Automating the Generation and Sequencing of Test Cases from Model-Based Specifications

Jeremy Dick[1] and Alain Faivre[2]

[1] Bull Information Systems, Maxted Road, Hemel Hempstead, HP2 7DZ, UK
[2] Bull Corporate Research Centre, 78450 Les Clayes-sous-Bois, France

Abstract. Formal specifications contain a great deal of information that can be exploited in the testing of an implementation, either for the generation of test-cases, for sequencing the tests, or as an oracle in verifying the tests. This papers presents automatic techniques for partition analysis in state-based specifications, specifically VDM. Test domains for individual operations are calculated by reduction of their mathematical description to a Disjunctive Normal Form. Following this, a partition analysis of the system state can be performed which permits the construction of a Finite State Automaton from the specification. This, in turn, can be used to sequence the required tests in a valid and sensible way. A tool has been developed based on the techniques applied to VDM, which has been used to develop the examples presented in the paper.

1 Introduction

Formal methods promise high product confidence through mathematical proof of system correctness. The ultimate aim is to do away with the typically large amount of effort spent *a posteriori* in error detection, and replace it by effort spent *a priori* in the construction of systems which are error-free by design.

With the current state-of-the-art, however, there are reasons why the need for post-developmental validation cannot be eliminated. One such reason is that formally developed systems are seldom created and operated in total isolation, being dependent on compilers, operating systems and run-time environments that may not be trustworthy. It also frequently occurs that the cost of performing completely proven refinement steps is only warranted in exceptional cases of high financial or human risk, leading to systems that are formally specified, but whose development was, at least in part, informal. A similar scenario arises with reverse engineering, where a formal specification has been created for an existing system whose development is obscure.

The work reported here is an attempt to add further value to the existence of a formal specification by allowing it to be used in the testing process. We show how the partition analysis of individual operations can be automated, and how valid sequences of operations which cover all the necessary tests can be constructed. The key to this is to exploit the mathematics inherent in the specification in various constraint solving activities. There are four main aspects:

- *the partition analysis of individual operations.* This involves reducing the mathematical expression defining an operation into a Disjunctive Normal Form (DNF), which gives disjoint partitions representing domains of the operation that should be tested in the implementation;

- *the partition analysis of the system state.* Again, the mathematical expression defining the system state, viewed in the light of the pre- and postconditions of operations, is reduced into DNF, which yields disjoint partitions of state values which can be used to construct a Finite State Automaton (FSA) from the specification;
- *the scheduling of tests of different operations to avoid redundancy in the testing process.* This involves finding paths through the FSA which cover all the required tests, and composing the constraints resulting from the composition of these sequences to detect inconsistencies;
- *the generation of test values for use in the validation of the implementation.* This involves the selection of values relating to the operation *input* for each case. These test values must satisfy the constraints imposed by the mathematical definition of the specification in the given case. Once input values have been selected, the constraints describing the case can be simplified, leaving an expression constraining just the *result* values;
- *the verification of the results of an individual test against the specification.* This involves using the residual constraints to verify the values relating to the operation *results* under the test.

Our approach draws on ideas from other work. Initial inspiration came from [BGM1], which reports on test-case extraction from algebraic specifications. A generally applicable framework for the generation of sets of test data is described, in which one is required to state the hypotheses used to justify both the choice of a finite set of test data, and the existence of an oracle for verifying the results. A form of partition analysis is carried out by unfolding equations in the specification.

An earlier paper, [S1], uses case analysis of VDM specifications to partition the test domains. No precise details of how the case analysis could be performed and automated were given.

Reading [HH1] introduced us to the problem of sequencing the tests. This useful survey paper, and others by the same authors, [H1] [H2], discuss a wide range of issues in the use of formal specifications in testing. Hall and Hierons [HH1] suggest the construction of an FSA from model-based specifications, but give no method of doing so. They indicate that, given such an FSA, classical graph traversal algorithms can be used to find the required sequences of operations.

As a vehicle for exploring methods of partition analysis and test-case sequencing, we have chosen the specification language VDM-SL ([D1]), used in the Vienna Development Method ([J1]). We intend, however, the underlying method to be generally applicable to the model-based (sometimes called state-based) approach, which encompasses other notations, such as Z [S2], RSL [R1] and COLD [FJ1].

We use the ideas of [BGM1] to create a more precise formulation of the approach introduced in [S1]. In our approach, the formulae of the specification are the relations on states described by operations, and are expressed in first-order predicate calculus. These relations are reduced to a DNF, creating a set of disjoint sub-relations. Each sub-relation yields a set of constraints which describe a single test domain. The approach of reduction to DNF is a state-oriented equivalent of the case analysis of the algebraic approach. We also transfer the use of uniformity and regularity hypotheses from the algebraic work to our situation.

When applied to the most abstract level of specification, the test domains generated correspond to pure black-box testing; no information concerning the nature of the implementation is taken into consideration. If applied, however, to various levels of refinement, which is possible in the wide spectrum notation VDM-SL, more and more detail can be introduced into the partition analysis, leading, potentially, to complete white-box analysis.

We also describe a method of extracting an FSA from a state-based specification. This method uses the results of the partition analysis of individual operations, to perform a further partition analysis of the system state. The result is a set of disjoint states, each of which is either the before-state or the after-state of at least one of the tests to be performed. From these analyses, an FSA can be constructed.

Unlike in [BGM1] where the kind of formulae treated is more constrained, we cannot, in general, hope to be able to generate solutions to sets of constraints automatically in every possible case; we are working in the semi-decidable logic of the first-order predicate calculus. Despite this inherent limitation, we feel that in practice many specifications can be treated, and our goal is to create a practical approach that offers the maximum of assistance in exploiting a formal specification in the testing process.

We have developed a tool to assess the practicality of the approach, which is briefly outlined in the paper, but described more fully in [DF1] . Except where indicated, the examples shown in Sections 2 and 3 were calculated using this tool.

The paper is organised as follows. Section 2 describes the approach adopted for partition analysis of individual operations with an example. Section 3 describes a method of extracting an FSA, and finding appropriate sequences of tests, again with an example. Section 4 briefly describes the tool we have developed. Section 5 discusses limitations of the approach, and draws conclusions.

2 Partition Analysis

In this section we present our approach to the partition analysis of individual operations. We use VDM-SL, but make reference to how the approach may differ slightly for other notations.

In the state-based approach, operations describe a partial relation between system states. They are specified by giving a logical expression, *Spec-OP*, which characterises that relation. We can express this relation as a set of pairs of states as follows:

$$\begin{array}{l}\{\ (\mathit{before},\ \mathit{after}) \\ \mid\ \mathit{before}{:}\mathit{State},\ \mathit{after}{:}\mathit{State} \\ \bullet\ \mathit{inv\text{-}State}(\mathit{before}) \wedge \mathit{pre\text{-}OP}(\mathit{before}) \wedge \\ \quad \mathit{post\text{-}OP}(\mathit{before},\mathit{after}) \wedge \mathit{inv\text{-}State}(\mathit{after}) \\ \}\end{array}$$

where *pre-OP*, *post-OP* and *inv-State* are the expressions that characterise the pre- and postconditions of the the operation *OP* and the state invariant, respectively.

In other notations, such as Z, the precondition may not be explicitly stated, but can be calculated if needed from the specification (see, for instance, [W1]), and the invariant may be made accessible by unfolding the state definitions into the operation.

It will also help us if all types defined in the specification can be decomposed into the basic types, such as Natural numbers, and Booleans, or sets, sequences and maps of basic types. Where types carry invariants, or characterising expressions, these too will then become an explicit part of the operation definition.

This formulation of *spec-OP* corresponds to the idea that, whilst we are only interested in creating tests for the operation for cases in which the precondition (and before-state invariant) is true, we wish to make use of valuable information contained in the structure of the postcondition which leads to a further decomposition into cases. This is important for a number of reasons:

- it may permit the specifier to discover, by examination of the test domains generated, combinations of input and before-state values for which the specification is not satisfiable;
- if the specification is in an advanced state of refinement, it is the structure of the postcondition that will allow us to create, in effect, a white-box test-suite;
- information in the post-condition can provide constraints on the possible results of the operation, to be used later in the validation process.

We continue by splitting the above relation into a set of smaller relations, the union of which is equivalent to the original. This splitting is achieved by transforming its propositional structure into a Disjunctive Normal Form (DNF). The particular formulation is based on a treatment of *or* in which $A \vee B$ is transformed into three disjoint cases: $A \wedge B$, $\neg A \wedge B$ and $A \wedge \neg B$. This in turn transforms $A \Rightarrow B$ into two cases: $\neg A$ and $(A \wedge B)$. This allows us to treat each sub-relation entirely independently. If the ordinary rules for *or* are used, the separate treatment of each sub-relation has to be justified by the use of an appropriate hypothesis.

To illustrate reduction to DNF, consider the following example of a simple VDM operation, *MAX*, which sets a state component *max* to the larger of its two integer arguments, *a* and *b*:

$$
\begin{array}{l}
MAX(a : \mathbb{Z},\ b : \mathbb{Z}) \\
\quad \textbf{wr}\ max : \mathbb{Z} \\
\quad \textbf{post}\ (\ max = a \vee max = b\ ) \wedge \\
\qquad\quad max \geq a \wedge \\
\qquad\quad max \geq b
\end{array}
$$

Since there is no precondition or state-invariant, the characterising expression to be normalised in this example is simply the postcondition:

$$
\begin{array}{l}
(\ max = a \vee max = b\ ) \wedge \\
\qquad max \geq a \wedge \\
\qquad max \geq b
\end{array}
$$

Distributing the $\vee$ gives:

$$
\begin{array}{l}
(\ max = a \wedge max \geq a \wedge max \geq b\ ) \vee \\
(\ max = b \wedge max \geq a \wedge max \geq b\ )
\end{array}
$$

This gives rise to the following set of three disjoint cases:

$$\{\ max = a \wedge max = b \wedge max \geq a \wedge max \geq b,\\ max = a \wedge max \neq b \wedge max \geq a \wedge max \geq b,\\ max \neq a \wedge max = b \wedge max \geq a \wedge max \geq b\ \}$$

These in turn simplify to:

$$\{\ max = a \wedge max = b,\\ max = a \wedge max > b,\\ max = b \wedge max > a\ \}$$

For each disjoint case, we have an expression which characterises a sub-relation, *sub-spec-OP*, describing a single test domain.
To summarise, and add other incidental details:

Partition Analysis of an Operation

Step 1: Extract the definition of *spec-OP*, by collecting together all the relevant parts (pre/postconditions, invariants ...).
Step 2: Unfold all definitions, to eliminate all auxiliary predicate calls, and introduce basic types where possible. (The unfolding of recursive function and type definitions has, of course, to be limited. This is discussed below.)
Step 3: Transform the definition into DNF to obtain disjoint sub-relations.
Step 4: Simplify each sub-relation, possibly splitting it into further sub-relations, by using inference rules based on the first order predicate calculus. (How this is achieved in our tool will be briefly described in Sect. 5, and in detail in [DF1].)

Our treatment of recursion is motivated by what we are trying to achieve. Transformation into DNF is akin to the path-coverage analysis of a program. Typically, where loops are encountered in the program, paths that traverse the body of the loop up to a fixed number of times are selected. By analogy, we have chosen to unfold the body of every recursive definition a fixed number of times in every different context in which a call to it appears. This allows us to exploit the propositional structure of every definition, without getting lost in infinite structures. This approach is similar to the use of regularity hypotheses in [BGM1]. Each time such a limit is placed on unfolding, the underlying hypothesis should be justified.

We now illustrate partition analysis by an example. The *Triangle Problem* is becoming a classical example for test-case generation ([N1]). This is perhaps because it gives rise to a simple formal specification, and yet requires a good number of tests to be accurately validated.

The Triangle Problem consists in determining if three lengths given as input constitute the three sides of a triangle, and if the triangle is scalene, isosceles or equilateral.

The formal specification in VDM-SL of the Triangle Problem is given as follows:

types
Triangle = $\mathbb{Z}*$
  inv $t \underline{\triangle}$ **len** $t = 3 \wedge (\forall\ i \in$ **elems** $t \bullet (2 \times i) < sum(t))$
TriangleType = SCALENE | ISOSCELES | EQUILATERAL | INVALID

```
functions
sum : ℤ* → ℤ
sum(seq) ≜ if seq = [] then 0 else (hd seq) + sum(tl seq)

classify : ℤ* → TriangleType
classify(sides) ≜ if is-Triangle(sides) then variety(sides) else INVALID

variety : Triangle → TriangleType
variety(sides) ≜
            cases card(elems sides) :
                    ⟨1⟩ → EQUILATERAL,
                    ⟨2⟩ → ISOSCELES,
                    ⟨3⟩ → SCALENE
            end

operations
CHARACTERISATION(sides : ℤ*) type : TriangleType
        post type = classify(sides)
```

The function *sum* is recursive, and we have chosen to unfold it just once in every context. However, as will be seen below, in most of the cases generated, we know the exact length of its argument, which permits us to unfold *sum* completely.

Note that the specification of the Triangle Problem contains no system state. This means that the further treatment of specifications to be discussed in the Sect. 3 is irrelevant, since it works on the system state.

For convenience, we have specified the main functionality as an operation, *CHARACTERISATION*, with no state component. After extracting and unfolding the definition of the operation, partition analysis produces 22 cases at the end of Step 3. Using the deeper knowledge of the predicate calculus at Step 4, 14 of these cases turn out to be false, due to conflicting constraints not detectable during simple reduction into DNF, such as:

$$\text{len } sides' = 3 \ \wedge \ sides' = []$$

This leaves the 8 following subdomains:

Case 1: Scalene.

```
3 = card { sides'(1), sides'(2), sides'(3) }
type = SCALENE
elems sides' = { sides'(1), sides'(2), sides'(3) }
inds sides' = { 1, 2, 3 }
sides'(2) + sides'(3) > sides'(1)
sides'(1) + sides'(3) > sides'(2)
sides'(1) + sides'(2) > sides'(3)
sides'(1) ∈ ℕ₁
sides'(2) ∈ ℕ₁
sides'(3) ∈ ℕ₁
```

Case 2: Isosceles with sides 1 and 2 equal.

```
2 = card { sides'(2), sides'(3) }
type = ISOSCELES
elems sides' = { sides'(2), sides'(3) }
inds sides' = { 1, 2, 3 }
sides'(1) = sides'(2)
sides'(2) + sides'(2) > sides'(3)
sides'(2) ∈ ℕ₁
sides'(3) ∈ ℕ₁
```

Case 3: Isosceles with sides 2 and 3 equal.

```
2 = card { sides'(1), sides'(3) }
type = ISOSCELES
```

elems $sides' = \{ sides'(1), sides'(3) \}$
inds $sides' = \{ 1, 2, 3 \}$
$sides'(2) = sides'(3)$
$sides'(3) + sides'(3) > sides'(1)$
$sides'(1) \in \mathbb{N}_1$
$sides'(3) \in \mathbb{N}_1$

Case 4: Isosceles with sides 1 and 3 equal.
$2 =$ **card** $\{ sides'(2), sides'(3) \}$
$type =$ ISOSCELES
elems $sides' = \{ sides'(2), sides'(3) \}$
inds $sides' = \{ 1, 2, 3 \}$
$sides'(1) = sides'(3)$
$sides'(3) + sides'(3) > sides'(2)$
$sides'(2) \in \mathbb{N}_1$
$sides'(3) \in \mathbb{N}_1$

Case 5: Equilateral.
$1 =$ **card** $\{ sides'(3) \}$
$type =$ EQUILATERAL
elems $sides' = \{ sides'(3) \}$
inds $sides' = \{ 1, 2, 3 \}$
$sides'(1) = sides'(3)$
$sides'(2) = sides'(3)$
$sides'(3) \in \mathbb{N}_1$

Case 6: Invalid with sides not satisfying triangle property.
$type =$ INVALID
$sides' \neq []$
$\exists i \in$ **elems** $sides' \bullet sides'(1) + sum(\mathrm{tl}\ sides') \geq 2 \times i$
elems $sides' \subseteq \mathbb{Z}$

Case 7: Invalid with wrong number of sides satisfying triangle property.

$type =$ INVALID
$3 \neq$ **len** $sides'$
$sides' \neq []$
$\forall i \in$ **elems** $sides' \bullet 2 \times i < sides'(1) + sum(\mathrm{tl}\ sides')$
elems $sides' \subseteq \mathbb{Z}$

Case 8: Invalid with no sides.
$type =$ INVALID
$sides' = []$

These 8 cases represent test domains from which test-cases can be selected, taking into account boundary cases, minimum and maximum integers, etc. We could automate the generation of such boundary cases by a further inference step in which domains containing, for instance, $x \in \mathbb{Z}$, are further partitioned by

$$0 < x \leq MAXINT \qquad x = 0 \qquad 0 > x \geq MININT$$

3 Test Sequencing

We have so far considered the treatment of operations in isolation from one another: partition analysis is used to determine cases for which individual operations should be tested. There are, however, a number of important theoretical and practical issues to be addressed in the use of formal specifications in the testing process after partition analysis. These mainly relate to the problems involved in scheduling the sequence of tests to be performed so as to achieve the right internal state of the system to be tested. These issues have implications for the design of test-beds for systems.

Our motivation is to be able to place the physical system in the states required by the test domains. One solution is to provide test-bed functions which can place the system directly into any desired state, and use the partition analysis of individual operations to test operations directly. In many systems, this just will not be practical, perhaps because it has inaccessible imported components without the necessary functionality, or perhaps due to the sheer expense or complexity involved. In these cases, it is probably only achievable by the execution of sequences of operations, which could themselves be tests. The problem therefore becomes closely related to the issue of how best to sequence the tests to avoid unnecessary repetition. Here an analysis of the formal specification can help.

Suppose that we have already performed partition analysis on individual operations, giving rise to a set of subcases of each operation, which we call sub-operations. In this we should include a partition analysis of the initial state specification, treated as an operation without a before state. What interests us is an FSA whose transitions are precisely this set of sub-operations. A complete traversal of the FSA will then give us a complete sequence of the tests we wish to perform. The states of such an FSA, then, are those in which the pre- or postcondition of at least one sub-operation is satisfied.

We propose the following method of finding the FSA:

Calculating an FSA from a State-based Specification

Step 1. Perform partition analysis on all individual operations and initial state to obtain the set of sub-operations.

Step 2. Extract from each sub-operation two sets of constraints, one describing its before state, and the other describing its after state. This is done by existentially quantifying every variable external to the state in question, and simplifying.

Step 3. Perform partition analysis, by reduction to DNF, of the disjunction of the sets of constraints found in Step 2. The resulting partitions will describe disjoint states in which at least one sub-operation either creates the state (corresponding to the post-condition) or is executable in that state (corresponding to the pre-condition).

Step 4. Step 1 has provided the transitions of the FSA, and Step 3 has provided its states. The FSA can now be constructed by resolving the constraints of sub-operations against states. A transition labeled OP is created from $S1$ to $S2$ for every sub-operation OP and every state $S1$ and $S2$ satisfying

$$(S1, S2) \in \mathit{rel\text{-}OP}$$

where $\mathit{rel\text{-}OP}$ is the relation on states defined by the constraints on OP resulting from partition analysis.

Step 5. If possible, simplify the FSA using classical FSA reduction techniques.

Since partition analysis always gives a finite number of partitions, we can always find an FSA corresponding to a specification. The result is an FSA of exactly the right level of abstraction for the task in hand. But, since it is a finite state abstraction of the, in general, infinite number of states allowable by the specification, we must allow for the FSA to be non-deterministic; that is, two arcs with the same label (same sub-operation) lead from the same before state to different after states. This non-determinacy considerably increases the complexity of solving the sequence-finding problem, and will be discussed later.

For the purposes of testing, we need to find a path through the FSA which traverses every transition with the *minimum number of repetitions.* Sequences will start with an "initial" sub-operation, which places the system into a valid initial state, as defined by the specification, and will proceed by the composition of successive sub-operations until all sub-operations have been covered. More than one sequence may be necessary, branching in different directions, to cover all of the cases. For this reason, the duplication of some tests may be unavoidable. It may not be possible to select a single path that traverses all the arcs; or it may be better not to do so when several shorter paths will do.

Since there may be several transitions for each sub-operation, this means that each sub-operation may be used several times in different contexts. This corresponds to a finer partition of operations than is obtained from the partition analysis of individual operations.

As a path through the FSA is found, a sequence of operations is formed. As each new operation is added to the sequence, the composition of resulting constraints must be resolved. These constraints may then be used to resolve non-determinism in the FSA. Where the non-determinism cannot be disambiguated as the constraints are resolved, it is because there is looseness in the specification which can only be resolved by execution of the implementation. The effect of this is that a complete path through the FSA can only be found as the tests are performed, the remaining path being recalculated after each test. For this reason, it is not obvious that classical path-finding algorithms (see for instance [ADLU1]) could be easily adapted to assist.

Similar problems arise when a test fails. The test process may suddenly find the system in an unexpected state. One option would be simply to stop, and let the error be corrected. But this is not desirable, because it fragments the test process; it is better to push on to find as many errors as possible in a single test run. In this case, a recalculation of the remaining sequence will have to take place, leaving out the erroneous case, and as many as possible of those tests already performed.

The following procedure suggests itself for sequencing tests in a complete test suite.

Performing Tests

Step 1. Using the FSA, chart several paths (or perhaps just one), from some initial state, in which every test appears a minimum number of times, but at least once, in the set of paths selected. As the paths are constructed, the composition of the constraints is resolved at each stage, to ensure a valid and unambiguous path;

Step 2. Choose test data values for all the operations in the sequence which satisfy the composed constraints.

Step 3. Perform the first (next) test, and check the result against the specification. If the test is successful, then

- if the system is in the expected state for the currently selected sequence, continue the test sequence with the next test at Step 3;

  - if the system is not in the expected state for the currently selected sequence (due to non-determinism), repeat from Step 1, assuming the current state as the initial state, omitting where possible the tests already performed.

Otherwise, if the test is unsuccessful, check the after-state of the system, and

  - if it matches the current composite constraints (*i.e.* the operation failed in relation to its result value, but left the system in the correct state), continue the test sequence;
  - if it does not satisfy the current composite constraints, but does correspond to another state in the FSA, repeat from Step 1, assuming the current state as the initial state, omitting the erroneous case, and taking into account the tests already performed;
  - else repeat from Step 1, restarting from an initial state, omitting the erroneous case, and taking into account the tests already performed;

Step 4. Repeat from Step 2 for another sequence.

In this way, the maximum number of tests can be performed in a single test run, despite the occurrence of non-determinism and failure, and the formal specification is used to a maximum in scheduling the sequence of tests.

Important issues from a testing point of view arise from this approach. It may be desirable to test the system for all possible valid sequences of operations, giving rise to testing very much like that carried out in the process algebra world for protocols [SL1], and a different analysis of the FSA. Quite independently of this, as soon as operations are chained together, possible dependencies become visible which could give rise to further test cases, or the elimination of some suggested by partition analysis of the individual operations. The latter situation will arise when it is found that it is just not possible to put the system, using the given operations, into a state appropriate for a particular test. For an example of operation dependencies, what if the result of one operation is identified with the input of the next?

The main implication of these issues for an automated testing environment based on formal specifications is that the analysis tool must be fully integrated with the testing environment. It must provide the facility of solving logical constraints during the execution of the tests, for

- finding appropriate test sequences;
- the selection of appropriate test data;
- the verification of the result of the operation;
- the comparison of the state of the physical system with abstract states in the FSA.

We should also briefly mention the problem of relating the abstract values of the specification to the concrete values of the implementation. If testing from the formal specification is to be automated, the test-bed will have to be equipped with the means of converting between these abstract and concrete values. In ideal circumstances, retrieval functions could be implemented as part of the test-bed to expose the state of the system in terms relating to the specification. Such functions may even correspond to those used during a formal development. Even if these are true functions, in the sense that a concrete value has a single abstract representation, it cannot be guaranteed that they are implementable, or even calculable. Moving in the other direction, an abstract value could have many concrete representatives (possibly an infinite number). A means of selecting an appropriate concrete value will have to be provided.

At the time of writing, the tool we have developed allows the user to compose a sequence of tests, and checks at each stage that the composition is satisfiable. An implementation of the calculation of an FSA by partition analysis is planned in the very near future. In the example that now follows, the FSA was calculated by hand, and the tool was used to verify the selected sequence by operation composition.

The example concerns a system process scheduler. At any one time, the system may have some processes *ready* to be scheduled, some processes *waiting* for some external action before they become *ready* and, optionally, a single *active* process. Each process is identified by a unique *Pid* (process identifier). There is a state invariant indicating that the two sets *ready* and *waiting* are always disjoint (a process cannot be both ready and waiting), that the active process is not ready or waiting, and that the only time there is no active process is when there are none ready to be scheduled. It also contains an auxiliary function definition, *schedule*, which characterises the scheduling algorithm very abstractly: it simply chooses any process from the provided set.

There are three operations: *NEW* which introduces a new process into the system in waiting state, *READY* which moves a waiting process into a ready state, making it active if the system is idle, and *SWAP* which swaps the currently active process for a ready one, leaving the system idle if there are no ready processes. The initial state of the system is idle and empty of processes:

```
state System of
        active  : [Pid]
        ready   : Pid-set
        waiting : Pid-set
inv mk-System(active,ready,waiting) ≜
                    ready ∩ waiting = ∅   ∧
                    active ∉ (ready ∪ waiting)   ∧
                    (active = nil) ⇒ (ready = ∅)
init mk-System(active,ready,waiting) ≜
                    ready ∪ waiting = ∅   ∧
                    active = nil
end

types
   Pid = ℕ

functions
   schedule(from:Pid-set)sel:Pid
   pre    from ≠ ∅
   post   sel ∈ from

operations
   NEW(p:Pid)
   wr waiting:Pid-set
   rd active:[Pid]
   pre     p ≠ active ∧
           p ∉ (ready ∪ waiting)
   post   waiting = waiting' ∪ { p }

   READY(q:Pid)
   wr waiting:Pid-set
wr ready:Pid-set
wr active:[Pid]
pre   $q \in waiting$
post  $waiting = waiting' - \{ q \} \wedge$
      if $active' = \mathbf{nil}$
      then
            ( $ready = ready' \wedge$
              $active = q$ )
      else
            ( $ready = ready' \cup \{ q \} \wedge$
              $active = active'$ )

SWAP()
wr active:[Pid]
wr ready:Pid-set
wr waiting:Pid-set
pre   $active \neq \mathbf{nil}$
post  if $ready' = \emptyset$
      then  ( $active = \mathbf{nil} \wedge ready = \emptyset$ )
      else  ( $active = schedule(ready') \wedge$
                    $ready = ready' - \{active\}$ ) $\wedge$
      $waiting = waiting' \cup \{active'\}$
```

The result of partition analysis on the individual operations gives the obvious set of 7 sub-operations, two for each operation, and one for the initial state:

INIT-1	$active = \mathbf{nil}$
	$ready = \emptyset$
	$waiting = \emptyset$
NEW-1(p')	$active = \mathbf{nil}$
	$ready = \emptyset$
	$waiting = waiting' \cup \{ p' \}$
	$active' = \mathbf{nil}$
	$ready' = \emptyset$
	$p' \notin waiting'$
	$p' \in \mathbb{N}$
	$waiting' \subseteq \mathbb{N}$
NEW-2(p')	$waiting = waiting' \cup \{ p' \}$
	$active' = active$
	$ready' = ready$
	$ready \cap waiting' = \emptyset$
	$active \neq p'$
	$active \notin ready$
	$active \notin waiting'$
	$p' \notin ready$
	$p' \notin waiting'$
	$active \in \mathbb{N}$
	$ready \subseteq \mathbb{N}$
	$waiting' \subseteq \mathbb{N}$
READY-1(q')	$active = q'$
	$active' = \mathbf{nil}$
	$ready = \emptyset$
	$waiting = waiting' - \{ active \}$
	$ready' = \emptyset$
	$active \in \mathbb{N}$
	$active \in waiting'$
	$waiting' \subseteq \mathbb{N}$
READY-2(q')	$active = active'$
	$ready = ready' \cup \{ q' \}$
	$waiting = waiting' - \{ q' \}$
	$ready' \cap waiting' = \emptyset$
	$active' \neq \mathbf{nil}$
	$active' \notin ready'$
	$active' \notin waiting'$
	$waiting' \subseteq \mathbb{N}$
	$active' \in \mathbb{N}$
	$q' \in waiting'$
	$ready' \subseteq \mathbb{N}$
SWAP-1()	$active = \mathbf{nil}$
	$ready = \emptyset$
	$waiting = waiting' \cup \{ active' \}$
	$ready' = \emptyset$
	$active' \notin waiting'$
	$active' \in \mathbb{N}$
	$waiting' \subseteq \mathbb{N}$
SWAP-2()	$ready = ready' - \{ active \}$
	$waiting = waiting' \cup \{ active' \}$
	$ready' \cap waiting' = \emptyset$
	$active' \notin ready'$
	$active' \notin waiting'$
	$active \in ready'$
	$active' \in \mathbb{N}$
	$ready' \subseteq \mathbb{N}$
	$waiting' \subseteq \mathbb{N}$

Partition analysis of the system state for finding the FSA gives the following 6 states:

1 $active = \mathbf{nil}$	4 $active \neq \mathbf{nil}$
$ready = \emptyset$	$ready = \emptyset$
$waiting = \emptyset$	$waiting \neq \emptyset$
2 $active = \mathbf{nil}$	5 $active \neq \mathbf{nil}$
$ready = \emptyset$	$ready \neq \emptyset$
$waiting \neq \emptyset$	$waiting = \emptyset$
3 $active \neq \mathbf{nil}$	6 $active \neq \mathbf{nil}$
$ready = \emptyset$	$ready \neq \emptyset$
$waiting = \emptyset$	$waiting \neq \emptyset$

The associated FSA is therefore as shown in Fig. 1.

This FSA cannot be reduced by classic reduction techniques. Note that it is non-deterministic in states 2, 5 and 6. In fact the choice of arc to follow is always determined by the composed constraints of the operations that lead to non-deterministic states. What has happened here, in effect, is that the partition analysis that takes into account all the operations has resulted in a finer partitioning of individual operations. This is not only manifest in the above non-determinacy, but also in the duplicated arcs for *NEW*-1 on states 1 and 2, and for *NEW*-2 on states 4 and 6, for example. For a complete test, every arc should be traversed, including the duplicates.

The sequence of operations we have chosen is given below:

init-1 ; *NEW*-1(*p*-1) ; *READY*-1(*p*-1) ; *NEW*-2(*p*-3) ; *READY*-2(*p*-3) ; *SWAP*-2 ; *READY*-2(*p*-7) ; *NEW*-2(*p*-7) ; *READY*-2(*p*-7) ; *SWAP*-2 ; *NEW*-2(*p*-7) ; *READY*-2(*p*-7) ; *SWAP*-2 ; *SWAP*-2 ; *SWAP*-1 ; *NEW*-1(*p*-1) ; *READY*-1(*p*-7) ; *SWAP*-1 ; *READY*-1(*p*-7) ; *NEW*-2(*p*-7)

This sequences traverses all of the 18 arcs in 20 transitions. The duplicated transitions are *READY*-1 between states 2 and 3, and *READY*-2 between 4 and 5.

When test data has been randomly selected, the constraints associated with this sequence of tests is as follows:

	Operation	**Constraints**
1	*INIT*()	$active\text{-}1 = \mathbf{nil} \wedge ready\text{-}1 = \emptyset \wedge waiting\text{-}1 = \emptyset$
2	*NEW*-1(15)	$active\text{-}2 = \mathbf{nil} \wedge ready\text{-}2 = \emptyset \wedge waiting\text{-}2 = \{ 15 \}$
3	*READY*-1(15)	$active\text{-}3 = 15 \wedge ready\text{-}3 = \emptyset \wedge waiting\text{-}3 = \emptyset$
4	*NEW*-2(9)	$active\text{-}4 = 15 \wedge ready\text{-}4 = \emptyset \wedge waiting\text{-}4 = \{ 9 \}$
5	*READY*-2(9)	$active\text{-}5 = 15 \wedge ready\text{-}5 = \{ 9 \} \wedge waiting\text{-}5 = \emptyset$
6	*SWAP*-2()	$active\text{-}6 = 9 \wedge ready\text{-}6 = \emptyset \wedge waiting\text{-}6 = \{ 15 \}$
7	*READY*-2(15)	$active\text{-}7 = 9 \wedge ready\text{-}7 = \{ 15 \} \wedge waiting\text{-}7 = \emptyset$
8	*NEW*-2(4)	$active\text{-}8 = 9 \wedge ready\text{-}8 = \{ 15 \} \wedge waiting\text{-}8 = \{ 4 \}$
9	*READY*-2(4)	$active\text{-}9 = 9 \wedge ready\text{-}9 = \{ 4, 15 \} \wedge waiting\text{-}9 = \emptyset$
10	*SWAP*-2()	$active\text{-}10 \in \{ 4, 15 \} \wedge ready\text{-}10 = \{ 4, 15 \} - \{active\text{-}10\} \wedge waiting\text{-}10 = \{ 9 \}$
11	*NEW*-2(67)	$active\text{-}11 = active\text{-}10 \wedge ready\text{-}11 = \{ 4, 15 \} - \{active\text{-}10\} \wedge waiting\text{-}11 = \{ 9, 67 \}$
12	*READY*-2(9)	$active\text{-}12 = active\text{-}10 \wedge ready\text{-}12 = \{ 4, 9, 15 \} - \{active\text{-}10\} \wedge waiting\text{-}12 = \{ 67 \}$
13	*SWAP*-2()	$active\text{-}13 \in \{ 4, 9, 15 \} \wedge ready\text{-}13 = \{ 4, 9, 15 \} - \{ active\text{-}10, active\text{-}13 \} \wedge waiting\text{-}13 = \{ active\text{-}10, 67 \} \wedge active\text{-}13 \neq active\text{-}10$
14	*SWAP*-2()	$active\text{-}14 \in \{ 4, 9, 15 \} \wedge ready\text{-}14 = \emptyset \wedge waiting\text{-}14 = \{ active\text{-}10, active\text{-}13, 67 \} \wedge active\text{-}14 \neq active\text{-}13 \wedge active\text{-}14 \neq active\text{-}10 \wedge active\text{-}13 \neq active\text{-}10$
15	*SWAP*-1()	$active\text{-}15 = \mathbf{nil} \wedge ready\text{-}15 = \emptyset \wedge waiting\text{-}15 = \{ 4, 9, 15, 67 \}$
16	*NEW*-1(12)	$active\text{-}16 = \mathbf{nil} \wedge ready\text{-}16 = \emptyset \wedge waiting\text{-}16 = \{ 4, 9, 12, 15, 67 \}$
17	*READY*-1(15)	$active\text{-}17 = 15 \wedge ready\text{-}17 = \emptyset \wedge waiting\text{-}17 = \{ 4, 9, 12, 67 \}$
18	*SWAP*-1()	$active\text{-}18 = \mathbf{nil} \wedge ready\text{-}18 = \emptyset \wedge waiting\text{-}18 = \{ 4, 9, 12, 15, 67 \}$
19	*READY*-1(12)	$active\text{-}19 = 12 \wedge ready\text{-}19 = \emptyset \wedge waiting\text{-}19 = \{ 4, 9, 15, 67 \}$
20	*NEW*-2(26)	$active\text{-}20 = 12 \wedge ready\text{-}20 = \emptyset \wedge waiting\text{-}20 = \{ 4, 9, 15, 26, 67 \}$

Note the looseness in the specification which is manifest in steps 10 and 13. In this example, the looseness does not lead to non-determinism in the FSA. All of the

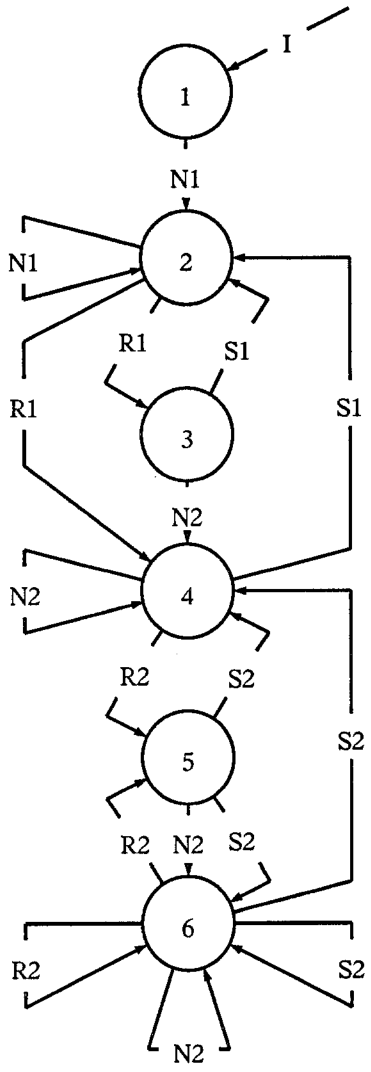

Fig. 1. The FSA created for the specification *System.*

non-determinism is resolved by composed constraints during the path construction.

4 The Tool

The test case generation tool has been written in SEPIA Prolog with KEGI, an XView interfacing tool, and amounts to about 9000 lines of code. It has been integrated in the Atmosphere VDM tool-set [L1] which includes:

- a VDM multi-font editor;

- a VDM type checker;
- "VDM through Pictures", [DL1] a tool which allows specifications to be represented as diagrams, with transformations between text and visual notations.

The tool hooks onto the back of the VDM parser to extract representations of VDM specifications. Reduction to DNF is performed in Prolog, and VDM knowledge is encoded as a set of about 200 inference rules, which are used in Step 4 of partition analysis. This specialised theorem-proving capability does not aspire to be complete, since it is handling the full first-order predicate logic. The set of inference rules may be extended to treat particular cases as needed. As the tool "gains experience" in this way, the range of problems treatable is expanded.

At the time of writing, the techniques in the paper are all implemented except the generation of FSAs and test values. In addition, we have a tool which allows sequences of operations to be composed, and their combined constraints resolved.

The tool is described in detail in [DF1].

5 Conclusions

We have presented techniques for automatic partition analysis of model-based specifications with a view to the generation and sequencing of test-cases. We have implemented a tool specialising these techniques for VDM. The main novelty lies in the use of a special DNF for partition analysis, and in the technique described for extracting an FSA from the specification.

The work is in its early stages. Many more experiments will have to be done to assess the theoretical and practical viability of the techniques. The tool in its present form cannot treat very large specifications. This is because of the size of expressions generating during reduction to DNF. There are now, however, very efficient techniques for treating this problem, which can rapidly handle huge expressions [CM1].

One limitation inherent to this approach is the fact that with VDM-like specification languages, we are working in a semi-decidable logic, which in general cannot be fully automated. This incompleteness problem, as briefly mentioned in the previous section, is handled in our tool by using an extendible set of inference rules. The implications of the incompleteness are, however, as follows:

- same test domains may not be reduced to false, resulting in test cases that will (should) always fail. This is moderated a little by the fact that the falsehood of the domain will almost certainly be detected when actual test values are substituted into the constraints;
- the partition analysis may not be as detailed as possible, since the deeper propositional structure of some test domains may not have been exploited to the full in generating further sub-domains. To notice this problem may require a considerable amount of expertise and careful observation on the part of the practitioner.

Another limitation is our treatment of recursion. Whilst being adequate for the partition analysis that we perform, it does require prior reflection and intervention from the user. The absence of a full treatment of recursion in the theorem-proving aspects of the tool will also severely restrict the use of the techniques in any more general theorem proving domain.

There are other potential advantages of partition analysis in formal specifications. The generation of test cases is for internal verification of a product, *i.e.* is the implementation correct with respect to the specification? The analysis techniques described may also useful as a means of external validation, *i.e.* is the specification correct with respect to requirements? This is because the presentation of the specification as a set of cases provides an alternative view of its implications, and can help the specifier reflect on its meaning. Moreover, the ability to generate an FSA corresponding to a specification may provide valuable insight into its dynamic aspects. Indeed, it would be interesting to link the generation of the FSA to the Operation State Diagrams used in [DL1] in the visualisation of VDM.

Another interesting line of study here, would be to look at the effect of refinement on partition analysis. Does refining a specification create a super-set of the partitions of the previous level? Can partition analysis be used to guide the refinement process?

Acknowledgements: We wish to thank Bruno Marre and Marie-Claude Gaudel for useful discussions during the early stages of this work, Pat Hall for his very encouraging letter, Dick Hamlet for his comments as a referee on a previous incarnation of the paper, and the FME'93 referees for their constructive critisms, particularly for the suggestion of the *MAX* example. Alain Faivre conducted his part of this work whilst studying for his Diplôme d'Ingénieur at the Conservatoire National des Arts et Metiers, Paris, supervised by Prof. Philippe Facon.

References

[ADLU1] Aho A. V., Dahbura A. T., Lee D., Uyar M.U., *An Optimisation Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours*, Proc. Protocol Specification, Test and Verification VIII, 1988

[BGM1] Bernot G, Gaudel M-C, Marre B, *Software Testing Based on Formal Specifications: a Theory and a Tool*, Software Eng. Journal, Nov 1991

[CM1] O. Coudert, J. C. Madre, "Towards a Symbolic Logic Minimization Algorithm", in Proc. of *VLSI Design'93*, Bombay, India, January 1993.

[D1] John Dawes, *The VDM-SL Reference Guide*, Pitman, 1991.

[D2] Tim Denvir, *Introduction to Discrete Mathematics for Software Engineering*, Mac Millan, 1986.

[DF1] Jeremy Dick, Alain Faivre, *Automatic Partition Analysis of VDM Specifications*, Research Report RAD/DMA/92027, Bull Research Centre, Les Clayes-sous-Bois, France, Oct 1992

[DL1] Jeremy Dick, Jérôme Loubersac, *Integrating Structured and Formal Methods: A Visual Approach to VDM*, Proc. ESEC'91, Milan, Springer-Verlag LNCS 550, pp.37-59, Oct 1991

[FJ1] L.M.G. Feijs, H.B.M. Jonkers. *Specification and Design with COLD-K*, Philips Research Laboratories, Eindhoven, The Netherlands.

[H1] Hall, Patrick A. V., *Towards a Theory of Test Data Selection,* Second IEE/BCS Conf. Software Engineering 88. IEE Conf. Publication Number 290. pp. 159-163. 1988

[H2] Hall, Patrick A. V., *Relationship between specifications and testing,* Information and Software Technology, Jan/Feb 1991

[HH1] Hall P. A. V., Hierons R., *Formal Methods and Testing,* The Open Univ. Computing Dept. Tech Report No 91/16, August 1991

[J1] Cliff B. Jones, *Systematic Software Development using VDM,* Second Edition, Prentice Hall Int., 1990.

[L1] J. Loubersac, *VtP Users' Guide*, Atmosphere deliverable No. I4.1.4.2.3.1, Bull Research Centre, 1992.

[N1] N. D. North, *Automatic Test Generation for the Triangle Problem*, National Physical Laboratory Report DITC 161/90, February 1990.

[P1] M. Phillips, *CICS/ESA 3.1 Experience*, Procs. Z Users' Group, Oxford, 1989.

[S1] G. T. Scullard, *Test Case Selection using VDM*, In Proc. VDM'88, LNCS 328, Springer Verlag.

[SL1] Sidhu D. P., Lenung T. K., *Formal Methods in Protocol Testing: a Detailed Study,* IEEE Trans. SE Vol 15 No 4, 1989

[S2] J. M. Spivey, *The Z Notation*, Prentice Hall, 1989.

[R1] The RAISE Language Group, *The RAISE Specification Language*, Report No. CRI/RAISE/DOC/1/v1, CRI, Denmark 1991.

[W1] J. Woodcock, M. Loomes *Software Engineering Mathematics* Pitman, 1988.