanation is necessary since it will reveal an underlying assumption (with possible consequences for the way testing is performed) and that probabilities must surely underpin the rationale of any testing strategy (although the absolute value of the probability of finding an error in a particular application or domain is usually unknown). We could have been content however to rely on the notion of orthogonality of sub-domains.

Let A be a domain representing various dependent variables. Let B also be a domain representing various dependent variables. We suppose that A and B are independent. Let $\{A_1..A_n\}$ with $1 \leq n$ and $\{B_1..B_m\}$ with $1 \leq m$ be the partitions generated using our technique of the domains A and B respectively. To probabilise this space we choose our probabilistic event to denote the presence of an error in a given domain. For example, $Pr(x < 3)$ denotes the probability that the implementation does not respect the specification when the input variable x is less than 3.

According to the theory of partitioning, our sub-domains are homogenous hence the probability that a sub-domain contains an input which does not respect the specification can be tested using a unique sample of the domain. In other words, only one sample per sub-domain needs to be taken and subsequently tested to detect an error belonging to the sub-domain. Hence, the probability that a sub-domain contains an input which will cause an erroneous behaviour in the implementation with respect to the specification, is equal to the probability that a single, random, test sample taken from the sub-domain will reveal an erroneous behaviour. This is indeed the rationale of partitioning testing.

A basic law of probability theory is: $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$. We recall that the domain divisions of A constitute a partition of A hence $A = \bigcup_{i=1}^{n} A_i$ and similarly: $B = \bigcup_{i=1}^{m} B_i$. Thus, $Pr(A \cup B) = Pr(\bigcup_{i=1}^{n} A_i \cup \bigcup_{j=1}^{m} B_j)$. Using the basic probability law above, we find:

$$\begin{aligned} Pr(A \cup B) =& \textstyle\sum_{i=1}^{n} Pr(A_i) + \sum_{j=1}^{m} Pr(B_j) + \sum_{i=1}^{n} \sum_{j=1}^{m} Pr(Ai \cap B_j) + \\ & \textstyle\sum_{i=1}^{n-1} \sum_{k=i+1}^{n} Pr(A_i \cap A_k) + \sum_{j=1}^{m-1} \sum_{k=j+1}^{m} Pr(B_j \cap B_k) \end{aligned}$$

The intersections of the kind $A_1 \cap A_2$ are always empty as the family of A_i is a partition of A and hence the sub-domains are two by two disjoint. Further, the intersections of the forms $A_1 \cap B_2$ are also empty because A and B are disjoint since independent ($Pr(A_1 \cap B_2)$ denotes the probability that there is an error in the implementation which belongs to A_1 and B_2). As the probability of an event occuring within an empty set is null and without knowing the probability that an error is present in a particular domain we can write that when the domains A and B are independent: $Pr(A \cup B) = \sum_{i=1}^{n} Pr(A_i) + \sum_{j=1}^{m} Pr(B_j)$.

Thus, any covering of $A \times B$ (cartesian product) will yield the same probability of finding an error. For example with $n = m = 2$,

$$Pr(A \cup B) = Pr(A_1 \cup B_1) + Pr(A_2 \cup B_2)$$

which implies that we only need to select the sub-domains A_1 with B_1 and A_2 with B_2 for example, take a single sample of each combination and perform the two tests to obtain the maximum likelihood of revealing an error in the underlying implementation. Performing another test from, say, $A_1 \cup B_2$ will not increase our chances of finding an error.

Hence, we have demonstrated that assuming the domains are independent, a one to one combination of our partitions is just as likely to find an error in the underlying implementation as an exhaustive combination.

In contrast, if two domains are dependent, say C and D then we need to sample the intersection of each sub-domain because then the $Pr(C_i \cap D_j)$ are not the probability of an empty domain of events but is given by the Bayesain's law : $Pr(C_i \cap D_j) = Pr(C_i|D_j) * Pr(D_j)$ where $Pr(C_i|D_j)$ denotes the probability $Pr(C_i)$ if $Pr(D_j) = 1$ (i.e. if there is an error in the implementation whose input domain lies within D_j). This in fact demonstrates the need for full combination of dependent sub-domains.

We have overlooked, however, one implicit assumption made when justifying our technique: that the independence of domains at the specification level must be preserved at the implementation level. It is possible for, say, two domains to be independent in the specification but not in the implementation (in which case the implementation is probably erroneous). In practice, the independence of domains is likely to be preserved. We make two remarks to justify this assumption.

From a logical point of view, if two domains are separate in the specification it is difficult to envisage how a correct implementation could transgress the independence of the domains. Further, it is usually clear from a specification point of view if some variables do not constrain or interfere with others, hence for an implementation written from the specification this dichotomy should be clear to the developer and no transgression would normally occur.

It is however possible that an implementation transgresses the independence of some variables. Consider a predicate in a specification where x and y are independent such as $x > 5 \wedge y > 6$, in an implementation an extraneous constraint could emerge as in $x > 5 \wedge y > 6 \wedge x + y < 1000$. The error is unlikely to be detected even if we considered the domains of x and y to be dependent in the specification as is the case in the implementation. It is therefore important to stress that in some cases our assumption will be invalid but that then the full combination alternative to our context dependent combination is just as likely to reveal an error in the underlying implementation.

To conclude on this matter, and as the example above shows, it could be of benefit, from a correctness point view, to investigate instances when the independence of variables in the specification is transgressed in the implementation. This is however outside the scope of this work but points to static data flow analysis of programs according to their specification (see [27, 135] for example). Further, for function calls, functions which are not implemented independently from the rest of the system (to be deemed as being implemented independently from the rest of the system, an implemented function must model its specification. e.g. in the case of sum, the implementation must be able to deal with any sequence of natural numbers) according to their specification, using knowledge of the implementation or our judgement, can only be considered as a shorthand notation from a test generation purpose. I.e. in those cases the partitioning of the function is still performed only once, but the partition obtained simply replaces the function call with no consequences for the minimum coverage of the specification. It can be envisaged that the specification and the implementation be made closer to enhance the testability of the system.

The net effect of context dependent combination is to allow us to reduce the size of some test sets without, according to our assumptions, reducing the quality of the test sets generated.

5.1.3 Notation

In the next section, we investigate various techniques to construct test sets. To do so, we denote a sub-domain belonging to a particular group of dependency by a lower alphabetical letter. So

$$\begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{Bmatrix} \times \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{Bmatrix}$$

could denote our earlier example: $P(x \neq 6 \wedge y \neq 0)$ with a_1 representing $x < 5$, a_2, $x = 6$ etc.

5.2 Constructing Test Sets: Initial Approaches

Implicitly so far, we have taken the approach of systematically generating the final partition and then identifying redundant classes to obtain a minimal set of classes.

In this section, we attempt to reduce the size of the set of classes generated by performing partitioning, consistency checking and combining sub-domains, according to our context dependent combination principle, while analysing VDM-SL expressions. To do so we first attempt to use the fact that minimum coverage of the sub-domains of a specification can be achieved if in $P_1 \times P_2$, where P_1 and P_2 are partition expressions, systematic combination of the sub-domains is only preformed for dependent classes.

But, given an extract of a specification such as:

$$\mathit{if}\ \mathit{exp}(x)\ \mathit{then}\ \mathit{exp}(z)\ \mathit{else}\ (\mathit{exp}(y) \wedge \mathit{exp}(x, y, z))$$

we cannot construct directly the set of classes because the inter-dependence of x, y and z is not apparent in the first operand of the $\wedge$ operator. Performing a one to one combination on the sub-domains revealed by the partitioning of the predicates $\mathit{exp}(x)$ and $\mathit{exp}(y)$ (because so far x and y are independent in the specification), violates our rule that dependent sub-domains must be exhaustively combined.

One issue which deserves to be investigated further is whether sub-domain dependence could be restricted to paths in the specification hence enhancing the potential for test set size reduction. For example in the expression above, of the three paths:

$$\left\{\begin{array}{c} P(exp(x) \wedge exp(z)) \\ P(\neg exp(x) \wedge exp(y) \wedge exp(x,y,z)) \\ P(exp(x) * \wedge exp(y) \wedge exp(x,y,z)) \end{array}\right\}$$

only in the last two does the dependence of the sub-domains of x, y and z appear, thus it could be envisaged that in the first path the combination be performed on a one to one basis. This would amount to defining sub-domain dependence with a path wide scope rather than specification wide. We have decided in this work to reject the temptation and define sub-domain dependence as a specification wide property (with some restrictions for called functions as already mentioned). We feel that domain dependence at the path level is unlikely to be preserved in the implementation not least because logical paths in the specification are unlikely to be preserved in the implementation. This is purely a pessimistic decision and could be reversed in cases where knowledge of the implementation is available. Below we examine several approaches to obtain a minimal set of classes respecting our criteria.

In the next two sections we assume that we have a means to determine the dependence of sub-domains as they are generated. Also, for simplicity we will not consider function calls.

5.2.1 Naive Approach

Assuming a partitioning skeleton of the form:

$$\bigcup\left\{\begin{array}{c} \left\{\begin{array}{c} a_1 \\ a_2 \end{array}\right\} \times \left\{\begin{array}{c} a_3 \\ a_4 \end{array}\right\} \\ \left\{\begin{array}{c} a_5 \\ a_6 \end{array}\right\} \times \left\{\begin{array}{c} a_7 \\ a_8 \end{array}\right\} \end{array}\right\} \times \left\{\begin{array}{c} a_9 \\ a_{10} \end{array}\right\} \times \left\{\begin{array}{c} b_1 \\ b_2 \end{array}\right\} \times \left\{\begin{array}{c} c_1 \\ c_2 \end{array}\right\}$$

with the following unsatisfiable expressions a_1a_3, a_1a_4, a_6a_7 and a_6a_{10}. Then we can proceed by doing local combinations while partitioning the expressions. For example after having generated

$$\left\{\begin{array}{c} a_1 \\ a_2 \end{array}\right\} \times \left\{\begin{array}{c} a_3 \\ a_4 \end{array}\right\}$$

we can combine the sub-domains to obtain:

$$\begin{Bmatrix} a_2a_3 \\ a_2a_4 \end{Bmatrix}$$

after local consistency checking. Similarly, after having generated

$$\begin{Bmatrix} a_5 \\ a_6 \end{Bmatrix} \times \begin{Bmatrix} a_7 \\ a_8 \end{Bmatrix}$$

from an expression of the form $exp_1(x) \wedge exp_2(x)$ where both non-path expressions reveal two sub-domains, we obtain

$$\begin{Bmatrix} a_5a_7 \\ a_5a_8 \\ a_6a_8 \end{Bmatrix}$$

Finally performing the distributed unions the partition now becomes:

$$\begin{Bmatrix} a_2a_3 \\ a_2a_4 \\ a_5a_7 \\ a_5a_8 \\ a_6a_8 \end{Bmatrix} \times \begin{Bmatrix} a_9 \\ a_{10} \end{Bmatrix} \times \begin{Bmatrix} b_1 \\ b_2 \end{Bmatrix} \times \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix}$$

performing the next combination yields:

$$\begin{Bmatrix} a_2a_3a_9 \\ a_2a_3a_{10} \\ a_2a_4a_9 \\ a_2a_4a_{10} \\ a_5a_7a_9 \\ a_5a_7a_{10} \\ a_5a_8a_9 \\ a_5a_8a_{10} \\ a_6a_8a_9 \end{Bmatrix} \times \begin{Bmatrix} b_1 \\ b_2 \end{Bmatrix} \times \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix}$$

Because the consistency checks are performed whenever combination occurs, many sub-classes have already been eliminated thus, minimising the subsequent amount of effort.

The next combination to be performed is between independent domains, so as long as the sub-domains are covered the combination is valid:

$$\left\{\begin{array}{c} a_2a_3a_9b_1 \\ a_2a_3a_{10}b_1 \\ a_2a_4a_9b_1 \\ a_2a_4a_{10}b_1 \\ a_5a_7a_9b_1 \\ a_5a_7a_{10}b_1 \\ a_5a_8a_9b_1 \\ a_5a_8a_{10}b_1 \\ a_6a_8a_9b_2 \end{array}\right\} \times \left\{\begin{array}{c} c_1 \\ c_2 \end{array}\right\}$$

and finally:

$$\left\{\begin{array}{c} a_2a_3a_9b_1c_1 \\ a_2a_3a_{10}b_1c_2 \\ a_2a_4a_9b_1c_1 \\ a_2a_4a_{10}b_1c_2 \\ a_5a_7a_9b_1c_1 \\ a_5a_7a_{10}b_1c_2 \\ a_5a_8a_9b_1c_1 \\ a_5a_8a_{10}b_1c_2 \\ a_6a_8a_9b_2c_1 \end{array}\right\}$$

is a minimal set of classes.

However, some expressions are not amendable to this approach because partial combinations cannot in general be performed before all the total combinations have been completed and checked for satisfiability.

For example, the naive construction of a test set for an expression of the form $exp_1(x) \wedge exp_2(y) \wedge exp_3(x)$ is as follows:

$$P(exp_1(x) \wedge exp_2(y) \wedge exp_3(x)) = P(exp_1(x) \wedge exp_2(y)) \times P(exp_3(x))$$

and $P(exp_1(x) \wedge exp_2(y)) = P(exp_1(x)) \times P(exp_2(y))$

where, say, $P(exp_1(x)) = \left\{\begin{array}{c} a_1 \\ a_2 \end{array}\right\}$ and $P(exp_2(y)) = \left\{\begin{array}{c} b_1 \\ b_2 \end{array}\right\}$

so that $P(exp_1(x) \wedge exp_2(y)) = \begin{Bmatrix} a_1b_1 \\ a_2b_2 \end{Bmatrix}$

for example (as the sub-domains are independent any combination is valid). We obtain:

$$P(exp_1(x) \wedge exp_2(y) \wedge exp_3(x)) = \begin{Bmatrix} a_1b_1 \\ a_2b_2 \end{Bmatrix} \times P(exp_3(x))$$

which if $P(exp_3(x)) = \begin{Bmatrix} a_3 \\ b_4 \end{Bmatrix}$

could yield

$$\begin{Bmatrix} a_1b_1a_3 \\ a_1b_1a_4 \\ a_2b_2a_3 \\ a_2b_2a_4 \end{Bmatrix}$$

Unfortunately, if a_1a_3 and a_1a_4 are unsatisfiable classes the sub-domain b_1 is not covered by our set of classes which is thus inadequate. One test set which is adequate in these circumstances is:

$$\begin{Bmatrix} a_2b_1a_3 \\ a_2b_2a_4 \end{Bmatrix}$$

What we have done is perform the partial combination (hence unnecessarily committing ourselves) while we were unsure of the global satisfiability of the intermediate classes. Better would have been to delay the partial combinations until certain that the potential classes generated so far are satisfiable.

Thus this naive approach, where the combinations are performed as soon as some sub-domains have been generated, is inadequate. Partial combinations cannot be performed immediately but must wait until all combinations between dependent sub-domains have been performed and checked for consistency.

5.2.2 Using Graphs

Our proposal for using graphs stems from the fact that partitioning expressions can be represented using graphs and may offer an efficient way to generate test sets.

For example the partitioning expression:

$$\begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} \times \begin{Bmatrix} b_1 \\ b_2 \end{Bmatrix} \times \begin{Bmatrix} a_3 \\ a_4 \end{Bmatrix}$$

which could be the underlying partition expression of $exp_1(x) \wedge exp_2(y) \wedge exp_3(x)$ analysed in the previous section, can be represented by a directed graph as in Figure 5.1 where the vertices represent the sub-domains and the arcs possible combinations.

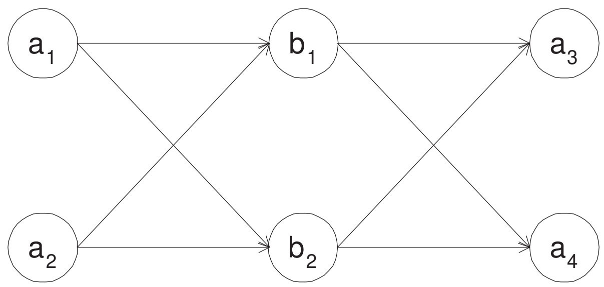

Figure 5.1: Simple Partitioning Graph

One way to perform the combinations for dependent sub-domains first is to regroup the dependent sub-domains in singleton vertices as in Figure 5.2 and then find a minimal (in terms of the number of classes necessary) cover of the graph.

We have however found this process challenging and difficult. One way to get closer to a solution is to transform the graph such that all the vertices dependent on a given set of variables are made adjacent: merging is then straightforward.

Partitioning expressions can become very complex and their subsequent graph difficult to manipulate. For example consider the following partitioning expression:

$$\begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} \times \bigcup \left\{ \begin{matrix} \begin{Bmatrix} b_1 \\ b_2 \end{Bmatrix} \times \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix} \\ \begin{Bmatrix} b_3 \\ b_4 \end{Bmatrix} \times \begin{Bmatrix} d_1 \\ d_2 \end{Bmatrix} \end{matrix} \right\} \times \bigcup \left\{ \begin{matrix} \begin{Bmatrix} e_1 \\ e_2 \end{Bmatrix} \times \begin{Bmatrix} d_3 \\ d_4 \end{Bmatrix} \\ \begin{Bmatrix} e_3 \\ e_4 \end{Bmatrix} \times \begin{Bmatrix} f_1 \\ f_2 \end{Bmatrix} \end{matrix} \right\}$$

where the only dependency is between the sub-domains denoted by the d_is. It is then difficult to find a general solution to our merging of vertices problem. In

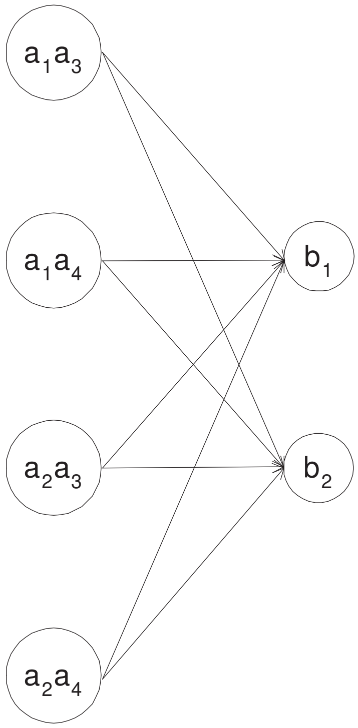

Figure 5.2: Merging Dependent Vertices

particular the manipulation of the intermediate sub-paths between the dependent vertices is problematic. For illustration purposes the graph corresponding to this partitioning expression is given in 5.3 and its associated graph with merged vertices is given in Figure 5.4.

Assuming that the combinations performed are satisfiable and recalling that the remaining sub-domains are disjoint (as we have assumed them to be independent) a minimal path covering algorithm could be applied to obtain, for example, the following set of paths:

$$\left\{\begin{array}{c} a_1b_1c_1d_3e_1 \\ a_1b_2c_2d_4e_2 \\ a_2b_3d_1e_3f_1 \\ a_2b_4d_2e_4f_2 \\ a_1b_3d_1d_3e_1 \\ a_1b_3d_1d_4e_2 \\ a_2b_4d_2d_3e_1 \\ a_2b_4d_2d_4e_2 \end{array}\right\}$$

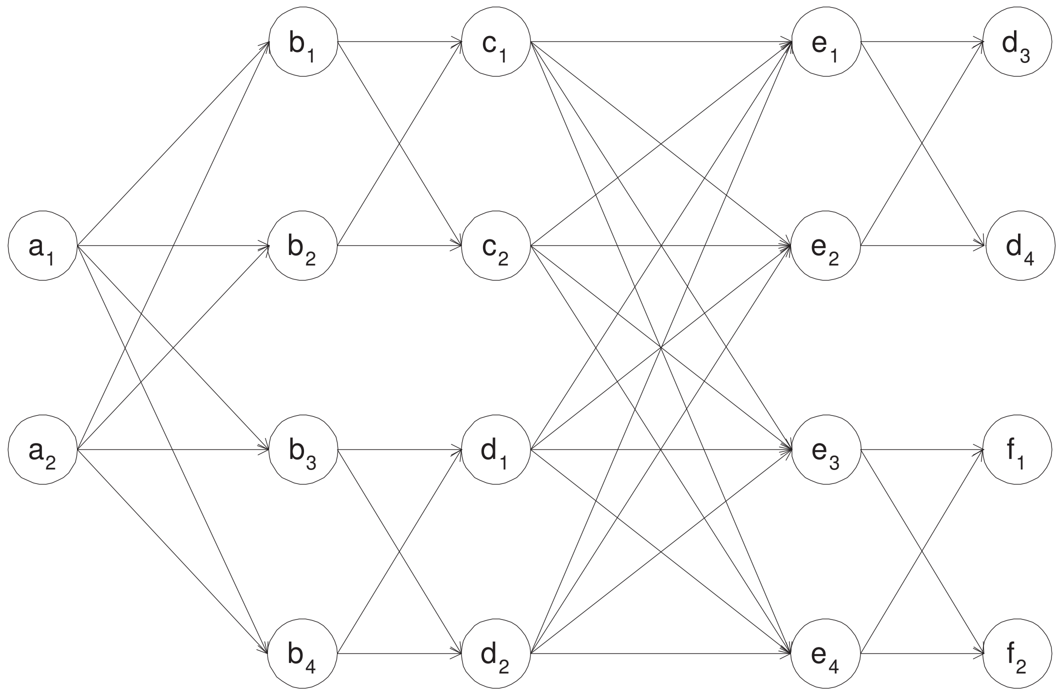

Figure 5.3: A Complex Graph

While a deeper study of graph theory may be rewarded with an efficient and general algorithmic solution to our problem (especially the study of bipartite graphs and set covers) we have abandoned this particular approach because of its complexity. To pursue with this approach we would also need to consider function calls which, in the case of multiple calls, introduce cycles in the graphs.

We now return to our initial approach whereby the final partition is first generated, using our systematic approach, and then a minimal set of classes identified,

5.3 Systematic Test Cases Generation

As seen in the previous section, discovering an algorithm for the direct construction of a minimal set of classes is not as straightforward as might have been anticipated. Below we will present our algorithm for the generation of test sets.

This algorithm is based on our systematic partitioning technique presented in the previous chapter and a redundancy analysis phase which will detect and eliminate redundant classes from the partition to obtain a minimal set of classes

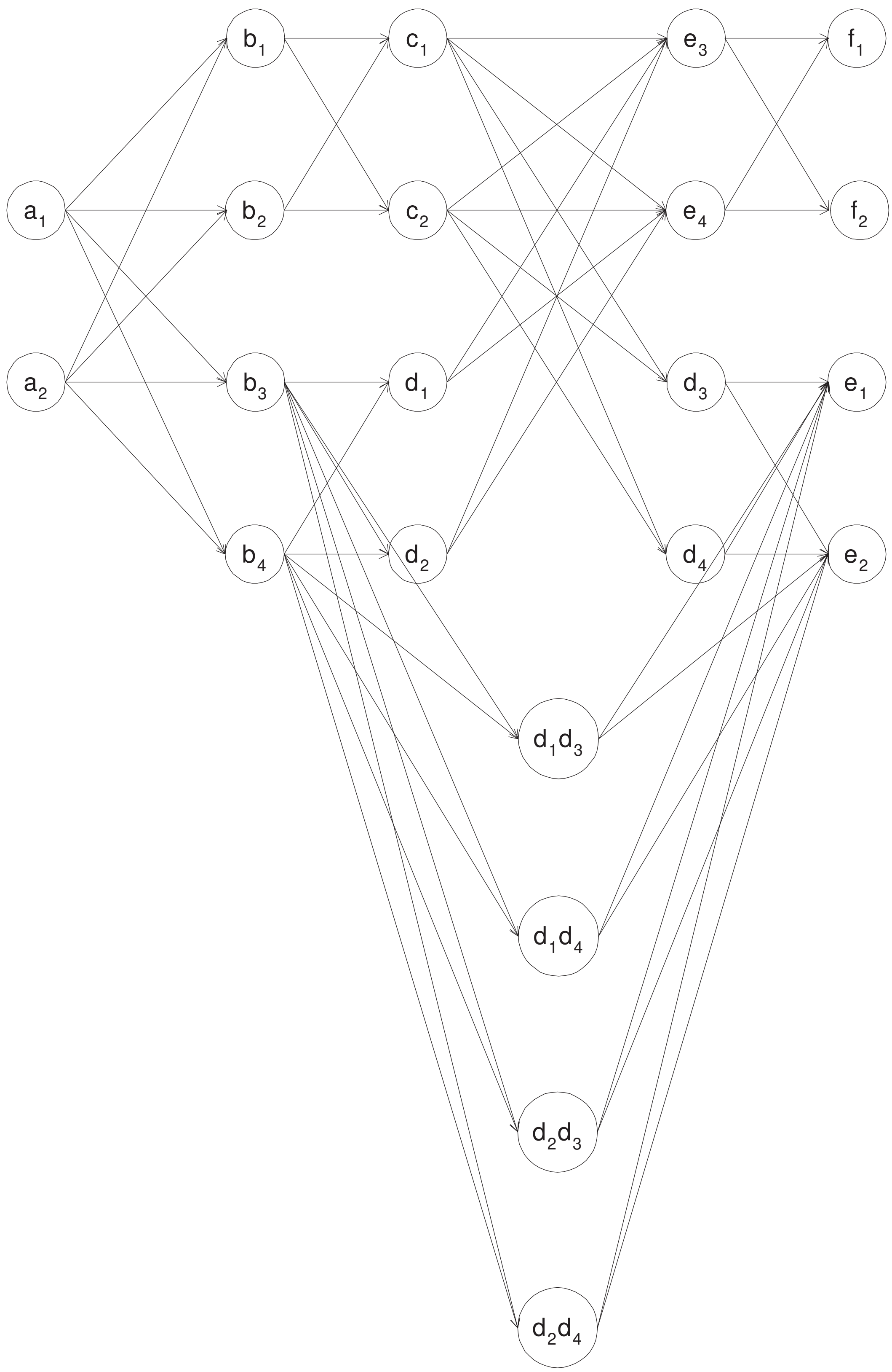

Figure 5.4: Merging Dependent Vertices in Complex Graph

suitable for sampling.

To highlight the relation between the syntax of VDM-SL expressions and their partitioning we will use the concept of parsing.

5.3.1 Parsing

A parser is a program for recognising sentences of a particular language while performing some actions. Parsing is principally used in compiler technology and is best explained by an example.

First we need a description of the syntax of the language to be parsed. We will give this syntax in a similar fashion as the commonly used Backus-Naur Form notation. Although we will eventually work on the VDM-SL syntax, we illustrate the principle of parsing using a small language. A syntax is a series of production rules as in:

```
assign = VARIABLE  ':='  exp
exp = NUMBER
exp = exp  '+'  exp
exp = exp  '-'  exp
exp = exp  '*'  exp
exp = exp  '/'  exp
exp =  '('  exp  ')'
```

These rules specify that an `assign` sentence is composed of a variable followed by the $':='$ symbol followed by an `exp` sentence which is defined elsewhere in the grammar. An `exp` sentence can be any of the alternatives listed. It can be seen that an `exp` sentence is recursively defined.

In the above grammar, `NUMBER` is a terminal for which no syntax rule is given, and `exp` is a non-terminal which is defined elsewhere in the grammar.

For the language above the following expressions are valid: $x := 3 * 9 + 6$, $y := 3 * (9 - (9 + 6))/(9)$. The following expressions are non valid according to the grammar above $x := (9 + 8$ and $7 := x$.

A parser generator is a program for converting a grammatical specification of a language, as given above, into a parser that will parse sentences of a language while performing some actions. The actions to be performed can be added to the syntax rules. In our example to calculate the value of the right hand side of an assignement, actions noted within square brackets are given below:

`assign = VARIABLE` $':='$ `exp` $[print(S1 := S2)]$

`exp = NUMBER` $[S0 := S1]$

`exp = exp` $'+'$ `exp` $[S0 := S1 + S2]$

`exp = exp` $'-'$ `exp` $[S0 := S1 - S2]$

`exp = exp` $'*'$ `exp` $[S0 := S1 * S2]$

`exp = exp` $'/'$ `exp` $[S0 := S1/S2]$

`exp =` $'('$ `exp` $')'$ $[S0 := S1]$

Values can be attached to non-terminals and terminals. In the above $S0$ denotes the value returned by the syntax rule, this value is attached to the left most non-terminal in the rule. Within the right hand side of the syntax rule the value of the first non-terminal or terminal is denoted by $S1$, the second value by $S2$ etc. This particular notation is a simplification of the notation used in Yacc which is a parser generator distributed under Unix.

The parser generated using the rules above will not behave appropriately if the precedence and associativity of operators are not taken into account. Although we will not explain them, notations are also available for specifying the precedence and associativity of operators ensuring that the correct rule is applied in case of alternatives. This ensures that the parser generated is deterministic.

We will use the notation as presented to specify our parsers. The syntax rules given would need to be transformed into the particular input format of the parser generator adopted. We note that these transformations are straightforward, but for a complex language such as VDM-SL the effort involved cannot be underestimated. We can now give our partitioning rule using syntax rules. We will show how our formalism presented so far can be mapped to syntax rules in order to underline the algorithmic nature of our technique.

The parsing rules above are more difficult to manipulate than the partitioning rules given previously using the P notation. However they have the merit of clearly linking our partitioning process to the syntax of VDM-SL.

Also we will assume that the specification given as input is correct: no syntax checking will be perfomed.

5.3.2 Algorithm Overview

The first phase is to establish the dependency relations of the variables present in a specification.

The second phase consists of generating the final partition of the specification. This partition will be composed of classes with associated dependency information represented using labels.

In the third phase we will perform a redundancy analysis on the final partition to detect and eliminate redundant classes. The outcome of this penultimate phase is a minimal set of classes.

The fourth and final phase consists of generating the actual test cases. This will be performed by sampling every remaining class using a solver.

5.3.3 First Phase: Establishing Dependencies

The outcome of this phase is a function, *GetLabel* that will, given a VDM-SL predicate, return a dependency label denoting the dependency characteristic of the predicate. We proceed in a number of steps.

In a first step we will parse the specification, and collect in a set, *DirectD*, the sets of variables appearing in every non-path expression.

In a second step, *DirectD* is transformed into a partition of variables: every equivalence class in the partition is a, non empty, set of dependent variables, the equivalence classes are mutually disjoint, and every variable is represented in the partition.

First Step

DirectD is a set of sets of variables, it is initially empty. In the syntax rules below, `ANYTHING` denotes an outer level non-path expression. These simple syntax rules are based on the VDM-SL syntax [112].

```
exp = exp  '∧'  exp
exp = exp  '∨'  exp
exp = exp  '⇒'  exp
exp = exp  '⇔'  exp
exp =  '¬'  exp
exp =  'if'  exp  'then'  exp  'else'  exp
exp =  '('  exp  ')'
```

exp = ANYTHING $[DirectD := DirectD \cup variables\ \ appearing\ \ in\ \ S1]$

(For brievety we have omitted some rules; for example, to deal with *cases* expressions)

Second Step

The following algorithm builds a partition of variables `Dependence` from the initial call `Merge(DirectD)`.

```
Merge(DirectD: set of sets of vars) :  set of sets of vars
  Dependence := DirectD
  While DirectD <> {} Do
    Take one set of variables s0 from DirectD
    ss := {s1 | s1 ∈ DirectD · s0 ∩ s1 ≠ ∅}
    Dependence := Dependence - ss
    DirectD := DirectD - ss
    Dependence := Dependence ∪ ⋃ ss
  end while
  return Dependence
end Merge
```

Every equivalence class in the `Dependence` partition is then mapped to a unique identifier (in our examples we have chosen a letter).

GetLabel

Given a VDM-SL expression (which should not be an outer level path expression) `GetLabel` returns a label composed of:

- the unique identifier (we have chosen a letter) assigned to the equivalence class of `Dependence` which contains the variables appearing in the VDM-SL expression. This identifier represents the dependency characteristic of the expression.

- a unique identifier (we have chosen an integer subscript) for this particular call to `GetLabel` and particular dependency characteristic.

If the expression does not contain any variables nothing is returned. In our examples, dependency characteristics are denoted by lower case letters and the particular call is denoted by an integer subscript.

5.3.4 Second Phase: Systematic Partitioning with Labels

In this phase a VDM-SL expression will be partitioned according to our systematic partitioning rules given in the previous chapter. We express these rules in the form of syntax rules suitable for parser generation.

Negate Function

We will need a negate function to return the negation of a VDM-SL expression.

The *negate* function can be defined in terms of a parser which we specify using syntax rules. For clarity we prefix the syntax rules by an abreviation of the name of the current parser being defined. Here `N:` prefixes the rules for the *negate* parser

For example, the VDM-SL syntax for conjunctive expressions is:

`N:exp = exp '∧' exp`

to which we can associate the action:

$[S0 := S1 \vee S2]$

so that the disjunction of the negated operands is returned. Similarly the following rules can be devised:

`N:exp = exp '∨' exp`

$[S0 := S1 \wedge S2]$

`N:exp = ANYTHING '⇒' exp`

$[S0 := S1 \wedge S2]$

This last rule illustrates how to express the fact that as the negation of $A \Rightarrow B$ is $A \wedge \neg B$, the first expression does not need to be negated and can therefore be considered as a terminal in our notation. Implicitly $S1$ in this case, is simply an echo of the expression itself.

Our aim here is not to present all the rules necessary for the negation of VDM-SL expressions, but to illustrate how it can be performed using parsing techniques. Some simple rules are given below to illustrate further this process.

N:exp = $'\neg'$ ANYTHING

$[S0 := S1]$

N:exp = VARIABLE

$[S0 := S1 = false]$

N:exp = $'true'$

$[S0 := false]$

N:exp = $'false'$

$[S0 := true]$

N:exp = ANYTHING $'='$ ANYTHING

$[S0 := S1 \neq S2]$

N:exp = ANYTHING $'\geq'$ ANYTHING

$[S0 := S1 < S2]$

N:exp = ANYTHING $'<'$ ANYTHING

$[S0 := S1 \geq S2]$

Below we give a simple rule for *if* ... *then* ... *else* ... expressions:

N:exp = $'if'$ ANYTHING $'then'$ exp $'else'$ exp

$[S0 :=$ *if S1 then S2 else S3*$]$

This rule would need to be extended for the general VDM-SL *if* expressions which may contain *elseif* parts. Similarly, parsing rules for *cases* expressions would need to be devised. We do not forsee any problems for extending the parser to these expressions apart from the length and fastidiousness of the parsing rules.

To conclude we also give the parsing rules for negating quantified expressions:

N:exp = $'\forall'$ ANYTHING $'\cdot'$ exp

$[S0 := \exists S1 \cdot S2]$

N:exp = $'\exists'$ ANYTHING $'\cdot'$ exp

$[S0 := \forall S1 \cdot S2]$

Parsers for Partitioning

We will keep track of the dependence charateristics of the sub-domains generated during the partitioning of non-path expressions by associating a label (using the *GetLabel* function) to the predicate representing a particular sub-domain.

A class, c, will therefore be composed of a VDM-SL predicate denoted by *c.pred* and a set of labels, *c.lab* denoting the original sub-domains making up the class c.

As we recall from 4.1.2. a partition, P, of a predicate E is a set of equivalence classes composed of sub-domains, such that:

$$(\forall c \in P \cdot consistent(c.pred)) \wedge \bigoplus_{c_i \in P} c_i.pred \Leftrightarrow E$$

Also the $\times$ operator is:

$\_ \times \_ : Partition \times Partition$

$P_1 \times P_2 == \{(c_1.pred \wedge c_2.pred, c_1.lab \cup c_2.lab)$

$\qquad\qquad | \; c_1 \in P_1, c_2 \in P_2 \cdot consistent(c_1.pred \wedge c_2.pred)\}$

The $\bigcup$ operator denotes, as usual, the mathematical distributed union over sets.

In accordance with our systematic partitioning technique we differentiate between two levels of partitioning. Coarse partitioning rules are applied on path expressions, and refined partitioning rules applied on non-path expressions.

We also need to differentiate between parsing an expression which we want to evaluate to true and an expression which needs to be undefined. The suffixes *_true* and *_undef* are used for this purpose.

We therefore need to define four parsers: *coarse_true*, *coarse_undef*, *refine_true* and *refine_undef*. The abreviations used will be `CT`, `CU`, `RT`, `RU` respectively.

In the syntax rules below, the *.part* of a returned value denotes the partition of the expression and *.exp* designates the expression literal prior to parsing. The actions to produce the expression literals are repetitive and only consist of recopying the expressions; we therefore omit them. As we will see from the syntax rules, when a non-path expression is encountered in a coarse parser, the appropriate *refine* parser is called on the expression to generate the sub-domains according to our refined partitioning rules.

Coarse_true Parser

The *coarse_true* parser is relatively simple, for example some of the rules for logical operators are:

```
CT:exp = exp  '∧'  exp
```

$$\left[S0.part := S1.part \times S2.part \right]$$

and:

```
CT:exp = exp  '∨'  exp
```

$$\left[\begin{array}{l} S3 := coarse\_true(negate(S1.exp)); \\ S4 := coarse\_true(negate(S2.exp)); \\ S5 := coarse\_undef(S1.exp); \\ S6 := coarse\_undef(S2.exp); \\ S0.part := \bigcup \left\{ \begin{array}{c} S1.part \times S4.part \\ S1.part \times S2.part \\ S3.part \times S2.part \\ S1.part \times S6.part \\ S5.part \times S2.part \end{array} \right\} \end{array} \right]$$

This latest rule calls the *coarse_undef* parser, which we will define next, to take into account the potential LPF behaviour of logical conjunctions. All the rules given are simple transformations of their corresponding coarse partitioning rule given in section 4.1.3.

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

if expressions are dealt with in the following manner:

```
CT:exp =  'if'  exp  'then'  exp  'else'  exp
```

$$\left[\begin{array}{l} S4 := coarse\_true(negate(S1.exp)); \\ S5 := coarse\_undef(S1.exp); \\ S0.part := \bigcup \left\{ \begin{array}{c} S1.part \times S2.part \\ S4.part \times S3.part \\ S5.part \times S3.part \end{array} \right\} \end{array} \right]$$

Again here, we only present a subset of the necessary syntax.

`CT:exp = '¬' ANYTHING`

$$\left[S0 := \mathit{coarse\_true}(\mathit{negate}(S1))\right]$$

Finally, anything that is not a path expression is parsed using the *refine_true* parser.

`CT:exp = ANYTHING`

$$\left[\mathit{S0} := \mathit{refine\_true}(\mathit{S1})\right]$$

Coarse_undef Parser

In a similar fashion we give some of the grammatical rules for the *coarse_undef* parser. These rules are simple tranformations of our original partitioning rules.

For example, the rules for expressions based on the logical operators are (they can be checked using the partitioning rules given in the previous chapter which where themselves derived from the table 4.1 detailing the three valued logic conventions of VDM-SL):

`CU:exp = exp '∧' exp`

$$\left[\begin{array}{l} S3 := \mathit{coarse\_true}(\mathit{S1.exp}); \\ S4 := \mathit{coarse\_true}(\mathit{S2.exp})); \\ S0.part := \bigcup \left\{\begin{array}{l} S3.part \times S2.part \\ S1.part \times S4.part \\ S1.part \times S2.part \end{array}\right\} \end{array}\right]$$

`CU:exp = exp '∨' exp`

$$\left[\begin{array}{l} S3 := \mathit{coarse\_true}(\mathit{negate}(\mathit{S1.exp})); \\ S4 := \mathit{coarse\_true}(\mathit{negate}(\mathit{S2.exp})); \\ S0.part := \bigcup \left\{\begin{array}{l} S3.part \times S2.part \\ S1.part \times S4.part \\ S1.part \times S2.part \end{array}\right\} \end{array}\right]$$

`CU:exp = '¬' exp`

$$\left[S0 := S1\right]$$

The rule to deal with *if* expressions is:

CU:exp = $'if'$ exp $'then'$ exp $'else'$ exp

$$\left[\begin{array}{l} S4 := \mathit{coarse\_true}(\mathit{S1.exp}); \\ S5 := \mathit{coarse\_true}(\mathit{negate}(\mathit{S1.exp})); \\ S0.part := \bigcup \left\{\begin{array}{l} S4.part \times S2.part \\ S5.part \times S3.part \\ S1.part \times S3.part \end{array}\right\} \end{array}\right]$$

Finally, anything that is not a path expression is partitioned using the *refine_undef* parser:

CU:exp = ANYTHING

$$\left[S0 := \mathit{refine\_undef}(\mathit{S1})\right]$$

Refine_true **Parser**

We assign labels to the predicates within a sub-domain whenever a new sub-domain is created. This is done using the *GetLabel* function obtained during the first phase. In what follows the argument of *GetLabel* is usually implicit: it is the predicate part of the current sub-domain. The labels will allow us to differentiate the sub-domains generated during the third phase which is concerned with the detection of redundant classes.

For example, the syntax rule for the $<$ arithmetic relation is:

RT:exp = exp $'<'$ exp

to which we associate the action:

$$\left[\begin{array}{l} S0.part = \left\{\begin{array}{l} (S1.exp + d = S2.exp, GetLabel) \\ (S1.exp + d < S2.exp, GetLabel) \end{array}\right\} \\ \qquad\qquad \times S1.part \times S2.part \end{array}\right]$$

where d is equal to *1* if the expression is over integers and a specified precision amount otherwise as explained in the previous chapter. We do not detail here the way the type of an expression can be determined but the modifications to the parser are similar to those necessary in compiler technology.

Further, the syntax rule for the $\leq$ arithmetic relation is:

RT:exp = exp $'\leq'$ exp

to which we associate the action:

$$\left[\begin{array}{l} S0.part = \left\{\begin{array}{l}(S1.exp < S2.exp, GetLabel) \\ (S1.exp = S2.exp, GetLabel)\end{array}\right\} \\ \qquad\qquad \times S1.part \times S2.part \end{array}\right]$$

Some rules are very simple as we do not partition every VDM-SL expression. For example in our systematic partitioning technique described in the previous chapter we have decided against partitioning arithmetic additions. Thus:

```
RT:exp = exp '+' exp
```

$$\left[S0.part = S1.part \times S2.part\right]$$

Some expressions are partitioned into many sub-domains. Below we give the example of the set union expression which is partitioned according to the sub-domains identified in 4.2.2.

```
RT:exp = exp '∪' exp
```

is associated with the action:

$$\left[\begin{array}{l} S0.part = \left\{\begin{array}{l}(S1.exp = \{\} \wedge S2.exp = \{\}, GetLabel) \\ (S1.exp = \{\} \wedge S2.exp \neq \{\}, GetLabel) \\ (S1.exp \neq \{\} \wedge S2.exp = \{\}, GetLabel) \\ (S1.exp \neq \{\} \wedge S2.exp \neq \{\} \wedge S1.exp \cap S2.exp = \{\}, \\ \qquad GetLabel), \\ (S1.exp \neq \{\} \wedge S2.exp \neq \{\} \wedge S1.exp \subset S2.exp, \\ \qquad GetLabel), \\ (S1.exp \neq \{\} \wedge S2.exp \neq \{\} \wedge S2.exp \subset S1.exp, \\ \qquad GetLabel), \\ (S1.exp \neq \{\} \wedge S2.exp \neq \{\} \wedge S1.exp = S2.exp, \\ \qquad GetLabel, \\ (S1.exp \neq \{\} \wedge S2.exp \neq \{\} \wedge S1.exp \cap S2.exp \neq \{\} \wedge \\ \qquad \neg(S1.exp \subset S2.exp) \wedge \neg(S2.exp \subset S1.exp) \wedge \\ \qquad S1.exp \neq S2.exp, GetLabel) \end{array}\right\} \\ \qquad\qquad \times S1.part \times S2.part \end{array}\right]$$

which generates the 8 specific sub-domains for a set union expression. Below we give some rules to deal with syntax terminals.

`RT:exp = VARIDENTIFIER`

$$\left[S0.part = \{(true, GetLabel(S1))\} \right]$$

`RT:exp = NUMBER`

$$\left[S0.part = \{(true, \{\})\} \right]$$

So for example, if y is an integer:

$$Refine\_true(y < 5).part =$$

$$\begin{Bmatrix} (y+1=5, \{y_2\}) \\ (y+1<5, \{y_3\}) \end{Bmatrix} \times \{(true, y_1)\} \times \{(true, \{\})\}$$

Thus $Refine\_true(y < 5).part$ can be represented as:

$$\begin{Bmatrix} (y=4, \{y_2 y_1\}) \\ (y<4, \{y_3 y_1\}) \end{Bmatrix}$$

We also need to provide syntax rules for embedded path expressions, we therefore need to include the rules of *coarse_true* within the *refine_true* parser. Those rules are included with a minimum of changes: we only need to ensure that the correct parser is called. For example, below we give a simplified rule (in the sense that the *elseif* part is omitted) for *if ... then ... else ...* expressions:

`RT:exp =` $'if'$ `exp` $'then'$ `exp` $'else'$ `exp`

$$\left[\begin{array}{l} S4 := refine\_true(negate(S1.exp)); \\ S5 := refine\_undef(S1.exp); \\ S0.part := \bigcup \begin{Bmatrix} S1.part \times S2.part \\ S4.part \times S3.part \\ S5.part \times S3.part \end{Bmatrix} \end{array} \right]$$

As another example of a rule for embedded path expressions we give the simple rule for logical conjunctions below.

`RT:exp = exp` $'\wedge'$ `exp`

$$\left[S0.part := S1.part \times S2.part \right]$$

We deal with function calls in the following fashion:

`RT:exp = FunctionID` $'('$ `parameter_list` $')'$

$$\left[S0.part := S1.part \times S2.part \right]$$

Where the partition returned by the terminal `FunctionID` is the final partition of the specification of the function obtained from a separate analysis.

`RT:parameter_list = exp` $\left[S0 := S1 \right]$

`RT:parameter_list = parameter_list` $','$ `exp`

$$\left[S0.part := S1.part \times S2.part \right]$$

To conclude we give the more complex refined partitioning rules for quantified expressions. The rules below are transformations of the rules given in section 4.3.1 (apart from the use of labels the semantics is equivalent). We have also changed the presentation style of the partitioning rules by performing most of the combinations in situ (using the set comprehension notation): this gives a more direct approach. Below $\bigwedge_{c \in ts} \ldots$ denotes the conjunction of every expression for every c in ts.

```
RT:exp =  '∀'  ANYTHING  '∈'  exp  '·'  exp
```

$$\left[\begin{array}{l} tmp\_part := \{((\bigwedge_{c \in ts} \exists S1 \in S2.exp \cdot c.pred) \wedge (\bigwedge_{c \in fs} \forall S1 \in S2.exp \cdot \neg c.pred), \\ \qquad\qquad \bigcup_{c \in ts} c.lab) \mid ts \subseteq S3.part \wedge ts \neq \emptyset \wedge fs = S3.part/ts\}; \\ S0.part := \bigcup \left\{\begin{array}{l} \{(S2.exp = \emptyset, GetLabel)\} \\ \{(\forall S1 \in S2.exp \cdot S3.exp, GetLabel)\} \;\; \times tmp\_part \end{array}\right\} \\ \qquad\qquad \times S2.part \end{array}\right]$$

Also:

```
RT:exp =  '∃'  ANYTHING  '∈'  exp  '·'  exp
```

$$\left[\begin{array}{l} tmp\_part := \{((\bigwedge_{c \in ts} \exists S1 \in S2.exp \cdot c.pred) \wedge (\bigwedge_{c \in fs} \forall S1 \in S2.exp \cdot \neg c.pred), \\ \qquad\qquad \bigcup_{c \in ts} c.lab) \mid ts \subseteq S3.part \wedge ts \neq \emptyset \wedge fs = S3.part/ts\}; \\ S0.part := \left\{\begin{array}{l} ((\forall S1 \in S2.exp \cdot S3.exp) \wedge S2.exp \neq \emptyset, GetLabel) \\ (\exists S1 \in S2.exp \cdot \neg S3.exp \vee S3.exp*, GetLabel) \end{array}\right\} \\ \qquad\qquad \times tmp\_part \times S2\_part \end{array}\right]$$

Refine_undef Parser

With this parser we want to partition non-path expressions to obtain undefined expressions. For example we can create a rule to deal with all arithmetic relations. An arithmetic relation is undefined if any of its operands is undefined or not a number.

RU:arith_rel = $'<' \quad | \quad '\leq' \quad | \quad '>' \quad | \quad '\geq'$

RU:exp = exp arith_rel exp

$$\left[\begin{array}{l} S3.part := \{(is\_num(S1.exp), GetLabel)\} \\ S4.part := \{(\neg is\_num(S1.exp), GetLabel)\} \\ S5.part := \{(is\_num(S2.exp), GetLabel)\} \\ S6.part := \{(\neg is\_num(S2.exp), GetLabel)\} \\ S0.part := \bigcup \left\{\begin{array}{l} S1.part \times S5.part \\ S1.part \times S2.part \\ S1.part \times S6.part \\ S4.part \times S5.part \\ S4.part \times S2.part \\ S4.part \times S6.part \\ S3.part \times S2.part \\ S3.part \times S6.part \end{array}\right\} \end{array}\right]$$

Most binary operators behave in the same fashion, only a few, such as the divide by operator when the divisor is 0, lead to undefined expressions in special circumstances. As an example, we give below the syntax rule for the partitioning of integer division expressions.

`RU:exp = exp` $'DIV'$ `exp`

$$\left[\begin{array}{l} S3.part := \{(is\_int(S1.exp), GetLabel)\} \\ S4.part := \{(\neg is\_int(S1.exp), GetLabel)\} \\ S5.part := \{(is\_int(S2.exp), GetLabel)\} \\ S6.part := \{(\neg is\_int(S2.exp), GetLabel)\} \\ S7.part := \{(S2.exp = 0, GetLabel)\} \\ S0.part := \bigcup \left\{ \begin{array}{l} S1.part \times S5.part \\ S1.part \times S2.part \\ S1.part \times S6.part \\ S1.part \times S7.part \\ S4.part \times S5.part \\ S4.part \times S2.part \\ S4.part \times S6.part \\ S4.part \times S7.part \\ S3.part \times S2.part \\ S3.part \times S6.part \\ S3.part \times S7.part \end{array} \right\} \end{array}\right]$$

Some non recursive rules are:

`RU:exp = VARIDENTIFIER`

$$\left[S0.part = \{(S1*, GetLabel)\}\right]$$

`RU:exp = NUMBER`

$$\left[S0.part = \{(false, GetLabel)\}\right]$$

We also need of course to reproduce the rules from *coarse_undef* for this parser to deal with embedded path expressions:

`RU:exp = exp '∧' exp`

$$\left[\begin{array}{l} S3 := \mathit{refine\_true}(\mathit{S1.exp}); \\ S4 := \mathit{refine\_true}(\mathit{S2.exp})); \\ S0.part := \bigcup \left\{\begin{array}{l} S3.part \times S2.part \\ S1.part \times S4.part \\ S1.part \times S2.part \end{array}\right\} \end{array}\right]$$

`RU:exp = exp '∨' exp`

$$\left[\begin{array}{l} S3 := \mathit{refine\_true}(\mathit{negate}(\mathit{S1.exp})); \\ S4 := \mathit{refine\_true}(\mathit{negate}(\mathit{S2.exp})); \\ S0.part := \bigcup \left\{\begin{array}{l} S3.part \times S2.part \\ S1.part \times S4.part \\ S1.part \times S2.part \end{array}\right\} \end{array}\right]$$

`RU:exp = '¬' exp`

$$\left[S0 := S1 \right]$$

The rule to deal with *if* expressions is:

`RU:exp = 'if' exp 'then' exp 'else' exp`

$$\left[\begin{array}{l} S4 := \mathit{refine\_true}(\mathit{S1.exp}); \\ S5 := \mathit{refine\_true}(\mathit{negate}(\mathit{S1.exp})); \\ S0.part := \bigcup \left\{\begin{array}{l} S4.part \times S2.part \\ S5.part \times S3.part \\ S1.part \times S3.part \end{array}\right\} \end{array}\right]$$

5.3.5 Third Phase: Redundancy Analysis

The outcome of our second phase is the final partition. Each equivalence class in the partition is composed of a satisfiable predicate and a set of labels representing the sub-domains that were combined to obtain the predicate.

Using the set of labels we need to detect and eliminate redundant classes from the partition to obtain a minimal set of classes suitable for sampling.

First Step

We will first need to denote every group of dependent labels appearing in the classes of the final partition by a unique label (dependent labels are readily identifiable using our convention: this is the role of the predicate `are_dependent` below).

We give below an algorithm for the *Regroup* function which takes a set of labels associated with a predicate within a class and returns a set of sets of dependent labels (*Regroup* is quite similar to the *GetLabel* function).

```
Regroup(L: set of labels) :  set of sets of labels
  GL := ∅
  While L <> ∅ Do
    Take one label l from L
    G := {l1 | l1 ∈ L · are_dependent(l, l1)}
    L := L - G
    GL := GL ∪ {G}
  end while
  return GL
end Regroup
```

So for example $Regroup(\{a_2 b_3 a_3 c_1 b_1\})$ is equal to $\{\{c_1\}, \{a_2 a_3\}, \{b_1 b_3\}\}$.

So first, the set of groups of dependent labels in the final partition P represented by the set $GS = \{G \mid c \in P, G \in Regroup(c.lab)\}$ is constructed.

Then every group in GS is mapped to a unique label (we will use the same convention as before in terms of letters and subscripts).

Finally, the set of labels, $c.lab$, of a class c in the final partition P is replaced by the new set of labels associated with $Regroup(c.lab)$ according to the mapping defined above.

Second Step

In this final step of the third phase of our technique we will identify and eliminate redundant classes to obtain a minimal set of classes. But for interest we can first formally characterise a minimal set of classes, MCs of a final partition P.

Let CCs be the set of sets of classes which cover the groups of dependent sub-domains appearing in the final partition P:

$$CCs = \{Cs \mid Cs \subset P \cdot \bigcup\{c.lab \mid c \in Cs\} = \bigcup\{c.lab \mid c \in P\}\}$$

A minimal set of classes MCs of P is such that:

$$MCs \in CCs \wedge \neg(\exists MCs_1 \in CCs \cdot card MCs_1 < card MCs)$$

(i.e. there is no smaller set of classes in CCs)

As we have already mentioned there can be many minimal sets of classes for a given final partition (and they are viewed as equivalent from a testing point of view): the algorithm we propose below to derive a minimal set of classes is therefore non deterministic.

```
  initial call:  Minimal(P, {})
Minimal(RC, MC : set of classes) :  set of classes
  if RC = ∅ then return MC
  if ∃C ∈ RC · ∃C1 ∈ RC ∪ MC · C.lab ⊆ C1.lab then
   return Minimal(RC − {c}, MC)
  if ∃C ∈ RC · ∃label ∈ C.lab · ¬(∃C1 ∈ RC ∪ MC · label ∈ C1.lab)
   then return Minimal(RC − {C}, MC ∪ {C})
  let C ∈ RC in return Minimal(RC − {C}, MC)
end Minimal
```

5.3.6 Fourth Phase: Sampling

The result of the third phase is a minimal set of classes. This minimal set of classes can be sampled using a solver to obtain a set of test cases.

5.4 Examples

In this section we illustrate our technique on some small examples. Because of a lack of space we cannot illustrate every atomic step as would be generated by following step by step the parsing process.

To keep the second phase (i.e. the systematic partitioning) relatively short, the examples below are based on VDM-SL arithmetic expressions with LPF behaviour disabled.

Our aim here is to show the kind of minimal set classes obtained using our technique: we therefore omit the fourth phase.

The labels associated with the predicates are represented alongside the predicates. We will also numerate the equivalence classes for clarity.

5.4.1 No Function, No Dependence

We consider the expression: $x > 0 \vee y < 5$.

First Phase

The set of set of variables is $\{\{x\},\{y\}\}$ which is mapped to the labels x_is and y_is respectively.

Second Phase

$$coarse\_true(x > 0 \vee y < 5).part = \left\{ \begin{pmatrix} x = 1 \wedge y = 5 \\ x = 1 \wedge y > 5 \\ x > 1 \wedge y = 5 \\ x > 1 \wedge y > 5 \\ x = 1 \wedge y = 4 \\ x = 1 \wedge y < 4 \\ x > 1 \wedge y = 4 \\ x > 1 \wedge y < 4 \\ x = 0 \wedge y = 4 \\ x = 0 \wedge y < 4 \\ x < 0 \wedge y = 4 \\ x < 0 \wedge y < 4 \end{pmatrix} \begin{matrix} \{x_1x_2y_1y_2\}\ 1 \\ \{x_1x_2y_1y_3\}\ 2 \\ \{x_1x_3y_1y_2\}\ 3 \\ \{x_1x_3y_1y_3\}\ 4 \\ \{x_1x_2y_4y_5\}\ 5 \\ \{x_1x_2y_4y_6\}\ 6 \\ \{x_1x_3y_4y_5\}\ 7 \\ \{x_1x_3y_4y_6\}\ 8 \\ \{x_4x_5y_4y_5\}\ 9 \\ \{x_4x_5y_4y_6\}10 \\ \{x_4x_6y_4y_5\}11 \\ \{x_4x_6y_4y_6\}12 \end{matrix} \right.$$

Third Phase

After the first step the final partition is:

$$\left\{\begin{array}{l|l} x=1 \wedge y=5 & \{a_1b_1\}\ 1 \\ x=1 \wedge y>5 & \{a_1b_2\}\ 2 \\ x>1 \wedge y=5 & \{a_2b_1\}\ 3 \\ x>1 \wedge y>5 & \{a_2b_2\}\ 4 \\ x=1 \wedge y=4 & \{a_1b_3\}\ 5 \\ x=1 \wedge y<4 & \{a_1b_4\}\ 6 \\ x>1 \wedge y=4 & \{a_2b_3\}\ 7 \\ x>1 \wedge y<4 & \{a_2b_4\}\ 8 \\ x=0 \wedge y=4 & \{a_3b_3\}\ 9 \\ x=0 \wedge y<4 & \{a_3b_4\}10 \\ x<0 \wedge y=4 & \{a_4b_3\}11 \\ x<0 \wedge y<4 & \{a_4b_4\}12 \end{array}\right.$$

Using our algorithm, a minimal set of classes is

$$\left\{\begin{array}{l|l} x=1 \wedge y=5 & \{a_1b_1\}\ 1 \\ x>1 \wedge y>5 & \{a_2b_2\}\ 4 \\ x=0 \wedge y=4 & \{a_3b_3\}\ 9 \\ x<0 \wedge y<4 & \{a_4b_4\}12 \end{array}\right.$$

5.4.2 No Function, Dependence

We consider the expression: $x > 0 \vee x < 5$.

First Phase

The set of sets of variables is $\{\{x\}\}$ which is mapped to the labels x_is.

Second Phase

$$coarse\_true(x > 0 \lor x < 5) = \begin{Bmatrix} x > 1 \land x = 5 \\ x > 1 \land x > 5 \\ x = 1 \land x < 4 \\ x > 1 \land x = 4 \\ x > 1 \land x < 4 \\ x = 0 \land x < 4 \\ x < 0 \land x < 4 \end{Bmatrix} \begin{matrix} \{x_1 x_2 x_4 x_5\}\ 1 \\ \{x_1 x_2 x_4 x_6\}\ 2 \\ \{x_1 x_3 x_9 x_8\}\ 3 \\ \{x_1 x_2 x_9 x_7\}\ 4 \\ \{x_1 x_2 x_9 x_8\}\ 5 \\ \{x_{10} x_{11} x_9 x_8\} 6 \\ \{x_{10} x_{12} x_9 x_8\} 7 \end{matrix}$$

Third Phase

After the first step the final partition is:

$$\begin{Bmatrix} x > 1 \land x = 5 \\ x > 1 \land x > 5 \\ x = 1 \land x < 4 \\ x > 1 \land x = 4 \\ x > 1 \land x < 4 \\ x = 0 \land x < 4 \\ x < 0 \land x < 4 \end{Bmatrix} \begin{matrix} \{a_1\}1 \\ \{a_2\}2 \\ \{a_3\}3 \\ \{a_4\}4 \\ \{a_5\}5 \\ \{a_6\}6 \\ \{a_7\}7 \end{matrix}$$

Redundancy analysis does detect any redundant class: the above is a minimal set of classes.

5.4.3 With Function Call

We consider the expression: $(x > 0 \lor y < 5) \land f(x, y)$. Also,

$f(X, Y) = \mathit{if}\ X > 0\ \mathit{then}\ X + Y > 0\ \mathit{else}\ 2X + Y < 0$

and is deemed by the human tester to have been implemented separately.

To generate the final partition of $f(X, Y)$ we consider the expression:

$\mathit{if}\ X > 0\ \mathit{then}\ X + Y > 0\ \mathit{else}\ 2X + Y < 0$

First Phase

The set of sets of variables is $\{\{X, Y\}\}$ which is mapped to the label a_i

Second Phase

$coarse\_true(if\ X > 0\ then\ X + Y > 0\ else\ 2X + Y < 0) =$

$$\left\{\begin{array}{c} X = 1 \wedge X + Y = 1 \\ X = 1 \wedge X + Y > 1 \\ X > 1 \wedge X + Y = 1 \\ X > 1 \wedge X + Y > 1 \\ X = 0 \wedge 2X + Y = -1 \\ X = 0 \wedge 2X + Y < -1 \\ X < 0 \wedge 2X + Y = -1 \\ X < 0 \wedge 2X + Y < -1 \end{array}\right\} \begin{array}{l} \{a_9 a_1 a_{11} a_5\}\, 1 \\ \{a_9 a_1 a_{11} a_6\}\, 2 \\ \{a_9 a_2 a_{11} a_5\}\, 3 \\ \{a_9 a_2 a_{11} a_6\}\, 4 \\ \{a_{10} a_3 a_{12} a_7\} 5 \\ \{a_{10} a_3 a_{12} a_8\} 6 \\ \{a_{10} a_4 a_{12} a_7\} 7 \\ \{a_{10} a_4 a_{12} a_8\} 8 \end{array}$$

Third Phase

After the first step the final partition is:

$$\left\{\begin{array}{c} X = 1 \wedge X + Y = 1 \\ X = 1 \wedge X + Y > 1 \\ X > 1 \wedge X + Y = 1 \\ X > 1 \wedge X + Y > 1 \\ X = 0 \wedge 2X + Y = -1 \\ X = 0 \wedge 2X + Y < -1 \\ X < 0 \wedge 2X + Y = -1 \\ X < 0 \wedge 2X + Y < -1 \end{array}\right\} \begin{array}{l} \{F_1\}1 \\ \{F_2\}2 \\ \{F_3\}3 \\ \{F_4\}4 \\ \{F_5\}5 \\ \{F_6\}6 \\ \{F_7\}7 \\ \{F_8\}8 \end{array}$$

It is this final partition that is used whenever f is called. Also the labels F_is are independent from each other (to respect the remarks in 5.1.1. about called functions).

We can now return to the original expression: $(x > 0 \vee y < 5) \wedge f(x, y)$

First Phase

The set of sets of variables is $\{\{x\}, \{y\}\}$ which is mapped to the labels x_is and y_is resprectively.

Second Phase

$coarse\_true((x > 0 \vee y < 5) \wedge f(x,y)) =$

$$\left\{\begin{array}{l|l} x = 1 \wedge y = 5 & \{x_5 x_1 y_5 y_1\}\ 1 \\ x = 1 \wedge y > 5 & \{x_5 x_1 y_5 y_2\}\ 2 \\ x > 1 \wedge y = 5 & \{x_5 x_2 y_5 y_1\}\ 3 \\ x > 1 \wedge y > 5 & \{x_5 x_2 y_5 y_2\}\ 4 \\ x = 1 \wedge y = 4 & \{x_5 x_1 y_6 y_3\}\ 5 \\ x = 1 \wedge y < 4 & \{x_5 x_1 y_6 y_4\}\ 6 \\ x > 1 \wedge y = 4 & \{x_5 x_2 y_6 y_3\}\ 7 \\ x > 1 \wedge y < 4 & \{x_5 x_2 y_6 y_4\}\ 8 \\ x = 0 \wedge y = 4 & \{x_6 x_3 y_6 y_3\}\ 9 \\ x = 0 \wedge y < 4 & \{x_6 x_3 y_6 y_4\}10 \\ x < 0 \wedge y = 4 & \{x_6 x_4 y_6 y_3\}11 \\ x < 0 \wedge y < 4 & \{x_6 x_4 y_6 y_4\}12 \end{array}\right\} \times \{X = x \wedge Y = y\}\{x_7 y_7\} \times$$

$$\left\{\begin{array}{c|l} X = 1 \wedge X + Y = 1 & \{F_1\}1 \\ X = 1 \wedge X + Y > 1 & \{F_2\}2 \\ X > 1 \wedge X + Y = 1 & \{F_3\}3 \\ X > 1 \wedge X + Y > 1 & \{F_4\}4 \\ X = 0 \wedge 2X + Y = -1 & \{F_5\}5 \\ X = 0 \wedge 2X + Y < -1 & \{F_6\}6 \\ X < 0 \wedge 2X + Y = -1 & \{F_7\}7 \\ X < 0 \wedge 2X + Y < -1 & \{F_8\}8 \end{array}\right\}$$

Thus (the classes combined are given as an indication and some simplifications are performed for readability) , $Coarse\_true(x > 0 \vee y < 5) \wedge f(x,y)) =$

$$
\left\{\begin{array}{c}
x = 1 \wedge y = 5 \\
x = 1 \wedge y > 5 \\
x > 1 \wedge y = 5 \\
x > 1 \wedge y > 5 \\
x = 1 \wedge y = 4 \\
x = 1 \wedge y = 0 \\
x = 1 \wedge y > 0 \wedge y < 4 \\
x > 1 \wedge y = 4 \\
x > 1 \wedge y < 4 \wedge x + y = 1 \\
x > 1 \wedge y < 4 \wedge x + y > 1 \\
x = 0 \wedge y = -1 \\
x = 0 \wedge y < -1 \\
x < 0 \wedge y = 4 \\
x < 0 \wedge y < 4 \wedge 2x + y = -1 \\
x < 0 \wedge y < 4 \wedge 2x + y < -1
\end{array}\right\}
\begin{array}{l}
\{x_5 x_1 y_5 y_1 F_2\}\ 1,2 \\
\{x_5 x_1 y_5 y_2 F_2\}\ 2,2 \\
\{x_5 x_2 y_5 y_1 F_4\}\ 3,4 \\
\{x_5 x_2 y_5 y_2 F_4\}\ 4,4 \\
\{x_5 x_1 y_6 y_3 F_2\}\ 5,2 \\
\{x_5 x_1 y_6 y_4 F_1\}\ 6,1 \\
\{x_5 x_1 y_6 y_4 F_2\}\ 6,2 \\
\{x_5 x_2 y_6 y_3 F_4\}\ 7,4 \\
\{x_5 x_2 y_6 y_4 F_3\}\ 8,3 \\
\{x_5 x_2 y_6 y_4 F_4\}\ 8,4 \\
\{x_6 x_3 y_6 y_4 F_5\} 10,5 \\
\{x_6 x_3 y_6 y_4 F_6\} 10,6 \\
\{x_6 x_4 y_6 y_3 F_7\} 11,8 \\
\{x_6 x_4 y_6 y_4 F_8\} 12,7 \\
\{x_6 x_4 y_6 y_4 F_7\} 12,8
\end{array}
$$

Third Phase

After the first step the final partition is:

$$
\left\{
\begin{array}{c}
x = 1 \wedge y = 5 \\
x = 1 \wedge y > 5 \\
x > 1 \wedge y = 5 \\
x > 1 \wedge y > 5 \\
x = 1 \wedge y = 4 \\
x = 1 \wedge y = 0 \\
x = 1 \wedge y > 0 \wedge y < 4 \\
x > 1 \wedge y = 4 \\
x > 1 \wedge y < 4 \wedge x + y = 1 \\
x > 1 \wedge y < 4 \wedge x + y > 1 \\
x = 0 \wedge y = -1 \\
x = 0 \wedge y < -1 \\
x < 0 \wedge y = 4 \\
x < 0 \wedge y < 4 \wedge 2x + y = -1 \\
x < 0 \wedge y < 4 \wedge 2x + y < -1
\end{array}
\right\}
\begin{array}{l}
\{a_1 b_1 F_2\}\ 1,2 \\
\{a_1 b_2 F_2\}\ 2,2 \\
\{a_2 b_1 F_4\}\ 3,4 \\
\{a_2 b_2 F_4\}\ 4,4 \\
\{a_1 b_3 F_2\}\ 5,2 \\
\{a_1 b_4 F_1\}\ 6,1 \\
\{a_1 b_4 F_2\}\ 6,2 \\
\{a_2 b_3 F_4\}\ 7,4 \\
\{a_2 b_4 F_3\}\ 8,3 \\
\{a_2 b_4 F_4\}\ 8,4 \\
\{a_3 b_4 F_5\} 10,5 \\
\{a_3 b_4 F_6\} 10,6 \\
\{a_4 b_3 F_7\} 11,8 \\
\{a_4 b_4 F_8\} 12,7 \\
\{a_4 b_4 F_7\} 12,8
\end{array}
$$

The second step yields the following minimal set of classes:

$$
\left\{
\begin{array}{c}
x = 1 \wedge y = 5 \\
x > 1 \wedge y > 5 \\
x = 1 \wedge y = 0 \\
x > 1 \wedge y < 4 \wedge x + y = 1 \\
x = 0 \wedge y = -1 \\
x = 0 \wedge y < -1 \\
x < 0 \wedge y = 4 \\
x < 0 \wedge y < 4 \wedge 2x + y = -1
\end{array}
\right\}
\begin{array}{l}
\{a_1 b_1 F_2\}\ 1,2 \\
\{a_2 b_2 F_4\}\ 4,4 \\
\{a_1 b_4 F_1\}\ 6,1 \\
\{a_2 b_4 F_3\}\ 8,3 \\
\{a_3 b_4 F_5\} 10,5 \\
\{a_3 b_4 F_6\} 10,6 \\
\{a_4 b_3 F_8\} 11,8 \\
\{a_4 b_4 F_7\} 12,7
\end{array}
$$

This latest systematic partitioning followed by our redundancy analysis phase is as complex as can reasonably be presented here. To be able to present our tests generation process for more complex expressions we will not from here on detail every phase as much as we did. Further, we will regroup dependent labels as they are generated when possible by denoting them by a unique label (this does not influence our results and is done for reasons of clarity).

5.4.4 Functions and Dependence

We consider $f(x) \vee g(x)$ where $f(x) = x > 0$ and $g(x) = x < 5$.

$$coarse\_true(f(x) \vee g(x)).part = \left\{ \begin{array}{l} x > 1 \wedge x \geq 5 \\ x = 1 \wedge x < 4 \\ x > 1 \wedge x = 4 \\ x > 1 \wedge x < 4 \\ x \leq 0 \wedge x < 4 \end{array} \right\} \begin{array}{l} \{x_1 x_2 F_2 F_4\}1 \\ \{x_1 x_2 F_1 F_6\}2 \\ \{x_1 x_2 F_2 F_5\}3 \\ \{x_1 x_2 F_2 F_6\}4 \\ \{x_1 x_2 F_3 F_6\}5 \end{array}$$

Regrouping labels we get:

$$\left\{ \begin{array}{l} x > 1 \wedge x \geq 5 \\ x = 1 \wedge x < 4 \\ x > 1 \wedge x = 4 \\ x > 1 \wedge x < 4 \\ x \leq 0 \wedge x < 4 \end{array} \right\} \begin{array}{l} \{a_1 F_2 F_4\}1 \\ \{a_1 F_1 F_6\}2 \\ \{a_1 F_2 F_5\}3 \\ \{a_1 F_2 F_6\}4 \\ \{a_1 F_3 F_6\}5 \end{array}$$

The class number 4 is detected as redundant, thus leaving the following simplified minimal set classes for test data generation:

$$\left\{ \begin{array}{l} x \geq 5 \\ x = 1 \\ x = 4 \\ x \leq 0 \end{array} \right\} \begin{array}{l} \{a_1 F_2 F_4\}1 \\ \{a_1 F_1 F_6\}2 \\ \{a_1 F_2 F_5\}3 \\ \{a_1 F_3 F_6\}5 \end{array}$$

5.4.5 Multiple Function Calls

Finally we consider: $f(x) \wedge f(y) \wedge f(z)$ with $f(X) = X > 5$

$$coarse\_true(X > 5).part = \left\{ \begin{array}{l} X = 6 \\ X > 6 \end{array} \right\} \begin{array}{l} \{F_1\} \\ \{F_2\} \end{array}$$

$$coarse\_true(f(x) \wedge f(y) \wedge f(z)).part =$$

$$\left\{\begin{array}{l} x = 6 \wedge y = 6 \wedge z = 6 \\ x = 6 \wedge y = 6 \wedge z > 6 \\ x = 6 \wedge y > 6 \wedge z = 6 \\ x = 6 \wedge y > 6 \wedge z > 6 \\ x > 6 \wedge y = 6 \wedge z = 6 \\ x > 6 \wedge y = 6 \wedge z > 6 \\ x > 6 \wedge y > 6 \wedge z = 6 \\ x > 6 \wedge y > 6 \wedge z > 6 \end{array}\right\} \begin{array}{c} \{x_1 y_1 z_1 F_1\} \\ \{x_1 y_1 z_1 F_1 F_2\} \\ \{x_1 y_1 z_1 F_1 F_2\} \\ \{x_1 y_1 z_1 F_1 F_2\} \\ \{x_1 y_1 z_1 F_1 F_2\} \\ \{x_1 y_1 z_1 F_1 F_2\} \\ \{x_1 y_1 z_1 F_1 F_2\} \\ \{x_1 y_1 z_1 F_2\} \end{array}$$

A minimal set of classes for the final partition above is:

$$\left\{ x = 6 \wedge y > 6 \wedge z > 6 \right\} \{x_1 y_1 z_1 F_1 F_2\}$$

5.4.6 Conclusion

The previous examples have illustrated the effects of context dependent combination. While many of them could have been dealt with by the direct approach, using graphs, as described in 5.2.2, the introduction of functions which may induce contradictions in classes renders it impracticable.

In the next chapter, recursion and looseness in specifications will be examined. Finally, we will systematically generate a test set for the Triangle Problem (as described in Chapter 3) using our technique and evaluate our efforts against North's, manually derived, test set [11] and Dick and Faivre's, automatically generated, test set [13].

Chapter 6

The Triangle Problem and Other Aspects

In this chapter we discuss the remaining aspects of generating test cases from VDM-SL specifications with a view to integrating them in our technique, namely: recursion and looseness.

Finally, test cases from the Triangle Problem will be generated using our technique and the results analysed.

6.1 Remaining Aspects

Before analysing the specification of the Triangle Problem using our test cases generation technique and evaluating its adequacy, we must discuss how we propose to deal with recursive functions as well as with non-deterministic behaviour (looseness) in specifications.

While recursion is considered in the work of Dick and Faivre [13], looseness is not mentioned.

6.1.1 Recursion

Recursivity is always a difficult feature to deal appropriately with in analysing techniques. For example, the symbolic execution of imperative programming language code is usually curtailed for recursive functions. The difficulty is in deciding how many times recursive functions should be symbolically unfolded.

Typically, recursive functions are unfolded a fixed number of times and the result analysed. This is indeed the approach taken in [13].

We will argue that for the purpose of identifying the sub-domains of a recursive function, no recursive calls need to be unfolded. However, as we will see this may not be satisfactory for the generation of adequate test cases.

We will illustrate our argument using the *sum* example of the Triangle Problem:

$sum\ : \mathbb{N}* \to \mathbb{N}$

$sum(seq) ==$

$\quad if\ \ seq = []$

$\qquad then\ 0$

$\quad else\ hd\ \ seq + sum(tl\ \ seq)$

Using our partitioning rules up to the recursive call to *sum* we obtain:

$$\left\{\begin{array}{l} s = [] \wedge r = 0 \\ \left\{\begin{array}{l} len\ \ s = 1 \\ len\ \ s > 1 \\ s* \end{array}\right\} \times \left\{\begin{array}{l} len\ \ s = 1 \wedge r = hd\ \ s + sum([]) \\ len\ \ s > 1 \wedge r = hd\ \ s + sum(tl\ \ s) \end{array}\right\} \end{array}\right\}$$

In general, if we apply our partitioning rules without unfolding recursive calls we will be able to generate a partition of the function which will contain the adequate sub-domains.

$$P(sum(s) = r) = \{s \in \mathbb{N}* \wedge r \in \mathbb{N}\} \times \left\{\begin{array}{l} s = [] \wedge r = 0 \\ len\ \ s = 1 \wedge r = hd\ \ s \\ len\ \ s > 1 \wedge r = hd\ \ s + sum(tl\ \ s) \end{array}\right\}$$

However, whenever recursive functions are not tested in isolation from the rest of the specification but only through its calls, we may encounter problems in generating adequate tests. The *sum* function above, does not raise any difficulties because any argument will always eventually cover the case when the sequence is empty. Hence this particular class will always be implicitly covered even if an argument cannot be the empty sequence in any of the calling contexts. However, a function of the style:

$sum \ : \mathbb{N}* \to \mathbb{N}$

$sum(seq) ==$

$\quad if \ \ seq = []$

$\quad\quad then \ 0$

$\quad elseif \ \ seq = [9,9,9]$

$\quad\quad then \ 999$

$\quad else \ hd \ \ seq + sum(tl \ \ seq)$

will create problems if the sequence $[9,9,9]$ cannot be passed as argument in any of the calling contexts. If, for example, the length of any of the sequences in all calling contexts must be 4 then we will have no means to generate a sequence of the type $[n,9,9,9]$, where n is a natural, which would exercise this particular aspect of this revised *sum* function. The partition for this revised *sum* function is:

$$P(r = sum(s)) = \{s \in \mathbb{N}* \wedge r \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{l} s = [] \wedge r = 0 \\ s = [9,9,9] \wedge r = 999 \\ len \ \ s = 1 \wedge r = hd \ \ s \\ len \ \ s > 1 \wedge r = hd \ \ s + sum(tl \ \ s) \end{array}\right\}$$

While in this particular instance, *sum* could be unfolded once to reveal extraneous classes which would allow the generation of appropriate test cases, we cannot in general decide how many times a recursive function should be unfolded to allow the generation of such equivalence classes.

So while we can generate an adequate partition for recursive functions, we cannot ensure, in some circumstances, the generation of test cases which will exercise every equivalence class generated.

We therefore propose that recursive function calls never be unfolded and accept that, sometimes, some classes from recursive functions which cannot be tested separately from the rest of the system under test will not be exercised by the test cases generated. In such circumstances, an analysis of the labels covered by the test cases generated—which would always reveal which of the sub-domains from those generated are not exercised during testing—could highlight the classes of recursive functions which we have been unable to cover. We would then have to decide if the non-covered classes can be exercised at all through calls in the specification and manually generate tests to exercise those non-covered classes. It could well be that some classes cannot be covered because of inconsistencies in the specification rather than because of the shortcomings of our approach in dealing with recursive functions.

6.1.2 Looseness

We have already seen, in chapter 4, how looseness in specifications may induce a very high level of complexity which may well be beyond what mechanical analysis can be envisaged to ever handle appropriately. We will nevertheless examine the effects of looseness in simple circumstances for our test generation technique.

As an illustration we propose the following example:

$f(x, y : \mathbb{N}) r : \mathbb{Z}$

post

  if $y \neq 0$

    then $r = x \ DIV \ y$

  else $r \leq 0$

Whenever y is equal to 0 the result of this function is non-deterministic in that it is only specified that r should be less or equal than 0.

Partially partitioning f without LPF behaviour for simplicity we obtain:

$$\left\{\begin{array}{l} \left\{\begin{array}{l} y = 0 \\ y > 0 \end{array}\right\} \times P(r = x \ DIV \ y) \\ \{y = 0\} \times \left\{\begin{array}{l} r = 0 \\ r < 0 \end{array}\right\} \end{array}\right\}$$

We are interested here in the two classes: $y = 0 \wedge r = 0$ and $y = 0 \wedge r < 0$ from which two test cases will be generated after sampling of the input variables of f. The two tests along with their oracle are given below.

$$x = 42, y = 0 \ r = 0$$
$$x = 67, y = 0 \ r < 0$$

There are two serious problems with these two tests.

Firstly, the oracle is wrong: if an implementation outputs $r = -3$ with $x = 42, y = 0$ or $r = 0$ with $x = 67, y = 0$ as inputs then, according to the specification, the implementation has behaved correctly in these instances. We should not have partitioned the expression $r \leq 0$ because it will have no effect on the sub-domains of the input variables of the function under analysis.

In general, to identify which expressions affect the input sub-domains of a particular function during partitioning is probably infeasible for complex expressions and impractical to perform routinely in simple circumstances.

Hence we accept that for loose specifications the oracle may well be biased towards an unspecified behaviour and is thus of no value. Whenever the specification has been identified by the human tester as non-deterministic (in complex specifications this is particularly difficult to infer), the test input could be fed back to the specification and, using symbolic execution on the specification, an expression specifying the expected output obtained.

The second problem with the two particular tests generated above, is that even if a correct oracle can be generated from the specification, i.e. to obtain the following:

$$x = 42, y = 0 \; r \leq 0$$
$$x = 67, y = 0 \; r \leq 0$$

then it becomes apparent that the two tests cover the same sub-domains and hence that one of them is redundant. This redundancy however could be identified using our proposal for redundancy analysis of the previous section.

This simple example has allowed us to show that even in elementary circumstances, loose specifications raise difficult problems for the automatic generation of test cases. Our proposal to deal with these problems relies on human intervention for the identification of loose specifications.

6.2 Triangle Problem

We now return to the Triangle Problem and illustrate the kind of consistency checks which must be performed automatically, or at least with mechanical assistance, for the generation of test cases using our approach.

We take into account the potential LPF behaviour of the specification to illustrate its importance and the difficulties it raises. We note that the specification is deterministic.

According to the VDM-SL standard and our understanding, is_*type(*)* is undefined but $f(*)$ (a function call) proceeds with the undefined value as argument (i.e. undefined is not returned automatically).

To be able to compare our test set with those produced by North [11] and by Dick and Faivre [13, 14] we use the following specification of *Classify* as given in Chapter 3:

$Classify\ : token \rightarrow Triangle\_type$

$Classify(s) ==$

$\quad if\ is\_seq\_nat(s) \wedge is\_triangle(s)$

$\quad\quad then\ variety(s)$

$\quad else\ INVALID$

We cannot give the test cases generation process in too many details here for lack of space. Also, to shorten the generation of test cases for the Triangle Problem, we will in places use our insight into our tests generation process to avoid deriving equivalent classes which we know will be found to be inconsistent if we were using our systematic approach.

It is important to note that this does not affect the final result.

6.2.1 Initial Checks

Using the *if* ... *then* ... *else* ... rule we obtain:

$$P(Classify(s)) = P(s \in token) \times P(R \in Triangle\_type) \times$$

$$\bigcup \left\{ \begin{array}{l} P(is\_seq\_nat(s) \wedge is\_triangle(s)) \times P(variety(s) = R) \\ \cup \left\{ \begin{array}{l} P(\neg is\_seq\_nat(s) \vee \neg is\_triangle(s)) \\ P((is\_seq\_nat(s) \wedge is\_triangle(s))*) \end{array} \right\} \times P(R = INVALID) \end{array} \right\}$$

By convention: $P(s \in token) = \{s \in token\}$

Further because the specification is not loose, we can choose:

$P(R \in Triangle\_type) = \{R \in Triangle\_type\}$

(generating the four sub-domains of this expression would only lengthen the derivation process without affecting the final result).

Using the *or* rule: $P(Classify(s)) = \{s \in token \wedge R \in Triangle\_type\} \times$

$$\bigcup \left\{ \begin{array}{l} P(is\_seq\_nat(s) \wedge is\_triangle(s)) \times P(variety(s) = R) \\ \bigcup \left\{ \begin{array}{l} P(\neg is\_seq\_nat(s)) \times P(is\_triangle(s)) \\ P(\neg is\_seq\_nat(s)) \times P(\neg is\_triangle(s)) \\ P(is\_seq\_nat(s)) \times P(\neg is\_triangle(s)) \\ P(\neg is\_seq\_nat(s)) \times P((\neg is\_triangle(s))*) \\ P((\neg is\_seq\_nat(s))*) \times P(\neg is\_triangle(s)) \\ P(is\_seq\_nat(s)) \times P(is\_triangle(s)*) \\ P(is\_seq\_nat(s)*) \times P(is\_triangle(s)) \\ P(is\_seq\_nat(s)*) \times P(is\_triangle(s)*) \end{array} \right\} \times P(R = INVALID) \end{array} \right\}$$

According to VDM-SL semantics $(\neg A)* = A*$, thus in the above: $P((\neg is\_seq\_nat(s))*)$ is replaced by $P(is\_seq\_nat(s)*)$ and $P((\neg is\_triangle(s))*)$ is replaced by $P(is\_triangle(s)*)$.

We now use the definition of $is\_seq\_nat(s) = is\_seq(s) \wedge \forall x \in elems(s) \cdot is\_nat(x)$ and the definition of $is\_triangle(s) = len(s) = 3 \wedge \forall i \in elems(s) \cdot 2 * i < sum(s)$. We will use the following abbreviation:

$$\begin{array}{l} A \; for \;\; is\_seq(s) \\ B \; for \; \forall x \in elems(s) \cdot is\_nat(x) \\ C \; for \;\; len(s) = 3 \\ D \; for \; \forall i \in elems(s) \cdot 2 * i < sum(s) \end{array}$$

We consider $P(variety(s) = R)$. *variety* is a function in the specification but is deemed not to be implemented as such in the system under test, thus using a simplified *cases* rule (which applies in this context) to shorten the derivation:

$P(variety(s) = R) = \{s \in Triangle \wedge R \in Triangle\_type\} \times$

$$\bigcup \left\{ \begin{array}{l} P(card(elems(s)) = 1) \times P(R = EQUILATERAL) \\ P(card(elems(s)) = 2) \times P(card(elems(s)) \neq 1) \times P(R = ISOSCELES) \\ P(card(elems(s)) = 3) \times P(card(elems(s)) \neq 1) \times P(card(elems(s) \neq 2) \\ \quad \times P(R = SCALENE) \end{array} \right\}$$

which we can simplify to:

$$P(variety(s) = R) = \{s \in Triangle \wedge R \in Triangle\_type\} \times$$

$$\bigcup \begin{Bmatrix} P(card(elems(s)) = 1) \times P(R = EQUILATERAL) \\ P(card(elems(s)) = 2) \times P(R = ISOSCELES) \\ P(card(elems(s)) = 3) \times P(R = SCALENE) \end{Bmatrix}$$

Hence,

$$P(Classify(s)) = \{s \in token \wedge R \in Triangle\_type\} \times$$

$$\bigcup \left\{ \begin{array}{l} P(A) \times P(B) \times P(C) \times P(D) \times \bigcup \begin{Bmatrix} P(card(elems(s)) = 1) \times \\ \quad P(R = EQUILATERAL) \\ P(card(elems(s)) = 2) \times \\ \quad P(R = ISOSCELES) \\ P(card(elems(s)) = 3) \times \\ \quad P(R = SCALENE) \end{Bmatrix} \\ \bigcup \begin{Bmatrix} P(\neg A \vee \neg B) \times P(C) \times P(D) \\ P(\neg A \vee \neg B) \times P(\neg C \vee \neg D) \\ P(A) \times P(B) \times P(\neg C \vee \neg D) \\ P(\neg A \vee \neg B) \times P((C \wedge D)*) \\ P((A \wedge B)*) \times P(\neg C \vee \neg D) \\ P(A) \times P(B) \times P((C \wedge D)*) \\ P((A \wedge B)*) \times P(C) \times P(D) \\ P((A \wedge B)*) \times P((C \wedge D)*) \end{Bmatrix} \times P(R = INVALID) \end{array} \right\}$$

$$\text{Further}: \ P(\neg A \vee \neg B) = \bigcup \begin{Bmatrix} P(\neg A) \times P(B) \\ P(\neg A) \times P(\neg B) \\ P(A) \times P(\neg B) \\ P(\neg A) \times P(B*) \\ P(A*) \times P(\neg B) \end{Bmatrix}$$

$$\text{and} \ P(\neg C \vee \neg D) = \bigcup \begin{Bmatrix} P(\neg C) \times P(D) \\ P(\neg C) \times P(\neg D) \\ P(C) \times P(\neg D) \\ P(\neg C) \times P(D*) \\ P(C*) \times P(\neg D) \end{Bmatrix}$$

and $P((C \wedge D)*) = \bigcup \left\{ \begin{array}{l} P(C) \times P(D*) \\ P(C*) \times P(D) \\ P(C*) \times P(D*) \end{array} \right\}$

and $P((A \wedge B)*) = \bigcup \left\{ \begin{array}{l} P(A) \times P(B*) \\ P(A*) \times P(B) \\ P(A*) \times P(B*) \end{array} \right\}$

Hence after simplifications at the predicate level:

$P(Classify(s)) = \{s \in token \wedge R \in Triangle\_type\} \times$

$$\bigcup \left\{ \begin{array}{l} P(A) \times P(B) \times P(C) \times P(D) \times \bigcup \left\{ \begin{array}{l} P(card(elems(s)) = 1) \times \\ \quad P(R = EQUILATERAL) \\ P(card(elems(s)) = 2) \times \\ \quad P(R = ISOSCELES) \\ P(card(elems(s)) = 3) \times \\ \quad P(R = SCALENE) \end{array} \right\} \\ \bigcup \left\{ \begin{array}{l} P(A) \times P(\neg B) \times P(\neg C) \times P(D*) \\ P(A) \times P(B) \times \bigcup \left\{ \begin{array}{l} P(\neg C) \times P(D) \\ P(\neg C) \times P(\neg D) \\ P(C) \times P(\neg D) \end{array} \right\} \\ \bigcup \left\{ \begin{array}{l} P(A) \times P(\neg B) \times P(C) \times P(D*) \\ P(\neg A) \times P(B*) \times P(C*) \times P(D*) \end{array} \right\} \\ P(A) \times P(B*) \times P(\neg C) \times P(D*) \\ \bigcup \left\{ \begin{array}{l} P(A) \times P(B*) \times P(C) \times P(D*) \\ P(A*) \times P(B*) \times P(C*) \times P(D*) \end{array} \right\} \end{array} \right\} \times P(R = INVALID) \end{array} \right\}$$

We need to generate sub-domains for the following expressions: A, $\neg A$, $A*$, B, $\neg B$, $B*$, C, $\neg C$, $C*$, D, $\neg D$ and $D*$.

6.2.2 A is $is\_seq(s)$

$P(A) = P(is\_seq(s)) = \left\{ \begin{array}{c} s = [] \\ is\_seq(s) \wedge s \neq [] \end{array} \right\} \begin{array}{l} \{s_1\} \\ \{s_2\} \end{array}$

$P(\neg A) = P(\neg is\_seq(s)) = \left\{ \neg is\_seq(s) \right\} \{s_3\}$

$P(A*) = P(is\_seq(s)*) = \left\{ s* \right\} \{s_4\}$

6.2.3 B is $\forall x \in elems(s) \cdot is\_nat(x)$

We assume that $is\_nat(x)$ has been implemented as a function in the system under test.

We will need: $P(is\_nat(x)) = \left\{\begin{array}{c} x = 0 \\ is\_nat(x) \wedge x > 0 \wedge x < M \\ x = M \end{array}\right\} \begin{array}{l} \{F_1\} \\ \{F_2\} \\ \{F_3\} \end{array}$

Because $is\_nat(x)$ is a function, the F_is labels will remain independent from all other labels throughout the rest of the specification.

Hence, $P(\forall x \in elems(s) \cdot is\_nat(x)) =$

$$\left\{\begin{array}{l} s = [] \\ \forall x \in elems(s) \cdot x = 0 \\ \forall x \in elems(s) \cdot is\_nat(x) \wedge x > 0 \wedge x < M \\ \forall x \in elems(s) \cdot x = M \\ (\forall x \in elems(s) \cdot is\_nat(x) \wedge x \neq M) \wedge (\exists x \in elems \cdot x = 0) \wedge \\ \quad (\exists \in elems(s) \cdot x > 0 \wedge x < M) \\ (\forall x \in elems(s) \cdot is\_nat(x) \wedge \neg(x > 0 \wedge x < M)) \wedge \\ \quad (\exists x \in elems \cdot x = 0) \wedge (\exists \in elems(s) \cdot x = M) \\ (\forall x \in elems(s) \cdot is\_nat(x) \wedge x \neq 0) \wedge (\exists x \in elems \cdot x > 0 \wedge x < M) \wedge \\ \quad (\exists \in elems(s) \cdot x = M) \\ (\forall x \in elems(s) \cdot is\_nat(x)) \wedge (\exists x \in elems \cdot x = 0) \wedge \\ \quad (\exists x \in elems \cdot x > 0 \wedge x < M) \wedge (\exists x \in elems \cdot x = M) \end{array}\right\}$$

We will represent these sub-domains as follows (internally, in an implementation, the sub-domains can be represented in any fashion):

$P(B) = P(\forall x \in elems(s) \cdot is\_nat(x)) =$

$$\left\{\begin{array}{l} elems(s) = \{\} \\ elems(s) = \{0\} \\ elems(s) = \{x_1, \ldots, x_n\} \\ elems(s) = \{M\} \\ elems(s) = \{0, x_1, \ldots, x_n\} \\ elems(s) = \{0, M\} \\ elems(s) = \{x_1, \ldots, x_n, M\} \\ elems(s) = \{0, x_1, \ldots, x_n, M\} \end{array}\right\} \begin{array}{c} \{s_5\} \\ \{s_{31}F_1\} \\ \{s_{31}F_2\} \\ \{s_{31}F_3\} \\ \{s_{31}F_1F_2\} \\ \{s_{31}F_1F_3\} \\ \{s_{31}F_2F_3\} \\ \{s_{31}F_1F_2F_3\} \end{array}$$

where in each sub-domain $n \geq 1$ and $\forall j \in \{1, \ldots n\} \cdot x_j > 0 \wedge x_j < M$

We now consider $P(\neg B)$. We will need:

$$P(\neg is\_nat(x)) = \left\{ \begin{array}{c} is\_int(x) \wedge x < 0 \\ is\_real(x) \\ \neg is\_nat(x) \wedge \neg is\_int(x) \wedge \neg is\_real(x) \end{array} \right\} \begin{array}{l} \{F_4\} \\ \{F_5\} \\ \{F_6\} \end{array}$$

Hence, using our modified partitioning expression for existentially quantified expressions with function call, as discussed in the previous chapter, we get: $P(\neg B) = P(\exists x \in elems(s) \cdot \neg is\_nat(x)) =$

$$\left\{ \begin{array}{l} elems(s) = \{i_1, \ldots, i_n\} \\ elems(s) = \{r_1, \ldots, r_m\} \\ elems(s) = \{t_1, \ldots, t_p\} \\ elems(s) = \{i_1, \ldots, i_n, r_1, \ldots, r_m\} \\ elems(s) = \{i_1, \ldots, i_n, t_1, \ldots, t_p\} \\ elems(s) = \{r_1, \ldots, r_m, t_1, \ldots, t_p\} \\ elems(s) = \{i_1, \ldots, i_n, r_1, \ldots, r_m, t_1, \ldots, t_p\} \\ elems(s) = \{i_1, \ldots, i_n\} \cup S \\ elems(s) = \{r_1, \ldots, r_m\} \cup S \\ elems(s) = \{t_1, \ldots, t_p\} \cup S \\ elems(s) = \{i_1, \ldots, i_n, r_1, \ldots, r_m\} \cup S \\ elems(s) = \{i_1, \ldots, i_n, t_1, \ldots, t_p\} \cup S \\ elems(s) = \{r_1, \ldots, r_m, t_1, \ldots, t_p\} \cup S \\ elems(s) = \{i_1, \ldots, i_n, r_1, \ldots, r_m, t_1, \ldots, t_p\} \cup S \end{array} \right\} \begin{array}{c} \{s_{32}F_4\} \\ \{s_{32}F_5\} \\ \{s_{32}F_6\} \\ \{s_{32}F_4F_5\} \\ \{s_{32}F_4F_6\} \\ \{s_{32}F_5F_6\} \\ \{s_{32}F_4F_5F_6\} \\ \{s_{33}F_4\} \\ \{s_{33}F_5\} \\ \{s_{33}F_6\} \\ \{s_{33}F_4F_5\} \\ \{s_{33}F_4F_6\} \\ \{s_{33}F_5F_6\} \\ \{s_{33}F_4F_5F_6\} \end{array}$$

where in each sub-domain $n \geq 1$, $m \geq 1$ and $p \geq 1$. Further:

$\forall j \in \{1, \ldots n\} \cdot is\_int(i_j) \wedge i_j < 0$

$\forall j \in \{1, \ldots m\} \cdot is\_real(r_j)$

$\forall j \in \{1, \ldots p\} \cdot \neg is\_int(t_j) \wedge \neg is\_real(t_j) \wedge \neg is\_nat(t_j)$

$\forall x \in S \cdot x* \vee is\_nat(x)$

We now consider $P(B*) = P((\forall x \in elems(s) \cdot is\_nat(x))*)$, we will need:

$$P(is\_nat(x)*) = \left\{ x* \right\} \{F_7\}$$

Thus, $P(B*) = P((\forall x \in elems(s) \cdot is\_nat(x))*) =$

$$\left\{\begin{array}{l} s* \\ \neg is\_seq(s) \\ elems(s) = \{n_1, \ldots, n_n, *\} \end{array}\right\} \begin{array}{c} \{s_6\} \\ \{s_7\} \\ \{s_{34}F_7\} \end{array}$$

6.2.4 C is $len(s) = 3$

This is straightforward:

$$P(C) = P(len(s) = 3) = \left\{ len(s) = 3 \right\} \{s_8\}$$

$$P(\neg C) = P(len(s) \neq 3) = \left\{\begin{array}{l} len(s) < 2 \\ len(s) = 2 \\ len(s) = 4 \\ len(s) > 4 \end{array}\right\} \begin{array}{c} \{s_9\} \\ \{s_{10}\} \\ \{s_{11}\} \\ \{s_{12}\} \end{array}$$

$$P(C*) = P((len(s) = 3)*) = \left\{\begin{array}{c} s* \\ \neg is\_seq(s) \end{array}\right\} \begin{array}{c} \{s_{13}\} \\ \{s_{14}\} \end{array}$$

6.2.5 D is $\forall i \in elems(s) \cdot 2 * i < sum(s)$

We will need $P(2 * i < sum(s))$. This expression is dependent on s.

A first partitioning step yields:

$$P(2 * i < sum(s)) = \left\{\begin{array}{l} 2 * i = r - 1 \\ 2 * i < r - 1 \end{array}\right\} \times P(r = sum(s))$$

We recall that sum is specified as follows:

$sum \; : \mathbb{N}* \to \mathbb{N}$

$sum(seq) ==$

$\quad if \;\; seq = []$

$\qquad then \; 0$

$\quad else \; hd \;\; seq + sum(tl \;\; seq)$

Hence $P(r = sum(s)) = \{s \in \mathbb{N}* \wedge r \in \mathbb{N}\} \times$

$$\bigcup \left\{\begin{array}{l} P(s = []) \times P(r = 0) \\ \bigcup \left\{\begin{array}{l} P(s \neq []) \\ P((s = [])*) \end{array}\right\} \times P(r = hd \; s + sum(tl \;\; s)) \end{array}\right\}$$

(as sum is not a function in the implementation, we do not test s nor r for type membership. This could be necessary however for the testing of the specification proper)

With :

$P(s = []) = \{s = []\}$

$P(r = 0) = \{r = 0\}$

$$P(s \neq []) = \begin{Bmatrix} len(s) = 1 \\ len(s) > 1 \end{Bmatrix}$$

$P((s = [])*) = \{s*\}$

$P(r = hd\ s + sum(tl\ s))$ is transformed into:

$P(r = r_1 + sum(r_2) \times P(r_1 = hd\ s) \times P(r_2 = (tl\ s))$ (the recursive call to sum is not re-partitioned). We have to ensure that potential LPF behaviour is preserved so we use:

$$P(r_1 = hd\ s) = \{r_1 = hd\ s\} \times \begin{Bmatrix} len\ s = 1 \\ len\ s > 1 \end{Bmatrix}$$

$$P(r_2 = tl\ s) = \{r_2 = tl\ s\} \times \begin{Bmatrix} len\ s = 1 \\ len\ s > 1 \end{Bmatrix}$$

to obtain:

$P(r = hd\ s + sum(tl\ s)) =$

$$\begin{Bmatrix} len\ s = 1 \wedge r = hd\ s + sum([]) \\ len\ s > 1 \wedge r = hd\ s + sum(tl\ s) \end{Bmatrix}$$

Thus: $P(r = sum(s)) = \{s \in \mathbb{N}* \wedge r \in \mathbb{N}\} \times$

$$\bigcup \left\{ \begin{array}{l} s = [] \wedge r = 0 \\ \begin{Bmatrix} len\ s = 1 \\ len\ s > 1 \\ s* \end{Bmatrix} \times \begin{Bmatrix} len\ s = 1 \wedge r = hd\ s + sum([]) \\ len\ s > 1 \wedge r = hd\ s + sum(tl\ s) \end{Bmatrix} \end{array} \right\}$$

Developing the partitioning expression we get:

$P(exp(s) = sum(s)) = \{s \in \mathbb{N}* \wedge r \in \mathbb{N}\} \times$

$$\begin{Bmatrix} s = [] \wedge r = 0 \\ len\ s = 1 \wedge r = hd\ s \\ len\ s > 1 \wedge r = hd\ s + sum(tl\ s) \end{Bmatrix}$$

$sum([])$ can be simplified by executing it since the argument is fully known.

Hence finally:

$P(2 * i < sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l} s = [] \wedge 2 * i = -1 \\ len\ s = 1 \wedge 2 * i = hd\ s - 1 \\ len\ s > 1 \wedge 2 * i = hd\ s + sum(tl\ s) - 1 \\ s = [] \wedge 2 * i < -1 \\ len\ s = 1 \wedge 2 * i < hd\ s - 1 \\ len\ s > 1 \wedge 2 * i < hd\ s + sum(tl\ s) - 1 \end{array}\right\}$$

We will not list all the potential combinations arising from:

$P(D) = P(\forall i \in elems(s) \cdot 2 * i < sum(s))$; instead we will note that some inconsistencies will arise: s cannot be the empty sequence in the partitioning proper, the sub-domains of $2 * i < sum(s)$ where the length of s is specified to be different (e.g. $len\ s = 1$ and $len\ s > 1$) cannot be combined. These contradictions could have been detected earlier.

Thus:

$P(D) = P(\forall i \in elems(s) \cdot 2 * i < sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l} s = [] \\ len\ s = 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s - 1) \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \\ len\ s = 1 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s - 1) \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ len\ s = 1 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s - 1)\ \wedge (\exists i \in elems(s) \cdot 2 * i < hd\ s - 1) \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \end{array}\right\}$$

which we can simplify to:

$P(D) = P(\forall i \in elems(s) \cdot 2 * i < sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{ \begin{array}{l} s = [] \\ s = [X] \wedge 2 * X = X - 1 \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \\ s = [X] \wedge 2 * X < X - 1 \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ s = [X] \wedge 2 * X < sum([X]) \wedge 2 * X = X - 1 \wedge 2 * X < X - 1 \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \end{array} \right\}$$

After consistency checking we obtain:

$P(D) = P(\forall i \in elems(s) \cdot 2 * i < sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{ \begin{array}{l} s = [] \\ s = [1, 1, 1] \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \end{array} \right\} \begin{array}{l} \{s_{15}\} \\ \{s_{16}\} \\ \{s_{17}\} \\ \{s_{18}\} \\ \\ \\ \end{array}$$

We now need to consider $P(\neg D) = P(\exists i \in elems(s) \cdot 2 * i \geq sum(s))$. We re-use $P(exp(s) = sum(s))$ derived earlier to obtain:

$P(2 * i \geq sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{ \begin{array}{l} s = [] \wedge 2 * i = 0 \\ len\ s = 1 \wedge 2 * i = hd\ s \\ len\ s > 1 \wedge 2 * i = hd\ s + sum(tl\ s) \\ s = [] \wedge 2 * i > 0 \\ len\ s = 1 \wedge 2 * i > hd\ s \\ len\ s > 1 \wedge 2 * i > hd\ s + sum(tl\ s) \end{array} \right\}$$

We use the fact: $\exists i \in elems(s) \cdot (2 * i \geq sum(s))*$ is equivalent to $\exists i \in elems(s) \cdot i*$ and after some consistency checking we obtain:

$$P(\neg D) = P(\exists i \in elems(s) \cdot 2 * i \geq sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{l}
len\ s = 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s) \\
len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\forall i \in elems(s) \cdot 2 * i > hd\ s) \\
len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot 2 * i = hd\ s) \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot 2 * i > hd\ s) \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot 2 * i = hd\ s) \\
\quad \wedge (\exists i \in elems(s) \cdot 2 * i > hd\ s) \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot 2 * i = hd\ s) \\
\quad \wedge (\exists i \in elems(s) \cdot i*) \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot i*) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot 2 * i > hd\ s) \\
\quad \wedge (\exists i \in elems(s) \cdot i*) \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot i*) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\
len\ s = 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge (\exists i \in elems(s) \cdot 2 * i = hd\ s) \\
\quad \wedge (\exists i \in elems(s) \cdot 2 * i > hd\ s) \wedge (\exists i \in elems(s) \cdot i*) \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge (\exists i \in elems(s) \cdot i*)
\end{array}\right\}$$

(One has to be meticulous when taking into account potential LPF behaviour of the specification. Here the existence of an undefined value in the sequence of naturals renders *sum* undefined not because the type of the argument should be a sequence of naturals (as the undefined value is part of the natural type) but because the addition operator propagates undefinedness. Thus, if there is an undefined value in s the entire expression $\neg D = \exists i \in elems(s) \cdot 2 * i \geq sum(s)$ becomes undefined (because of sum) which renders the sub-domains unsatisfiable).

Simplifying further:

$P(\neg D) = P(\exists i \in elems(s) \cdot 2 * i \geq sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l}
s = [0] \\
len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\
s = [X] \wedge X > 0 \\
len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\
\perp \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\
\perp \\
len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\
\perp \\
\perp \\
\perp \\
\perp \\
\perp \\
\perp \\
\perp \\
\perp
\end{array}\right\}$$

Hence we will use:

$$P(\neg D) = P(\exists i \in elems(s) \cdot 2 * i \geq sum(s)) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{l} s = [0] \\ len\ s > 1 \wedge (\forall i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\ s = [X] \wedge X > 0 \\ len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\ len\ s > 1 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \end{array}\right\} \begin{array}{l} \{s_{19}\} \\ \{s_{20}\} \\ \{s_{21}\} \\ \{s_{22}\} \\ \\ \{s_{23}\} \\ \\ \end{array}$$

We now consider $P(D*)$ which is straightforward:

$$P(D*) = P((\forall i \in elems(s) \cdot 2 * i < sum(s))*) =$$

$$\left\{\begin{array}{l} s* \\ \neg is\_seq(s) \\ (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \end{array}\right\} \begin{array}{l} \{s_{24}\} \\ \{s_{25}\} \\ \{s_{26}\} \end{array}$$

6.2.6 Pursuing $P(Classify(s))$

We recall:

$$P(Classify(s)) = \{s \in token \wedge R \in Triangle\_type\} \times$$

$$\bigcup \left\{\begin{array}{l} P(A) \times P(B) \times P(C) \times P(D) \times \bigcup \left\{\begin{array}{l} P(card(elems(s)) = 1) \times \\ \quad P(R = EQUILATERAL) \\ P(card(elems(s)) = 2) \times \\ \quad P(R = ISOSCELES) \\ P(card(elems(s)) = 3) \times \\ \quad P(R = SCALENE) \end{array}\right\} \\ \bigcup \left\{\begin{array}{l} P(A) \times P(\neg B) \times P(\neg C) \times P(D*) \\ P(A) \times P(B) \times \bigcup \left\{\begin{array}{l} P(\neg C) \times P(D) \\ P(\neg C) \times P(\neg D) \\ P(C) \times P(\neg D) \end{array}\right\} \\ \bigcup \left\{\begin{array}{l} P(A) \times P(\neg B) \wedge P(C) \times P(D*) \\ P(\neg A) \times P(B*) \wedge P(C*) \times P(D*) \end{array}\right\} \\ P(A) \times P(B*) \times P(\neg C) \times P(D*) \\ \bigcup \left\{\begin{array}{l} P(A) \times P(B*) \times P(C) \times P(D*) \\ P(A*) \times P(B*) \times P(C*) \times P(D*) \end{array}\right\} \end{array}\right\} \times P(R = INVALID) \end{array}\right\}$$

We now perform the combinations:

$P(C) \times P(D) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l} s = [1,1,1] \\ len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \end{array}\right\} \begin{array}{l} \{s_8 s_{16}\} \\ \{s_8 s_{17}\} \\ \{s_8 s_{18}\} \\ \\ \\ \end{array}$$

$P(C) \times P(\neg D) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l} s = [0,0,0] \\ len(s) = 3 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\ len(s) = 3 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \end{array}\right\} \begin{array}{l} \{s_8 s_{20}\} \\ \{s_8 s_{22}\} \\ \\ \{s_8 s_{23}\} \\ \\ \end{array}$$

$P(C) \times P(D*) =$

$$\left\{len(s) = 3 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*)\right\} \{s_8 s_{26}\}$$

$P(\neg C) \times P(D) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l} s = [] \\ len(s) = 4 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ len(s) = 4 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ len(s) > 4 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ len(s) > 4 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \end{array}\right\} \begin{array}{l} \{s_9 s_{15}\} \\ \{s_{11} s_{17}\} \\ \{s_{11} s_{18}\} \\ \\ \\ \{s_{12} s_{17}\} \\ \{s_{12} s_{18}\} \\ \\ \\ \end{array}$$

$P(\neg C) \times P(\neg D) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$

$$\left\{\begin{array}{l} s = [0] \\ s = [X] \wedge X > 0 \\ s = [X, X] \\ len(s) = 2 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\ s = [0, 0, 0, 0] \\ len(s) = 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\ len(s) = 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \\ len(s) > 4 \wedge elems(s) = \{0\} \\ len(s) > 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \\ len(s) > 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \end{array}\right\} \begin{array}{l} \{s_9 s_{19}\} \\ \{s_9 s_{21}\} \\ \{s_{10} s_{20}\} \\ \{s_{10} s_{23}\} \\ \\ \{s_{11} s_{20}\} \\ \{s_{11} s_{22}\} \\ \\ \{s_{11} s_{23}\} \\ \\ \{s_{12} s_{20}\} \\ \{s_{12} s_{22}\} \\ \\ \{s_{12} s_{23}\} \\ \\ \end{array}$$

$P(\neg C) \times P(D*) =$

$$\left\{\begin{array}{l} len(s) < 2 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \\ len(s) = 2 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \\ len(s) = 4 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \\ len(s) > 4 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \end{array}\right\} \begin{array}{l} \{s_9 s_{26}\} \\ \{s_{10} s_{26}\} \\ \{s_{11} s_{26}\} \\ \{s_{12} s_{26}\} \end{array}$$

$$P(C*) \times P(D*) = \left\{\begin{array}{l} s* \\ \neg is\_seq(s) \end{array}\right\} \begin{array}{l} \{s_{13} s_{24}\} \\ \{s_{14} s_{25}\} \end{array}$$

We also generate the sub-domains for the remaining partition expressions:

$P(card(elems(s)) = 1) = \{card(elems(s)) = 1\}\ \{s_{28}\}$

$P(card(elems(s)) = 2) = \{card(elems(s)) = 2\}\ \{s_{29}\}$

$P(card(elems(s)) = 3) = \{card(elems(s)) = 3\}\ \{s_{30}\}$

$P(R = EQUILATERAL) = \{R = EQUILATERAL\}\ \{r_1\}$

$P(R = ISOSCELES) = \{R = ISOSCELES\}\ \{r_2\}$

$P(R = SCALENE) = \{R = SCALENE\}\ \{r_3\}$

$P(R = INVALID) = \{R = INVALID\}\ \{r_4\}$

$$\text{Thus:}P(A) \times P(C) \times P(D) \times$$

$$\bigcup \left\{\begin{array}{l} P(card(elems(s)) = 1) \times \\ \quad P(R = EQUILATERAL) \\ P(card(elems(s)) = 2) \times \\ \quad P(R = ISOSCELES) \\ P(card(elems(s)) = 3) \times \\ \quad P(R = SCALENE) \end{array}\right\} = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{ll} s = [1,1,1] \wedge R = EQUILATERAL & \{s_2 s_8 s_{16} s_{28} r_1\} \\ s = [X,X,X] \wedge X > 1 \wedge R = EQUILATERAL & \{s_2 s_8 s_{17} s_{28} r_1\} \\ len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge & \{s_2 s_8 s_{17} s_{29} r_2\} \\ \quad card(elems(s)) = 2 \wedge R = ISOSCELES & \\ len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_8 s_{18} s_{29} r_2\} \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge & \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge & \\ \quad card(elems(s)) = 2 \wedge R = ISOSCELES & \\ len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge & \{s_2 s_8 s_{17} s_{30} r_3\} \\ \quad card(elems(s)) = 3 \wedge R = SCALENE & \\ len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_8 s_{18} s_{30} r_3\} \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge & \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge & \\ \quad card(elems(s)) = 3 \wedge R = SCALENE & \end{array}\right\}$$

$$P(A) \times P(\neg C) \times P(D*) \times P(R = INVALID) =$$

$$\left\{\begin{array}{ll} len(s) < 2 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \wedge R = INVALID & \{s_2 s_9 s_{26} r_4\} \\ len(s) = 2 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \wedge R = INVALID & \{s_2 s_{10} s_{26} r_4\} \\ len(s) = 4 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \wedge R = INVALID & \{s_2 s_{11} s_{26} r_4\} \\ len(s) > 4 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \wedge R = INVALID & \{s_2 s_{12} s_{26} r_4\} \end{array}\right\}$$

$$P(A) \times P(\neg C) \times P(D) \times P(R = INVALID) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{l} s = [] \wedge R = INVALID \\ len(s) = 4 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\ \quad R = INVALID \\ len(s) = 4 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\ \quad \times R = INVALID \\ len(s) > 4 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\ \quad R = INVALID \\ len(s) > 4 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\ \quad R = INVALID \end{array}\right\} \begin{array}{l} \{s_1 s_9 s_{15} r_4\} \\ \{s_2 s_{11} s_{17} r_4\} \\ \\ \{s_2 s_{11} s_{18} r_4\} \\ \\ \\ \\ \{s_2 s_{12} s_{17} r_4\} \\ \\ \{s_2 s_{12} s_{18} r_4\} \\ \\ \\ \\ \end{array}$$

$$P(A) \times P(C) \times P(\neg D) \times P(R = INVALID) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{l} s = [0,0,0] \wedge R = INVALID \\ len(s) = 3 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \\ len(s) = 3 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \end{array}\right\} \begin{array}{l} \{s_2 s_8 s_{20} r_4\} \\ \{s_2 s_8 s_{22} r_4\} \\ \\ \\ \{s_2 s_8 s_{23} r_4\} \\ \\ \\ \end{array}$$

$$P(A) \times P(C) \times P(D*) \times P(R = INVALID) =$$

$$\left\{ len(s) = 3 \wedge (\exists x \in elems(s) \cdot \neg is\_nat(x) \vee x*) \wedge R = INVALID \right\} \{s_2 s_8 s_{26} r_4\}$$

$$P(\neg A) \times P(C*) \times P(D*) \times P(R = INVALID) =$$

$$\left\{ \neg is\_seq(s) \wedge R = INVALID \right\} \{s_3 s_{14} s_{25} r_4\}$$

$$P(A*) \times P(C*) \times P(D*) \times P(R = INVALID) =$$

$$\left\{ s * \wedge R = INVALID \right\} \{s_4 s_{13} s_{24} r_4\}$$

$$P(A) \times P(\neg C) \times P(\neg D) \times P(R = INVALID) = \{s \in \mathbb{N}* \wedge sum(s) \in \mathbb{N}\} \times$$

$$\left\{\begin{array}{l} s = [0] \wedge R = INVALID \\ s = [X] \wedge X > 0 \wedge R = INVALID \\ s = [X, X] \wedge R = INVALID \\ len(s) = 2 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \\ s = [0, 0, 0, 0] \wedge R = INVALID \\ len(s) = 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \\ len(s) = 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \\ len(s) > 4 \wedge elems(s) = \{0\} \wedge R = INVALID \\ len(s) > 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \\ len(s) > 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\ \quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge \\ \quad R = INVALID \end{array}\right\} \begin{array}{l} \{s_2 s_9 s_{19} r_4\} \\ \{s_2 s_9 s_{21} r_4\} \\ \{s_2 s_{10} s_{20} r_4\} \\ \{s_2 s_{10} s_{23} r_4\} \\ \\ \\ \{s_2 s_{11} s_{20} r_4\} \\ \{s_2 s_{11} s_{22} r_4\} \\ \\ \\ \{s_2 s_{11} s_{23} r_4\} \\ \\ \\ \{s_2 s_{12} s_{20} r_4\} \\ \{s_2 s_{12} s_{22} r_4\} \\ \\ \\ \{s_2 s_{12} s_{23} r_4\} \\ \\ \\ \end{array}$$

We now combine the simple sub-domains of $P(B)$, $P(\neg B)$ and $P(B*)$: s_5, s_6 and s_7) to obtain the following set of equivalence classes:

$$\left\{\begin{array}{l} s = [] \wedge R = INVALID \\ \neg is\_seq(s) \wedge R = INVALID \\ s * \wedge R = INVALID \end{array}\right\} \begin{array}{l} \{s_1 s_5 s_9 s_{15} r_4\} \\ \{s_3 s_7 s_{14} s_{25} r_4\} \\ \{s_4 s_6 s_{13} s_{24} r_4\} \end{array}$$

We will detect redundant classes as we generate the minimal set of classes. Out of the sub-domains arising from $P(B)$, we only need to find one consistent combination with the sub-domain of label $\{F_1 F_2 F_3\}$ to consider $P(B)$ as covered. One such combination is:

$$\begin{array}{l} len(s) = 4 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\ \quad R = INVALID \wedge elem(s) = \{0, x_1, \ldots, x_n, M\} \end{array}$$

which has for label: $\{s_2 s_{11} s_{17} s_{31} r_4 F_1 F_2 F_3\}$

Thus the remaining combinations can be performed with any of the sub-domains of $P(B)$

We obtain: $\{elems(s) = \{x_1, \ldots, x_n\}\} \times$

$$\left\{\begin{array}{l}
s = [1,1,1] \wedge R = EQUILATERAL \\
s = [X,X,X] \wedge X > 1 \wedge X < M \wedge R = EQUILATERAL \\
len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\
\quad card(elems(s)) = 2 \wedge R = ISOSCELES \\
len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\
\quad card(elems(s)) = 2 \wedge R = ISOSCELES \\
len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\
\quad card(elems(s)) = 3 \wedge R = SCALENE \\
len(s) = 3 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\
\quad card(elems(s)) = 3 \wedge R = SCALENE \\
len(s) = 4 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge \\
\quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \\
\quad \times R = INVALID \\
len(s) > 4 \wedge (\forall i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge \\
\quad R = INVALID
\end{array}\right\}
\begin{array}{l}
\{s_2 s_8 s_{16} s_{28} s_{31} r_1 F_2\} \\
\{s_2 s_8 s_{17} s_{28} s_{31} r_1 F_2\} \\
\{s_2 s_8 s_{17} s_{29} s_{31} r_2 F_2\} \\
\\
\{s_2 s_8 s_{18} s_{29} s_{31} r_2 F_2\} \\
\\
\\
\\
\{s_2 s_8 s_{17} s_{30} s_{31} r_3 F_2\} \\
\\
\{s_2 s_8 s_{18} s_{30} s_{31} r_3 F_2\} \\
\\
\\
\\
\{s_2 s_{11} s_{18} s_{31} r_4 F_2\} \\
\\
\\
\\
\{s_2 s_{12} s_{17} s_{31} r_4 F_2\} \\
\\
\end{array}$$

Followed by: $\{\forall x \in elems(s) \cdot is\_nat(x)\} \times$

$$\left\{\begin{array}{ll}
len(s) > 4 \wedge (\forall i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_{12} s_{18} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s) - 1) \wedge & \\
\quad (\exists i \in elems(s) \cdot 2 * i < hd\ s + sum(tl\ s) - 1) \wedge & \\
\quad R = INVALID & \\
s = [0] \wedge R = INVALID & \{s_2 s_9 s_{19} s_{31} r_4 F_1\} \\
s = [X] \wedge X > 0 \wedge R = INVALID & \{s_2 s_9 s_{21} s_{31} r_4 F_2\} \\
s = [X, X] \wedge R = INVALID & \{s_2 s_{10} s_{20} s_{31} r_4 F_2\} \\
len(s) = 2 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_{10} s_{23} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID & \\
s = [0, 0, 0, 0] \wedge R = INVALID & \{s_2 s_{11} s_{20} s_{31} r_4 F_1\} \\
len(s) = 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_{11} s_{22} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID & \\
len(s) = 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_{11} s_{23} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID & \\
len(s) > 4 \wedge elems(s) = \{0\} \wedge R = INVALID & \{s_2 s_{12} s_{20} s_{31} r_4 F_1\} \\
len(s) > 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_{12} s_{22} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID & \\
len(s) > 4 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_{12} s_{23} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID & \\
s = [0, 0, 0] \wedge R = INVALID & \{s_2 s_8 s_{20} s_{31} r_4 F_1\} \\
len(s) = 3 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_8 s_{22} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i = hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID & \\
len(s) = 3 \wedge (\exists i \in elems(s) \cdot 2 * i < sum(s)) \wedge & \{s_2 s_8 s_{23} s_{31} r_4 F_2\} \\
\quad (\exists i \in elems(s) \cdot 2 * i > hd\ s + sum(tl\ s)) \wedge & \\
\quad R = INVALID &
\end{array}\right\}$$

We now have to find a covering for $P(B*)$ and $P(\neg B)$. One of them is:

$$\left\{\begin{array}{l} s = [i_1] \wedge R = \mathit{INVALID} \\ s = [r_1, r_2] \wedge R = \mathit{INVALID} \\ s = [t_1, n_1] \wedge R = \mathit{INVALID} \\ len(s) = 4 \wedge elems(s) = \{t_1, \ldots, t_p\} \wedge R = \mathit{INVALID} \\ len(s) = 4 \wedge elems(s) = \{*, t_1, \ldots, t_p\} \wedge R = \mathit{INVALID} \\ len(s) > 4 \wedge elems(s) = \{i_1, \ldots, i_n, r_1, \ldots, r_q\} \wedge R = \mathit{INVALID} \\ len(s) > 4 \wedge elems(s) = \{i_1, \ldots, i_n, n_1, \ldots, n_n\} \wedge R = \mathit{INVALID} \\ len(s) = 3 \wedge elems(s) = \{*, n_1, \ldots, n_n\} \wedge R = \mathit{INVALID} \end{array}\right\} \begin{array}{l} \{s_2 s_9 s_{26} s_{32} r_4 F_4\} \\ \{s_2 s_{10} s_{26} s_{32} r_4 F_5\} \\ \{s_2 s_{10} s_{26} s_{33} r_4 F_6\} \\ \{s_2 s_{11} s_{26} s_{32} r_4 F_6\} \\ \{s_2 s_{11} s_{26} s_{33} r_4 F_6\} \\ \{s_2 s_{12} s_{26} s_{32} r_4 F_4 F_5\} \\ \{s_2 s_{12} s_{26} s_{33} r_4 F_4\} \\ \{s_2 s_8 s_{26} s_{34} r_4 F_7\} \end{array}$$

We can now sample these final classes to obtain an adequate set of test cases for the Triangle Problem as shown in table 6.1 and 6.2.

6.3 Evaluation

To evaluate the adequacy of the tests we have generated following our technique we recall the test set derived by North in Table 6.3 in his feasibility study of test case generation from formal specifications [11]. We will also compare our results with those of Dick and Faivre [13, 14].

We remarked in chapter 4, that the fact that the 8 test cases generated using Dick and Faivre's tool (remark however that only the classes are generated: the sampling is actually manual) are roughly included in North's test set was encouraging. The absence of boundary tests and of many of the circumstances in which an *INVALID* outcome arises prompted us to refine the partitioning process. We must now examine our test set for inclusion in North's test set and discuss the differences arising.

6.3.1 Permutations

If we examine the *Isosceles* outcome first. Our tests number 7 and 8 are present in North's set (represented by say 25 and 28). But their permutations are not represented in our test set.

Globally, none of the permutations present in North's test set are represented in our set (this concerns North's tests number 3, 4, 6, 7, 9, 10, 12, 13, 26, 27, 29,

Id.	Test Input	Oracle	Class
1	[]	Invalid	$s_1 s_5 s_{15} r_4$
2	1.42	Invalid	$s_3 s_7 s_{14} s_{25} r_4$
3	$*$	Invalid	$s_4 s_6 s_{13} s_{24} r_4$
4	$[0, 70, 30, M]$	Invalid	$s_2 s_{11} s_{17} s_{31} r_4 F_1 F_2 F_3$
5	$[1, 1, 1]$	Equilateral	$s_2 s_8 s_{16} s_{28} s_{31} r_1 F_2$
6	$[42, 42, 42]$	Equilateral	$s_2 s_8 s_{17} s_{29} s_{31} r_1 F_2$
7	$[10, 4, 10]$	Isosceles	$s_2 s_8 s_{17} s_{29} s_{31} r_2 F_2$
8	$[5, 5, 9]$	Isosceles	$s_2 s_8 s_{18} s_{29} s_{31} r_2 F_2$
9	$[10, 7, 8]$	Scalene	$s_2 s_8 s_{17} s_{30} s_{31} r_3 F_2$
10	$[5, 10, 14]$	Scalene	$s_2 s_8 s_{18} s_{30} s_{31} r_3 F_2$
11	$[5, 10, 8, 22]$	Invalid	$s_2 s_{11} s_{18} s_{31} r_4 F_2$
12	$[2, 20, 5, 42, 41, 5]$	Invalid	$s_2 s_{12} s_{17} s_{31} r_4 F_2$
13	$[7, 10, 45, 20, 5, 8, 94]$	Invalid	$s_2 s_{12} s_{18} s_{31} r_4 F_2$
14	$[0]$	Invalid	$s_2 s_9 s_{19} s_{31} r_4 F_1$
15	$[50]$	Invalid	$s_2 s_9 s_{21} s_{31} r_4 F_2$
16	$[23, 23]$	Invalid	$s_2 s_{10} s_{20} s_{31} r_4 F_2$
17	$[10, 5]$	Invalid	$s_2 s_{10} s_{23} s_{31} r_4 F_2$
18	$[0, 0, 0, 0]$	Invalid	$s_2 s_{11} s_{20} s_{31} r_4 F_1$
19	$[3, 13, 4, 20]$	Invalid	$s_2 s_{11} s_{22} s_{31} r_4 F_2$

Table 6.1: Our Test Cases for the Triangle Problem Part 1

Id.	Test Input	Oracle	Class
20	$[100, 3, 5, 10]$	Invalid	$s_2 s_{11} s_{23} s_{31} r_4 F_2$
21	$[0, 0, 0, 0, 0, 0, 0]$	Invalid	$s_2 s_{12} s_{20} s_{31} r_4 F_1$
22	$[1, 5, 10, 9, 20, 45]$	Invalid	$s_2 s_{12} s_{22} s_{31} r_4 F_2$
23	$[1, 5, 10, 50, 8]$	Invalid	$s_2 s_{12} s_{23} s_{31} r_4 F_2$
24	$[0, 0, 0]$	Invalid	$s_2 s_8 s_{20} s_{31} r_4 F_1$
25	$[42, 20, 62]$	Invalid	$s_2 s_8 s_{22} s_{31} r_4 F_2$
26	$[20, 23, 100]$	Invalid	$s_2 s_8 s_{23} s_{31} r_4 F_2$
27	$[-10]$	Invalid	$s_2 s_9 s_{26} s_{32} r_4 F_4$
28	$[53.95, -78.9]$	Invalid	$s_2 s_{10} s_{26} s_{32} r_4 F_5$
29	$['H', 3]$	Invalid	$s_2 s_{10} s_{26} s_{33} r_4 F_6$
30	$['A','Z','E','R']$	Invalid	$s_2 s_{11} s_{26} s_{32} r_4 F_6$
31	$[*,'D', *,'J']$	Invalid	$s_2 s_{11} s_{26} s_{33} r_4 F_6$
32	$[4.56, 9, -3.14, 2.3, 13, 10.4]$	Invalid	$s_2 s_{12} s_{26} s_{32} r_4 F_4 F_5$
33	$[-4, 10, -3, 6, 42, 10]$	Invalid	$s_2 s_{12} s_{26} s_{33} r_4 F_4$
34	$[*, *, 9]$	Invalid	$s_2 s_8 s_{26} s_{33} r_4 F_7$

Table 6.2: Our Test Cases for the Triangle Problem Part 2

Id.	Test Input	Oracle	Id.	Test Input	Oracle
1	$[0, 0, 0]$	Invalid	19	$[2, 3]$	Invalid
2	$[0, 1, 1]$	Invalid	20	$[4, 4, 4, 4]$	Invalid
3	$[1, 0, 1]$	Invalid	21	$[M, M, 1]$	Isosceles
4	$[1, 1, 0]$	Invalid	22	$[M, M, M]$	Equilateral
5	$[3, 1, 2]$	Invalid	23	$[M + 1, M - 1, M]$	Scalene_or_Invalid
6	$[1, 3, 2]$	Invalid	24	$[1, 1, 1]$	Equilateral
7	$[2, 1, 3]$	Invalid	25	$[1, 2, 2]$	Isosceles
8	$[1, 2, 5]$	Invalid	26	$[2, 1, 2]$	Isosceles
9	$[5, 2, 1]$	Invalid	27	$[2, 2, 1]$	Isosceles
10	$[2, 5, 1]$	Invalid	28	$[3, 2, 2]$	Isosceles
11	$[5, 1, 1]$	Invalid	29	$[2, 3, 2]$	Isosceles
12	$[1, 5, 1]$	Invalid	30	$[2, 2, 3]$	Isosceles
13	$[1, 1, 5]$	Invalid	31	$[2, 3, 4]$	Scalene
14	$[1, 2, -6]$	Invalid	32	$[3, 2, 4]$	Scalene
15	$[-2, -2, -2]$	Invalid	33	$[3, 4, 2]$	Scalene
16	$[2, 2.3, 2]$	Invalid	34	$[4, 3, 2]$	Scalene
17	$['A', 2, 3]$	Invalid	35	$[4, 2, 3]$	Scalene
18	$['A','A','A']$	Invalid	36	$[2, 4, 3]$	Scalene

Table 6.3: Reminder of North's Test Cases for the Triangle Problem

30 and 32 to 26 which are only permutations of some original test). Dick and Faivre's test set did not either contain any permutations. Beside by examining our test generation process, this absence can be explained at a higher level by remarking that nothing in the specification implies a different behaviour of the system for permutations of the sequence parameter. To introduce permutations in the tests generated we have several options:

- we can introduce permutations as a boundary feature of sequences.
- we can decide to include this testing requirement during sampling: sampling will then yield several tests per class according to the permutations of sequences.

We make two important remarks concerning this particular testing requirement.

First, we could argue that because it is not present in the specification, the requirement that sequences be permuted is in fact an *error guessing* aspect of the test cases derivation process. This is indeed accepted by North in [11]. Hence no testing tool or systematic test generation technique should, as per design, include those tests. That we have proposed two ways through which those tests could be generated using our technique shows that error guessing, in general, could be incorporated in our proposal. It is however not part of the basic technique because its value is not underpinned by as much empirical evidence as partitioning or classical basic boundary testing are.

Secondly, we could argue that many of the permutations in North's test set are redundant. In particular, in many implementations, testing for permutations only once would be sufficient. For an unknown implementation however, North's test set seems adequate with respect to the permutation of sequences.

Thus, we accept that permutations of the input sequence would improve the adequacy of the test set generated (although we have some concerns with a systematic permutation of all sequences in the specification) and that it could be incorporated in our technique as an error guessing testing requirement. However, for the testing of specifications alone, it would be difficult to argue for the introduction of permutations in sequences.

6.3.2 Overflow

The *Equilateral* outcome illustrates another divergence between our test set and North's. Our two equilateral outcome tests are represented by the tests 22 and 21 of North's test set. While the test $[1, 1, 1]$ is perfectly matched, North's $[M, M, M]$ appears to be designed to test not only the boundary value of an Integer but the possibility of arithmetic overflows in the implementation under test. If the goal is to test that the representation of integers in the implementation can handle as large numbers as is specified in the specification then sensibly only one value need to take its maximum.

The design of test cases from formal specifications for the explicit testing of potential overflows in the system under test is we believe doomed to failure because an implementation is almost certain not to reproduce the specification in terms of computational steps. Better, would be to apply static analysis techniques on the code of the system under test to reveal instances where overflow could actually occur as in [27].

As the input is a sequence of numbers, it is sufficient to test that the implementation handles one, as large as possible, input correctly. That in our test set there may be a problem in the fact that such a large value is only to be found in an input of length 4 is a concern to which we will return.

6.3.3 Scalene Outcome

Beside the absence of permutations in our test set which we have already discussed, one of the differences which emerge between our test set and North's is the presence of two sequences in our results with a Scalene outcome compared with only one in North's test set.

At closer inspection North's test number 31 seems to cover our tests number 9 and 10 in that it represents the case where only only one digit change should alter the outcome (as in our test number 9) and that it is an almost flat triangle which our test number 10 seems to cover separately.

It is difficult to say if our test set does contain a redundant test in this circumstance. We are sure however that our two tests do cover North's single test and are thus safe. If we cannot find a high level argument for the removal of those kind of tests (this assumes that we can identify them in the first place) then we must refrain from changing our rules to suit this particular example. In designing our approach we have taken care not to make dubious choices to suit particular examples.

6.3.4 Invalid Outcome

This is the area where most of the differences between our test set and North's are to be found: if there were some differences for the other outcomes, the results were broadly similar with the exception of the absence of permutations in our test set.

The first difference is the inclusion of undefined elements in our tests number 31 and 34 and a completely undefined input in 3. The introduction of undefined values in our test set stems from our close adhesion to the semantics of VDM-SL. We however recognise that it may be difficult to take the potential LPF behaviour of specifications systematically into account in an eventual test generator. Therefore such tests may be very difficult to generate automatically.

Further, it could be argued that in most circumstances an undefined input value would have no equivalent in the system under test and thus that it would be difficult to exercise the implementation with such values. We note however, that for robust systems, undefined values could be of interest: a misreading from an input file could be said to be equivalent to an undefined value in the specification or that a totally spurious value in testing assembly code could be used.

If tests with undefined values as input are not desired then the specification could be changed to rule them out during the partitioning process.

Our test set includes the empty sequence and an input which is not a sequence at all (tests 1 and 2). These are absent from North's test set but should, in our opinion, be part of an adequate test set.

Sequences of length 2 and 4 are present in our set as recommended by North as well as sequences with other values than integers.

We also generate 3 tests of length three which should lead to an invalid outcome.

***Of Length Three* Problem**

The most striking difference however, is that our tests, as derived, do not reflect the fact that in an implementation if the input is not composed of exactly three values it will be correctly dismissed immediately as an invalid triangle. This implies that many of our tests with a length different than three are redundant and that some of the classes they cover should be covered by other tests of length three.

For example our test number 4, which contains the largest number that the implementation should be able to handle will probably never exercise the implementation on this aspect. The same applies for the non naturals of the tests 28, 29, 30, 32 and 33.

This is also the origin of many of our *INVALID* test cases.

We can propose no change in our approach which would remedy this problem. However, as remarked in chapter 3, if the specification of the Triangle Problem was more detailed in the error messages it issues then a specification of the form:

$$
\begin{array}{l}
\mathit{Classify} \ : \mathit{token} \rightarrow \mathit{Triangle\_type} | \mathit{NOT\_A\_SEQUENCE\_NAT} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad | \mathit{NOT\_OF\_LENGHT\_3} \\
\mathit{Classify}(s) == \\
\quad \mathit{if}\ \neg \mathit{is\_seq\_nat}(s) \\
\qquad \mathit{then}\ \mathit{NOT\_A\_SEQUENCE\_NAT} \\
\quad \mathit{elseif}\ \mathit{len}(s) \neq 3 \\
\qquad \mathit{then}\ \mathit{NOT\_OF\_LENGHT\_3} \\
\quad \mathit{elseif}\ \exists i \in \mathit{elems}(s) \cdot 2 * i \geq \mathit{sum}(s) \\
\qquad \mathit{then}\ \mathit{INVALID} \\
\quad \mathit{else}\ \mathit{variety}(s)
\end{array}
$$

would allow the generation of test cases closer to the ideal of North.

6.3.5 Conclusion

Given that while developing our approach, we have never introduced special features to suit the specification of the Triangle Problem as given by North, we find our results encouraging. Most of the differences between our test cases and North's adequate test set are minor, or it can be argued that our results do in fact increase the likelihood of finding an error in the system under test.

The Triangle Problem illustrates well our recommendations that detailed error messages should increase the adequacy of the test cases generated. The Triangle Problem, as specified by North, has proved challenging and yet brief enough to permit a detailed derivation of test cases to be included here.

Because our approach is general, we are confident that our technique is suitable for the generation of high adequacy test sets from formal specifications.

Chapter 7

Conclusions

Our introduction, in Chapter 1 of this thesis, outlined the following assumptions concerning test cases generation from formal specifications:

- *that it would strengthen both, the testing activity, and the use of formal software engineering methods.*
- *that a technique for the generation of high adequacy test sets could be found.*
- *that it could be automated.*

We will first review these concerns by highlighting our contributions and the issues still to be resolved, and then propose further work in this area.

7.1 Contributions

Chapter 2 and 3 underpinned our first assumption. Chapter 3, 4 and 5 were broadly concerned with finding an appropriate test cases generation technique. Finally, in parts of Chapter 5 and further in Chapter 6 we addressed our last concern.

We now review these three issues.

7.1.1 Closing the Gap

In Chapter 2, we outlined the current state-of-the-art in automatic test generators. We stressed the shortcomings of the current techniques and noted the relatively little amount of research carried out on automatic test cases generation from formal specifications. We also stressed the complementary nature of the various approaches to testing: random, white box and black box.

Chapter 3, showed how an ATG from high level specifications could be integrated in the the software development process as a whole and its potential benefits. The possibility of testing specifications themselves was introduced. In particular, in Section 3.1, the benefits of testing from formal specifications were discussed for their relevance to the testing process and as a means to add value to formal software engineering methods.

Being able to generate rigorous test cases at an early stage of the software development should go towards the rationalisation of the testing activity. Further, doing so automatically would mean that large case studies could be carried out with a view to improving the current testing theory which lacks empirical data. Even an automatic oracle based on formal specifications would be an advantage during the testing phase.

The kind of work we have undertaken should ultimately allow a rapprochement of current practices and academic wisdom; this is to be welcomed.

7.1.2 An Appropriate Technique

In Chapter 3, we clarified many aspects of testing from formal specifications and how to integrate it within the global testing activity.

Our contributions towards a technique for high adequacy test set generation can be summarised as follows:

- We have extended the work of Dick and Faivre, by taking into account the potential LPF behaviour of VDM-SL specifications. We have also introduced a formalism suitable for the study of partitioning rules from formal specifications.

- We have shown that the coarse partitioning rules developed in the above work are insufficient for the generation of adequate test sets and that refinements must be introduced.

- The work of Stock and Carrington on fine partitioning rules, was extended for expressions which introduce the notion of scope in the formal notation. In particular, we developed rules for quantified expressions. We have shown how the partitioning conventions for operator expressions can be systematically obtained by using their definition based on basic operators.

- We have shown, that to integrate the coarse partitioning rules and the necessary refinements discussed, heuristics for the detection of redundant classes are necessary. We have proposed one such heuristic underpinned by the probability of revealing an error in the system under test. Our heuristic was integrated into our formalism. We illustrated the potential benefits of our heuristic which includes a natural treatment of functions.

- For completeness, we also discussed the use of recursion in specifications. Also, we examined the consequences, for test cases generation from formal specifications, of the difficult area of non-determinism.

Our synthesis of the two previous basic results in the area of test sets generation from formal specifications was shown, using the Triangle Problem, to reach a high level of adequacy when compared with the manually derived test set of North. Further, as we refrained, in the development of our technique, to introduce non established ad hoc testing methods and, as far as possible, justified every aspect of our technique, we are confident in the generality of our results.

7.1.3 Automation

The pioneering work of Dick and Faivre, allowed us to make many suggestions towards the development of an efficient and pragmatic technique.

In particular, we have shown that DNF reduction of the specification prior to partitioning is not necessary. We have also introduced the Constraint Logic Programming pradigm to show how an eventual solver might be implemented.

Althought our technique is systematic we do have reservations about its possible automation. In particular, and as the study of the simple Triangle Problem has shown, the complexity of the consistency checks necessary is beyond current technology.

The strengths of VDM-SL as a specification language (mainly its high level level of abstractness) contribute to its weaknesses as far as a basis for test generation is concerned. In parlicular, the complexity of the language makes tools based on the notation difficult to design and implement. We note for example that the principal functionalities of the IFAD's toolbox [102] are only available on a subset of the language because of theoretical limitations.

We have suggested, and demonstrated on the simple Triangle Problem, that because of its generality, precision and standardisation VDM-SL would be a good basis for tests generation. This was already reported by North [11]. We must however stress that the language may well be too complex for a high level of automation of our test generation technique to be achieved.

7.2 Future Work

During our review of current Automatic Tests Generators, in section 2.3.5, we listed the following problems affecting current ATGs:

- their efficiency is difficult to assess.
- the level of automation achieved is generally not high enough.
- they are all based on a single strategy.
- their scope of application is usually limited.

We must admit that an eventual tool based on our proposed systematic test cases generation from VDM-SL specifications technique would also be affected by all these problems.

Thus, much work remains to be done. We propose two strands for future work.

7.2.1 Pragmatic Considerations

Of considerable benefit, would be to implement a prototype test cases generator adopting our approach. Currently, only the prototype tool of Dick and Faivre is available. As we reported, our approach makes numerous pragmatic suggestions towards the improvement of their achievements. Also, the rapid progress of Constraint Logic Programming languages since its implementation would have, if taken into account, a positive effect on its capabilities. Ultimately however, we do not forsee a possible full automation of our technique for the entire VDM notation.

Besides the many implementation considerations we have made in this work, we propose some compromises which would ease somewhat the heavy implementation efforts involved in building a prototype.

- To study the particular constraint solving requirements of an eventual tool, an automated oracle from a high level formal notation could be developed first. 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 as suggested in Chapter 4, by using annotations to indicate where LPF behaviour is intended). Also for basic operators, simpler rules than those we have used could be employed (e.g. for $\neq$).

- Integration of a tests generator in IFAD's toolbox [102] could also be envisaged. 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 [136] 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. Although we would now tend towards concentrating on pragmatic aspects of our work, we also suggest some theoretical considerations deserving some further attention.

7.2.2 Theoretical Advances

In this work, and as declared in Chapter 2, our main concern has been to find a technique, that could be automated, for the generation of component tests from formal specifications. Although we have made many suggestions as to which style of specifications are better than others, from a testing point of view (see for example sections 3.4.2, 3.4.4. and 3.5.1), these were rather low level suggestions mainly concerned with the generation of the component tests per se.

Of more interest perhaps, would be the study of specification styles suitable for building testable software and the whole area of system level testing in general. It has been argued [137, 138] that software built from specifications using extended Finite State Machines are easier (both from a practical and theroretical point of view) to test than others because of their intrinsic design for test features. These concerns are well known to the hardware community where testing is not an afterthought, as is too often the case for software, but is an integral part of system development.

It would also be worthwhile to investigate how, from a theoretical point of view, our context dependent combination heuristic could be extended in some circumstances. An analysis of a specification and its implementation, even if manual, could reveal parts of the specification which, although not specified separately using functions, are in effect independently implemented in the system under consideration. This would allow some further contraction of the test sets generated in some circumstances. The implications of using system specific information in the generation of tests from formal specifications (an essentially black box testing technique) could also be of interest.

We also suggest that, to refine the partitioning theory, some new forms of constraints or domain divisions could be investigated. For example, it could be of benefit to generate *optimum* constraints: on an expression of the form $x > 5$, a constraint such as x is as small as possible but greater than 5 could be of interest. Currently, only the sub-domains $x = 6$ and $x > 6$ are generated, but it may well be possible that, because of global unsatisfiability, we cannot generate a test where $x = 5$: in those cases we miss the opportunity of generating tests at the boundary of the input domain. As a further example, it could be interesting to generate sets, or sequences of the minimal size allowed by the specification rather than of the size explicitly implied by the predicates.

Finally, the availability of a tests generator tool could allow a statistical evaluation of the most beneficial domain divisions. This would lead to the simplification of some partitioning rules and the refinement of others. In particular, the quantified expression rules could be simplified if some of our sub-domains are statistically shown to be of little value. The development of different sets of rules for different areas of application could also be envisaged to deal, for example, with safety critical systems.

Bibliography

[1] O.-J. Dahl, E. Dijkstra, and C. Hoare, *Structured Programming*, vol. 8 of *APIC Studies in Data Processing.* Academic Press, 1972.

[2] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1402–1411, Dec. 1990.

[3] A. Tanenbaum, "In defense of program testing or correctness proofs considered harmful," *ACM SIGPLAN Notices*, vol. 11, pp. 64–68, May 1976.

[4] A. Hall, "Seven myths of formal methods," *IEEE Software*, pp. 11–19, Sept. 1990.

[5] J. Bowen and M. Hinchey, "Seven more myths of formal methods," *IEEE Software*, pp. 34–41, July 1995.

[6] R. Glass, "The many flavors of testing," *Journal of Systems Software*, vol. 20, no. 2, pp. 105–106, 1993.

[7] M. Ould, "Testing—a challenge to method and tool developers," *Software Engineering Journal*, vol. 6, pp. 59–64, Mar. 1991.

[8] D. Gelperin and B. Hetzel, "The growth of software testing," *Communications ACM*, vol. 31, pp. 687–695, June 1988.

[9] C. Ramamoorthy and S.-B. F. Ho, "Testing large software with automated software evaluation systems," *IEEE Transactions on Software Engineering*, vol. 1, pp. 46–58, Mar. 1975.

[10] R. Hamlet, "Special section on software testing," *Communications ACM*, vol. 31, pp. 662–667, June 1988.

[11] N. North, "Automatic test generation for the triangle problem," Tech. Rep. DITC 161/90, NPL, Feb. 1990.

[12] I. Spence and C. Meudec, "Generation of software tests from specifications," in *Software Quality Management II: Building Quality into Software* (M. Ross, C. Brebbia, G. Staples, and J. Stapleton, eds.), vol. 2, (Edinburgh, UK), pp. 517–530, Computational Mechanics Publications, July 1994.

[13] J. Dick and A. Faivre, "Automatic partition analysis of VDM specifications," Tech. Rep. TR 92027, BULL S.A., 1992.

[14] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model based specifications," in *FME'93 Industrial Strength Formal Methods* (J. Woodcock and P. Larsen, eds.), vol. 670 of *Lecture Notes in Computer Science*, pp. 268–284, Springer-Verlag, 1993.

[15] P. Stocks and D. Carrington, "Test template framework: A specification-based testing case study," in *ISSTA'93*, (USA), 1993.

[16] P. Stocks and D. Carrington, "A framework for specification-based testing," *IEEE Transactions on Software Engineering*, vol. 22, pp. 777–793, Nov. 1996.

[17] A. Bertolini, "An overview of automated software testing," *Journal of Systems Software*, vol. 15, no. 2, pp. 133–138, 1991.

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

[19] B. Beizer, *Software System Testing and Quality Assurance.* International Thomson Computer Press, 1996.

[20] M. Deutsch, *Software Verification and Validation; realistic project approaches.* Prentice-Hall Series in Software Engineering, 1982.

[21] H. Kopetz, *Sotware Reliability.* Macmillan Computer Science Series, 1979.

[22] W. Hetzel, *The Complete Guide to Software Testing.* QED Information Sciences Inc., 1984.

[23] P. Coward, "A review of software testing," *Information and Software Technology*, vol. 30, pp. 189–198, Apr. 1988.

[24] D. Ince, *Introduction to Software Project Management and Quality Assurance.* McGraw-Hill, 1993.

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

[26] British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), April 97, *Glossary of Terms Used in Software Testing.*

[27] J. Barnes, *High Integrity Ada - The SPARK Approach.* Addison-Wesley, 1997. ISBN 0–201–17517–7.

[28] 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*, pp. 68–78, ACM Press, May 1997.

[29] A. Omar and F. Mohammed, "A survey of software functional testing methods," *Software Engineering Notes*, vol. 16, pp. 75–82, Apr. 1991.

[30] E. Weyuker, "The evaluation of program-based software test data adequacy criteria," *Communications ACM*, vol. 31, pp. 668–675, June 1988.

[31] A. Parrish and S. Zweben, "Analysis and refinement of software test data adequacy properties," *IEEE Transactions on Software Engineering*, vol. 17, pp. 565–581, June 1991.

[32] H. Zhu and P. Hall, "Test data adequacy measurement," *Software Engineering Journal*, pp. 21–29, Jan. 1993.

[33] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 438–444, July 1984.

[34] E. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, pp. 703–711, July 1991.

[35] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: help for the practicing programmer," *IEEE Computer*, vol. C-11, pp. 34–41, Apr. 1978.

[36] B. Choi and A. Mathur, "High-performance mutation testing," *Journal of Systems Software*, vol. 20, no. 2, pp. 135–152, 1993.

[37] E. Krauser, A. Mathur, and V. Rego, "High performance software testing on SIMD machines," *IEEE Transactions on Software Engineering*, vol. 17, pp. 403–423, May 1991.

[38] R. DeMillo and A. Offutt, "Experimental results from an automatic test case generator," *IEEE Transactions on Software Engineering and Methodologies*, vol. 2, pp. 109–127, Apr. 1993.

[39] J. Miller, M. Roper, M. Wood, and A. Brooks, "Towards a benchmark for the evaluation of software testing techniques," *Information and Software Technology*, vol. 37, no. 1, pp. 5–13, 1995.

[40] B. Beizer, "Review," *Computing Reviews*, p. 67, Jan. 1992.

[41] R. DeMillo and A. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, pp. 900–910, Sept. 1991.

[42] K. King and A. Offutt, "A Fortran language system for mutation based software testing," *Software-Practice and Experience*, vol. 21, pp. 685–718, July 1991.

[43] M. Woodward, "Mutation testing: its origin and evolution," *Information and Software Technology*, vol. 35, pp. 163–169, Mar. 1993.

[44] D. Hoffman and P. Strooper, "Automated module testing in Prolog," *IEEE Transactions on Software Engineering*, vol. 17, pp. 934–943, Sept. 1991.

[45] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. 4, pp. 178–187, May 1978.

[46] K. Sabnani and A. Dahbura, "A protocol test generation procedure," vol. 15, no. 4, pp. 285–297, 1988.

[47] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering*, vol. 17, pp. 591–603, June 1991.

[48] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," vol. 14, no. 1, 1987.

[49] S. Budkowski and P.Dembinski, "An introduction to estelle: a specification language for distributed systems," vol. 14, no. 1, 1987.

[50] F. Belina and D. Hogrefe, "The CCITT-specification and description language SDL," vol. 16, 1989.

[51] P. Hall, "Relationship between specifications and testing," *Information and Software Technology*, vol. 33, pp. 47–52, Jan./Feb. 1991.

[52] G. Bernot, M. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Software Engineering Journal*, vol. 6, pp. 387–405, Nov. 1991.

[53] J. Wilson, "Formal methods in object oriented analysis," *BT Technology Journal*, vol. 11, pp. 18–31, July 1993.

[54] C. Minkowitz and P. Henderson, "A formal description of object-oriented programming using VDM," in *VDM'87; VDM—A Formal Method at Work*, vol. 252 of *Lecture Notes in Computer Science*, pp. 237–259, Springer-Verlag, Mar. 1987.

[55] C. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985.

[56] C. Middelburg, *Logic and Specification—Extending VDM-SL for Advanced Formal Specification*. Chapman & Hall, 1993.

[57] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, pp. 284–292, Sept. 1993.

[58] D. Brown, R. Roggio, J. C. II, and C. McCreary, "An automated oracle for software testing," *IEEE Transactions on Reliability*, vol. 41, pp. 272–280, June 1992.

[59] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing," *IEEE Transactions on Software Engineering*, vol. 8, pp. 343–353, July 1982.

[60] D. Bird and C. Munoz, "Automatic generation of random self-checking test cases," *IBM systems journal*, vol. 22, no. 3, pp. 229–245, 1983.

[61] P. Coward, "Symbolic execution systems—a review," *Software Engineering Journal*, vol. 3, pp. 229–239, Nov. 1988.

[62] C. Ramamoorthy, S. bun F. Ho, and W. Chen, "On the automated generation of program test data," *IEEE Transactions on Software Engineering*, vol. 2, pp. 293–300, Dec. 1976.

[63] J. King, "Symbolic execution and program testing," *Communications ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[64] R. Boyer, B. Elpas, and K. Levit, "Select—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*, pp. 234–244, Apr. 1975.

[65] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 215–222, Sept. 1976.

[66] P. Asirelli, P. Degano, G. Levi, A. Martelli, U. Montanari, G. Pacini, F. Sirovich, and F. Turini, "A flexible environment for program devel-

opment based on a symbolic interpreter," in *Proceedings of the Fourth International Conference on Software Engineering*, (Munich, Germany), pp. 251–263, Sept. 1979.

[67] D. Hedley, *Automatic test data generation and related topics.* PhD thesis, Liverpool University, UK, 1981.

[68] M. Hennell, D. Hedley, and I. Riddell, "The LDRA software testbeds: their roles and capabilities," in *Proceedings of the IEEE Software Fair 83 Conference*, (Arlington, VA, USA), July 1983.

[69] P. Coward, "Symbolic execution and testing," *Information and Software Technology*, vol. 33, pp. 53–64, Jan./Feb. 1991.

[70] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, pp. 870–879, Aug. 1990.

[71] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller, "SMOTL—a system to construct samples for data processing program debugging," *IEEE Transactions on Software Engineering*, vol. 15, pp. 60–66, 1979.

[72] M. Dyer, "Distribution-based statistical sampling: an approach to software functional test," *Journal of Systems Software*, vol. 20, no. 2, pp. 107–114, 1993.

[73] Z. Furukawa, K. Nogi, and K. Tokunaga, "AGENT: an advanced test-case generation system for functional testing," in *AFIPS press national computer conference*, vol. 54, pp. 525–535, 1985.

[74] S. Morasca and M. Pezze, "Using high-level petri nets for testing concurrent and real-time systems," in *Real-time systems, theory and applications* (H. Zendan, ed.), pp. 119–131, North-Holland, 1990.

[75] W. Tsai, D. Volovik, and T. Keefe, "Automated test case generation for programs specified by relational algebra queries," *IEEE Transactions on Software Engineering*, vol. 16, pp. 316–324, Mar. 1990.

[76] D. Pitt and D. Freestone, "The derivation of conformance tests from LOTOS specifications," *IEEE Transactions on Software Engineering*, vol. 12, pp. 1337–1343, Dec. 1990.

[77] X. Li, T. Higashino, and K. Taniguchi, "Automatic derivation of test cases for LOTOS expressions with data parameters," vol. 77, pp. 1–14, 1994.

[78] S. Ashford, "Automatic test case generation using prolog," Tech. Rep. DITC 215/93, NPL, Jan. 1993.

[79] P. Jalote, "Specification and testing of abstract data types," *Computer Language*, vol. 17, no. 1, pp. 75–82, 1992.

[80] P. Dauchy, M. Gaudel, and B. Marre, "Using algebraic specifications in software testing: a case study on the software of an automatic subway," *Journal of Systems Software*, vol. 21, no. 3, pp. 229–244, 1993.

[81] T. Ostrand and M. Balcer, "The category-partition method for specifying and generating functional tests," *Communications ACM*, vol. 31, pp. 676–686, June 1988.

[82] G. Laycock, "Formal specification and testing," *Journal of Software Testing, Verification and Reliability*, vol. 2, no. 3, pp. 7–23, 1992.

[83] J. Chang, D. Richardson, and S. Sankar, "Structural specification-based testing with ADL," *Software Engineering Notes*, vol. 21, pp. 62–70, Jan. 1996. Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96).

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

[85] I. Hayes, *Specification case studies.* Series in computer science, Prentice Hall International, 1987.

[86] J. Spivey, *The Z notation-reference manual.* Series in computer science, Prentice Hall International, 1989.

[87] Softbridge, Inc., Cambridge, UK, *Automated Test Facility.*

[88] Software Research, Inc., USA, *Software TestWorks Tool Suite*.

[89] Software Quality Automation, Inc., USA, *SQA TeamTest*.

[90] J. Graham, A. Drakeford, and C. Turner, "The verification, validation and testing of object oriented systems," *BT Technology Journal*, vol. 11, pp. 79–88, July 1993.

[91] S. Barbey and A. Strohmeimer, "The problematics of testing object-oriented software," in *Software Quality Management II: Building Quality into Software* (M. Ross, C. Brebbia, G. Staples, and J. Stapleton, eds.), vol. 2, (Edinburgh, UK), pp. 411–426, Computational Mechanics Publications, July 1994.

[92] Software Products Group (IPL), Bath, UK, *Cantata*.

[93] R. Glass, "The lost world of software debugging and testing," *Communications ACM*, vol. 23, pp. 264–271, 1980.

[94] R. Yang and C. Chung, "Path analysis testing of concurrent programs," *Information and Software Technology*, vol. 34, pp. 43–56, Jan. 1992.

[95] B. Korel, H. Weddle, and R. Ferguson, "Dynamic method of test data generation for distributed software," *Information and Software Technology*, vol. 34, pp. 523–531, Aug. 1992.

[96] Software Products Group (IPL), Bath, UK, *AdaTEST*.

[97] N. Plat and P. Larsen, "An overview of the ISO/VDM-SL standard," *ACM SIGPLAN Notices*, vol. 27, pp. 76–82, Aug. 1992.

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

[99] I. Hayes, "Specification directed module testing," *IEEE Transactions on Software Engineering*, vol. 12, pp. 124–133, Jan. 1986.

[100] The VDM-SL Tool Group, *Users Manual for the IFAD VDM-SL Tool.* The Institute of Applied Computer Science, Sept. 1994.

[101] The VDM-SL Tool Group, *The IFAD VDM-SL Language.* The Institute of Applied Computer Science, Sept. 1994.

[102] R. Elmstrøm, P. Larsen, and P. Lassen, "The IFAD VDM-SL toolbox: a practical approach to formal specification," *ACM SIGPLAN Notices*, vol. 29, no. 9, pp. 77–81, 1994.

[103] P. Mukherjee, "Computer-aided validation of formal specification," *Software Engineering Journal*, pp. 133–140, July 1995.

[104] P. Lindsay, "A survey of mechanical support for formal reasoning," *Software Engineering Journal*, pp. 3–27, Jan. 1988.

[105] S. Agerholm, "Translating specifications in VDM-SL to PVS," in *Theorem Proving in Higher Order Logics—TPHOLs'96* (J. von Wright, J. Grundy, and J. Harrison, eds.), vol. 1125 of *Lecture Notes in Computer Science*, pp. 1–16, Springer-Verlag, 1996.

[106] B. Monahan and R. Shaw, "Model-based specifications," in *Software Engineer's Reference Book* (J. McDermid, ed.), ch. 21, London: Butterworth-Heinemann, 1991.

[107] J. Bicarregui, J. Fitzgerald, P. Lindsay, R. Moore, and B. Ritchie, *Proof in VDM: A Practitioner's Guide.* Springer-Verlag, 1994.

[108] J.-P. Banâtre, S. Jones, and D. L. Métayer, *Prospects for Functional Programming in Software Engineering.* No. Project 302, Volume 1 in Research Reports, ESPRIT, Springer-Verlag, 1991.

[109] P. Lindsay, "Reasoning about Z specifications: a VDM perspective," Tech. Rep. 93-20, SVRC, The University of Queensland, Australia, Oct. 1993.

[110] I. Hayes, "VDM and Z: A comparative case study," *Formal Aspect of Computing*, vol. 4, pp. 76–99, 1992.

[111] I. Hayes, C. Jones, and J. Nicholls, "Understanding the differences between VDM and Z," Tech. Rep. UMCS-93-8-1, University of Manchester, Aug. 1993.

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

[113] J. Dawes, *The VDM-SL Reference Guide.* Pitman, 1991.

[114] P. Larsen and N. Plat, "Standards for non-executable specification languages," *The Computer Journal*, vol. 35, no. 6, pp. 567–573, 1992.

[115] R. Kneuper, "Symbolic execution of specifications: User interface and scenarios," Tech. Rep. UMCS–87–12–6, University of Manchester(UK), 1987.

[116] R. Nicholl, "Unreachable states in model oriented specifications," Tech. Rep. No. 175, The University of Western Ontario, London, Ontario, Canada N6A 5B9, Sept. 1987.

[117] C. Wilmot, "Analytical techniques for verification, validation and testing," *BT Technology Journal*, vol. 10, pp. 46–53, Apr. 1992.

[118] E. Freuder, "The Many Paths to Satisfaction," in *Proceedings ECAI'94 Workshop on Constraint Processing* (M. Meyer, ed.), (Amsterdam), Aug. 1994.

[119] G. B. Dantzig, *Linear Programming and Extensions.* Princeton, New Jersey: Princeton University Press, 1963.

[120] I. Bratko, *Prolog Programming For Artificial Intelligence.* International Computer Science Series, Addison-Wesley Publishing Company, second ed., 1990. ISBN 0-201-41606-9.

[121] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving.* John Wiley & Sons, Inc, 1987.

[122] J. Jaffar and J.-L. Lassez, "Constraint Logic Programming," in *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, (Munich), pp. 111–119, ACM, Jan. 1987.

[123] A. Colmerauer, "An Introduction to Prolog III," *Communications ACM*, vol. 33, pp. 69–90, July 1990.

[124] J. Cohen, "Constraint Logic Programming Languages," *Communications ACM*, vol. 33, pp. 52–68, July 1990.

[125] P. V. Hentenryck, H. Simonis, and M. Dincbas, "Constraint Satisfaction Using Constraint Logic Programming," *Artificial Intelligence*, vol. 58, pp. 113–159, 1992.

[126] N. Choquet, "Test data generation using Prolog with constraints," in *Workshop on Software Testing*, pp. 132–141, Banff: IEEE, 1986.

[127] ECRC Common Logic Programming System, *ECL^{i}PS^e 3.5–User Manual*, Feb. 1995.

[128] "The ECRC Project ECL^{i}PS^e."
URL http://www.ecrc.de/research/projects/eclipse/.

[129] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. Yap, "The CLP($\Re$) Language and System," *ACM Transactions on Programming Languages and Systems*, vol. 14, pp. 339–395, July 1992.

[130] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, "The Constraint Logic Programming Language CHIP," in *FGCS-88: Proceedings International Conference on Fifth Generation Computer Systems*, (Tokyo), pp. 693–702, ICOT, Dec. 1988.

[131] P. Jeavons, D. Cohen, and M. Cooper, "A Substitution Operation for Constraints," in *Principles and Practice of Constraint Programming—Second International Workshop, PPCP'94* (A. Borning, ed.), vol. 874 of *Lecture Notes in Computer Science*, pp. 1–9, Springer-Verlag, 1994.

[132] C. Gervet, "Conjunto: Constraint Logic Programming with Finite Set Domains," Tech. Rep. ECRC-94-15, ECRC, 1994.

[133] X. Yang, "The automatic generation of software test data from Z specifications," Tech. Rep. CS-95-2, University of Glamorgan, Wales, UK, Feb. 1995.

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

[135] J.-F. Bergeretti and B. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 37–61, Jan. 1985.

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

[137] M. Holcombe, "An integrated methodology for the specification, verification and testing of systems," *Journal of Software Testing, Verification and Reliability*, vol. 3, no. 3–4, pp. 149–163, 1993.

[138] L. Zucconi and K. Reed, "Building testable software," *Software Engineering Notes*, pp. 51–55, Sept. 1996.