# Automatic Software Test Data Generation from Z Specifications Using Evolutionary Algorithms

XILE YANG

A submission presented in partial fulfilment of the

requirements of the University of Glamorgan/Prifysgol Morgannwg

for the degree of Doctor of Philosophy

June 1998

To My Mother, Father

and My Husband

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Xile Yang

(CANDIDATE)

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The investigation was carried out partly at the Department of Computer Studies, University of Glamorgan.

I would like to thank my supervisors Prof. B. F. Jones and Mr. D. E. Eyres for all their time, guidance and help during this investigation. I am very luck, I can always get the best teachers.

I also would like to thank the Head of Department Mr. P. Hodson for allowing me generously to use the facilities in the Department of Computer Studies.

Finally, I would like to thank all members of Department of Computer Studies, especially Dr. Jim Moon for their help.

Automatic Software Test Data Generation from Z Specifications

Using Evolutionary Algorithms

Abstract

Test data sets have been automatically generated for both numerical and string data types to test the functionality of simple procedures and a good sized UNIX filing system from their Z specifications. Different structured properties of software systems are covered, such as arithmetic expressions, existential and universal quantifiers, set comprehension, union, intersection and difference, etc. A CASE tool ZTEST has been implemented to automatically generate test data sets.

Test cases can be derived from the functionality of the Z specifications automatically. The test data sets generated from the test cases check the behaviour of the software systems for both valid and invalid inputs. Test cases are generated for the four boundary values and an intermediate value of the input search domain. For integer input variables, high quality test data sets can be generated on the search domain boundary and on each side of the boundary for both valid and invalid tests. Adaptive methods such as Genetic Algorithms and Simulated Annealing are used to generate test data sets from the test cases. GA is chosen as the default test data generator of ZTEST. Direct assignment is used if it is possible to make ZTEST system more efficient.

Z is a formal language that can be used to precisely describe the functionality of computer systems. Therefore, the test data generation method can be used widely for test data generation of software systems. It will be very useful to the systems developed from Z specifications.

# CHAPTER 1. INTRODUCTION

## 1.1. The Role of Software Testing

Nowadays, the effect of computers extends far beyond the imagination of the inventors. From scientific laboratories to ordinary family homes, computers can be seen working everywhere. They play a more and more important role in our modern daily life. In many areas computers undertake the work that is impossible for humans.

At the same time, the demand for software development is increasing greatly. Instead of being developed by one person, now a software system is likely to be developed by many people working together. Much of software development becomes a co-operative activity.

To make the software more responsive to the needs of their users and free of errors, some standard development procedures and measurement of software quality are necessary as in other branches of engineering. To qualify as a branch of engineering, software development must have a systematic way of developing software, so it is possible to compare the design with the original requirements in each stage of the development.

The common software development stages [1] are:

Step 1: Requirement analysis ---- what are the user's needs.

Step 2: Specification ---- what the software system will provide.

Step 3: Design ---- how the requirements are to be met.

Step 4: Implementation ---- the programs are developed.

Step 5: Testing ---- the functions of software system are tested.

If the software system were developed properly, there would be no bugs. In practice, errors in the software system are inevitable due to human imperfection. The statistics indicate that programming, done well, will still have of the order of one to three bugs per hundred statements [2]. Testing consumes at least half of the labour required to produce a working software system. Therefore, testing is a very important stage in software development. This has been shown by software development in many industry safety critical systems [49].

What is software testing? It was said that program testing can be used to show the presence of bugs, but never their absence [3]. The purpose of software testing is to find errors in the software system, to make the software more reliable and to increase user confidence in the software. Some of the disasters caused by software failure are notorious. Rigorous software testing can reduce risks to the public.

There are different kinds of software testing, the module level testing that is done during the software development and the system level testing that is carried out after the system is built. On the other hand, tests can be designed from a functional or a structural point of view. Structural testing is based on execution paths within the software code, it looks at the implementation details in Step 4. For many systems, structural testing is not possible because there is no access to source code. In functional testing the system

boundary value analysis and error guessing to derive test cases for the triangle problem which was written in different kinds of specifications. Partition testing that dividing the input domain into sub-domains is one of the main approach to generating test cases from a formal specification. Laycock [6] applied category partition testing to a Z specification, that is to classify the behaviour of the specification into categories and apply partition testing to each category. Hörcher and Peleska [40] demonstrated test cases derived from a Z specification of a real time system, in which the test case sequencing needs to be taken into consideration. Work has been done in the area of test template framework [36] [37] [56]. A test template framework defines test cases and their relation to the operation in a specification. It therefore gives a structure within which test case generation algorithms or manual test data selection can be applied. For Z specifications, work has been done on Z animation, that is to translate Z into a programming language (i.e. Prolog or Lisp), and then execute it [7] [8]. There are Z automatic support tools [44] [45] for building, editing, checking and viewing Z specifications, and to automatically generate test framework that can be used by test engineers to create test cases manually [37]. Specification based automatic testing methods are not well developed.

One of the most important aspects of automatic testing is automatic test data generation. Arkko, Hirvisalo, Kuusela and Nuutila [60] described a system which generates test cases from specifications that are expressed in algebraic form, while Weyuker, Coradia and Singh [61] introduced a method for automatically generating test data from a Boolean specification.

is treated as a black box shown in Fig. 1. Test data are derived by analysing the specification of the software system in Step 2. For a given input we test the output against the expected output.


Fig. 1. Black box testing.

A system analyst usually decides what the software will do based on the user's requirements without dealing with the programming details. It is also possible and necessary for a system analyst to give some test cases from the specification which will test the functions of the software system. Functional testing is a higher order testing.

After deriving test cases, test data can either be created manually, or be generated using adaptive methods. The later one is automatic testing. The purpose of this research project is to generate test data automatically from formal software specifications.

## 1.2. Related Work

According to the literature, work has been done in the area of software testing from formal specifications [5] [6] [36] [37] [40] [55] [56]. Most of the work is focused on methods of analysing the specifications and deriving test cases which can be used by test engineers to produce test data manually. North [5] used equivalence partitioning,

Random testing is the simplest test generation method whose test data can be generated easily and rapidly. In some cases, random testing can be cost effective [52]. One problem of random testing is that a large amount of tests need to be checked manually by test engineers if no automatic tool is available. As discussed by Ince and Hekmatpour [53], the number of tests that need to be checked can be reduced to one quarter after eliminating the tests which achieve the same branch coverage. Another problem for random testing is that it is not suitable for generating tests inside a small valid sub-domain from a large search domain. For the example given in [52] to classify triangles with sides of length I, J, K, random testing can be cost effective if the range of input variables is $1 \leq I, J, K \leq 5$. It will be a different story if the range of input variables becomes $0 < I, J, K \leq 65535$ due to the difficulty of finding the solution for a right angle triangle in a large search domain. In such cases, test data generation algorithms that use knowledge from the objective functions can be more effective.

In general, test data generation is an optimisation process that searches for an optimum solution of the objective functions. Classical optimisation methods such as steepest gradient can be used for generating test data, but they only work for differentiable objective functions. Gallagher and Narasimhan [51] introduced an algorithm in which penalty functions were added to ensure a continuous objective function, subsequently a quasi-Newton method was used for the optimisation process. There is a limitation for using classical optimisation methods that the derivative information must be given. There is also an overhead of computation time used for calculating the derivative information. Another problem for these classical optimisation techniques is that they may be trapped in a local optimum and fail to find the required global optimum.

Distinguished from classical methods, evolutionary algorithms such as genetic algorithms do not use derivative information from the objective functions, so there is no requirement for the objective function to be continuous. No matter what form the objective function may be in, genetic algorithms can work provided a numeric measurement is given for how close each sample test data set is to the goal. Because genetic algorithms take the sample over the whole search domain, it is less likely to be trapped in a local optimum. Jones, Sthamer and Eyres [50] applied genetic algorithms successfully to generate test data sets for structural testing. Program instrumentation statements were inserted manually into the source code to calculate the fitness values and to count the branches executed.

On the basis of the relevant work, the goal of the research project is to find an automatic method for parsing Z specifications, deriving test cases from Z and generating test data sets using evolutionary algorithms. Due to the financial circumstances of the project, the existing Z automatic support tools can not be used in the project.

## 1.3. Aim of the Research Project

The aim of the project is to undertake investigation of the feasibility of the automatic software test data generation from formal specifications, and the feasibility of using evolutionary algorithms in test data generation.

The research project includes the investigation of automatically deriving test cases from Z specifications, and generating test data sets from the test cases. As a general

approach, the testing algorithm need to be able to cover different kinds of Z constructs as well as different types of inputs and outputs.

Optimisation methods need to be used in order to automatically generate test data from the objective functions given in the test cases. The investigation in the use of evolutionary algorithms, such as Genetic Algorithms and Simulated Annealing, to automatically generate test data sets is also included in the research project. Evolutionary algorithms will be compared with random testing and other deterministic approaches such as direct assignment. The research project proposes to develop and evaluate a strategy for generating high quality test data efficiently.

An automatic software testing tool needs to be developed in the research project to implement all the above testing strategies and algorithms. The testing tool will be applied to Z specifications including a sizeable Z specification for a software system to demonstrate and verify the testing algorithm and to cover scaling of the approach.

The main stages of this work are as follows:

-- Input Z specification, parse it automatically

-- Derive test cases automatically from the functions described in Z

-- Generate test data sets using evolutionary algorithms such as GA or SA

-- Output the test suite to be used for testing the software system

The project is an interesting approach to software testing. The contribution to knowledge is embodied in the automatic software test data generation from formal

specifications and the use of evolutionary algorithms such as GA or SA to generate test data sets. It will be very useful for testing software which are developed from Z specifications.

This research project was started from February 1994. A testing software system ZTEST has been developed using C++ in PC environment to undertake the task of automatic test data generation from Z specifications. Adequate results have been achieved for both numerical and string data types [24] [25] [26] [48].

# CHAPTER 2. Z SPECIFICATION

Mathematical specifications have the virtues of being concise (brief, precise) accurate and unambiguous (not have more then one possible meaning, undoubted). They are used in many computer systems.

Z is a formal specification language, which has been developed at Oxford University in the 80s [9] [12]. Based on set theory, it uses mathematical phrases and expressions to describe the properties of a software system in a precise way.

## 2.1. Z Schemas

A Z specification is decomposed into a series of schemas that are linked in a commentary. The schemas describe both static and dynamic aspects of a system. The static aspects include:

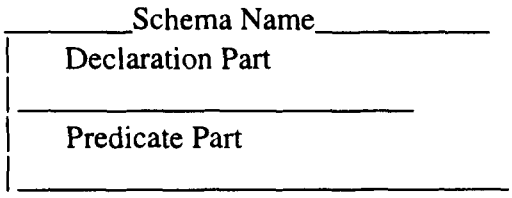
* the state that a system can occupy
* the invariant relationships which are maintained as the system moves from one state to another.

The dynamic aspects include:

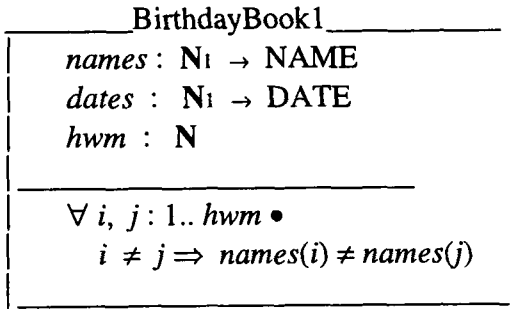
* the possible operations
* the relationships between inputs and outputs

* the possible changes of state.

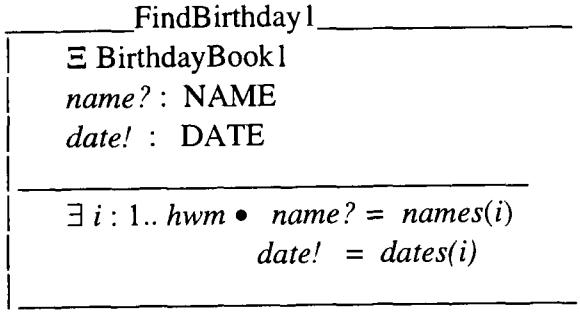
The schema language allows facets of a system to be described separately and then combined and related. In general, a schema contains a declaration part and a predicate part:




The birthday book [9] will be used as an example. The state space is given as:




In the upper declaration section of the schema three state variables are declared. Variable *names* and *dates* are arrays described by functions from the set $\mathbf{N}_1$ of strictly positive integers to NAME or DATE, while *hwm* is the length of the arrays. The predicate section says that there are no repetitions in the array *names*. An operation to find a person's birthday is described by the following schemas:



FindError

$\Xi$ BirthdayBook1

*name?* : NAME

*reply!* : MESSAGE

---

$\forall i : 1..hwm \bullet name? \neq names(i)$

*reply!* = Name not found

There is an input variable *name?* and an output variable *date!* in FindBirthday1. If there exists a *names(i)* which is the same as the input *name?*, then output *date!* will be *dates(i)*. This operation does not change the state. Similarly, schema FindError gives an error message using the universal quantifier.

Because nearly all of Z is drawn from standard mathematics, set and logic theory, it means that Z can be widely used to describe the constructions and functions of most computer systems. Z specifications have been used for specifying the functionality and for manual test generation of many critical software systems [10] [33] [36] [37] [39] [41]. This is why it is chosen in this research project.

## 2.2. The Input Format of Z Specifications

Most of the symbols and operators in Z specifications are standard mathematical symbols and operators coming from set and logic theory. Input variable names in Z are followed by a question mark, while output variable names are followed by an exclamation mark. State variable names before and after the operation are distinguished by a decoration symbol prime such as *statename* and *statename'* in Z.

There are many symbols and operators in Z that can not be input into computer directly. In order to allow the testing system ZTEST to read Z specifications, it is necessary to restrict the writing style of Z input files. An input format of Z has been defined.

The input table of Z symbols and operators used in the project is given in Appendix 1. Some of them are from IBM UK Laboratories[11]. Part of the input form of symbols and operators is shown in Table 1.

**Table 1. Symbols in Z Specifications**

| Name | Symbol | Input |
|---|---|---|
| integers | $\mathbb{Z}$ | &Int. |
| integer division | $\div$ | &div. |
| numeric comparison | $\leq$ | &leq. |
| | $\neq$ | &neq. |
| | $\geq$ | &geq. |
| and (conjunction) | $\wedge$ | &and. |
| or (disjunction) | $\vee$ | &or. |
| not (negation) | $\neg$ | &not. |

For the schemas in Z, an input format is defined for the testing system ZTEST. As an example, a schema PhoneBook can be written as follows:

```
_____PhoneBook_____
|  index? : N
|  num! : N
|  record : N → N
|-------------------
|  num! = record(index?)
|___________________
```

The input form of the schema is:

```
&sname. Phonebook
&bsdec.
   index? : &Nat.
   num!  : &Nat.
   record : &Nat. &tfun. &Nat.
&esdec.
&bspre.
   num! = record(index?)
&espre.
```

Where the string "&sname." introduces a schema name. The strings "&bsdec." and "&esdec." indicate the beginning and the end of schema declarations. The strings "&bspre." and "&espre." indicate the beginning and the end of schema predicates. The string "&Nat." represents the built-in data type of natural numbers N in Z.

In Z the whole system is defined by a commentary, a schema definition in horizontal form, which links all schemas together. For example, a triangle classifier can be defined as:

$$\text{Triangle} = \text{Triangle0} \vee \text{Error}$$

where Triangle, Triangle0 and Error are schema names. The input form for this definition is

```
Triangle &sdef. &lsch. Triangle0 &or. Error &rsch.
```

Where string "&sdef." introduces the schema definition (horizontal form). The strings "&lsch." and "&rsch." represent the left and the right brackets which hold the schema definition.

## 2.3. The Representation of Z Expressions

### 2.3.1. Definition of Given Set

Variables in Z can be defined as the data type of a given set. In Z, a given set is defined in a very general way using English descriptions. In the specification for birthday book, two given sets can be defined as following:

[NAME] the set of all possible names

[DATE] the set of all dates

Then a variable *names* can be defined to hold the record of names in the birthday book.

$$names : \mathbb{N}_1 \rightarrow NAME$$

This kind of definition for a given set is not practical for the system of automatic testing. In the project, given sets are defined in a more precise way. The type and the subset of the type are declared for the given set. As the search domain of the variable is been clearly defined, the process of test data generation can be carried out within this search domain. There are built-in data types in the system: &Int. for integer data type, &char. for character data type, and &alpha. for alphabetic data type.

For the above example, the definitions of these given sets in the input file are:

```
&lsqb. NAME &rsqb.
&bsetd.
    NAME : &seq. &alpha.
    # NAME = 15
&esetd.

&lsqb. DATE &rsqb.
&bsetd.
    DATE : &seq. &Nat.
```

```
    # DATE = 2
    DATE(1), DATE(2) > 0
    DATE(1) &leq. 31
    DATE(2) &leq. 12
&esetd.
```

So the given set NAME is defined as a sequence of alphabetic letters with the length of 15. Given set DATE is defined as a sequence of natural numbers with the length of 2. The first element of DATE is less than or equal to 31, which represents the date. The second element of DATE is less than or equal to 12, which represents the month. Both elements of DATE are larger then 0.

The definition of a given set can be nested to form multi-dimensional given sets. For example, a one dimensional given set SYL is defined as:

[CHAR] the set of all characters

SYL : seq CHAR

then a two dimensional given set NAME can be defined as:

NAME : seq SYL

These given set can be used to define variables of different dimensions in Z as the follows.

One dimensional variable *v1* can be defined as

*v1* : seq TYPE (where TYPE can be integer, char .... ), or

*v1* : SET_1D (where SET_1D can be any one dimensional sets).

Two dimensional variable *v2* can be defined as

*v2* : seq SET_1D (where SET_1D can be any one dimensional sets), or

*v2* : SET_2D (where SET_2D can be any two dimensional sets).

Three dimensional variable *v3* can be defined as

*v3* : seq SET_2D (where SET_2D can be any two dimensional sets).

2.3.2. Conditional Expression

In Z, conditional expressions

IF expression 1 THEN
    expression 2
ELSE
    expression 3
END IF

can also be presented using conjunction and disjunction. The above expressions can be written in Z specification as:

(expression 1 $\wedge$ expression 2) $\vee$
expression 3

The conjunction at the end of a line is optional as shown in the following example. Expressions such as

expression 1
expression 2

are the same as

expression 1 $\wedge$
expression 2

2.3.3. Existential Quantifier

**Syntax** ∃ D | C • P

**Description**

The existential quantifier is true if there exist at least one way of giving values to the variables introduced by the declaration D so that both the constraint C and the predicate P are true.

**Example** ∃ i : 1..m | S(i) ≠ 10 • x? > S(i) / (S(i) - 10)

**Input** The input form of the example is as follows:

```
&exi.
    i : 1 .. m
    &cbar. S(i) &neq. 10
    &bul. x? > S(i) / (S(i) - 10)
&eexi.
```

Where strings "&exi." and "&eexi." indicate the beginning and the end of the existential structure. The string "&cbar." introduces the constraint, while string "&bul." introduces the predicate.

2.3.4. Universal Quantifier

**Syntax** ∀ D | C • P

**Description**

The universal quantifier is true if, for all values taken by the variables introduced by the declaration D, both the constraint C and the predicate P are true.

**Example** ∀ i : 1..hwm • name? ≠ names(i)

**Input** The input form of the example is as follows:

```
&all.
    i : 1 .. hwm
    &bul. name? &neq. names(i)
&eall.
```

Where strings "&all." and "&eall." indicate the beginning and the end of the universal structure. The string "&cbar." introduces the constraint, while string "&bul." introduces the predicate.

2.3.5. Set Operations

**Assignment**

**Syntax** S = P

**Description** All members of the set S are the same values as in set P.

**Example** name? = { Mary}

**Input** name? = &lset. Mary &rset.

Where strings "&lset." and "&rset." indicate the left and the right of the set.

**Overriding**

**Syntax** $S = P \oplus G$

**Description** The set S is the same as set P for all values which are not in the domain of G, and is the same as set G for all values which are in the domain of G.

**Example** $names = names \oplus \{ hwm \mapsto name? \}$

**Input** names = names &fovr. &lset. hwm &map. name? &rset.

**Set Comprehension**

**Syntax** S = { D | C • E }

**Description** The members of the set S are the values taken by the expression E when the variables introduced by the declaration D take all possible values which make the constraint C true.

**Example** cards! = { i : 1 .. hwm | dates(i) = today? • names(i) }

**Input**

```
cards! = &setc.
          i : 1 .. hwm
         &cbar. dates(i) = today?
         &bul. names(i)
       &esetc.
```

Where strings "&setc." and "&esetc." indicate the beginning and the end of the set comprehension structure. The string "&cbar." introduces the constraint, while string "&bul." introduces the predicate.

2.3.6. Representation

In order to automatically derive objective functions for test cases from Z, a way of representing Z expressions is needed. Three arrays are used to store information about Z expressions. Assuming a Z expression has the form of

$$f1(x1, .. xn)\ \Theta f2(x1, .. xn)$$

in a n-dimensional search domain, the symbol $\Theta$ could be <, =, >, <=, >= or $\neq$. A two-dimensional integer array funct_1 is defined as:

funct_1[MAXEXP][MAXNUM]

Where MAXEXP is the maximum number of expressions in Z, MAXNUM is the maximum number of items in an expression. An item in the expression could be a constant, a variable or an operator.

In array funct_1, every row keeps the information for a Z expression. In a row of the array, every pair of elements represents an item in the expression. For the elements with even index numbers (funct_1[j][0], funct_1[j][2], ..), their values can represent different kind of items:

the item of a constant

the item of a variable

the item of an operator

For the elements with odd index numbers (funct_1[j][1], funct_1[j][3], ..), their values can represent the value of a constant integer, the index number of a variable or the index number of an operator (+, -, *, /, ..) according to the values of the previous elements. There is a similar array funct_2 for the right side of the expression. Another array funct_smb[MAXEXP] stores the index number of the symbol (<, =, >, <=, >=, ≠) between the two parts of the expression.

For example, the first expression x? > 3 in a Z schema can be represented in the following way:

```
funct_1[0][0] = INPVAR   ---- an input variable
funct_1[0][1] = 0        ---- the first input variable (x?)
funct_2[0][0] = CONINT   ---- a constant integer
funct_2[0][1] = 3        ---- the value of the constant integer
funct_smb[0] = IDGRE     ---- the symbol  greater than >
```

If the nth expression is in the mth existential structure, the first two elements in array funct_1 are given as

```
funct_1[n][0] = EXISTS    ---- the identifier of existential structures
funct_1[n][1] = m         ---- the index of the existential structure
```

Similarly, the identifiers for the universal structures and the set comprehension are FORALL and SETCOM, respectively. For the variables of two dimensional sequences, the sentinel of a constant character with value 8 is used to separate the sub sequences. For example, assignment names = { Mary, John, ..} is represented as:

```
funct_2[n][0] = CONCHA    ---- a constant character
funct_2[n][1] = 77        ---- the ASCII value of 'M'
funct_2[n][2] = CONCHA    ---- a constant character
funct_2[n][3] = 97        ---- the ASCII value of 'a'
funct_2[n][4] = CONCHA    ---- a constant character
funct_2[n][5] = 114       ---- the ASCII value of 'r'
funct_2[n][6] = CONCHA    ---- a constant character
funct_2[n][7] = 121       ---- the ASCII value of 'y'
funct_2[n][8] = CONCHA    ---- a constant character
funct_2[n][9] = 8         ---- a sentinel
funct_2[n][10] = CONCHA   ---- a constant character
funct_2[n][11] = 74       ---- the ASCII value of 'J'
.....
```

and so on.

Using these arrays, the testing system ZTEST can parse descriptions in Z automatically. By analysing these arrays, objective functions can be formed to allow adaptive algorithms being applied upon different criteria. It makes the process of test data generation from Z specifications more robust.

# CHAPTER 3. THE APPLICATION OF EVOLUTIONARY ALGORITHMS TO SOFTWARE TESTING

## 3.1. Test Case Derivation

There are different kinds of methods used in software testing. Only functional testing is employed in the project. Functional testing is a process of attempting to find discrepancies between the program and its external specification. An external specification is a precise description of the program's behaviour from the point of view of the outside world. It describes the relation between input and output variables and constraints on inputs. To perform a functional test, the specification is analysed to derive test cases by using knowledge of dependencies among the variables in the specification. The testing methods given later are especially relevant to functional testing. A sufficient amount of attention should be focused on invalid and unexpected input conditions. The purpose of the functional testing is to expose errors, not to demonstrate that the program matches its external specification.

### 3.1.1. Test Methodologies

The test methodologies suggested by Glenford J. Myers[14] for functional testing are:

* Equivalence partitioning.
* Boundary value analysis.
* Error guessing.

Equivalence partitioning

Equivalence partitioning is a process of dividing possible inputs into equivalence classes, each of which should be processed in the same way by the system under test, and then writing tests to cover the various classes, for example, the class of valid input and the class of invalid input. Fig. 2 shows a domain of inputs S and its two sub domain S1 and S2 for different functions of a software system.

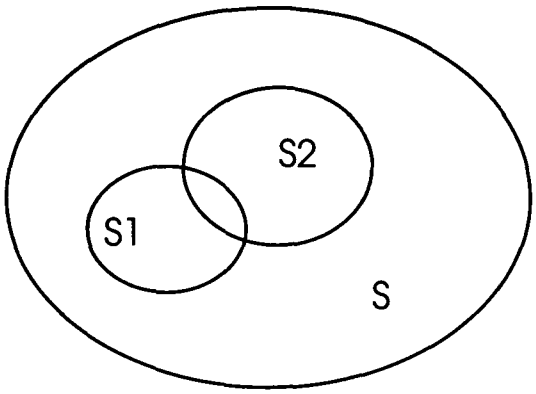



Fig. 2. Input domain S and sub domains S1, S2.

One test may cover many valid equivalence classes, but each invalid class should be covered by a separate test. This can prevent combinations of invalid inputs cancelling each other.

Boundary value analysis

Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not [14]. Boundary values are those directly on, above, and beneath the boundary of an equivalence class such as S1 in Fig. 2. They are more likely

to produce errors than others. It is the same for the input values that produce output on or near the boundary of an equivalence class. If the input is an integer, then the values which should be tested are MININT-1, MININT, MAXINT, MAXINT+1, where MININT and MAXINT are the minimum and the maximum integers available, respectively.

Error guessing

The idea of error guessing is to use your skill and judgement to write tests that are likely to uncover errors. This may be done by experience, guessing the implementation method or inspect the specification to see which parts seem most complex. It is the most subjective part of the test generation method. Error guessing can be used in two ways:

1. Building knowledge of "interesting" features of data types into the test generator.
2. Annotating the specification to indicate that certain input values are interesting.

There are many other kinds of testing methods. If these methods could be used properly, test data would be generated to cover the complete functions of the software to be tested.

### 3.1.2. Definition of Search Domain

Functional testing is to test the inputs related to its outputs in testing a specified part of the functionality of the software. So the testing is based on input variables. From the declaration part of Z specifications, the type of variables is known. The domain of test data generator is within the valid data type of input variables. For example, the search domain for an input variable *x?* of integer type will be

$$x? \in [-32768, 32767]$$

in 16 bit environment. The invalid data are given as the following:

Different types of input data

For integers, invalid types are character and real.

For characters, invalid types are integer and real.

Different number of input parameters

Empty input.

The number of input parameters + 1.

The number of input parameters - 1.

Boundary values

For integers, the invalid test data are MAXINT+1 and MININT-1. The valid test data are MAXINT and MININT.

For characters, the valid search domain is ch? $\in$ [32, 126] of ASCII values. The valid test data are 32(' ') and 126('~'), the invalid values are 31 and 127.

For alphabetic type, the valid search domain is alpha? $\in$ [ A, z]. The valid test data are 'A'(65) and 'z'(122), the invalid values are '@'(64) and '{'(123). The gap between 'Z'(90) and 'a'(97) are ignored here.

In the case of further restrictions on variable values, the search domain is defined in a subset of the data type. If variable x?, y? and z? are defined in Z as:

```
x?, y?, z? :  Z
x?, y?, z?  >= -200
x?, y?, z?  <=  200
```

then the search domain will be x?, y?, z? $\in$ [-200, 200], with -201 and 201 as invalid data.

Test cases are generated automatically by the testing system ZTEST to cover boundary values and all invalid data types, include a test case for empty input. For the above example, the test data set generated are

```
Test Case 1   Invalid { }
Test Case 2   Invalid { 68   22}
Test Case 3   Invalid {  6    8    99   85}
Test Case 4   Invalid {  I    O     v}
Test Case 5   Invalid { 0.6  0.29 0.14}
Test Case 6   Invalid {-201 -201 -201}
Test Case 7   Invalid { 201  201  201}
Test Case 8   Valid   {-200 -200 -200}
Test Case 9   Valid   { 200  200  200}.
```

For alphabetic string alpha? of the length three, test cases are given as

```
Test Case 1   Invalid { }
Test Case 2   Invalid { L   h}
Test Case 3   Invalid {  k  T   j   p}
Test Case 4   Invalid {  8   75   34}
Test Case 5   Invalid { 0.6  0.29 0.14}
Test Case 6   Invalid { @   @   @}
Test Case 7   Invalid { {   {   { }
Test Case 8   Valid   { A   A   A}
Test Case 9   Valid   { z   z   z }.
```

### 3.1.3. Coverage of Paths

All functional information of the software system to be tested is described in its Z specification. An analysis of Z yields a tree structure describing the full functionality of the software system. By following this tree, a complete test data set can be generated to cover every function of the tested Z specification.

Conjunction

For Z expressions related by conjunction, there is a single route to link these expressions together. For the form of Z:

| | |
|---|---|
| expression 1 | (1) |
| expression 2 | (2) |
| expression 3 | (3) |

the route will be (1, 2, 3).

Disjunction

For Z expressions related by disjunction, there are different routes in the tree for every branch of the specification. For the form of Z:

| | |
|---|---|
| (expression 1) $\vee$ | (1) |
| (expression 2) $\vee$ | (2) |
| (expression 3) | (3) |

the three routes in the tree will be either (1) or (2) or (3), respectively. This method can also be used when the system has a mixed combination of conjunction and disjunction. For example, the expressions are given in the form of

| | |
|---|---|
| expression 1 | (1) |
| expression 2 | (2) |
| (expression 3 $\wedge$ expression 4) $\vee$ | (3), (4) |
| (expression 5) | (5) |

There are two routes in the tree shown in Fig 3.

route 1: (1, 2, 3, 4)
route 2: (1, 2, 5)

So every branch is covered by the tree structure.

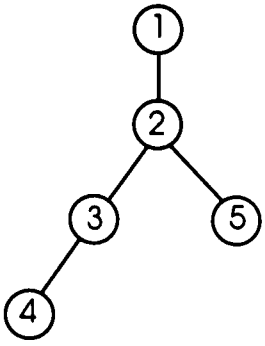



Fig. 3. Tree structure.

### 3.1.4. High Quality Test Data

Consider the objective function

$$x?, y? : \mathbf{Z} \qquad x? + y? > 10$$

Any two integer values that the sum of which is larger than 10 can be used as valid test data for x? and y?. On the other hand, test data on and near the search domain boundary are considered as high quality test data due to the higher possibility of error occurrence on the boundary. For the above objective function, test data set can be generated as:

| | |
|---|---|
| x? + y? = 11 | Valid |
| x? + y? = 10 | Invalid |
| x? + y? = 9 | Invalid |

where the successor and predecessor are used in each side of the search domain boundary. In order to get high quality test data three test cases are generated for the value itself, the successor and the predecessor as shown in Table 2.

For the objective function x? + y? <= z?, the three test cases are

| | | | |
|---|---|---|---|
| Test Case 1 | Valid | { 11 17 28} | ---- x? + y? = z? |
| Test Case 2 | Valid | { 43 12 56} | ---- x? + y? = z? - 1 |
| Test Case 3 | Invalid | { 8 26 33}. | ---- x? + y? = z? + 1 |

**Table 2. High Quality Test Data**

| Expression | Branch | State |
|---|---|---|
| f1 = f2 | f1 = f2 | Valid |
| | f1 = succ(f2) | Invalid |
| | f1 = pred(f2) | Invalid |
| f1 < f2 | f1 = pred(f2) | Valid |
| | f1 = f2 | Invalid |
| | f1 = succ(f2) | Invalid |
| f1 > f2 | f1 = succ(f2) | Valid |
| | f1 = f2 | Invalid |
| | f1 = pred(f2) | Invalid |
| f1 <= f2 | f1 = pred(f2) | Valid |
| | f1 = f2 | Valid |
| | f1 = succ(f2) | Invalid |
| f1 >= f2 | f1 = succ(f2) | Valid |
| | f1 = f2 | Valid |
| | f1 = pred(f2) | Invalid |
| f1 ≠ f2 | f1 = f2 | Invalid |
| | f1 = succ(f2) | Valid |
| | f1 = pred(f2) | Valid |

For some specific problems the successor or predecessor may not exist. For the fitness function B*B - 4AC = 0, the predecessor B*B - 4AC = -1 does not exist when A, B and C are integers which means that there is no integer N, for which N*N + 1 can be divided by 4 with no remainder. If the predecessor could not be found after certain number of attempts in ZTEST, the predecessor of the predecessor is used, until the nearest answer is found. The same rule is applied on the successor as well.

The three test cases for Quadratic Equation B*B - 4AC = 0 are:

| | | |
|---|---|---|
| Test Case 1 | Valid { -12 -24 -12} | ---- B*B - 4AC = 0 |
| Test Case 2 | Invalid { 94 -93 23} | ---- B*B - 4AC = 1 |
| Test Case 3 | Invalid { -7 19 -13} | ---- B*B - 4AC = -3 |

The successor or predecessor may not exist when the solution is on the boundary of the search domain. Only two test cases are given by ZTEST when the test data is detected

as the minimum value of the search domain, one test case is for the solution itself and the other one is for its successor. The test case for the predecessor which is out of the search domain is omitted. Similarly for the solution which is the maximum value of the search domain, the test data for the successor is omitted.

In the literature, there is no mention of the boundary values of character and alphabetic data types. Usually test cases are given for checking the different length of strings, the maximum length, the length of 1, etc.

In programming practice, any character is processed according to its ASCII value in a similar way with integers. For structural testing, test cases for the successor and predecessor of the ASCII value of the character can help to detect the jumps of the program to the right routines. On the other hand, the functional testing in the project does not deal with programming details. Only one valid test case is given for character and alphabetic data types, as well as for some structures such as the universal quantifiers and the existential quantifiers in which cases successors and predecessors are not meaningful.

### 3.1.5. Functional Testing

The purpose of functional testing is to generate test data set in order to investigate the performance of different functions in the software. For every function of the software, test data are given to check the correctness of this function.

A part of the Z specification for the Triangle Classifier [5] is shown below, where x?, y? and z? are integer inputs representing the sides of a triangle.

| | |
|---|---|
| $x? + y? > z?$ | (1) |
| $y? + z? > x?$ | (2) |
| $z? + x? > y?$ | (3) |
| $(x? \neq y? \wedge y? \neq z? \wedge z? \neq x?) \vee$ | (4), (5), (6) |
| $(x? = y? \wedge x? \neq z?) \vee$ | (7), (8) |
| $(x? = y? \wedge x? = z?) \vee$ | (9), (10) |
| $(x? * x? + y? * y? = z? * z?)$ | (11) |

There are four routes in the tree to cover all these eleven expressions as shown in Fig. 4.

route 1: (1, 2, 3, 4, 5, 6)
route 2: (1, 2, 3, 7, 8)
route 3: (1, 2, 3, 9, 10)
route 4: (1, 2, 3, 11)

Route 1 gives the definition of a Scalene Triangle, and the rest of them define the Isosceles, Equilateral and Right Angle Triangle, respectively. Four test cases are needed to check these four functions. For each function, an objective function is formed to give the criteria of test data generation. Each objective function consists of the conjunction of all expressions from the route related to the function.

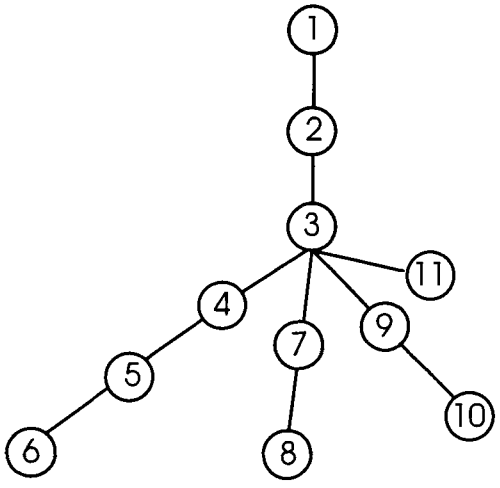



Fig. 4. Tree structure of the triangle classifier.

For example, the objective function for Isosceles Triangle is:

$$x? + y? > z? \land y? + z? > x? \land z? + x? > y? \land x? = y? \land x? \neq z?$$

Three test cases can be generated for each function test, which are on and near the boundary of search domain. For the objective function of Isosceles Triangle, the search domain is defined as:

$$\begin{aligned} & x?, y?, z? \in [1, \text{MAXINT}] \land \\ & x? + y? > z? \land \\ & y? + z? > x? \land \\ & z? + x? > y? \land \\ & x? = y? \land \\ & x? \neq z? \end{aligned}$$

The three test cases are:

Test Case 1 Valid:
$$\begin{aligned} & x? + y? > z? \land \\ & y? + z? > x? \land \\ & z? + x? > y? \land \\ & x? = y? \land \\ & x? = z? + 1 \end{aligned}$$

Test Case 2 Valid:
$$\begin{aligned} & x? + y? > z? \land \\ & y? + z? > x? \land \\ & z? + x? > y? \land \\ & x? = y? \land \\ & x? = z? - 1 \end{aligned}$$

Test Case 3 Invalid:
$$\begin{aligned} & x? + y? > z? \land \\ & y? + z? > x? \land \\ & z? + x? > y? \land \\ & x? = y? \land \\ & x? = z? \end{aligned}$$

3.1.6. Branch Testing of the Functionality

As described before, the routes in a tree structure will cover all functions given in the Z specification. A route is a control flow of the software under the condition that all nodes (expressions) are true. There should be branches for false conditions in all nodes.

Branch testing is to generate test data set which covers all functionality of the software. The strategy of branch testing used in the project is given below:

Following the routes of the tree structure derived from Z, valid test data are generated for each node which is on the boundary of the search domain. Two test cases are given on each side of the boundary if the high quality test data are used. For the nodes previous to the tested node in the route, only valid cases are considered. The nodes following the test node are neglected. Thus, every possible situation is covered by the test data set.

As an example of branch testing, the three objective functions of node 2 in the Triangle Classifier are:

$$x? + y? > z? \wedge y? + z? = succ(x?)$$
$$x? + y? > z? \wedge y? + z? = x?$$
$$x? + y? > z? \wedge y? + z? = pred(x?)$$

The first function gives valid test data, the second and the third one give invalid test data.

### 3.1.7. Classes of Objective Functions

Besides input variables, there are three other kinds of variables which the test data generator need to deal with. They are output variables that send out relevant messages; state variables that set up or change state parameters of the system and iteration variables that set the value of iteration counter in iteration structures. The states of these variables are considered in the process of test case derivation.

## 3.2. Automatic Test Data Generation

The aim of test data generation is to search the desired solution from the input domain. There are two ways of generating test data: direct assignment or adaptive automatic generation methods. They are used for different types of objective functions in ZTEST to make the system more efficient. The direct assignment in ZTEST automatically calculates the value on the right side of the objective function and assign the value to the input variable on the left side of the objective function. For the objective functions such as

$$x? = 5$$
$$name? = names(3)$$

direct assignment is used to generate the test data of input variables. For the objective functions such as

$$x1?*x1? + x2?*x2? = x3?*x3?$$
$$name? \neq names(i)$$

the automatic test data generator, adaptive methods such as genetic algorithm(GA) or simulated annealing(SA), are used to generate the test data set.

### 3.2.1. Genetic Algorithms

Genetic algorithms are a type of stochastic search technique that is modelled on the process of natural evolution of biological genes [16][17]. A genetic algorithm is an optimisation method, that is a method of finding the best answer to a problem based on the feedback from its repeated estimates to a solution. The judgement of the estimated

solutions is based on a "fitness function". A GA does not know how to get the solution of a problem, but it does know how close (how fit) it is to a better solution from the fitness function, and it can work toward making fitter solutions.

The procedure of a GA is:

1. Find a chromosomal representation of solution to the problem, usually bit strings.
2. Generate an initial population of solutions.
3. Design a fitness function to evaluate the fitness of a solution.
4. Use the fitness to decide how likely a solution is to reproduce, and select reproduction pairs according to their fitness.
5. Apply genetic operators to the pairs to create "offspring" solutions. The most often used operators are crossover which exchanges a randomly selected segment between the pair, and mutation which randomly alters several bits of the solution.
6. Replace the weakest solutions with the offspring to form a new generation, until the best solution is found.

Now, genetic algorithms are widely used in many engineering branches as an optimisation method. As some other adaptive methods, it has the advantage that it does not need to know the differential form of the objective functions.

Genetic algorithms are selected as one of the generators to automatically generate test data set. The current genetic algorithm used in ZTEST is introduced below.

3.2.1.1. Representation of Solution

At first, a representation is chosen to describe solutions for the problem. As usual, a binary string is used. A variable of integer type can be represented as a binary string. The high-order bit is used as the sign bit. If the sign bit is 0, then the integer is positive; if it is 1, then the integer is negative. A negative number is represented by two's complement arithmetic, that is having all bits reversed and one is added to the number. For example, -1 in binary is 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 in 16-bit. The two's complement form of negative numbers is converted to the ordinary binary form in ZTEST. That is -1 in binary is 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 in 16-bit.

For character type, ASCII values can be used to represent the variable, or an 8-bit string. For example, character A in bit string is 0 1 0 0 0 0 0 1. Therefore, after the types of variables are defined, a binary string can be formed to represent them in GA process.

The operators of GA are applied to a population of chromosomes, that is a population of bit strings. The size of population N remains the same during the process. A typical size of population is usually between 50 and 200 [19]. At present, the size of population is set to the length of the binary string:

$$N = \text{String Length}$$

The array of chromosomes is usually a square matrix. Therefore, a problem having three integer variables will have the size of population 48.

Considering the limitation of PC memory, this rule of deciding N may be not practical for more complicated problems with many variables. An experiment has been done to investigate the effect of reduced population size N. Genetic algorithms are applied to minimise the function f(x1, x2) = x1 - x2 - 1 in different population size, results are shown in Table 3 for 50 runs.

**Table 3. Comparison of GA Results for Different Population Size**

| Size of Population | Total Generations | Average Generations |
|---|---|---|
| 32 | 156 | 3.12 |
| 16 | 258 | 5.16 |

When population size N is reduced to half of the string length, average generations to get the answer is increased. Considering the execution time will also be reduced with smaller N, it is still a practical approach to reduce N for complicated problems.

3.2.1.2. Evaluation of the Fitness

In GA process, a new generation of chromosomes is selected according to their fitness. The fitness of a chromosome is evaluated using the fitness functions. Two kinds of evaluation are used in ZTEST.

Comparison

Assume an objective function of the form

$$f1(x1, .. xn)\ \Theta f2(x1, .. xn)$$

in a n-dimensional search domain, where the symbol $\Theta$ could be <, =, >, <=, >= or $\neq$. The fitness function fit($x1, .. xn$) is chosen as follows according to the native of $\Theta$ :

$f1(x1, .. xn) = f2(x1, .. xn)$, fit $= \dfrac{1000.0}{abs(f1 - f2) + 10}$

$f1(x1, .. xn) \langle f2(x1, .. xn)$, if $(f1 \langle f2)$ then fit = 100.0

else fit $= \dfrac{1000.0}{f1 - f2 + 11}$

$f1(x1, .. xn) \rangle f2(x1, .. xn)$, if $(f1 \rangle f2)$ then fit = 100.0

else fit $= \dfrac{1000.0}{f2 - f1 + 11}$

$f1(x1, .. xn) \langle= f2(x1, .. xn)$, if $(f1 \langle f2)$ then fit = 100.0

else fit $= \dfrac{1000.0}{f1 - f2 + 10}$

$f1(x1, .. xn) \rangle= f2(x1, .. xn)$, if $(f1 \rangle f2)$ then fit = 100.0

else fit $= \dfrac{1000.0}{f2 - f1 + 10}$

$f1(x1, .. xn) \neq f2(x1, .. xn)$, if $(f1 \neq f2)$ then fit = 100.0

else fit $= \dfrac{1000.0}{11} = 90.9$

Using these fitness functions, the fitness of chromosome is kept between 0.0 and 100.0. When a chromosome satisfies the objective function, it gets the fitness value 100.0, the criteria have been satisfied and the GA search will be stopped.

Hamming Distance

Hamming distance is a way of quantifying the difference between two bit strings. It counts the number of different bits between the two strings. If the two strings are identical, then Hamming distance is 0.

The fitness function using Hamming distance is

$$f1(x1, .. xn) = f2(x1, .. xn), \qquad \text{fit} = \frac{1000.0}{\text{Hamming}(f1, f2) + 10}$$

where Hamming($f1$, $f2$) is the Hamming distance between bit string $f1$ and $f2$. The value of fitness is still in the range of (0.0, 100.0), with 100.0 as the best solution for the objective function. For other cases, the successor or predecessor is used in Hamming distance as shown in Table 2, section 3.1.4. In this way, high quality test data near the search domain boundary can be found.

For the fitness function that is a conjunction of multiple sub functions, a chromosome will satisfy the fitness function only if it can satisfy everyone of the sub functions. To make the GA search efficient, the evaluation process will be stopped if a chromosome fails any one of the functions.

3.2.1.3. Penalty Function

In GA process, the search for the solution should be kept inside the valid domain of the input variables. To guarantee that all chromosomes chosen in each generation of GA search are in the valid domain, a penalty function $\mathbf{P}(x)$ is added to the fitness functions fit($x$). The new fitness function becomes:

$$\text{fitness}(x) = \text{fit}(x) + \mathbf{P}(x)$$

The penalty function is a threshold function given as

$$\mathbf{P}(x) = \begin{cases} 0 & \text{when } x \in \mathbf{D} \\ -\text{fit}(x) - 1 & \text{when } x \notin \mathbf{D} \end{cases}$$

where **D** is the valid search domain. By using the penalty function, any chromosome outside the search domain gets a low fitness of -1. So it will not be picked up in the survive procedure.

3.2.1.4. Crossover and Mutation

Two recombination operators are employed in GA: crossover and mutation. Crossover is the primary operator in GA process. A pair of chromosomes is chosen randomly as parents, as well as a crossover point within the bit string. The parents swap their tails from the crossover point to generate two offspring as shown in Fig. 5. The probability of this happening is the crossover probability *p1* and is set to 0.5, for example.

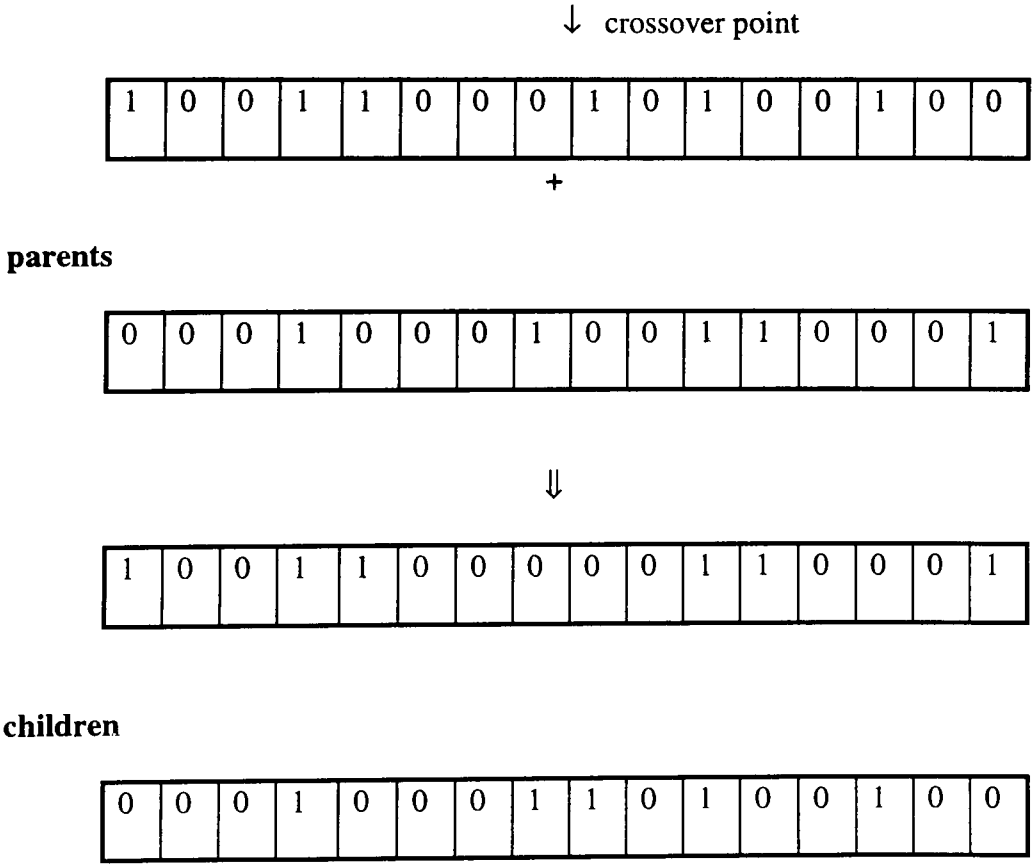



Fig. 5. Crossover.

This kind of crossover is called one point crossover. Similarly, there are two-point crossover, multi-point crossover and uniform crossover. Uniform crossover is to swap every bit of the string. It was said uniform crossover has advantage over others[20]. It is used in ZTEST.

In GA, crossover is applied to every pair of chromosomes at the probability of $p1$. The mutation operation changes a bit in a chromosome from 1 to 0 (or 0 to 1) with a low probability $p2$ as shown in Fig. 6. A mutation rate of 1/pop-size and a crossover rate of approximately 0.6 are suggested[19].

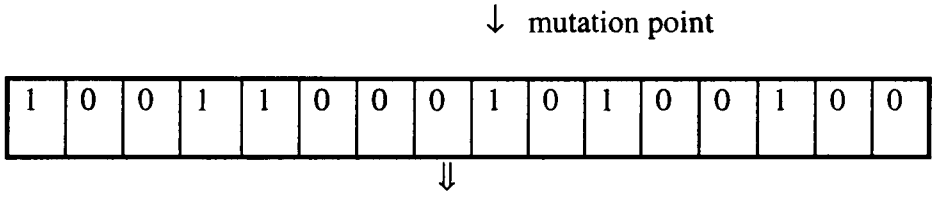



Fig. 6. Mutation.

In ZTEST, crossover and mutation probabilities are

$$p1 = 0.5$$

$$p2 = \frac{1}{\text{length of bit string}}$$

Usually, uniform crossover and mutation operations are applied to the whole bit string of a chromosome. When the range of the variable has been defined, the search domain is limited to a subset of the variable type. This feature is used in ZTEST to make the search of GA more efficient and effective. From the boundary values of a

variable MAXIMUM and MINIMUM, the most significant bit can be known. For example when MAXIMUM is equal to 200, in binary it is 0 0 0 0 1 1 0 0 1 0 0 0. So the most significant bit is 8.

Both crossover and mutation are applied only on bit 1 to the most significant bit of each variable in a chromosome. This constraint on search domain results in reducing calculation time of GA and preserving variables within the valid ranges. For the objective function

$$x1 + x2 + x3 + x4 + x5 = 100 \qquad 0 < x1, x2, x3, x4, x5 <= 50$$

the execution time of one generation in GA is reduced about 30 percent after using the most significant bit as the constraint.

3.2.1.5. Selection of New Population

After the combination operations, the next step of GA is to select a new population of chromosomes from the old population and the offspring. The criterion of selection is that the fittest survive. In ZTEST, all chromosomes are sorted into a list in a decreasing order according to their fitness values, then N fittest chromosomes are chosen as the new generation of population. No duplicates or chromosomes outside the search domain are allowed.

One drawback of genetic algorithms is that genetic search sometimes is trapped to a local optimum. Other algorithms such as Simulated Annealing and TABU Search can overcome this difficulty because they allow the value of objective function not only

increasing, but also decreasing during the process. Therefore a local optimum can be overcome.

In order to improve the GA's performance, some chromosomes with a low fitness are selected to survive deliberately to preserve diversity of chromosomes. A proportion of chromosomes with a poor fitness are chosen randomly, while others are still chosen by the fittest surviving criterion.

Experiments have been done to investigate GA performance with different proportion of chromosomes with low fitness. Results of 100 runs for the objective function

$$x*x + y*y = z*z \qquad \text{with } 0 < x, y, z <= 50$$

are given in Table 4.

**Table 4. The Results of GA for 100 Runs Using Different Proportion of Less Fitted Chromosomes**

| Proportion of chromosomes with poor fitness | 1/2 | 1/3 | 1/4 | 1/5 | 1/10 | 0 | After 30 generations 1/3 |
|---|---|---|---|---|---|---|---|
| Average generations | 13.97 | 13.12 | 15.53 | 14.66 | 14.68 | 13.44 | 12.80 |
| Average generations without over 50 | 12.98 | 12.53 | 14.11 | 14.44 | 11.80 | 12.08 | 12.17 |
| Maximum generations | 59 | 67 | 99 | 51 | 188 | 84 | 68 |
| Number of generations over 50 | 2 | 1 | 2 | 1 | 3 | 2 | 1 |

In Table 4, the first row is the proportions of low fitness chromosomes added to the new generation, the second row is the average generations when GA find the solution over

100 runs, the third row is the average generations without the extremely bad cases of generation over 50, the fourth row is the maximum generations and the last row is the number of runs whose generation are over 50.

It seems that with a higher proportion (1/2 and 1/3) of less fitted chromosomes, the average generations for 100 runs are decreased slightly as well as the maximum generations. But without considering the extremely bad cases (generations over 50), low proportion (1/10 and 0) of poor chromosomes will require less average generations. This is the advantage of the fittest surviving criterion.

Because of the randomness of genetic search, it is difficult to say which proportion of poor chromosomes will get the best result. A compromise has been made. In the first N*g* generations, the whole population is selected according to the fittest surviving criterion in order to keep the efficiency of GA. If the solution still can not be found after N*g* generations, then 1/3 of the population will be substituted by randomly chosen poor chromosomes. The result of this approach is slightly more satisfactory, as shown in the last column of Table 4 with N*g* equals 30.

3.2.1.6. The Process of GA Search

GA search is a random process. It starts from a randomly generated initial population, does crossover and mutation according to the probabilities. The number of generations to get the best solution is different in each attempt even for the same problem. The process is some times longer, some times shorter.

The GA search may be trapped into a local optimum if the solution has not been found after a long time, for example, over 300 generations. It is not worth pursuing the search and a fresh start from another initial population is more efficient in this case. ZTEST will abandon the GA search and begins a fresh start after 300 generations.

Another criterion of the GA process is the average fitness value of the population. According to experiments, the average fitness will increase when chromosomes converge to the solution. The average fitness value is checked in every generation. The attempt of GA search will be abandoned if the average fitness stays the same for over 100 generations.

3.2.1.7. Considerations in Choosing Fitness Functions

Considerations should be taken in choosing fitness functions of genetic algorithms. The fitness function should not be too narrow, nor too flat. The reciprocal function is used as the fitness function in GA as described in section 3.2.1.2. For the objective function

$$f1(x) = f2(x),$$

the fitness function is $$\text{fit(x)} = \frac{1000.0}{abs(f1 - f2) + 10}$$

The curve of the fitness function fit(x) is shown in Fig. 7.

For the objective function

$$\exists x? : \mathrm{N} \mid x? \neq 10 \bullet x? \,/\, (x? - 10) > 3,$$

the curve of function $f1(x?) = x? \,/\, (x? - 10)$ is flat as shown in the thin lines in Fig. 8.

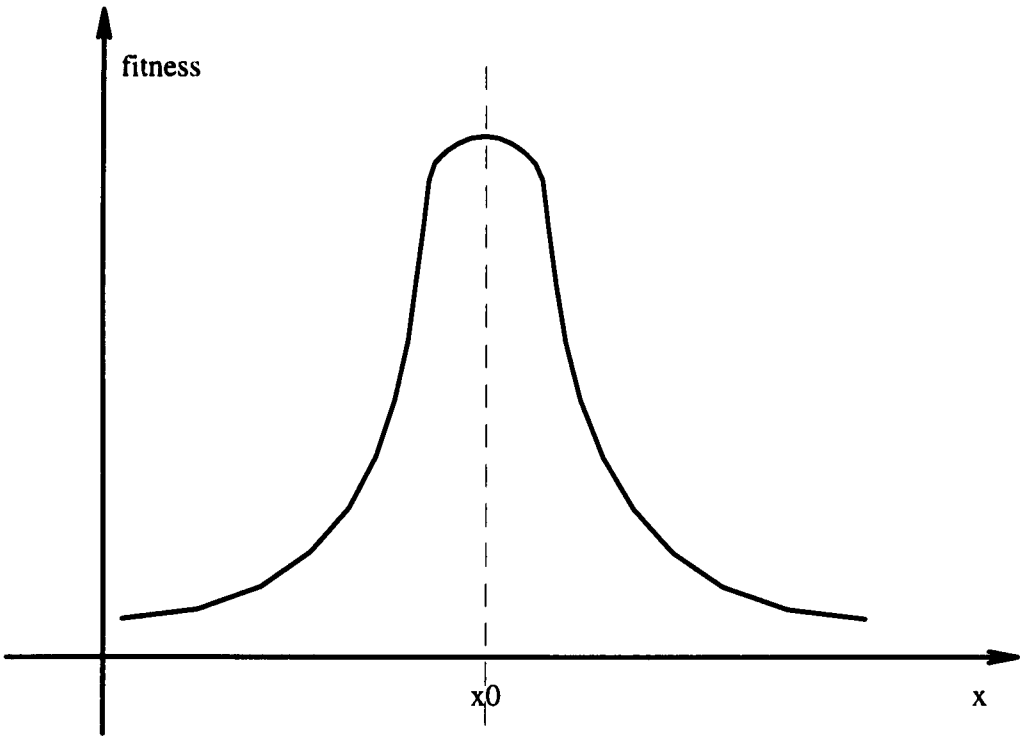



Fig.7. Fitness function fit(x).

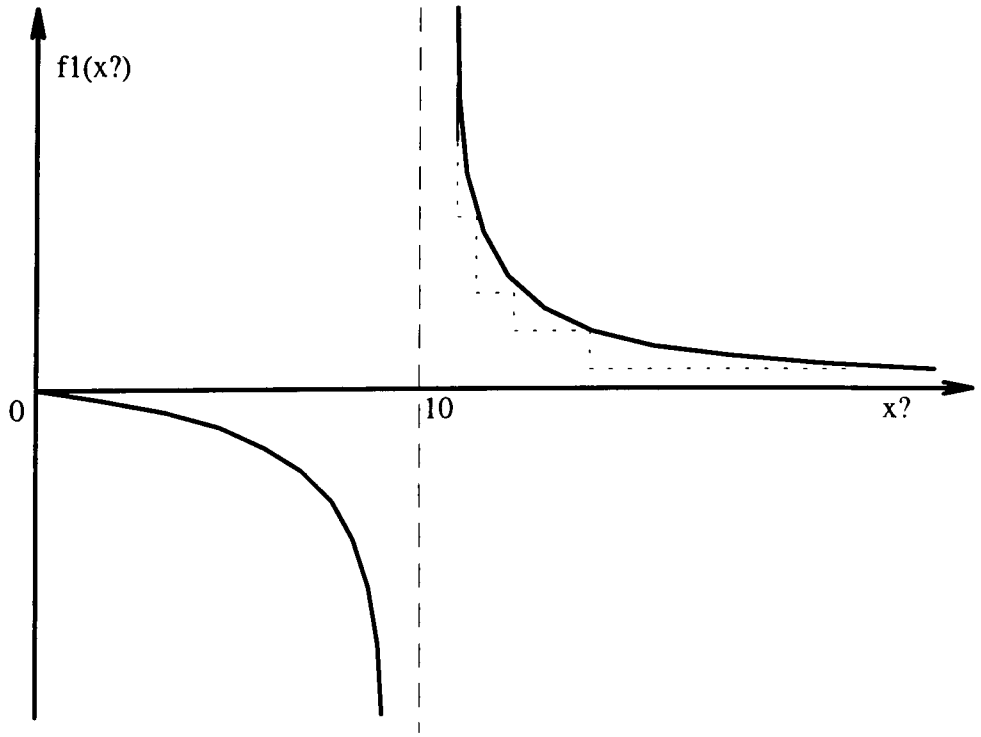



Fig. 8. Fitness function f1(x?) = x? / ( x? - 10 ).

The value of *fl(x?)* remains 1 in a wide range when *x?* has a large value. Therefore, it is difficult for GA to evaluate how close a chromosome is from the best solution. The real type operations are used in calculation of the fitness values. When *x?* is a real number, the fitness is the thick lines in Fig. 8 and the results of GA search is more efficient. The value of *x?* is converted back to integer afterwards.

### 3.2.2. Simulated Annealing

Simulated annealing (SA) is a stochastic computational technique derived from modelling the natural process of thermodynamics for finding near global minimum solutions to optimisation problems [17] [23].

According to statistical mechanics, if a fluid system of particles is in thermal equilibrium at given temperature T, then the probability $\pi_T(s)$ that the system is in a given configuration $s$ depends upon the energy E($s$) of the configuration and follows the Boltzmann distribution:

$$\pi_T(s) = \frac{e^{\frac{-E(s)}{kT}}}{\sum_{\omega \in S} e^{\frac{-E(\omega)}{kT}}}$$

where k is Boltzmann's constant, and S is the set of all possible configurations. One can simulate the behaviour of a system of particles in thermal equilibrium using a stochastic relaxation technique developed by Metropolis [17]:

Suppose that at time t the system is in configuration $q$. A candidate $r$ for the configuration at time t+1 is generated randomly. The ratio $p$ between probability of being in $r$ and in $q$ is

$$p = \frac{\pi_T(r)}{\pi_T(q)} = e^{\frac{-(E(r) - E(q))}{kT}}$$

If $p>1$, the energy of $r$ is strictly less than that of $q$, then $r$ is accepted as the new configuration for time t+1. If $p \leq 1$, then $r$ is accepted with probability $p$. Therefore, configurations with lower energy can be obtained.

It can be shown that as $t \rightarrow \infty$, the probability that the system is in a given configuration s equals $\pi_T(s)$, regardless of starting configuration, and that the distribution of configurations generated converges to the Boltzmann distribution.

The optimisation method of SA is:

1. * The energy function becomes the objective function.
   * The configurations of particles become the configurations of parameter values.
   * Finding a low-energy configuration becomes seeking a global optimal solution.
   * Temperature becomes the control parameter for the process.
2. Having a way of generating and selecting new configurations.
3. From a random solution following an annealing schedule consisting of a decreasing set of T.
4. At each T doing local search to achieve lower energy solution until the best solution.

SA's generation-acceptance iteration is repeated often enough to guarantee that the temporal distribution becomes Boltzmann. Thereafter, the temperature is reduced gradually, keeping the distribution in equilibrium, thereby guaranteeing the discovery of globally optimal structure as the temperature goes to zero.

3.2.2.1. Energy Function

Assuming the objective function of the problem is of the form

$$f1(x1, .. xn)\ \Theta f2(x1, .. xn)$$

in a n-dimensional search domain, the symbol $\Theta$ could be $<$, $=$, $>$, $<=$, $>=$ or $\neq$. The energy function $E(x1, .. xn)$ is chosen as follows:

$f1(x1, .. xn) = f2(x1, .. xn)$, $\quad E(x1, .. xn) = abs\,(f1 - f2)$

$f1(x1, .. xn) < f2(x1, .. xn)$, $\quad$ if $(f1 < f2)$ then $E(x1, .. xn) = 0.0$

$\quad$ else $E(x1, .. xn) = f1 - f2 + 1.0$

$f1(x1, .. xn) > f2(x1, .. xn)$, $\quad$ if $(f1 > f2)$ then $E(x1, .. xn) = 0.0$

$\quad$ else $E(x1, .. xn) = f2 - f1 + 1.0$

$f1(x1, .. xn) <= f2(x1, .. xn)$, $\quad$ if $(f1 < f2)$ then $E(x1, .. xn) = 0.0$

$\quad$ else $E(x1, .. xn) = f1 - f2$

$f1(x1, .. xn) >= f2(x1, .. xn)$, $\quad$ if $(f1 > f2)$ then $E(x1, .. xn) = 0.0$

$\quad$ else $E(x1, .. xn) = f2 - f1$

$f1(x1, .. xn) \neq f2(x1, .. xn)$, $\quad$ if $(f1 \neq f2)$ then $E(x1, .. xn) = 0.0$

$\quad$ else $E(x1, .. xn) = 1.0$

The energy of every configuration can be evaluated using these energy functions. When a configuration satisfies the objective function, it gets the lowest energy value 0.0. It means the criteria have met, the process of simulated annealing will be stopped.

The energy function using Hamming distance is

$$f1(x1, .. xn) = f2(x1, .. xn), \qquad E(x1, .. xn) = \text{Hamming}(f1, f2).$$

where Hamming($f1$, $f2$) is the Hamming distance between bit string $f1$ and $f2$. For other cases, the successor or predecessor is used in Hamming distance as shown in Table 2, section 3.1.4. In this way, high quality test data near the boundary can be found.

3.2.2.2. Penalty Function

Similar to GA process, search for the solution should be kept inside the valid domain of the input variables. To guarantee all configurations chosen in SA search are in the valid domain a penalty function $\mathbf{P}(x)$ is added to the energy functions $E(x)$. The new energy function becomes:

$$\text{energy}(x) = E(x) + \mathbf{P}(x)$$

The penalty function is a threshold function given as:

$$\mathbf{P}(x) = \begin{cases} 0 & \text{when } x \in \mathbf{D} \\ -E(x) + 50000.0 & \text{when } x \notin \mathbf{D} \end{cases}$$

where **D** is the valid search domain. By using the penalty function, any configuration outside the search domain gets a high energy of 50000.0. It will not be picked up during the SA process.

3.2.2.3. Annealing Process

To achieve low-energy configurations, it is not sufficient to simply lower the temperature T, an annealing process must be used, where T is elevated, and then gradually lowered, spending enough time at each T to reach thermal equilibrium, otherwise the probability of attaining a very low energy configuration is greatly reduced. For the convenience, a control parameter T = $k$T is used commonly in the process of simulated annealing.

The process of SA search is as following.

```
T = T0
while T > 0 and Enew > 0,  loop
    i = 1;
    while  i <= Nm and Enew > 0,  loop
       generate a new configuration Xnew;
       get energy  Enew;
       if  Enew < Eold,  then
          accept Xnew;
       else
          if  random(0, 1) <  exp(-(Enew-Eold) / T), then
             accept Xnew;
          else
             keep Xold;
          end if;
       end if;
       i = i + 1;
   end loop;
   T = Co × T;
end loop;
```

where T is the control parameter(analogue of temperature), To is the initial value of T, Nm is the number of local search in any T and Co is the coefficient by which T reduced each time. A new configuration Xnew is generated randomly as:

$$Xnew = Xold \pm \Delta X$$
$$\Delta X = (Xmax\text{-}Xmin) \times random(0, 2)\%$$

There are suggestions on parameters To and Co[17][23]. For example, To should be larger than ΔEmax. Experiments have been done to investigate the effects of parameters To, Nm and Co for the objective function

$$A*A + B*B - C*C = 0, \qquad \text{where } A, B, C \in (10, 100).$$

The results are shown in Table 5. Every experiment in Table 5 has 100 success runs.

**Table 5. Results of SA for A*A + B*B - C*C = 0**

| | Nm | To | Co | Nr | Nrf | tave | tavef | Nc | Ncf | tmax | Ncmax |
|---|---|---|---|---|---|---|---|---|---|---|---|
| E1 | 300 | 2000 | 0.95 | 128 | 75 | 10.9 | 1.91 | 11188 | 2141 | 94.08 | 95484 |
| E2 | 100 | 2000 | 0.95 | 137 | 73 | 7.31 | 1.74 | 7488 | 1956 | 46.63 | 46225 |
| E3 | 100 | 2000 | 0.9 | 137 | 75 | 4.02 | 1.34 | 4154 | 1493 | 22.91 | 23031 |
| E4 | 100 | 1200 | 0.9 | 143 | 66 | 3.97 | 1.03 | 4025 | 1146 | 21.53 | 21296 |

In Table 5, Nr is the total number of runs in an experiment and Nrf is the number of runs that get the solution in the first attempt. In theory as $t \to \infty$, that is $T \to 0$, the distribution of configurations generated converges to the Boltzmann distribution regardless of starting configuration. This is only true when To is high enough, T is decreased slow enough and enough time is spent at each T. These pre conditions become not feasible in practice. Therefore the solution may still not be found after the whole cycle of SA search. With 100 success runs in each experiment, the total runs Nr is larger than 100. It means that sometimes it takes more than one attempt to get the solution. Therefore the value of Nrf is less than 100.

The values of $t_{ave}$ and Nc are the average time and the average configurations of the runs, while the values of $t_{avef}$ and Ncf are the average time and the average configurations of the runs that get the solution in the first attempt. The values of $t_{max}$ and $Nc_{max}$ are the maximum time and the maximum configurations from the worst case.

It can be seen from Table 5 that the average time and the average configurations reduced as the parameters Nm, $T_0$ and Co decreasing. This makes SA search faster. There are also down sides of these changes: the total number of runs Nr increased and the number of runs that get the solution in the first attempt Nrf decreased. Further improvements are needed to make the SA search more efficient.

### 3.2.3. Parallel Annealing

Efforts have been made in trying to get a better search method by combining the ideas from different algorithms [22]. Experiments have been done in the project to improve the performance of SA search. A new method called Parallel Annealing (PA) is given here.

#### 3.2.3.1. Wider Sampling

Because of the randomness of the process of simulated annealing the configurations sometimes converge to the solution quickly, and sometimes are trapped in local optimums. Fig. 9 shows two extreme cases of the SA process.

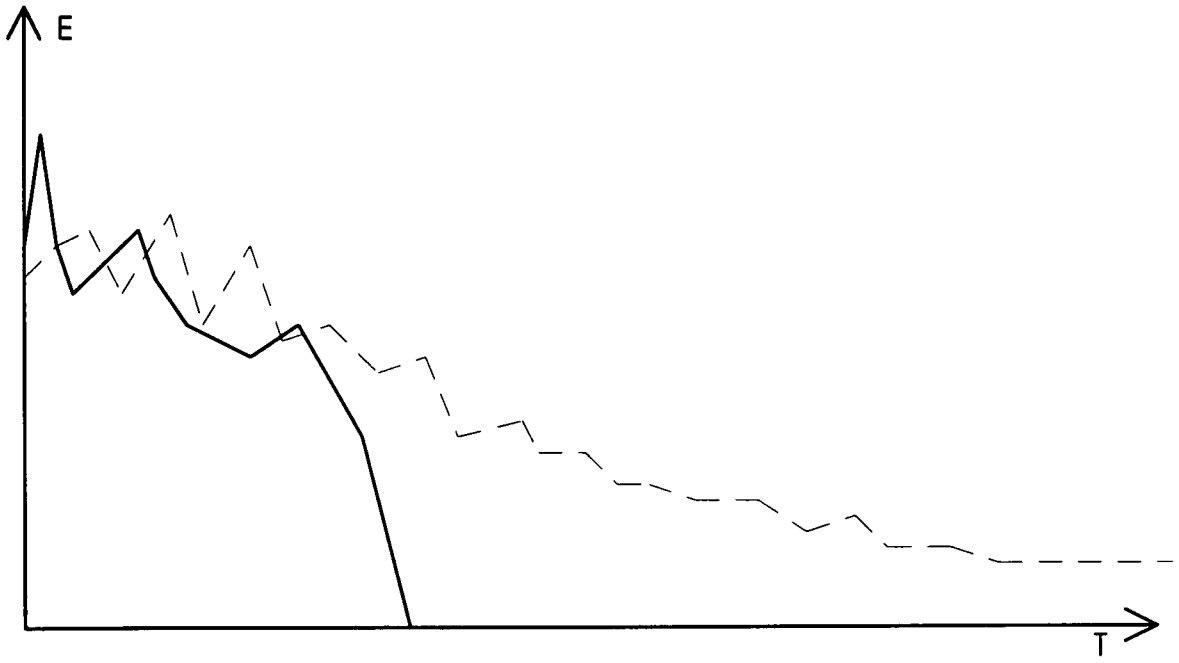



Fig. 9. Two extreme cases of SA process.

In genetic algorithms, all operations are applied on a population of chromosomes that the best solution can be chosen from. To take the advantage of this wider sampling, four configurations are used each time in a parallel SA search instant of just one. The results of this wider sampling are given in experiments E5 to E8 of Table 6 for the same objective function as the experiments in Table 5.

The new method shows better results comparing with the results of standard SA in Table 5. It can be seen from Table 6 that the number of runs that get the solution in the first time Nrf has been increased, while Nr, tave and Nc are all decreased.

**Table 6. Results of PA for A*A + B*B - C*C = 0**

| | Nm | To | Co | Nr | Nrf | tave | tavef | Nc | Ncf | tmax | Ncmax |
|---|---|---|---|---|---|---|---|---|---|---|---|
| E5 | 100 | 2000 | 0.9 | 100 | 100 | 3.02 | 3.02 | 3526 | 3526 | 11.92 | 13380 |
| E6 | 100 | 2000 | 0.8 | 102 | 98 | 2.45 | 2.19 | 2823 | 2556 | 15.43 | 16320 |
| E7 | 100 | 2000 | 0.7 | 107 | 93 | 2.31 | 1.75 | 2606 | 2027 | 12.03 | 12992 |
| E8 | 100 | 1200 | 0.7 | 105 | 95 | 2.11 | 1.71 | 2370 | 1958 | 10.27 | 10888 |
| E9* | 100 | 1200 | 0.7 | 110 | 90 | 1.42 | 0.83 | 2052 | 1215 | 8.68 | 12356 |

3.2.3.2. Simple Criterion

In simulated annealing, the probability of acceptance when Enew >= Eold is

$$\text{If random}(0,1) < e^{\frac{-(Enew - Eold)}{T}}, \text{ then Xnew is accepted.}$$

the exponent expression is time consuming in the search process. In parallel annealing, wider sampling is used to catch the runs that converge to the solution in an early stage as shown in the solid line in Fig. 9. Therefore a linear approach can be used for the exponent function in the reduced range of T. The probability of acceptance in parallel annealing becomes

When Enew < Eold, Xnew is accepted.

When Enew >= Eold,

$$\text{If random}(0,1) > \frac{Enew - Eold}{T}, \text{ then Xnew is accepted.}$$

The result of this change is shown in E9 of Table 6. The execution time $t_{ave}$, $t_{avef}$ and $t_{max}$ are all reduced. The results show that parallel annealing is a feasible approach of adaptive methods.

3.2.4. Comparison of Different Methods

Experiments have been done to compare the performance of different search methods. Random Testing is the method that generates random numbers in the search domain continuously until the solution is found. Genetic algorithm is described in section 3.2.1.

Simulated annealing, as described in 3.2.2, has the parameters set to Nm = 100, $T_0$ = 1200 and Co = 0.9. Parallel annealing that has wider sampling and simple criterion has the parameters set to Nm = 100, $T_0$ = 1200 and Co = 0.7.

The results of the average and the maximum execution times $t_{ave}$, $t_{max}$ as well as the average and the maximum configurations $N_{cave}$, $N_{cmax}$ generated in 100 success runs for each algorithm are given in Table 7 and Table 8 for two different objective functions. The objective function of Table 7 is A*A + B*B - C*C = 0, where A, B, C $\in$ (10, 100) and the objective function of Table 8 is $A \neq 0$; $B \neq 0$; $C \neq 0$ and B*B - 4*A*C = 0, where A, B, C $\in$ (-100, 100).

**Table 7. Comparison of Different Algorithms for A*A + B*B - C*C = 0**

| Algorithm | $t_{ave}$ | $N_{cave}$ | $t_{max}$ | $N_{cmax}$ |
|---|---|---|---|---|
| Random Test | 2.55 | 8508 | 13.13 | 43885 |
| Genetic Algorithm | 67.19 | 2959 | 344.77 | 15030 |
| Simulated Annealing | 3.97 | 4025 | 21.53 | 21296 |
| Parallel Annealing | 1.42 | 2052 | 8.68 | 12356 |

**Table 8. Comparison of Different Algorithms for B*B - 4*A*C = 0**

| Algorithm | $t_{ave}$ | $N_{cave}$ | $t_{max}$ | $N_{cmax}$ |
|---|---|---|---|---|
| Random Test | 2.69 | 8929 | 12.58 | 41980 |
| Genetic Algorithm | 16.52 | 808 | 148.52 | 7045 |
| Simulated Annealing | 1.84 | 1876 | 15.27 | 14880 |
| Parallel Annealing | 1.47 | 2104 | 7.19 | 10104 |

It is obvious that adaptive methods, such as GA, SA and PA, have the advantage of less evaluation time (fewer configurations) in the search process over Random Testing that shows the effects of these algorithms.

It is shown in Table 7 and Table 8 that SA and PA can get the solution faster than GA, especially when GA search is trapped in local optima. On the other hand, GA search may need less evaluation time (fewer configurations) than SA and PA as shown in Table 8. So for the simple objective functions such as the above examples, SA or PA may get the answer more quickly than GA. For the more complicated objective functions that need more evaluation time for each configuration, GA search may have the advantage over SA and PA.

Because of the local search algorithm in SA and PA, they can not have the diversity of configurations as in GA process that caused by crossover and mutation operations. SA and PA search are not very effective when the search domain is very large. For the objective function

$$\exists\ x? : \mathbf{Z} \mid x? \neq 10 \bullet x? / (x? - 10) > 3$$

the valid domain (11, 14) is in a large search domain that is the whole range of integers. Experiments show that GA search can get the solution in about 10 generations, while SA and PA often fail to find the solution.

It is difficult to say which method is better. Their performances are varied for different kind of objective functions. In ZTEST, a choice can be made from either GA, SA or PA as the test data generator with GA as the default algorithm.

Objective functions can be derived from different structured expressions by ZTEST in order to represent the variety of properties of software systems. This chapter introduces the Z constructs that can be tested by ZTEST system, as shown in Table 9.

Table 9. Z Constructs Covered by ZTEST

| Category | Name | Expression |
|---|---|---|
| Numeric Expressions | Algebraic Expression | x? + y? = z? |
| | Comparative Expressions | y? <= 100 |
| | Logical Relations | x? + y?<3 ∧ x? > 0 |
| Set Operations | Set Assignment | name? = "Mary" |
| | Size of a Set | # name? = 5 |
| | Minimum | n = min(S) |
| | Maximum | m = max(S) |
| | Union | S = P ∪ Q |
| | Intersection | S = P ∩ Q |
| | Difference | S = P \ Q |
| | Overriding | S = P ⊕ { m ↦ v?} |
| | Domain Restriction | S = P ◁ Q |
| | Domain Subtraction | S = P ⩤ Q |
| | Concatenate | S = $S_1$ ^ $S_2$ |
| | Head | head(S) |
| | Last | last(S) |
| | Front | front(S) |
| | Tail | tail(S) |
| | Set Comprehension | {j: $m_1$..$m_2$ \| $fc_1$(j) • $fe_1$(j)} = {i: $n_1$..$n_2$ \| $fc_2$(i) • $fe_2$(i)} |
| Iteration Structures | Existential Quantifier | ∃ *i* : 1.. *hwm* • *name?* = *names(i)* |
| | Universal Quantifier | ∀ *j* : 1.. *hwm* • *name?* ≠ *names(j)* |

## 4.1. Numeric Expressions

**Algebraic Expression** such as: x1? + x2? = x3?

Each expression is treated as a node in the tree structure used to form the objective functions during the test case derivation .

**Comparative Expression** such as: y? <= 100

Comparative expressions in the predicate part of Z schemas are used to form the objective functions for GA search while comparative expressions in given set definitions are used to define the boundary values of the given set. For instance, from the expressions in the definition of a given set **SIDES**

**SIDES : Z**
**SIDES** > 0
**SIDES** <= 100

the search domain of **SIDES** is defined as **SIDES** $\in$ [1, 100].

**Logical Relation** such as: (x1? + x2? <= x3?) $\vee$ (x2? + x3? <= x1?)

The tree structure is formed according to the logical relations between expressions.

**Examples for Numeric Expressions**

Two Z specifications are given here as the examples of testing Z numeric expressions by ZTEST: Triangle Classifier and Quadratic Equation.

1. Triangle Classifier

The English specification of Triangle Classifier is as follows[5]. The program reads three integer values. 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, equilateral or right angle triangle. If the input does not represent a

valid triangle, then the program prints a message saying that the input is invalid. The criterion for a valid triangle is that the sum of any two sides is larger than the third one. The Z specification of the Triangle Classifier is given as the following:

[SIDES] the set of all possible triangle sides
MESSAGE ::= Invalid Triangle | Equilateral Triangle | Isosceles Triangle | Scalene Triangle | Right Angle Triangle

Triangle0

$x?, y?, z? : SIDES$

---

$x? + y? > z? \wedge$
$y? + z? > x? \wedge$
$z? + x? > y?$

ScalTri

$\Xi$ Triangle0
reply! : MESSAGE

---

$x? \neq y?$
$y? \neq z?$
$z? \neq x?$
reply! = Scalene Triangle

IsosTri

$\Xi$ Triangle0
reply! : MESSAGE

---

$x? = y?$
$y? \neq z?$
reply! = Isosceles Triangle

EquiTri

$\Xi$ Triangle0
reply! : MESSAGE

---

$x? = y?$
$y? = z?$
reply! = Equilateral Triangle

RightTri

Ξ Triangle0

reply! : MESSAGE

---

x? * x? + y? * y? = z? * z?

reply! = Right Angle Triangle

TriangleError

x?, y?, z? : SIDES

reply! : MESSAGE

---

(x? + y? ≤ z?) ∨

(y? + z? ≤ x?) ∨

(z? + x? ≤ y?)

reply! = Invalid Triangle

Triangle = ScalTri ∨ IsosTri ∨ EquiTri ∨ RightTri ∨ TriangleError

The results of functional testing for the Triangle Classifier are given below. Test data type is Valid and Invalid for high quality test data with the setting of SIZES ∈ [1, 100].

Test Cases

| | | | | | |
|---|---|---|---|---|---|
| Test Case 1 | Invalid | { } | | | |
| Test Case 2 | Invalid | { 68 | 22} | | |
| Test Case 3 | Invalid | { 6 | 6 | 99 | 89} |
| Test Case 4 | Invalid | { 1 | O | v} | |
| Test Case 5 | Invalid | { 0.6 | 0.29 | 0.14} | |
| Test Case 6 | Invalid | { 101 | 101 | 101} | |
| Test Case 7 | Invalid | { 0 | 0 | 0} | |
| Test Case 8 | Valid | { 100 | 100 | 100} | |
| Test Case 9 | Valid | { 1 | 1 | 1} | |
| Test Case 10 | Invalid | { 70 | 42 | 70} | |
| Test Case 11 | Valid | { 55 | 58 | 54} | |
| Test Case 12 | Valid | { 21 | 42 | 22} | |
| reply! = Scalene Triangle | | | | | |
| Test Case 13 | Invalid | { 46 | 46 | 46} | |
| Test Case 14 | Valid | { 41 | 41 | 42} | |
| Test Case 15 | Valid | { 11 | 11 | 10} | |
| reply! = Isosceles Triangle | | | | | |
| Test Case 14 | Valid | { 93 | 93 | 93} | |

Test Case 17 Invalid { 22 22 23}
Test Case 18 Invalid { 19 19 18}
reply! = Equilateral Triangle

Test Case 19 Valid { 24 32 40}
Test Case 20 Invalid { 12 72 73}
Test Case 21 Invalid { 17 34 38}
reply! = Right Angle Triangle

Test Case 22 Valid { 11 17 28}
Test Case 23 Valid { 43 12 56}
Test Case 24 Invalid { 8 26 33}
reply! = Invalid Triangle

Test Case 25 Valid { 81 3 78}
Test Case 26 Valid { 31 10 20}
Test Case 27 Invalid { 31 31 1}
reply! = Invalid Triangle

Test Case 28 Valid { 73 86 13}
Test Case 29 Valid { 2 37 34}
Test Case 30 Invalid { 5 68 64}
reply! = Invalid Triangle

2. Quadratic Equation

The Z Specification of Quadratic Equation is given as:

[NUMBER] the set of all possible coefficients
MESSAGE ::= Same Solutions | Different Solutions | Complex Solutions |
Linear Equation

Same

A?, B?, C? : NUMBER
reply! : MESSAGE

---

A? ≠ 0
C? ≠ 0
B?*B? - 4*A?*C? = 0
reply! = Same Solutions

```
_____Different_____
|  A?, B?, C? : NUMBER
|  reply! : MESSAGE
|_____________________
|  A? ≠ 0
|  C? ≠ 0
|  B?*B? - 4*A?*C? > 0
|  reply! = Different Solutions
|_____________________

_____Complex_____
|  A?, B?, C? : NUMBER
|  reply! : MESSAGE
|_____________________
|  A? ≠ 0
|  C? ≠ 0
|  B?*B? - 4*A?*C? < 0
|  reply! = Complex Solutions
|_____________________

_____Linear_____
|  A? : NUMBER
|  reply! : MESSAGE
|_____________________
|  A? = 0
|  reply! = Linear Equation
|_____________________
```

Quadratic = Same ∨ Different ∨ Complex ∨ Linear

The results of functional testing for the Quadratic Equation are given below. Test data type is Valid only with the setting of NUMBER ∈ [-200, 200].

Test Cases

| | | | | | |
|---|---|---|---|---|---|
| Test Case 1 | Invalid | { } | | | |
| Test Case 2 | Invalid | { | -2 | 110} | |
| Test Case 3 | Invalid | { | -191 | 188 | -90 -32} |
| Test Case 4 | Invalid | { | J | D | I} |
| Test Case 5 | Invalid | { | 0.14 | 0.19 | 0.17} |
| Test Case 6 | Invalid | { | 201 | 201 | 201} |
| Test Case 7 | Invalid | { | -201 | -201 | -201} |
| Test Case 8 | Valid | { | 200 | 200 | 200} |
| Test Case 9 | Valid | { | -200 | -200 | -200} |
| Test Case 10 | Valid | { | 16 | 8 | 1} |

reply! = Same Solutions

Test Case 11 Valid { 3 -5 2}
reply! = Different Solutions

Test Case 12 Valid { 3 -3 1}
reply! = Complex Solutions

Test Case 13 Valid { 0 98 -161}
reply! = Linear Equation

4.2. Set Operations

A set is an unordered collection of objects, each of the same type. A function is a relation in which each member of the from-set (domain) maps to at most one member of the to-set (range). A sequence is a special kind of set that is a finite ordered collection of objects each of the same type. Mathematically, a sequence is a function in which the domain is a set of consecutive natural numbers. Sequences can be used to describe the features of software systems more precisely.

**Set Assignment** such as: name? = "Mary"

Set assignment expressions are decomposed into a series of sub expressions that are used to form the objective functions in test data generation. The above assignment can be decomposed into

name?(1) = M
name?(2) = a
name?(3) = r
name?(4) = y

**Size of a Set** such as: # name? = 5

The expression is used to define the number of members in a set. A default value MAXMEMB for the maximum number of members in a set is defined in ZTEST for every set variable. It can be overridden by the expression of size definition.

**Minimum** such as: $n = \min(S)$

where n is the minimum value of a set of integers S. For example, if S = { 8, 17, 10, 2, 5, 22, 9 }, then $n = \min(S) = 2$.

**Maximum** such as: $m = \max(S)$

where m is the maximum value of a set of integers S. For example, if S is the same as the above example, then $m = \max(S) = 22$.

**Union** such as: $S = P \cup Q$

The set consists of elements that belong to P, or Q, or both P and Q. The expression can be decomposed into two steps. The first step, set assignment S = P. The second step, check every element from Q against every element in P, add the element into S if there is no element in P equals to the element from Q. For example, if P = { 3, 6, 8, 5, 10 } and Q = { 2, 3, 4, 5, 6 }, then $S = P \cup Q$ = { 3, 6, 8, 5, 10, 2, 4 }.

**Intersection** such as: $S = P \cap Q$

The set consists of elements that belong to both P and Q. The expression can be implemented by checking every element from P against Q, add the element into S if there is an element in Q equals to the element from P. For example, if P and Q are the same as the above example, then $S = P \cap Q$ = { 3, 6, 5 }.

**Difference** such as: $S = P \setminus Q$

The set consists of elements that belong to P and do not belong to Q. The expression can be decomposed into two steps. The first step, set assignment S = P. The second step, check every element from P against Q, remove the element from S if there is an element in Q equals to the element from P. For example, if P and Q are the same as the above example, then $S = P \setminus Q = \{ 8, 10 \}$.

**Overriding** such as: $S = P \oplus \{ m \mapsto v? \}$

The expression can be decomposed into

$$S = P$$
$$S(m) = v?$$

The set assignment S = P can be further decomposed. The objective function of the overriding consists of conjunction of all sub expressions which come from the decomposition. Assume P is a set which has 5 members, the function $S = P \oplus \{ 3 \mapsto v? \}$ can be decomposed into

$$S(1) = P(1)$$
$$S(2) = P(2)$$
$$S(3) = v?$$
$$S(4) = P(4)$$
$$S(5) = P(5)$$

**Domain Restriction** such as: $S = P \lhd Q$

The function S consists of the ordered pairs from Q whose first members are in set P. The expression can be implemented by adding every pair from Q into S if its related first member is inside P. For example if Q and S are sequences and Q = { (1,k), (2,D), (3,r), (4,C), (5,h), (6,O), (7,L), (8,A) }, P = { 1, 2, 3, 5, 8}, then $S = P \lhd Q$ = { (1,k), (2,D), (3,r), (4,h), (5,A) }.

**Domain Subtraction** such as: $S = P \lhd Q$

The function S consists of the ordered pairs from Q whose first members are not in set P. The expression can be implemented by assignment S = Q, then removing every pair from S if the associated first member is inside P. For example if Q is the same as the above example and P = { 3, 4}, then $S = P \lhd Q$ = { (1,k), (2,D), (3,h), (4,O), (5,L), (6,A) }.

**Concatenate** such as: $S = S_1 \wedge S_2$

The sequence S is formed by appending the sequence $S_2$ to the sequence $S_1$. For example if $S_1$ = { D O C U } and $S_2$ = { C V }, then $S = S_1 \wedge S_2$ = { D O C U C V }.

**Head, Last, Front and Tail**

Having a sequence S = { $s_1, s_2, \ldots s_n$ }, then we have

head (S) = $s_1$
last (S) = $s_n$
front (S) = { $s_1, s_2, \ldots s_{n-1}$ }
tail (S) = { $s_2, s_3, \ldots s_n$ }

**Set Comprehension** such as: $\{j: m_1..m_2 \mid fc_1(j) \bullet fe_1(j)\} = \{i: n_1..n_2 \mid fc_2(i) \bullet fe_2(i)\}$

Where j: $m_1..m_2$ and i: $n_1..n_2$ are declaration lists, $fc_1(j)$ and $fc_2(i)$ are constraints, and $fe_1(j)$ and $fe_2(i)$ are expressions. The set comprehension can be decomposed into an iteration structure:

```
j = m1;
while fc1(j) is false, loop
   j = j + 1;
end loop;
for i = n1 to n2, loop
```

```
        if fc2(i) is true, then
            fe1(j) = fc2(i);
            j = j + 1;
            while fc1(j) is false, loop
              j = j + 1;
            end loop;
        end if;
    end loop;
    m2 = j - 1;
```

The objective function that consists of a series of sub expressions can be obtained from this structure. For the expression *contents!* = {*i*: 1.. *hwm* | *i* ≠ 4 • *names(i)*}, its sub expressions obtained from the decomposition are

*contents!* (1) = *names*(1)
*contents!* (2) = *names*(2)
*contents!* (3) = *names*(3)
*contents!* (4) = *names*(5)
*contents!* (5) = *names*(6)
....
*contents!* (*hwm*-1) = *names*(*hwm*).

4.3. Iteration Structures

**Existential Quantifier** such as: $\exists i : 1..\ hwm \bullet name? = names(i)$

There are three possible conditions on the search domain of an existential quantifier: the search domain is defined; the search domain is the whole domain of the set and the search domain is unknown.

**1. The boundary of the search domain is defined,** such as:

$$\exists i : 1..\ hwm \bullet name? = names(i)$$

This results in a series of test cases arising from different choices of *i* that belong to the set comprising extremes and typical values: {1, 2, *j*, *hwm*-1, *hwm*}, where $2 < j < hwm-1$. These values are equitable to the boundary value analysis plus an intermediate value

from an equivalence partition [14]. They are all applied to the predicate *name? = names(i).* For instance, five test cases are generated on the node of the existential quantifier.

| | | | |
|---|---|---|---|
| Test Case 1 | Valid | *name? = names(1)* | *(i = minimum)* |
| Test Case 2 | Valid | *name? = names(2)* | *(i = minimum+1)* |
| Test Case 3 | Valid | *name? = names(4)* | *(i = a typical value)* |
| Test Case 4 | Valid | *name? = names(hwm-1)* | *(i = maximum-1)* |
| Test Case 5 | Valid | *name? = names(hwm)* | *(i = maximum)* |

The test data can be generated by direct assignment.

**2. The search domain is the whole domain of the set,** such as:

$$\exists\ i : \mathbf{N}_1 \bullet name? \neq names(i)$$

Five test cases are chosen by taking the value of *i* from {1, 2, j, Ln-1, Ln}, where 2<j<Ln-1 and Ln is the length of the array *names* that can be found from the definition of the array by ZTEST. The test data can be generated by direct assignment.

**3. The boundary of the search domain is unknown,** such as:

$$\exists\ x? : \mathbf{Z} \mid x? \neq 10 \bullet x? / (x? - 10) > 3$$

Only one valid test case (a typical value of the iteration variable) is given as an ordinary predicate. The objective function of the example is given as:

$$x? \neq 10 \wedge x? / (x? - 10) > 3$$

The test data are generated by ZTEST form the objective function using adaptive methods such as GA and SA.

For the search domain conditions 1 and 2, all values of the iteration variable (e.g. i = 1, 2, 4, *hwm*-1and *hwm*) are checked to find out if the constraint of the existential quantifier is true during the test case derivation. The test case is excluded when the

value of iteration variable makes the constraint false. Therefore, the number of test cases for existential quantifiers may be less then five. For example,

$$\exists\, i : 1..\, hwm \mid i \neq 2 \bullet name? = names(i)$$

Test Case 1 Valid $name? = names(1)$
Test Case 2 Valid $name? = names(4)$
Test Case 3 Valid $name? = names(hwm\text{-}1)$
Test Case 4 Valid $name? = names(hwm)$

When the node being tested is after the existential structure, the existential structure is treated as one expression that the iteration variable takes the minimum value as shown in Fig. 10. Expressions in a Z schema are given as:

```
symbols = {A, B, C, D, E, F, G, H}                    (1)
values = {65, 66, 67, 68, 69, 70, 71 72}              (2)
n = 8                                                 (3)
∃ i : 1.. n | ch? = symbols(i)                        (4)
          • x? = values(i)                            (5)
            y? = values(i) + 32                       (6)
z? = x? * y?                                          (7)
```

The tree structure is shown in Fig. 10 where expressions (4), (5) and (6) are in an existential quantifier structure. Five test cases are derived for expressions (4), (5) and (6) when iteration variable i takes the value from {minimum, minimum+1, j, maximum-1 and maximum} as shown in Fig. 10 circled by the dot line. Only one test case is given for expression (7) which is after the existential quantifier when i equals to minimum for (4), (5) and (6) as shown in Fig. 10.

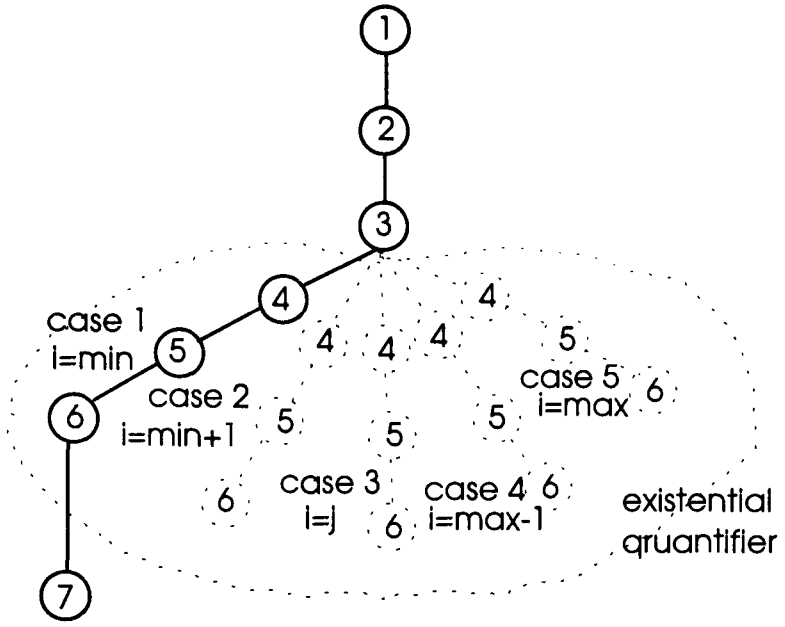



Fig. 10. Tree structure of testing existential quantifiers.

**4. Nested existential quantifiers**

If the existential quantifiers are nested to each other, there are five values of the iteration variable to be tested in each existential quantifier. The test suite are generated for every combination of the values of each iteration variable. For example, the structure of existential quantifiers is as the follows

$$\exists\, i : 1 \,..\, hwm \mid index? = i \bullet$$
$$\exists\, j : 1 \,..\, numnames \mid name? = names\ (j) \bullet$$
$$char = name?\ (\ index?\ )$$

There are five values of i (1, 2, 4, hwm-1, hwm) and five values of j (1, 2, 4, numnames-1, numnames) to be tested. Twenty five test cases are generated for the nested existential structure as shown in the following Table 10. This method is feasible for two or three nested existential structures. Test cases can be reduced for more complicated nested existential structures.

Table 10. Test Cases for the Nested Existential Quantifiers

| Test Case | i | j |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 4 |
| 4 | 1 | numnames-1 |
| 5 | 1 | numnames |
| 6 | 2 | 1 |
| 7 | 2 | 2 |
| 8 | 2 | 4 |
| 9 | 2 | numnames-1 |
| 10 | 2 | numnames |
| 11 | 4 | 1 |
| 12 | 4 | 2 |
| 13 | 4 | 4 |
| 14 | 4 | numnames-1 |
| 15 | 4 | numnames |
| 16 | hwm-1 | 1 |
| 17 | hwm-1 | 2 |
| 18 | hwm-1 | 4 |
| 19 | hwm-1 | numnames-1 |
| 20 | hwm-1 | numnames |
| 21 | hwm | 1 |
| 22 | hwm | 2 |
| 23 | hwm | 4 |
| 24 | hwm | numnames-1 |
| 25 | hwm | numnames |

## 5. State typed existential quantifiers

Predicates in Z can be categorised into three types: input predicates, state predicates and output predicates. The existential quantifiers mentioned before are all of input type, that is there are input variables on the left side of the predicates. Test cases are generated from input typed existential quantifiers. The existential quantifier is of state type if there are only state variables on the left side of the predicates. No test cases are generated from state typed existential quantifiers, the goal is to get the value of the state

variable to make the predicates true. During the test data generation, ZTEST checks values of the state variable until the value is found to make the predicates true. For example, a state existential quantifier is nested inside an input existential quantifier:

$$array = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$
$$\exists i : 1 .. N_1 \mid number? = i * i \bullet$$
$$\exists j : 1 .. \#array \mid array(j) = number? \bullet$$
$$\ldots\ldots$$

For the first input existential quantifier, five test cases ( i = 1, 2, 4, MaxNumber-1, MaxNumber, where MaxNumber is the maximum nature number. ) are generated. For each test case, the value of j need to be found to make the predicate *array*(j) = *number?* true. The value of j is inside the domain of state variable *array* which its members are set up before as the start state. Unlike the input typed existential quantifiers which their domain may be very large, the domain of state variables usually is known and is within 1 to MAXMEMB in ZTEST. It is practical to use the exhaustive testing method here to check values of j against the predicates. In case the value of j that makes the predicates true is not found in the whole domain, the test case is abandoned. For the above example, the test cases are:

Test Case 1: number? = 1 ( i=1, j=1 )
Test Case 2: number? = 4 ( i=2, j=4 )
Test Case 3: number? = 16 ( i=4, j=16 )

The test cases for i=MaxNumber-1 and i=MaxNumber are abandoned because no value of j is found to make the predicate true.

**Universal Quantifier** such as: $\forall i : 1.. hwm \bullet name? \neq names(i)$

This results in a test case that is the conjunction of all possible conditions of the predicate in the universal quantifier, such as:

$$name? \neq names(1) \wedge name? \neq names(2) \wedge .. \wedge name? \neq names(hwm)$$

Only one valid test case is needed for the universal quantifier. There are three possible conditions on the search domain of an universal quantifier: the search domain is defined; the search domain is the whole domain of the set and the search domain is unknown.

**1. The boundary of the search domain is defined,** such as:

$$\forall i : 1..10 \mid i \neq 5 \bullet x? + y? \neq i$$

The test case is:

$$x? + y? \neq 1 \wedge .. \wedge x? + y? \neq 4 \wedge x? + y? \neq 6 \wedge .. \wedge x? + y? \neq 10$$

The test data is generated using adaptive method.

**2. The search domain is the whole domain of the set,** such as:

$$\forall i : N_1 \bullet name? \neq names(i)$$

The valid test case is:

$$name? \neq names(1) \wedge name? \neq names(2) \wedge .. \wedge name? \neq names(Ln)$$

where Ln is the length of the array names. The test data is generated using adaptive method. If the length of the array is too large to be tested under this kind of exhaustive testing method, for instance Ln is greater than a threshold value MAXCASE, an invalid test case is given as:

$$\text{Test Case 1 Invalid} \quad name? = names(j)$$

where j is a value in the range of [1, Ln].

**3. The boundary of the search domain is unknown,** such as:

$$\forall x? : S \bullet \text{Predicate } fp(x?)$$

It is not feasible to have all members in a general set tested exhaustively in such an universal quantifier. Instant of the valid test case, an invalid complementary test case is generated for this kind of universal quantifiers.

Test Case 1 Invalid $\neg$ Predicate fp(x?)

**4. State typed universal quantifiers**

The universal quantifier is of state type if there are only state variables on the left side of the predicates. No test cases are generated from state typed universal quantifiers, the goal is to get the value of the state variable to make the predicates true. For an universal quantifier, the search domain usually is defined as outside a known domain (such as the domain of a state variable or an input variable). The solution of the universal quantifier need to be generated by adaptive methods. ZTEST generates the value of the state variable in an universal quantifier at random. For example,

$$
\begin{aligned}
&fid : N \\
&usedfids = \{ 5, 4, 8, 10, 6\} \\
&\forall i : 1 .. \#usedfids \mid fid \neq usedfids(i) \\
&\qquad \ldots.
\end{aligned}
$$

The search domain of fid is the whole domain of nature numbers except the numbers in usedfids, so the solution is easily found by random testing.

**5. Nested structures of existential quantifiers and universal quantifiers**

Existential quantifiers and universal quantifiers can be nested together to form a mixed structure. The same rules of test data generation are applied. The final objective functions are the conjunction of all sub functions from every existential and universal quantifiers. For example, the nested structure is shown as:

$$\exists\, i : 1 \,..\, numfid \mid fid? = fids(i) \bullet$$
$$\forall\, j : 1 \,..\, numused \mid fid? \neq usedfids(j) \bullet$$
$$....$$

The objective functions for each test case are:

Test Case 1:
$$fid? = fids(1) \wedge$$
$$fid? \neq usedfids(1) \wedge$$
$$fid? \neq usedfids(2) \wedge$$
$$....$$
$$fid? \neq usedfids(numused)$$

Test Case 2:
$$fid? = fids(2) \wedge$$
$$fid? \neq usedfids(1) \wedge$$
$$fid? \neq usedfids(2) \wedge$$
$$....$$
$$fid? \neq usedfids(numused)$$

Test Case 3:
$$fid? = fids(4) \wedge$$
$$fid? \neq usedfids(1) \wedge$$
$$fid? \neq usedfids(2) \wedge$$
$$....$$
$$fid? \neq usedfids(numused)$$

Test Case 4:
$$fid? = fids(numfid\text{-}1) \wedge$$
$$fid? \neq usedfids(1) \wedge$$
$$fid? \neq usedfids(2) \wedge$$
$$....$$
$$fid? \neq usedfids(numused)$$

Test Case 5:
$$fid? = fids(numfid) \wedge$$
$$fid? \neq usedfids(1) \wedge$$
$$fid? \neq usedfids(2) \wedge$$
$$....$$
$$fid? \neq usedfids(numused)$$

## 6. Complementary test cases

The exhaustive testing used in the universal quantifiers becomes impractical if the number of members of the set in a universal quantifier is too large, or the boundary of the search domain is unknown. In such cases, a complementary test case is raised. For instance, the universal quantifier is given as

$$\forall i : 1..hwm \bullet name? \neq names(i)$$

If the value hwm is larger than the threshold value MAXCASE, a complementary test case is given as invalid test case:

$$\exists i : 1..hwm \bullet name? = names(i)$$

Test data generated is:

Test Case 1 Invalid name? = names(j)

where j is a value in the range of [1, hwn].

On the other hand, test data may be difficult to find for certain objective functions. An invalid test case is generated if the solution has not been found after several failed attempts by the test data generator. For instance, when the number of failed attempts is equal to a threshold value MAXFAIL in the following case:

$$\exists x? : N \mid x? \neq 10 \bullet x? / (x? - 10) > 10 \qquad \text{Valid}$$

where there is only one possible solution ( $x? = 11$), the invalid test case is raised as:

$$\exists x? : N \mid x? \neq 10 \bullet x? / (x? - 10) < 10 \qquad \text{Invalid.}$$

When the number of failed attempts equals to MAXFAIL in an invalid complementary test case of universal quantifiers, the search of the test data generator is stopped. It may mean that the solution for such test case does not exist. In this case, a single valid test case is generated. For the universal quantifier

$$\forall x? : N \bullet (2*x? + 1) \bmod 2 \neq 0 \qquad \text{Valid}$$

ZTEST will try to find the invalid test data of the complementary test case:

$$\exists x? : N \bullet (2*x? + 1) \bmod 2 = 0 \qquad \text{Invalid.}$$

The solution for this complementary test case does not exist. One valid test data is generated after several failed attempts by the test data generator.

$$\exists x? : N \bullet (2*x? + 1) \bmod 2 \neq 0 \qquad \text{Valid}$$

## Example of Set Operations and Iteration Structures

A birthday book system[9] is taken as an example of testing Z schemas with set operations and iteration structures. The schema of the definition of state variables is given as:

BirthdayBook

$$names : \mathbf{N}1 \rightarrow NAME$$
$$dates : \mathbf{N}1 \rightarrow DATE$$
$$hwm : \mathbf{N}$$

---

$$\forall i, j : 1..hwm \bullet$$
$$i \neq j \Rightarrow names(i) \neq names(j)$$

Three state variables are declared in the schema. Variable *names* and *dates* are arrays described by functions from the set N1 of strictly positive integers to NAME or DATE, while *hwm* is the length of the arrays. The predicate section says that there are no repetitions in the array *names*.

A schema that gives the initial values of state variables is shown as:

Initial

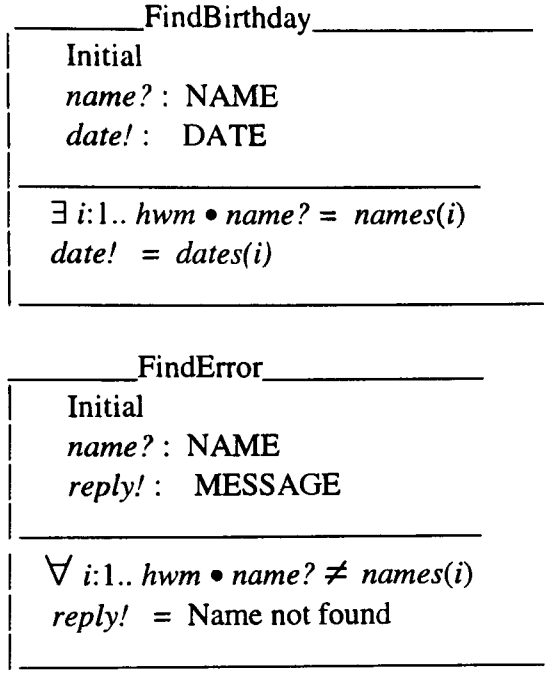
$$\Xi \text{ BirthdayBook}$$
$$names : \mathbf{N}1 \rightarrow NAME$$
$$dates : \mathbf{N}1 \rightarrow DATE$$
$$hwm : \mathbf{N}$$

---

$$names = \{ \text{Anna, Joe, Mary, Mike, Tony,}$$
$$\text{John, Tom, Xile, Lee, Jin, Sam} \}$$
$$dates = \{ 20,10, 15, 8, 30, 6, 1, 12, 5, 1, 18, 3,$$
$$27, 2, 11, 5, 15, 8, 9, 11, 30, 1 \}$$
$$hwm = 11$$

**Find Birthday**

An operation to find a person's birthday is described by the following schemas:




There is an input variable *name?* and an output variable *date!* in FindBirthday. If there exists a *names(i)* that is the same as the input *name?*, then output *date!* will be *dates(i).* Schema FindError gives an error message using the universal quantifier. The results of functional testing for Find Birthday are given below. Test data type is Valid Only.

Test Cases

| | | | | | | |
|---|---|---|---|---|---|---|
| Test Case 1 | Invalid | { } | | | | |
| Test Case 2 | Invalid | { | A | I | J} | |
| Test Case 3 | Invalid | { | N | s | H | c O} |
| Test Case 4 | Invalid | { | 82 | 407 | 342 | 382} |
| Test Case 5 | Invalid | { | 0.82 | 0.95 | 0.21 | 0.42} |
| Test Case 6 | Invalid | { | @ | @ | @ | @ } |
| Test Case 7 | Invalid | { | { | { | { | { } |
| Test Case 8 | Valid | { | A | A | A | A } |
| Test Case 9 | Valid | { | z | z | z | z } |
| Test Case 10 | Valid | { | A | n | n | a } |
| date! = 20 10 | | | | | | |
| Test Case 11 | Valid | { | J | o | e | } |
| date! = 15 8 | | | | | | |
| Test Case 12 | Valid | { | M | i | k | e} |
| date! = 1 12 | | | | | | |

```
Test Case 13 Valid  {   J    i    n    }
     date! =  9 11
Test Case 14 Valid  {   S    a    m    }
     date! = 30  1
Test Case 15 Valid  {   K    f    o    t}
     reply! = Name not found
```

**Add Birthday**

An operation to add a person's birthday is described by the following schemas:

AddBirthday

Initial

*name?* : NAME

*date?* : DATE

---

$\forall i{:}1..\ hwm \bullet name? \neq names(i)$

$hwm = hwm + 1$

$names = names \oplus \{ hwm \mapsto name? \}$

$dates = dates \oplus \{ hwm \mapsto date? \}$

AddError

Initial

*name?* : NAME

*reply!* : MESSAGE

---

$\exists i{:}1..\ hwm \bullet name? = names(i)$

*reply!* = Name already known

There are two input variables *name?* and *date?* in AddBirthday. If the input *name?* can not be found in *names*, then *name? and date?* will be added to *names(i) and dates(i)*, respectively. Schema AddError gives an error message using the existential quantifier. The results of functional testing for Add Birthday are given below.

Test Cases

```
Test Case  1  Invalid { }
Test Case  2  Invalid {   A    I    J    N   10}
Test Case  3  Invalid {   Y    L    s    B    6    4    3}
Test Case  4  Invalid {  413  479  109  213    y    y}
Test Case  5  Invalid { 0.45  0.6 0.66 0.59 0.54 0.72}
Test Case  6  Invalid {   @    @    @    @   32   13}
```

```
Test Case  7  Invalid {    {    {    {    {   0   0}
Test Case  8  Valid   {    A    A    A    A   31  12}
Test Case  9  Valid   {    z    z    z    z   1   1}
Test Case  10 Valid   {    Q    h    a    w   9   5}
  hwm = 12
        names = {Anna,Joe,Mary,Mike,Tony,John,Tom,Xile,Lee,Jin,
                 Sam,Qhaw}
  dates = { 20 10, 15  8, 30  6,  1 12,  5  1, 18  3, 27  2, 11  5,
            15  8,  9 11, 30  1,  9  5}
Test Case  11 Valid   {    A    n    n    a   22   5}
            reply! = Name already known
Test Case  12 Valid   {    J    o    e        12   2}
      reply! = Name already known
Test Case  13 Valid   {    M    i    k    e   14   8}
      reply! = Name already known
Test Case  14 Valid   {    S    a    m        13   6}
      reply! = Name already known
Test Case  15 Valid   {    Q    h    a    w   5   9}
      reply! = Name already known
```

**Card List**

An operation to find out today's card list from the birthday book is described by the following schemas:

CardList

$\Xi$ BirthdayBook
*today?* : DATE
*cardlist!* : $\mathbb{N} \rightarrow$ NAME
*ncard!* : $\mathbb{N}$

---

$\exists i:1..hwm \bullet today? = dates(i)$
$\{ j : 1..ncard! \bullet cardlist!(j) \} =$
$\{ k : 1..hwm \mid dates(k) = today? \bullet names(k) \}$

CardError

$\Xi$ BirthdayBook
*today?* : DATE
*reply!* : MESSAGE

---

$\forall i:1..hwm \bullet today? \neq dates(i)$
*reply!* = No card today

There is an input variable *today?* and two output variables *cardlist!* and *ncard!* in CardList. For any *dates(i)* that is the same as the input *today?*, the *names(i)* will be in

the output *cardlist!* and *ncard!* is the total number of cards. Schema CardError gives an error message when *today?* is not found in *dates* of the Birthday Book. The results of functional testing for Card List are given below.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 31}
Test Case 3 Invalid { 31 8 12}
Test Case 4 Invalid { J N}
Test Case 5 Invalid { 0.7 0.94}
Test Case 6 Invalid { 32 13}
Test Case 7 Invalid { 0 0}
Test Case 8 Valid { 31 12}
Test Case 9 Valid { 1 1}
Test Case 10 Valid { 20 10}
cardlist! = Anna
ncard! = 1
Test Case 11 Valid { 15 8}
cardlist! = Joe Lee
ncard! = 2
Test Case 12 Valid { 1 12}
cardlist! = Mike
ncard! = 1
Test Case 13 Valid { 9 11}
cardlist! = Jin
ncard! = 1
Test Case 14 Valid { 30 1}
cardlist! = Sam
ncard! = 1
Test Case 15 Valid { 19 7}
reply! = No Card Today

# CHAPTER 5. ZTEST SYSTEM

The software system ZTEST is developed to automatically generate test data sets from Z specifications using adaptive algorithms.

## 5.1. The Process of Test Data Generation

Supposing the functionality of a software system is described in its Z specification, ZTEST can read Z schemas and parse them automatically to identify the input variables as well as their pre and post conditions. Test data sets are generated according to these conditions. There are four kinds of variables in Z specifications, input variables such as $x?$; output variables such as $y!$; state variables such as *hwm* and iteration variables such as $i$. All expressions in Z schemas are classified into the following categories by ZTEST:

**1. Input Type**

The expression is of input type if there are input variables on the left side of the expression. The following expressions are input functions.

$$x? + 5 > 10$$

$$x? + y? = hwm + 3$$

$$name? \neq names(i)$$

Every input function is used to form the objective functions for test data generation. Test data sets are generated from the objective functions either by direct assignment or by using the test data generators such as GA or SA.

**2. State Type**

The expression is of state type if there are only state variables on the left side of the expression. The following expressions are state functions.

$$hwm = 10$$

$$hwm + 8 = x? + y?$$

$$names(j) \neq names(i)$$

State functions are not involved into test data generation directly. ZTEST resets values of state variables after these state operations.

**3. Output Type**

The expression is of output type if there are only output variables on the left side of the expression. The following expressions are output functions.

$$y! = x? - 3$$

$$y! = hwm$$

$$reply! = \text{Name not known}$$

Output functions are not involved into test data generation directly. ZTEST displays values of output variables after these output calculations.

**4. Iteration Type**

The expression is of iteration type if there are only iteration variables on the left side of the expression. The following expressions are iteration functions.

$$i = j$$

$$i <= hwm$$

$$i \neq 2$$

Iteration functions are not involved into test data generation directly. ZTEST substitutes iteration variables in the other kinds of functions with their values. The values of

iteration variable are checked against these iteration functions, the value will be excluded if it makes the iteration function false.

The process of automatic test data generation is shown in the flow chart of ZTEST.

## 5.2. The Flow Chart of ZTEST

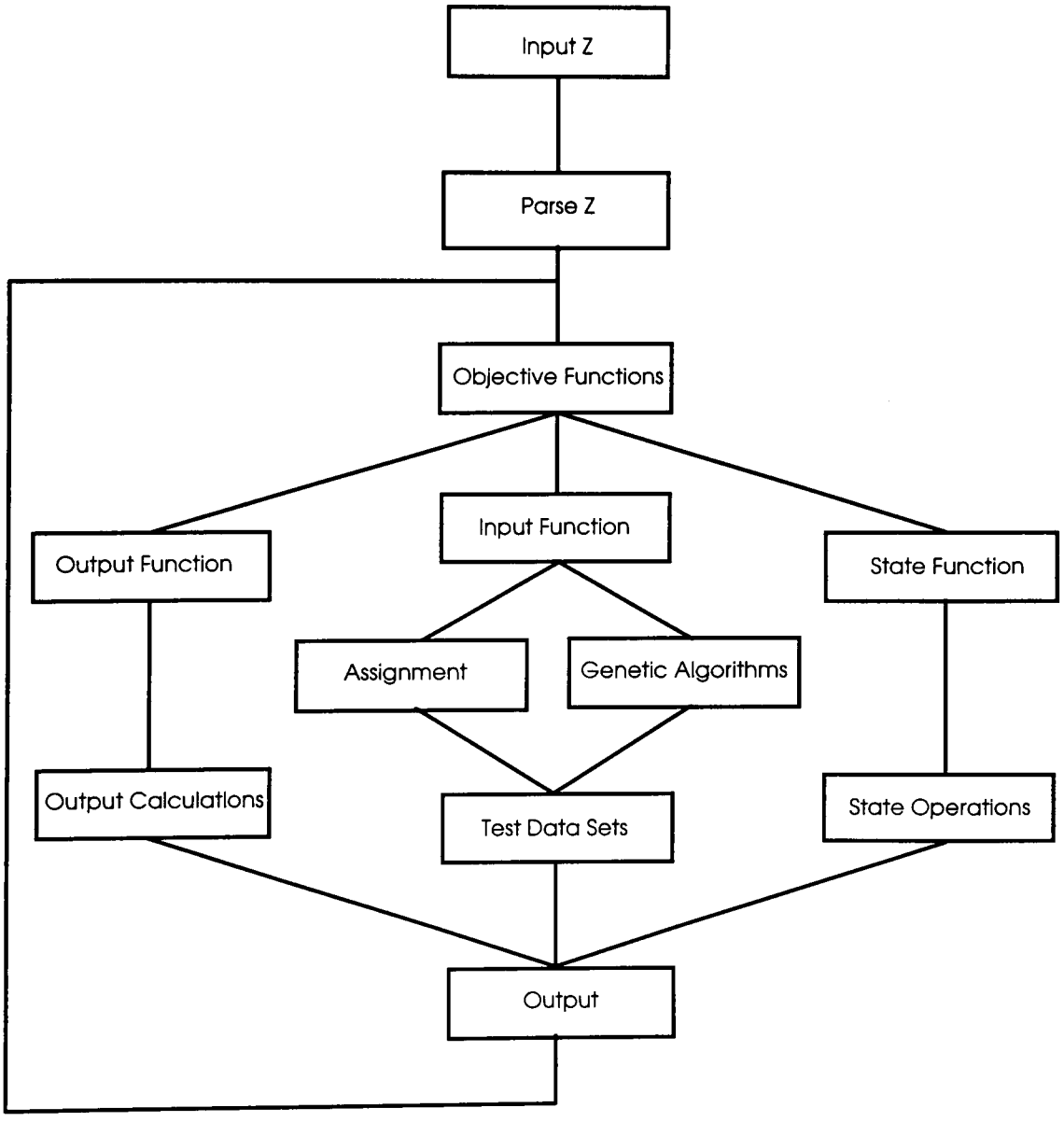



Fig. 11. Flow chart of ZTEST system.

## 5.3. Using ZTEST

The interface of ZTEST is shown in Fig. 12. The procedure of using ZTEST is as follows.

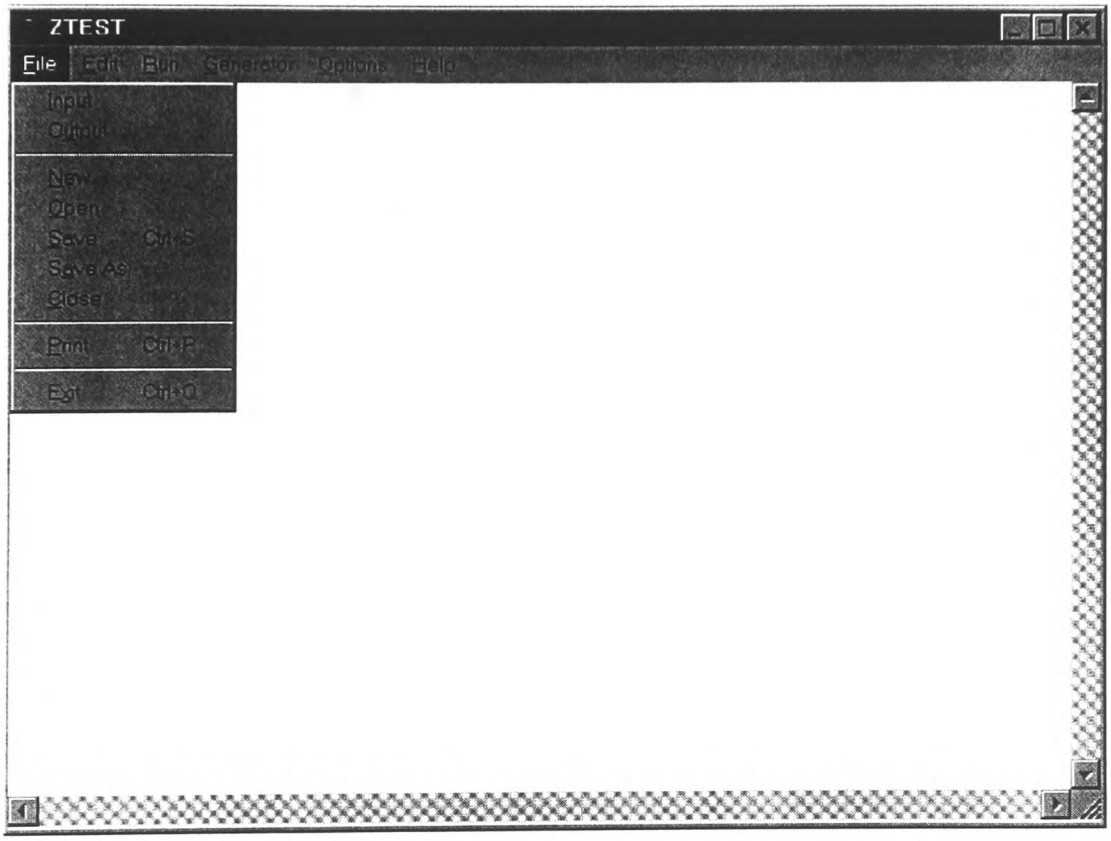



Fig. 12. The Main Window of ZTEST.

1. From the File menu, choose Input File. A dialogue box appears asking you to type in the input file name.
2. From the File menu, choose Output File. A dialogue box appears asking you to type in the output file name. There is a default output file called ZZ.OUT.
3. Choose Run from the main menu to run ZTEST.

There are options in running ZTEST:

1. From the Options menu, choose Functional Testing or Branch Testing as the testing type. The default setting is Functional Testing.
2. From the Options menu, choose Valid Only or Valid & Invalid as the test data type. The default setting is Valid Only.

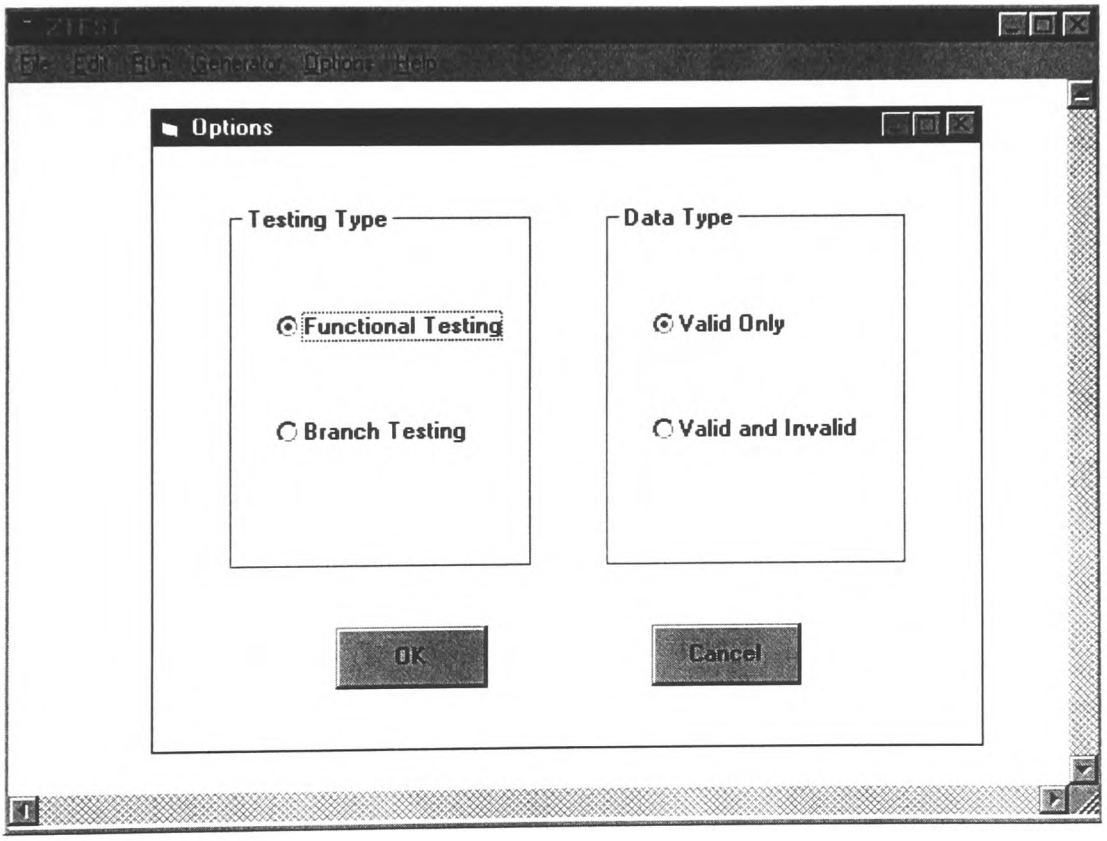



Fig. 13. The Options Window of ZTEST.

3. From the Generator menu, choose GA, SA or PA as the test data generator. The default setting is GA.

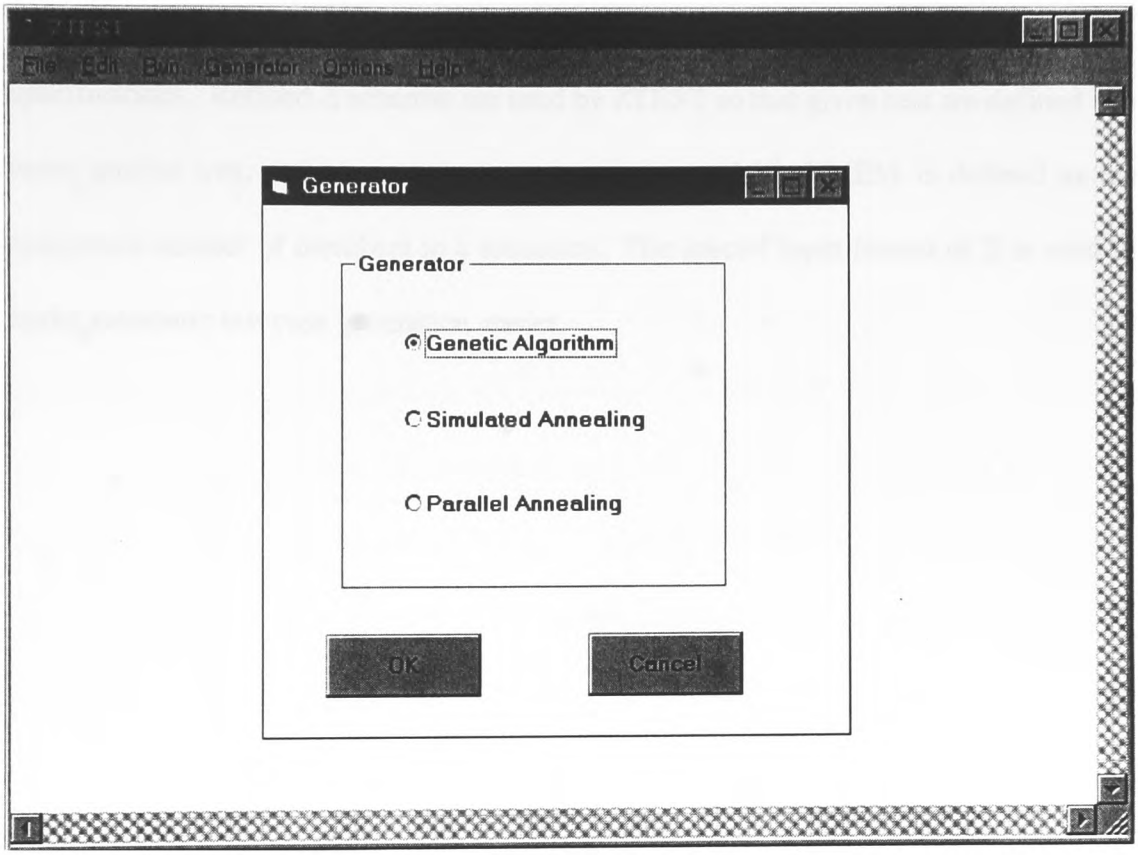



Fig. 14. The Generator Window of ZTEST.

From the Help menu, choose Help to get a brief description of how to use ZTEST. Error messages are given by ZTEST when errors occur during the execution, such as an unrecognised string or the index of an array is out of range, etc.

As an approach of automatic test data generation, ZTEST generates test data sets directly from Z specifications. Having the advantages of efficient and robust, automatic testing methods also have their drawbacks. Z is a formal specification based on mathematical set theory and supported by natural language descriptions. It is more

complicated than pure mathematical expressions to be automatically analysed. Further more, many capabilities of Z functions are optional, it means that the user can write different style Z notations for a same problem. In order to generate test cases automatically, it is necessary to make some restrictions to the writing style of Z specifications. Refined Z schemas are used by ZTEST so that given sets are defined in a more precise way, sets are represent by sequences and MAXMEM is defined as the maximum number of members in a sequence. The special input format of Z is used to make automatic test case generation easier.

# CHAPTER 6. UNIX FILING SYSTEM

## 6.1. UNIX Filing System

The Z specification of UNIX filing system [33] is chosen as an experiment of automatic test data generation using ZTEST. The UNIX operating system [34] is widely known and its filing system is well understood. The experiment demonstrates how to use a formal mathematical specification to capture important aspects of the behaviour of a good sized real software system, and automatically generate test data sets for the software system. The inputs from the test cases generated by ZTEST might be applied to UNIX system to validate the system by comparing the outcome with the output and state variables in the test cases. Refinement is made from the Z specification in reference [33] to meet the requirement of ZTEST.

### 6.1.1. File Storage System

In the UNIX system, the set of all bytes is defined as a given set BYTE

$$[BYTE] == 0 .. 255$$

A file is a finite sequence of bytes of any length including the null sequence $< >$ of 0 length. The set of all files is defined as

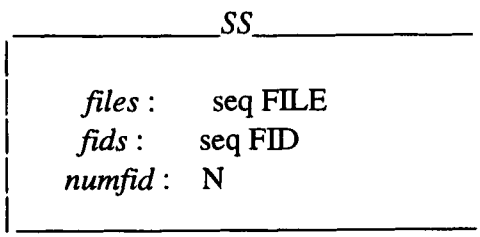
$$[FILE] == seq\ BYTE$$

For any file $f$ of the type FILE, $\#f$ is the size of the file, $f(1)$ is the first byte of the file and $f(\#f)$ is the last byte.

In the file storage system, files are stored and retrieved using file identifiers that are natural numbers. The set of all file identifiers is defined as

$$[FID] == N$$

Three state variables (invariant) are named in file storage system as *files*, *fids* and *numfid* to represent all files, file identifiers and the number of files in the storage system. The file storage system is described as the following Z schema *SS*.




6.1.2. Channel System

In order to allow easy sequential access to files, channels are defined to remember the current positions in the files. The state of the file in a channel is represented by its file identifier *cfid* and its position *posn* that is a natural number.

As in the file storage system, the channel system allows channels to be stored and retrieved using channel identifiers. A channel identifier *cid* is a UNIX 'file descriptor', and the set of all channel identifiers is defined as

$$[CID] == N$$

Four state variables are named in the channel system as *cfids*, *posns*, *cids* and *numcid* to represent all file identifiers and positions in the channel system, all channel identifiers and the number of channels in the system.

The channel system is described using the following Z schema *CS*.

*CS*

*cfids* : seq FID
*posns* : seq N
*cids* : seq CID
*numcid* : N

6.1.3. Naming System

Each file in the filing system has a name that is normally given by the user. The file naming system associates file names with file identifiers in the file storage system, so files can be stored and retrieved by their names. Besides file names, there are also directory names associated with directories to form the hierarchical filing structure. The full path name of a file includes its directory names and the file name. The set of all file names (full path names) NAME is defined as the sequence of syllables SYL that are sequence of characters.

$$[NAME] == seq\ SYL$$

$$[SYL] == seq\ CHAR$$

Three state variables *names*, *nfids*, *numname* are used to represent the file names, their identifiers and the number of names. The front of a name sequence is the directory name. The last of the name sequence is the file name. Only the empty sequence (the root directory "/") has no front.

Directories in UNIX system are stored as files and they are also associated with file identifiers. The content of each directory file describes the file structure stored under the directory such as file names and associated file identifiers. In order to clearly

describe the directories, two state variables *dnames* and *numdir* are used for directory names and the number of directories. For each directory three state variables *enames*, *efids* and *numentry* are used for entry file names, entry file identifiers and the number of file entries in the directory. The naming system is described using the following Z schema *NS*.

*NS*

| | |
|---|---|
| *names* : | seq NAME |
| *nfids* : | seq FID |
| *numname* : | N |
| *dnames* : | seq NAME |
| *numdir* : | N |
| *enames* : | seq seq SYL |
| *efids* : | seq seq FID |
| *numentry* : | seq N |

The complete UNIX filing system is the combination of the three separate systems given as the Z schema *FS*.

*FS*

*SS*
*CS*
*NS*

6.1.4. Initialisation of the Filing System

To generate test data sets automatically using ZTEST, the filing system needs to be set in a start state. The state variables in the filing system are initialised by a Z schema *Initial*.

*Initial*

*SS*
*CS*
*NS*

---

*files* = {65,68,73,80,65,66,107; ; ; 5, 87, 190 ; ; ; ; ; ; ;
63,68,63,88,65,66,10,98,129,100; 0 }
*fids* = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 20 }
*numfid* = 12
*cfids* = {1, 3, 12, 5, 20 }
*posns* = {2, 0, 8, 0, 0 }
*cids* = {1, 2, 3, 4, 5 }
*numcid* = 5
*names* = { /YANG/ZT; /YANG/ZT/ADD; /; /YANG; /YANG/ZT/FIND;
/TEMP; /YANG/DOCU; /YANG/DOCU/CV }
*nfids* = {1, 2, 3, 5, 8, 10, 12, 20 }
*numname* = 8
*dnames* = {/; /YANG; /YANG/ZT; /YANG/DOCU }
*numdir* = 4
*enames* = { YANG, TEMP; ZT, DOCU; ADD, FIND; CV }
*efids* = {5, 10; 1, 12; 2, 8; 20 }
*numentry* = {2, 2, 2, 1 }

---

The initial state of the filing system has 12 files, 5 channels and 8 names in storage. There are 4 directories among the 8 file names. The hierarchical structure of the files in the system is shown in Fig 15.

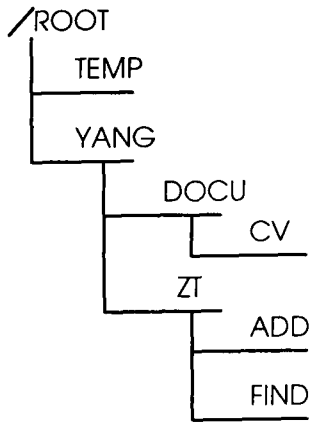



Fig 15. The hierarchical structure of files in the filing system.

## 6.2. Operations Covered in the Filing System

The twelve operations of UNIX filing system covered here by ZTEST for test data generation are open, close, seek, fstat, ls, link, unlink, move, create, destroy, read and write. Three of them are UNIX commands: ls, move('mv'), destroy('rm'), and others are UNIX system calls.

### 6.2.1. Operation 'open'

The operation 'open' opens a file from the filing system and creates a new channel for reading and writing. The operation takes an input *name?* which is a file name from the naming system, and gives an output *cid!* which is a new channel identifier created by the filing system. The filing system looks up the file identifier *nfid* of the file *name?* from the naming system, then stores it into channel system as *cfid*. The current position of the opened file in the new channel is initialised to zero. The operation is described in the following schemas *open0* and *NameError*.

In schema *open0* a universal quantifier is used to create a new channel id *cid!* which is not equal to any channel id in the channel system, then the number of channels is incremented by one ( *numcid'* ). The file name to be opened *name?* is defined as one of the *names* in the naming system by the existential quantifier. After the creation of the new channel and having known the file name to be opened, other state variables in the channel system are also moved to the new status by using function overriding. The channel file id for the new channel *cfids(numcid')* is set to the file id of *name?* which is

*nfids(j)*. The channel id for the new channel *cids(numcid')* is set to *cid!* generated. The current position of the new channel *posns(numcid')* is set to zero. The operation changes the status of the channel system *CS*.

*open0*

$\Xi SS$
$\Delta CS$
$\Xi NS$
*name?* : NAME
*cid!* : CID

---

$\forall$ i : 1 .. *numcid* | *cid!* $\neq$ *cids*(i)
*numcid'* = *numcid* + 1
$\exists$ j : 1 .. *numname* | *name?* = *names*(j)
*cfids'* = *cfids* $\oplus$ { *numcid'* $\mapsto$ *nfids*(j) }
*cids'* = *cids* $\oplus$ { *numcid'* $\mapsto$ *cid!* }
*posns'* = *posns* $\oplus$ { *numcid'* $\mapsto$ 0 }

REPORT ::== No Such Cid | No Such Name | No Such Dir | No Such Fid

*NameError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*name?* : NAME
*report!* : REPORT

---

$\forall$ i : 1 .. *numname* | *name?* $\neq$ *names*(i)
*report!* = No Such Name

The schema *NameError* gives an output *report!* as an error report for unsuccessful operations using an universal quantifier, in this case, the input variable *name?* is

unknown in the naming system. The operation does not change the filing system. The complete operation is defined as

$$open \triangleq open0 \vee NameError$$

Fifteen test cases are generated by ZTEST for the input *name?*.

Test Cases

```
Test Case  1  Invalid { }
Test Case  2  Invalid {   A    I    J    N    s    H    c    O    e    r    v}
Test Case  3  Invalid {   F    y    y    l    r    o    C    d    M    O    W
                         E    S}
Test Case  4  Invalid {  269  322  311    1  393  134  230  193  188  290
                        301  139}
Test Case  5  Invalid { 0.17  0.8 0.29 0.33  0.4 0.24 0.41 0.05 0.49 0.98
                        0.0  0.4}
Test Case  6  Invalid {   @    @    @    @    @    @    @    @    @    @    @}
Test Case  7  Invalid {   {    {    {    {    {    {    {    {    {    {    { }
Test Case  8  Valid   {   A    A    A    A    A    A    A    A    A    A    A    A}
Test Case  9  Valid   {   z    z    z    z    z    z    z    z    z    z    z    z}
```

route 1 Fitness function:

```
For all i, : 1 .. numcid
      cid != cids (i )

There exists j, : 1 .. numname
      name? = names (j )
      cfids' = cfids &fovr. numc' &map. nfids (j )
      cids' = cids &fovr. numc' &map. cid!
      posns' = posns &fovr. numc' &map. 0
```

```
Test Case  10 Valid  { Y A N G Z T        }
  cid! =    8
  cfids' = {   1  3  12   5  20   1}
  cids'  = {   1  2   3   4   5   8}
  posns' = {   2  0   8   0   0   0}

Test Case  11 Valid  { Y A N G Z T   A D D }
  cid! =    8
  cfids' = {   1   3   12  5  20   2}
  cids'  = {   1   2   3   4   5   8}
  posns' = {   2   0   8   0   0   0}

Test Case  12 Valid  { Y A N G          }
  cid! =    8
```

cfids' = { 1 3 12 5 20 5}
cids' = { 1 2 3 4 5 8}
posns' = { 2 0 8 0 0 0}

Test Case 13 Valid { Y A N G D O C U }
cid! = 8
cfids' = { 1 3 12 5 20 12}
cids' = { 1 2 3 4 5 8}
posns' = { 2 0 8 0 0 0}

Test Case 14 Valid { Y A N G D O C U C V }
cid! = 8
cfids' = { 1 3 12 5 20 20}
cids' = { 1 2 3 4 5 8}
posns' = { 2 0 8 0 0 0}

route 2 Fitness function:

For all i, : 1 .. numname
name? != names (i )
report! = No Such Name

Test Case 15 Valid { W Y I K A G A P J G Z S}
report! = No Such Name

The first seven test cases are generated for invalid input *name?* which is an alphabetic string of the length twelve. Test cases 10 to 14 are generated for valid input *name?* to test the input typed predicate in schema *open0*. The test data set is given by direct assignment from the fitness function of the existential quantifier that it's boundary of search domain is known. These test cases test the four boundary values and an intermediate value of the search domain ( *name?* = *names*(j); j = 1, 2, 4, 7, 8 ). The output value cid! is generated at random from the objective function of the universal quantifier and the values of state variables are set by direct assignment from the objective functions of function overriding in schema *open0*. The last test case is for valid input of schema *NameError*, *name?* is a name unknown in state variable *names*. The test data is generated using GA to test the input typed predicate in schema

*NameError*, that is the universal quantifier. The test case gives an output "No Such Name" by direct assignment.

### 6.2.2. Operation 'seek'

The operation 'seek' seeks a given position of the file in a channel. The operation takes two inputs *cid?* and *newposn?* which are the channel id and the new position for the file. The filing system looks up the given channel id *cid?* from the channel system, then sets the new position *newposn?* for the channel. The operation is described in the following schemas *seek0* and *CidError*.

*seek0*

$$\Xi SS$$
$$\Delta CS$$
$$\Xi NS$$
$$cid? : \mathrm{CID}$$
$$newposn? : \mathrm{N}$$

---

$$\exists i : 1 \,..\, numcid \mid cid? = cids(i)$$
$$posns' = posns \oplus \{ i \mapsto newposn? \}$$
$$cfids' = cfids$$
$$cids' = cids$$
$$numcid' = numcid$$

In schema *seek0* the input channel id *cid?* is defined as one of the channel ids in the channel system using the existential quantifier. Having known the channel id, the current position associated is set to the new position *newposn?* by function overriding. The operation only changes the current position *posns* in the channel system, other state variables *cids*, *cfids* and *numcid* stay the same. The schema *CidError* gives an output

error message for unsuccessful operation using a universal quantifier, in this case, the input variable *cid?* is unknown in the channel system.

*CidError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*cid?* : CID
*report!* : REPORT

---

$\forall$ i : 1 .. *numcid* | *cid?* $\neq$ *cids*(i)
*report!* = No Such Cid

The complete operation is defined as:

$$seek \triangleq seek0 \vee CidError$$

Fifteen test cases are generated by ZTEST for inputs *cid?* and *newposn?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 5 }
Test Case 3 Invalid { 8 21786 31678}
Test Case 4 Invalid { J N}
Test Case 5 Invalid { 0.7 0.94}
Test Case 6 Invalid { 11 32768}
Test Case 7 Invalid { 0 -1}
Test Case 8 Valid { 10 32767}
Test Case 9 Valid { 1 0 }

route 1 Fitness function:

There exists i, : 1 .. numc
cid? = cids (i )
posns' = posns &fovr. i &map. newposn?
cids' = cids
cfids' = cfids
numcid' = numcid

```
Test Case  10  Valid   {     1  15903}
  posns' = { 15903    0   8  0   0}
  cids'  = {    1    2    3    4    5}
  cfids' = {    1    3   12    5   20}
  numcid' = {    5}

Test Case  11  Valid   {     2  10036}
  posns' = {    2 10036   8  0   0 }
  cids' = {    1    2    3    4    5}
  cfids' = {    1    3   12    5   20}
  numcid' = {    5}

Test Case  12  Valid   {     3  27625}
  posns' = {    2   0  27625  0    0 }
  cids' =  {    1    2    3    4    5}
  cfids' = {    1    3   12    5   20}
  numcid' = {    5}

Test Case  13  Valid   {     4   1002}
  posns' = {    2    0    8  1002  0}
  cids' = {    1    2    3    4    5}
  cfids' = {    1    3   12    5   20}
  numcid' = {    5}

Test Case  14  Valid   {     5  11137}
  posns' = {    2    0    8  0  11137}
  cids' =  {    1    2    3    4    5}
  cfids' =  {    1    3   12    5   20}
  numcid' = {    5}
```

route 2  Fitness function:

```
For all i, : 1 .. numc
cid? != cids (i )
```

```
Test Case  15  Valid   {     9  30403}
  report! = No Such Cid
```

The input *cid?* is defined as the channel id which is a natural number in the range of [1, 10], and the new position *newposn?* is defined as a natural number. The first seven test cases are generated for invalid inputs. Test cases 10 to 14 are generated for valid inputs *cid?* and *newposn?* to test the input typed predicate in schema *seek0*. The test data are given by direct assignment from the fitness function of the existential quantifier that it's

boundary of search domain is known. These test cases test the boundary values and a value in the middle of the search domain of *cid?* (*cid?* = *cids*(i); i = 1, 2, 3, 4, 5). The value of state variable *posns* in the channel system is also changed after the operation according to the values of inputs for each test case. The last test case is for valid input of schema *CidError*, *cid?* is a number that unknown in state variable *cids* and is generated using GA from the fitness function of the universal quantifier in schema *CidError*. The test case gives an output "No Such Cid" by direct assignment.

### 6.2.3. Operation 'fstat

The operation 'fstat' returns the size of the file in a channel accessed with a given channel id. The operation takes an input *cid?* which is the channel id and looks up for the file associated with the *cid?*, finally gives an output *size!* as the size of the file in the channel. The operation is described in the following schemas *fstat0* and *CidError*.

*fstat0*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*cid?* : CID
*size!* : N

---

$\exists$ i : 1 .. *numcid* | *cid?* = *cids*(i)
  $\exists$ j : 1 .. *numfid* | *fids*(j) = *cfids*(i)
    *size!* = #*files*(j)

In schema *fstat0* the input channel id *cid?* is defined as one of the channel ids in the channel system using the existential quantifier. Having known the channel id, the index j of the file in the filing system *files* is defined using another existential quantifier in

schema *fstat0*. There exists a j from 1 to *numfid*, thus the file id *fids*(j) is the same as the file id *cfids*(i) associated with the channel *cid?*. The output *size!* equals to the size of the file *files*(j). The operation does not change any state variables in the filing system.

*CidError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*cid?* : CID
*report!* : REPORT

---

$\forall$ i : 1 .. *numcid* | *cid?* $\neq$ *cids*(i)
*report!* = No Such Cid

The schema *CidError* gives an output error message for unsuccessful operation. The complete operation is defined as:

$$fstat \triangleq fstat0 \vee CidError$$

Fourteen test cases are generated by ZTEST for the input *cid?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 10 6}
Test Case 3 Invalid { I}
Test Case 4 Invalid { 0.35}
Test Case 5 Invalid { 11}
Test Case 6 Invalid { 0}
Test Case 7 Valid { 10}
Test Case 8 Valid { 1}

route 1 Fitness function:

There exists i, : 1 .. numc
cid? = cids (i )

There exists j, : 1 .. numf
fids (j )= cfids (i )

```
                size! = # files (j )

Test Case  9 Valid   {     1}
   size! =    7

Test Case  10 Valid   {     2}
   size! =    0

Test Case  11 Valid   {     3}
   size! =    10

Test Case  12 Valid   {     4}
    size! =    0

Test Case  13 Valid   {     5}
   size! =    1

route 2   Fitness function:

                For all i, : 1 .. numc
                cid? != cids (i )

Test Case  14 Valid   {     7}
  report! = No Such Cid
```

The input *cid?* is defined as the channel id which is a natural number in the range of [1, 10]. The first six test cases are generated for invalid inputs. Test cases 9 to 13 are generated for valid input *cid?* to test the input typed predicate in schema *fstat0*. The test data are given by direct assignment from the fitness function of the first existential quantifier in schema *fstat0* with a given search domain. These test cases test the boundary values and an intermediate value in the search domain of *cid?*. For each value i, ZTEST system checks values of j to find the j that makes *fids*(j) = *cfids*(i) true as described in the second existential quantifier of schema *fstat0*. The output *size!* is set to the size of the file *files*(j) by direct assignment.

ZTEST system treats the two existential quantifiers differently in schema *fstst0*. The first one is of input type and five test cases are generated from it for the input variable

*cid?*. The second one is of state type, it does not change the value of any input and state variables. No extra test cases are generated, ZTEST checks values in the search domain of the second existential quantifier until the value j is found to make the predicate *fids*(j) = *cfids*(i) true. All values of state variables are the same after the operation. The last test case is for valid input of schema *CidError*, *cid?* is a number that unknown in state variable *cids* and is generated using GA from the fitness function of the universal quantifier in schema *CidError*. The test case gives an output "No Such Cid" by direct assignment.

### 6.2.4. Operation 'close'

The operation 'close' removes a given channel from the channel system. The operation takes an input *cid?* which is the channel id. The system looks up the given channel id *cid?* from the channel system, then removes the channel from *cids*, *cfids* and *posns* of the channel system. The value of *numcid* is decreased by one. The operation is described in the following schemas *close0* and *CidError*.

*close0*

$\Xi SS$
$\Delta CS$
$\Xi NS$
*cid?* : CID

---

$\exists$ i : 1 .. *numcid* | *cid?* = *cids*(i)
*cids'* = { i } $\lhd$ *cids*
*cfids'* = { i } $\lhd$ *cfids*
*posns'* = { i } $\lhd$ *posns*
*numcid'* = *numcid* - 1

In schema *close0* the input *cid?* is defined as one of the channel ids in the channel system using the existential quantifier. Having known the index i of *cid?* in the channel system, the values of *cids*(i), *cfids*(i) and *posns*(i) associated with the given channel are removed by domain subtractions. The operation changes the status of the channel system *CS*. The schema *CidError* gives an output error message for unsuccessful operation. The complete operation is defined as:

$$close \triangleq close0 \vee CidError$$

*CidError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*cid?* : CID
*report!* : REPORT

---

$\forall$ i : 1 .. *numcid* | *cid?* $\neq$ *cids*(i)
*report!* = No Such Cid

Fourteen test cases are generated by ZTEST for the input *cid?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 2 9 }
Test Case 3 Invalid { I}
Test Case 4 Invalid { 0.35}
Test Case 5 Invalid { 11}
Test Case 6 Invalid { 0}
Test Case 7 Valid { 10}
Test Case 8 Valid { 1}

route 1 Fitness function:

There exists j, : 1 .. numc
cid? = cids (j )
cids' = {j , } dsub cids
cfids' = {j , } dsub cfids
posns' = {j , } dsub posns
numc' = numc - 1

```
Test Case  9 Valid   {     1}
  cids' = {    2    3    4    5}
  cfids' = {    3   12    5   20}
  posns' = {    0    8    0    0}
  numc' = {    4}

Test Case  10 Valid   {     2}
  cids' = {    1    3    4    5}
  cfids' = {    1   12    5   20}
  posns' = {    2    8    0    0}
  numc' = {    4}

Test Case  11 Valid   {     3}
  cids' = {    1    2    4    5}
  cfids' = {    1    3    5   20}
  posns' = {    2    0    0    0}
  numc' = {    4}

Test Case  12 Valid   {     4}
  cids' = {    1    2    3    5}
  cfids' = {    1    3   12   20}
  posns' = {    2    0    8    0}
  numc' = {    4}

Test Case  13 Valid   {     5}
  cids' = {    1    2    3    4}
  cfids' = {    1    3   12    5}
  posns' = {    2    0    8    0}
  numc' = {    4}

route 2   Fitness function:

                For all i,  : 1 .. numc
                cid? != cids (i )

Test Case  14 Valid   {     9}
  report! = No Such Cid
```

The input *cid?* is defined as the channel id which is a natural number in the range of [1, 10]. The first six test cases are generated for invalid inputs. Test cases 9 to 13 are generated for valid input *cid?* to test the input typed predicate in schema *close0* which is the same as in schema *fstat0*. The values of state variable *cids*, *cfids* and *posns* in the channel system are changed after the operation by domain subtraction according to the

values of input for each test case. The value of state variable *numcid* is reduced by one. The last test case is for valid input *cid?* in schema *CidError*, which is the same as in operation 'fstat'.

### 6.2.5. Operation 'ls'

The operation 'ls' lists the names of the files in a given directory. The operation takes an input *dir?* which is the name of the directory, looks up it from the directory names in the naming system, then gives an output *contents!* which is the entry file list under the directory *dir?*. The operation is described in the following schemas *ls0* and *DirError*.

*ls0*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*dir?* : DIR
*contents!* : seq SYL

---

$\exists$ i : 1 .. *numdir* | *dir?* = *dnames*(i)
*contents!* = { j : 1 .. *numentry*(i) • *enames*(i)(j) }

*DirError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*dir?* : DIR
*report!* : REPORT

---

$\forall$ i : 1 .. *numdir* | *dir?* $\neq$ *dnames*(i)
*report!* = No Such Dir

In schema *ls0* the input *dir?* is defined as one of the directory names in the naming system using the existential quantifier. Having known the index i of *dir?* in the naming system, the output *contents!* is set to the list of all file entry names under the directory using the set comprehension. The operation does not change the status of the filing system.

The schema *DirError* gives an output error message for unsuccessful operation using a universal quantifier, in this case, the input variable *dir?* is unknown in the naming system. The complete operation is defined as:

$$ls \triangleq ls0 \vee DirError$$

Fourteen test cases are generated by ZTEST for the input *dir?*.

Test Cases

```
Test Case  1  Invalid { }
Test Case  2  Invalid {    A    I    J    N    s    H    c  }
Test Case  3  Invalid {    F    y    y    l    r    o    C    d    M  }
Test Case  4  Invalid {  269  322  311    1  393  134  230  193  }
Test Case  5  Invalid {  0.17  0.8  0.29  0.33  0.4  0.24  0.41  0.05  }
Test Case  6  Invalid {    @    @    @    @    @    @    @    @  }
Test Case  7  Invalid {    {    {    {    {    {    {    {    {  }
Test Case  8  Valid   {    A    A    A    A    A    A    A    A  }
Test Case  9  Valid   {    z    z    z    z    z    z    z    z  }

 route 1   Fitness function:
                There exists i, : 1 .. numd
                dir? = dnames (i )
               contents! = { j: 1 .. nume (i )&bul. enames (i )(j ) }

Test Case  10  Valid   { /                }
    contents! = YANG TEMP

Test Case  11 Valid   { Y A N G          }
    contents! = ZT  DOCU

 Test Case  12  Valid   { Y A N G Z T       }
    contents! = ADD  FIND
```

Test Case 13 Valid { Y A N G D O C U }
contents! = CV

route 2 Fitness function:
For all i, : 1 .. numd
dir? != dnames (i )

Test Case 14 Valid { G A O D X D T L}
report! = No Such Dir

The first seven test cases are generated for invalid input *dir?* which is an alphabetic string of the length eight. Test cases 10 to 13 are generated for valid input *dir?* to test the input typed predicate in schema *ls0*. The test data are given by direct assignment from the fitness function of the existential quantifier in schema *ls0* with a given search domain. These test cases test the four values of *dir?* from *dnames* (the number of directories is 4). The output values of *contents!* are set to the file entry names under the *dir?* after the operation by set comprehension according to the values of input for each test case. The last test case is for valid input of schema *DirError*, *dir?* is unknown in the state variable *dnames* and is generated using GA from the fitness function of the universal quantifier in schema *DirError*. The test case gives an output "No Such Dir" by direct assignment.

### 6.2.6. Operation 'link'

The operation 'link' makes a new filename refer to the same file as does a old filename. The operation takes three inputs *oldname?*, *newdir?* and *newname?* which are the old file name, the new directory name and the new file entry name. The filing system looks up the *oldname?* and *newdir?* from the names and directory names in the naming system. The new file entry name *newname?* is a unknown entry name under the

*newdir?*. New entries are set for state variables in the naming system. The operation is described in the following schemas *link0, NameError* and *DirError*.

*link0*

$\Xi SS$
$\Xi CS$
$\Delta NS$
*oldname?* : NAME
*newdir?* : DIR
*newname?* : SYL

---

$\exists$ i : 1 .. *numname* | *oldname?* = *names* (i)
$\exists$ j : 1 .. *numdir* | *newdir?* = *dnames*(j)
$\forall$ k : 1 .. *numentry*(j) | *newname?* $\neq$ *enames*(j)(k)
*numname'* = *numname* + 1
*nfids'* = *nfids* $\oplus$ { *numname'* $\mapsto$ *nfids*(i) }
*names'* = *names* $\oplus$ { *numname'* $\mapsto$ (*newdir?* ^ *newname?*) }
*numentry'*(j) = *numentry*(j) + 1
*enames'*(j) = *enames*(j) $\oplus$ { *numentry'*(j) $\mapsto$ *newname?* }
*efids'*(j) = *efids*(j) $\oplus$ { *numentry'*(j) $\mapsto$ *nfids*(i) }

*NameError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*oldname?* : NAME
*report!* : REPORT

---

$\forall$ i : 1 .. *numname* | *oldname?* $\neq$ *names*(i)
*report!* = No Such Name

In schema *link0* the input *oldname?* is defined as one of the file names and the input *newdir?* is defined as one of the directory names in the naming system using the existential quantifiers. The input *newname?* is defined as a unknown entry name under

the *newdir?* using the universal quantifier. Having known the *oldname?*, *newdir?* and having created the *newname?*, other state variables in the naming system are set to the new values by using function overriding. The number of names *numname* is increased by one. The operation changes the status of the naming system.

*DirError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*newdir?* : DIR
*report!* : REPORT

---

$\forall$ i : 1 .. *numdir* | *newdir?* $\neq$ *dnames*(i)
*report!* = No Such Dir

The schema *NameError* and *DirError* give error messages for unsuccessful operations. The complete operation is defined as:

$$link \triangleq link0 \vee NameError \vee DirError$$

Thirty one test cases are generated by ZTEST for inputs *oldname?*, *newdir?* and *newname?*.

Test Cases

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Test Case | 1 | Invalid { } | | | | | | | | | |
| Test Case | 2 | Invalid { | A | I | J | N | s | H | c | O | e | r |
| | | | v | F | y | y | l | r | o | C | d | M |
| | | | O | W | E} | | | | | | | |
| Test Case | 3 | Invalid { | S | n | Q | U | L | j | P | e | H | K |
| | | | K | m | A | B | g | J | F | F | t | x |
| | | | l | A | L | P | c} | | | | | |
| Test Case | 4 | Invalid { | 324 | 443 | 437 | 21 | 276 | 255 | 310 | 385 | 126 | 108 |
| | | | 117 | 167 | 283 | 66 | 307 | 15 | 474 | 106 | 404 | 314 |
| | | | 357 | 366 | 275 | 285} | | | | | | |
| Test Case | 5 | Invalid { | 0.16 | 0.33 | 0.86 | 0.82 | 0.61 | 0.95 | 0.99 | 0.98 | 0.73 | 0.5 |
| | | | 0.29 | 0.71 | 0.28 | 0.63 | 0.49 | 0.17 | 0.55 | 0.38 | 0.77 | 0.38 |

```
                                0.65  0.12  0.14  0.33}
Test Case  6  Invalid {   @   @   @   @   @   @   @   @   @   @
                          @   @   @   @   @   @   @   @   @   @
                          @   @   @   @}
Test Case  7  Invalid {   {   {   {   {   {   {   {   {   {   {
                          {   {   {   {   {   {   {   {   {   {
                          {   {   {   {}
Test Case  8  Valid   {   A   A   A   A   A   A   A   A   A   A
                          A   A   A   A   A   A   A   A   A   A
                          A   A   A   A}
Test Case  9  Valid   {   z   z   z   z   z   z   z   z   z   z
                          z   z   z   z   z   z   z   z   z   z
                          z   z   z   z  }
```

route 1  Fitness function:

There exists i, : 1 .. numn
oldname? = names (i )

There exists j, : 1 .. numd
newdir? = dnames (j )

For all k, : 1 .. nume (j )
newname? != enames (j )(k )
numn' = numn + 1
nfids' = nfids &fovr. numn' &map. nfids (i ),
names' = names &fovr. numn' &map. newdir? &cat. newname? ,
nume' = nume &fovr. j &map. nume (j )+ 1,
enames' (j )= enames' (j )&fovr. nume' (j )&map. newname? ,
efids' (j )= efids' (j )&fovr. nume' (j )&map. nfids (i ),

```
Test Case  10  Valid   { Y A N G Z T        /        T J S D}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   1}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;/  ,  ,TJSD}
  nume' = {   3   2   2   1}
  enames' = {YANG,TEMP,TJSD;ZT ,DOCU;ADD ,FIND;CV}
  efids' = {   5   10   1,   1   12,   2   8,   20}

Test Case  11  Valid   { Y A N G Z T        Y A N G      A K J O}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   1}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,   ,AKJO}
  nume' = {   2   3   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU,AKJO;ADD ,FIND;CV}
  efids' = {   5   10,   1   12   1,   2   8,   20}
```

```
Test Case 12 Valid  { Y A N G Z T      Y A N G Z T  E I G Y}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   1}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,ZT ,EIGY}
  nume' = {   2   2   3   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,EIGY;CV}
  efids' = {   5   10,   1   12,   2   8   1,   20}

Test Case 13 Valid  { Y A N G Z T      Y A N G D O C U B E M T}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   1}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,DOCU,BEMT}
  nume' = {   2   2   2   2}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,BEMT}
  efids' = {   5   10,   1   12,   2   8,   20   1}

Test Case 14 Valid  { Y A N G Z T   A D D  /      I F M F}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   2}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;/  ,   ,IFMF}
  nume' = {   3   2   2   1}
  enames' = {YANG,TEMP,IFMF;ZT ,DOCU;ADD ,FIND;CV}
  efids' = {   5   10   2,   1   12,   2   8,   20}

Test Case 15 Valid  { Y A N G Z T   A D D  Y A N G    C P V U}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   2}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,   ,CPVU}
  nume' = {   2   3   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU,CPVU;ADD ,FIND;CV}
  efids' = {   5   10,   1   12   2,   2   8,   20}

Test Case 16 Valid  { Y A N G Z T   A D D  Y A N G Z T   B R B Z}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   2}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,ZT ,BRBZ}
  nume' = {   2   2   3   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,BRBZ;CV}
  efids' = {   5   10,   1   12,   2   8   2,   20}

Test Case 17 Valid  { Y A N G Z T   A D D  Y A N G D O C U T R F K}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   2}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
```

```
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,DOCU,TRFK}
  nume' = {   2   2   2   2}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,TRFK}
  efids' = {   5  10,   1  12,   2   8,  20   2}

Test Case 18 Valid  { Y A N G        /        O D F C}
  numn' = {   9}
  nfids' = {   1   2   3   5   8  10  12  20   5}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;/  ,  ,ODFC}
  nume' = {   3   2   2   1}
  enames' = {YANG,TEMP,ODFC;ZT ,DOCU;ADD ,FIND;CV}
  efids' = {   5  10   5,   1  12,   2   8,  20}

Test Case 19 Valid  { Y A N G        Y A N G     U U H I}
  numn' = {   9}
  nfids' = {   1   2   3   5   8  10  12  20   5}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,  ,UUHI}
  nume' = {   2   3   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU,UUHI;ADD ,FIND;CV}
  efids' = {   5  10,   1  12   5,   2   8,  20}

Test Case 20 Valid  { Y A N G        Y A N G Z T   D U A E}
  numn' = {   9}
  nfids' = {   1   2   3   5   8  10  12  20   5}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,ZT ,DUAE}
  nume' = {   2   2   3   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,DUAE;CV}
  efids' = {   5  10,   1  12,   2   8   5,  20}

Test Case 21 Valid  { Y A N G        Y A N G D O C U R F K I}
  numn' = {   9}
  nfids' = {   1   2   3   5   8  10  12  20   5}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,DOCU,RFKI}
  nume' = {   2   2   2   2}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,RFKI}
  efids' = {   5  10,   1  12,   2   8,  20   5}

Test Case 22 Valid  { Y A N G D O C U     /        X N U B}
  numn' = {   9}
  nfids' = {   1   2   3   5   8  10  12  20  12}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;/  ,  ,XNUB}
  nume' = {   3   2   2   1}
  enames' = {YANG,TEMP,XNUB;ZT ,DOCU;ADD ,FIND;CV}
  efids' = {   5  10  12,   1  12,   2   8,  20}
```

```
Test Case 23 Valid  { Y A N G D O C U     Y A N G     N N M X}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   12}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,   ,NNMX}
  nume' = {   2   3   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU,NNMX;ADD ,FIND;CV}
  efids' = {   5   10,   1   12   12,   2   8,   20}

Test Case 24 Valid  { Y A N G D O C U     Y A N G Z T   P Z E I}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   12}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,ZT ,PZEI}
  nume' = {   2   2   3   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,PZEI;CV}
  efids' = {   5   10,   1   12,   2   8   12,   20}

Test Case 25 Valid  { Y A N G D O C U     Y A N G D O C U O I J K}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   12}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,DOCU,OIJK}
  nume' = {   2   2   2   2}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,OIJK}
  efids' = {   5   10,   1   12,   2   8,   20   12}

Test Case 26 Valid  { Y A N G D O C U C V   /       I C B T}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   20}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;/  ,   ,ICBT}
  nume' = {   3   2   2   1}
  enames' = {YANG,TEMP,ICBT;ZT ,DOCU;ADD ,FIND;CV}
  efids' = {   5   10   20,   1   12,   2   8,   20}

Test Case 27 Valid  { Y A N G D O C U C V   Y A N G     O Q O X}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   20}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,   ,OQOX}
  nume' = {   2   3   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU,OQOX;ADD ,FIND;CV}
  efids' = {   5   10,   1   12   20,   2   8,   20}

Test Case 28 Valid  { Y A N G D O C U C V   Y A N G Z T   E W X Y}
  numn' = {   9}
  nfids' = {   1   2   3   5   8   10   12   20   20}
```

```
names' = {YANG,ZT ;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
          TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,ZT ,EWXY}
nume' = {  2  2  3  1}
enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,EWXY;CV}
efids' = {  5  10,  1  12,  2  8  20,  20}

Test Case 29 Valid  { Y A N G D O C U C V  Y A N G D O C U P F T X}
  numn' = {  9}
  nfids' = {  1  2  3  5  8  10  12  20  20}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV;YANG,DOCU,PFTX}
  nume' = {  2  2  2  2}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,PFTX}
  efids' = {  5  10,  1  12,  2  8,  20  20}

route 2  Fitness function:
           For all i, : 1 .. numn
           oldname? != names (i )

Test Case 30 Valid  { T C L D S X I H S I V O B T Y A U B A V W Q W E}
  report! = No Such Name

route 3  Fitness function:
           For all i, : 1 .. numd
           newdir? != dnames (i )

Test Case 31 Valid  { J K L Y G V Q J M G M R B G U K G M J J A D U H}
  report! = No Such Dir
```

The first seven test cases are generated for invalid inputs *oldname?*, *newdir?* and *newname?* which are alphabetic strings of the length twelve, eight and four, respectively. Test cases 10 to 29 are generated for valid inputs *oldname?*, *newdir?* and *newname?* to test the input typed predicates in schema *link0*. These test cases test four boundary values and an intermediate value of *oldname?* from it's search domain in the first input existential quantifier in schema *link0*. For each value of *oldname?*, the test cases test four values of *newdir?* from it's search domain *dnames* in the second input existential quantifier. The test data for *oldname?* and *newdir?* are given by direct assignment, there are twenty valid test cases in total. In each valid test case, a *newname?* is generated using GA as a four character string from the fitness function of

the universal quantifier in schema *link0* for the file entry name. New entry values are added to state variables in the naming system after the operation by function overriding, and the number of names *numname* is increased by one. The last two test cases are for valid input of schema *NameError* and *DirError*, *oldname?* and *newdir?* are unknown from their search domain. The test cases are generated using GA from the fitness functions of universal quantifiers in schema *NameError* and *DirError*.

### 6.2.7. Operation 'unlink'

The operation 'unlink' removes a given file from the naming system, but does not destroy the file from the storage system. The operation takes an input *name?* which is the file name, looks it up from the naming system and removes the associated entry values from other state variables. The operation is described in the following schemas *unlink0* and *NameError*.

*unlink0*

$\Xi SS$
$\Xi CS$
$\Delta NS$
*name?* : NAME

---

$\exists$ i : 1 .. *numname* | *name?* = *names*(i)
*names'* = { i } $\lhd$ *names*
*nfids'* = { i } $\lhd$ *nfids*
*numname'* = *numname* - 1
$\exists$ j : 1 .. *numdir* | *dnames*(j) = front( *name?*)
$\exists$ k : 1 .. *numentry*(j) | *enames*(j)(k) = last( *name?*)
*numentry'*(j) = *numentry*(j) - 1
*enames'*(j) = { k } $\lhd$ *enames*(j)
*efids'*(j) = { k } $\lhd$ *efids*(j)

*NameError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*name?* : NAME
*report!* : REPORT

---

$\forall$ i : 1 .. *numname* | *name?* $\neq$ *names*(i)
*report!* = No Such Name

In schema *unlink0* the input *name?* is defined as one of the file names in the naming system by using the existential quantifier. Having known the index i of *name?* in *names*, the entries associated with the index are removed by using domain subtraction. The file entry name *enames*(j)(k) associated with *name?* is defined by the second and the third existential quantifiers assuming that the front of the character sequence front(*name?*) is the directory name and the last of the sequence last(*name?*) is the file entry name. These two existential quantifiers are of state type, ZTEST system checks the value of j to make *dnames*(j) = front(*name?*) true and the value of k to make *enames*(j)(k) = last(*name?*) true. The entries *enames*(j)(k) and *efids*(j)(k) are removed by using the domain subtraction. The number of file entries *numentry*(j) is reduced by one. The operation changes the status of the naming system. The schema *NameError* gives an output error message for unsuccessful operations. The complete operation is defined as:

$$unlink \triangleq unlink0 \vee NameError$$

Fourteen test cases are generated by ZTEST for the input *name?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { A I J N s H c O e r

```
                                v}
Test Case  3  Invalid {   F    y    y    l    r    o    C    d    M    O
                          W    E    S}
Test Case  4  Invalid {  269  322  311    1  393  134  230  193  188  290
                  301  139}
Test Case  5  Invalid {  0.17  0.8  0.29  0.33  0.4  0.24  0.41  0.05  0.49  0.98
                         0.0  0.4}
Test Case  6  Invalid {   @    @    @    @    @    @    @    @    @    @
                          @    @}
Test Case  7  Invalid {   {    {    {    {    {    {    {    {    {    {
                          {    {}
Test Case  8  Valid   {   A    A    A    A    A    A    A    A    A    A
                          A    A}
Test Case  9  Valid   {   z    z    z    z    z    z    z    z    z    z
                          z    z }
```

route 1  Fitness function:

```
There exists i, : 1 .. numn
name? = names (i )
names' = {i , } dsub names
nfids' = {i , } dsub nfids
numn' = numn - 1
There exists j, : 1 .. numd
dnames (j )= &front. (name? )
There exists k, : 1 .. nume (j )
enames (j )(k )= &last. (name? )
nume' (j )= nume (j )- 1
enames' (j )= {k , } dsub enames (j )
efids' (j )= {k , } dsub efids (j )
```

```
Test Case  10  Valid   { Y A N G Z T        }
  names' = {YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV}
  nfids' = {    2    3    5    8   10   12   20}
  numn' = {     7}
  nume' = {     2    1    2    1}
  enames' = {YANG,TEMP;DOCU;ADD ,FIND;CV}
  efids' = {    5   10,   12,    2    8,   20}

Test Case  11  Valid   { Y A N G Z T   A D D }
  names' = {YANG,ZT  ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV}
  nfids' = {    1    3    5    8   10   12   20}
  numn' = {     7}
  nume' = {     2    2    1    1}
  enames' = {YANG,TEMP;ZT  ,DOCU;FIND;CV}
  efids' = {    5   10,    1   12,    8,   20}
```

```
Test Case 12 Valid  { Y A N G          }
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,CV}
  nfids' = {   1   2   3   8   10   12   20}
  numn' = {   7}
Solution not found in state exists function.

Test Case 12 Valid  { Y A N G D O C U     }
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU,CV}
  nfids' = {   1   2   3   5   8   10   20}
  numn' = {   7}
  nume' = {   2   1   2   1}
  enames' = {YANG,TEMP;ZT ;ADD ,FIND;CV}
  efids' = {   5   10,   1,   2   8,   20}

Test Case 13 Valid  { Y A N G D O C U C V  }
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU}
  nfids' = {   1   2   3   5   8   10   12}
  numn' = {   7}
  nume' = {   2   2   2   0}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;}
  efids' = {   5   10,   1   12,   2   8,}

Test Case 14 Valid  { L K S P P H J I E A G L}
  report! = No Such Name
```

The first seven test cases are generated for invalid input *name?* which is a alphabetic string of the length twelve. Test cases 10 to 13 are generated for valid input *name?* to test the input predicate in schema *unlink0*. The test data are given by direct assignment from the fitness function of the first existential quantifier in schema *unlink0*, and test the boundary values and an intermediate value of *name?* in it's search domain. The values of state variables in the naming system are also set to new values after the operation for each test case using direct assignment from domain subtraction and the two state typed existential quantifiers. Note that in test case 12 (i=4) the *name?* is YANG which is a directory name without an attached file entry name, it means front(*name?*) = *name?* and last(*name?*) is empty. ZTEST failed the search for a file entry name in the third

existential quantifier ∃ k : ℓ .. *numentry*(j) | *enames*(j)(k) = last( *name?*) because the value of last( *name?*) is empty. The test case was abandoned and a new one is generated (i=7). The last test case is for valid input of schema *NameError*, *name?* is a name unknown in the state variable *names*, which is the same as in operation 'link'.

### 6.2.8. Operation 'move'

The operation 'move' renames a file from a old filename to a new filename. The operation takes three inputs *oldname?*, *newdir?* and *newname?* which are the old file name, the new directory name and the new file entry name. The filing system looks up the *oldname?* and *newdir?* from the names and directory names in the naming system and generates a *newname?* which is a unknown entry name under the *newdir?*. The number of names *numname* and file ids *nfids* stay the same after the operation, other entry values of state variables associated with *oldname?* are reset to be associated with *newdir?* and *newname?* in the naming system. The operation is described in the following schemas *move0*, *NameError* and *DirError*.

In schema *move0* the input *oldname?* is defined as one of the file names and the input *newdir?* is defined as one of the directory names in the naming system using the existential quantifiers. The input *newname?* is defined as an unknown entry name under the *newdir?* using the universal quantifier. Having known the *oldname?*, *newdir?* and having created the *newname?*, the *oldname?* in *names* is substituted by *newdir?* and *newname?* using function overriding. The number of file entries under the *newdir?* *numentry*(j) is increased by one, *enames*(j) and *efids*(j) are also reset to *newname?* and *nfids*(i). The file entry name *enames*(l)(m) associated with *oldname?* is defined by the

third and the fourth existential quantifiers assuming that front(*oldname?*) is the directory name and last(*oldname?*) is the file entry name. The entries *enames*(l)(m) and *efids*(l)(m) are removed by using domain subtraction. The number of file entries *numentry*(l) is reduced by one.

*move0*

$\Xi SS$
$\Xi CS$
$\Delta NS$
*oldname?* : NAME
*newdir?* : DIR
*newname?* : SYL

---

$\exists$ i : 1 .. *numname* | *oldname?* = *names* (i)
$\exists$ j : 1 .. *numdir* | *newdir?* = *dnames*(j)
$\forall$ k : 1 .. *numentry*(j) | *newname?* $\neq$ *enames*(j)(k)
*names'* = *names* $\oplus$ { i $\mapsto$ (*newdir?* ^ *newname?*) }
*numentry'* (j) = *numentry*(j) + 1
*enames'*(j) = *enames*(j) $\oplus$ { *numentry'*(j) $\mapsto$ *newname?* }
*efids'*(j) = *efids*(j) $\oplus$ { *numentry'*(j) $\mapsto$ *nfids*(i) }
$\exists$ l : 1 .. *numdir* | *dnames*(l) = front( *oldname?*)
$\exists$ m : 1 .. *numentry*(l) | *enames*(l)(m) = last(*oldname?*)
*numentry'*(l) = *numentry'*(l) - 1
*enames'*(l) = { m } $\lhd$ *enames'*(l)
*efids'*(l) = { m } $\lhd$ *efids'*(l)

---

*NameError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*name?* : NAME
*report!* : REPORT

---

$\forall$ i : 1 .. *numname* | *oldname?* $\neq$ *names*(i)
*report!* = No Such Name

---

*DirError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*newdir?* : DIR
*report!* : REPORT

---

$\forall$ i : 1 .. *numdir* | *newdir?* $\neq$ *dnames*(i)
*report!* = No Such Dir

The schema *NameError* and *DirError* give error messages for unsuccessful operations.

The complete operation is defined as:

$$move \triangleq move0 \vee NameError \vee DirError$$

Twenty seven test cases are generated by ZTEST for inputs *oldname?*, *newdir?* and *newname?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { A I J N s H c O e r
v F y y l r o C d M
O W E}
Test Case 3 Invalid { S n Q U L j P e H K
K m A B g J F F t x
l A L P c}
Test Case 4 Invalid { 324 443 437 21 276 255 310 385 126 108
117 167 283 66 307 15 474 106 404 314
357 366 275 285}
Test Case 5 Invalid { 0.16 0.33 0.86 0.82 0.61 0.95 0.99 0.98 0.73 0.5
0.29 0.71 0.28 0.63 0.49 0.17 0.55 0.38 0.77 0.38
0.65 0.12 0.14 0.33}
Test Case 6 Invalid { @ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @
@ @ @ @}
Test Case 7 Invalid { { { { { { { { { {
{ { { { { { { { { {
{ { { {}
Test Case 8 Valid { A A A A A A A A A A
A A A A A A A A A A
A A A A }

Test Case 9 Valid { z z z z z z z z z z
z z z z z z z z z z
z z z z }

route 1 Fitness function:

There exists i, : 1 .. numn
oldname? = names (i )
There exists j, : 1 .. numd
newdir? = dnames (j )
For all k, : 1 .. nume (j )
newname? != enames (j )(k )
names' = names &fovr. i &map. newdir? &cat. newname? ,
nume' = nume &fovr. j &map. nume (j )+ 1,
enames' (j )= enames' (j )&fovr. nume' (j )&map. newname? ,
efids' (j )= efids' (j )&fovr. nume' (j )&map. nfids (i ),

There exists l, : 1 .. numd
dnames (l )= &front. (oldname? )
There exists m, : 1 .. nume (l )
enames (l )(m )= &last. (oldname? )
nume' (l )= nume' (l )- 1
enames' (l )= {m , } dsub enames' (l )
efids' (l )= {m , } dsub efids' (l )

Test Case 10 Valid { Y A N G Z T / O X D P}
names' = {/ , ,OXDP;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = { 3 2 2 1}
enames' = {YANG,TEMP,OXDP;ZT ,DOCU;ADD ,FIND;CV}
efids' = { 5 10 1, 1 12, 2 8, 20}
nume' = { 3 1 2 1}
enames' = {YANG,TEMP,OXDP;DOCU;ADD ,FIND;CV}
efids' = { 5 10 1, 12, 2 8, 20}

Test Case 11 Valid { Y A N G Z T Y A N G U W C J}
names' = {YANG, ,UWCJ;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = { 2 3 2 1}
enames' = {YANG,TEMP;ZT ,DOCU,UWCJ;ADD ,FIND;CV}
efids' = { 5 10, 1 12 1, 2 8, 20}
nume' = { 2 2 2 1}
enames' = {YANG,TEMP;DOCU,UWCJ;ADD ,FIND;CV}
efids' = { 5 10, 12 1, 2 8, 20}

Test Case 12 Valid { Y A N G Z T Y A N G Z T A M B E}
names' = {YANG,ZT ,AMBE;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = { 2 2 3 1}

```
enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,AMBE;CV}
efids' = {   5   10,   1   12,   2   8   1,   20}
nume' = {   2   1   3   1}
enames' = {YANG,TEMP;DOCU;ADD ,FIND,AMBE;CV}
efids' = {   5   10,   12,   2   8   1,   20}

Test Case 13 Valid  { Y A N G Z T        Y A N G D O C U R C O J}
names' = {YANG,DOCU,RCOJ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
          TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = {   2   2   2   2}
enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,RCOJ}
efids' = {   5   10,   1   12,   2   8,   20   1}
nume' = {   2   1   2   2}
enames' = {YANG,TEMP;DOCU;ADD ,FIND;CV ,RCOJ}
efids' = {   5   10,   12,   2   8,   20   1}

Test Case 14 Valid  { Y A N G Z T   A D D  /        L V R Q}
names' = {YANG,ZT ;/  ,  ,LVRQ;/  ;YANG;YANG,ZT ,FIND;
          TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = {   3   2   2   1}
enames' = {YANG,TEMP,LVRQ;ZT ,DOCU;ADD ,FIND;CV}
efids' = {   5   10   2,   1   12,   2   8,   20}
nume' = {   3   2   1   1}
enames' = {YANG,TEMP,LVRQ;ZT ,DOCU;FIND;CV}
efids' = {   5   10   2,   1   12,   8,   20}

Test Case 15 Valid  { Y A N G Z T   A D D  Y A N G     Y D A S}
names' = {YANG,ZT ;YANG,   ,YDAS;/  ;YANG;YANG,ZT ,FIND;
          TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = {   2   3   2   1}
enames' = {YANG,TEMP;ZT ,DOCU,YDAS;ADD ,FIND;CV}
efids' = {   5   10,   1   12   2,   2   8,   20}
nume' = {   2   3   1   1}
enames' = {YANG,TEMP;ZT ,DOCU,YDAS;FIND;CV}
efids' = {   5   10,   1   12   2,   8,   20}

Test Case 16 Valid  { Y A N G Z T   A D D  Y A N G Z T   S H R B}
names' = {YANG,ZT ;YANG,ZT ,SHRB;/  ;YANG;YANG,ZT ,FIND;
          TEMP;YANG,DOCU;YANG,DOCU,CV}
nume' = {   2   2   3   1}
enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,SHRB;CV}
efids' = {   5   10,   1   12,   2   8   2,   20}
nume' = {   2   2   2   1}
enames' = {YANG,TEMP;ZT ,DOCU;FIND,SHRB;CV}
efids' = {   5   10,   1   12,   8   2,   20}

Test Case 17 Valid  { Y A N G Z T   A D D  Y A N G D O C U L F Z C}
names' = {YANG,ZT ;YANG,DOCU,LFZC;/  ;YANG;YANG,ZT ,FIND;
          TEMP;YANG,DOCU;YANG,DOCU,CV}
```

```
   nume' = {   2   2   2   2}
   enames' = {YANG,TEMP;ZT  ,DOCU;ADD ,FIND;CV  ,LFZC}
   efids' = {   5   10,   1   12,   2   8,   20   2}
   nume' = {   2   2   1   2}
   enames' = {YANG,TEMP;ZT  ,DOCU;FIND;CV  ,LFZC}
   efids' = {   5   10,   1   12,   8,   20   2}

Test Case 18 Valid  { Y A N G          /          I P U Z}
   names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;/  ,  ,IPUZ;YANG,ZT  ,FIND;
             TEMP;YANG,DOCU;YANG,DOCU,CV}
   nume' = {   3   2   2   1}
   enames' = {YANG,TEMP,IPUZ;ZT  ,DOCU;ADD ,FIND;CV}
   efids' = {   5   10   5,   1   12,   2   8,   20}
Solution not found in state exists function.

Test Case 18 Valid  { Y A N G          Y A N G     X Y E D}
   names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG,   ,XYED;YANG,ZT  ,FIND;
             TEMP;YANG,DOCU;YANG,DOCU,CV}
   nume' = {   2   3   2   1}
   enames' = {YANG,TEMP;ZT  ,DOCU,XYED;ADD ,FIND;CV}
   efids' = {   5   10,   1   12   5,   2   8,   20}
Solution not found in state exists function.

Test Case 18 Valid  { Y A N G          Y A N G Z T   M J V R}
   names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG,ZT  ,MJVR;
             YANG,ZT  ,FIND;TEMP;YANG,DOCU;YANG,DOCU,CV}
   nume' = {   2   2   3   1}
   enames' = {YANG,TEMP;ZT  ,DOCU;ADD ,FIND,MJVR;CV}
   efids' = {   5   10,   1   12,   2   8   5,   20}
Solution not found in state exists function.

Test Case 18 Valid  { Y A N G          Y A N G D O C U O Y S N}
   names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG,DOCU,OYSN;
             YANG,ZT  ,FIND;TEMP;YANG,DOCU;YANG,DOCU,CV}
   nume' = {   2   2   2   2}
   enames' = {YANG,TEMP;ZT  ,DOCU;ADD ,FIND;CV  ,OYSN}
   efids' = {   5   10,   1   12,   2   8,   20   5}
Solution not found in state exists function.

Test Case 18 Valid  { Y A N G D O C U     /          M J U J}
   names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;TEMP;
             /  ,  ,MJUJ;YANG,DOCU,CV}
   nume' = {   3   2   2   1}
   enames' = {YANG,TEMP,MJUJ;ZT  ,DOCU;ADD ,FIND;CV}
   efids' = {   5   10   12,   1   12,   2   8,   20}
   nume' = {   3   1   2   1}
   enames' = {YANG,TEMP,MJUJ;ZT  ;ADD ,FIND;CV}
   efids' = {   5   10   12,   1,   2   8,   20}
```

```
Test Case 19 Valid  { Y A N G D O C U      Y A N G      R K G Y}
  names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,    ,RKGY;YANG,DOCU,CV}
  nume' = {    2   3   2   1}
  enames' = {YANG,TEMP;ZT  ,DOCU,RKGY;ADD ,FIND;CV}
  efids' = {    5   10,   1   12   12,   2   8,   20}
  nume' = {    2   2   2   1}
  enames' = {YANG,TEMP;ZT  ,RKGY;ADD ,FIND;CV}
  efids' = {    5   10,   1   12,   2   8,   20}

Test Case 20 Valid  { Y A N G D O C U      Y A N G Z T    J L T R}
  names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,ZT  ,JLTR;YANG,DOCU,CV}
  nume' = {    2   2   3   1}
  enames' = {YANG,TEMP;ZT  ,DOCU;ADD ,FIND,JLTR;CV}
  efids' = {    5   10,   1   12,   2   8   12,   20}
  nume' = {    2   1   3   1}
  enames' = {YANG,TEMP;ZT  ;ADD ,FIND,JLTR;CV}
  efids' = {    5   10,   1,   2   8   12,   20}

Test Case 21 Valid  { Y A N G D O C U      Y A N G D O C U X M W W}
  names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,DOCU,XMWW;YANG,DOCU,CV}
  nume' = {    2   2   2   2}
  enames' = {YANG,TEMP;ZT  ,DOCU;ADD ,FIND;CV  ,XMWW}
  efids' = {    5   10,   1   12,   2   8,   20   12}
  nume' = {    2   1   2   2}
  enames' = {YANG,TEMP;ZT  ;ADD ,FIND;CV  ,XMWW}
  efids' = {    5   10,   1,   2   8,   20   12}

Test Case 22 Valid  { Y A N G D O C U C V   /          D U P I}
  names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,DOCU;/  ,  ,DUPI}
  nume' = {    3   2   2   1}
  enames' = {YANG,TEMP,DUPI;ZT  ,DOCU;ADD ,FIND;CV}
  efids' = {    5   10   20,   1   12,   2   8,   20}
  nume' = {    3   2   2   0}
  enames' = {YANG,TEMP,DUPI;ZT  ,DOCU;ADD ,FIND;}
  efids' = {    5   10   20,   1   12,   2   8,}

Test Case 23 Valid  { Y A N G D O C U C V   Y A N G      M F T F}
  names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,DOCU;YANG,    ,MFTF}
  nume' = {    2   3   2   1}
  enames' = {YANG,TEMP;ZT  ,DOCU,MFTF;ADD ,FIND;CV}
  efids' = {    5   10,   1   12   20,   2   8,   20}
  nume' = {    2   3   2   0}
  enames' = {YANG,TEMP;ZT  ,DOCU,MFTF;ADD ,FIND;}
  efids' = {    5   10,   1   12   20,   2   8,}
```

```
Test Case 24 Valid  { Y A N G D O C U C V   Y A N G Z T   T X E M}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,ZT ,TXEM}
  nume' = {   2   2   3   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,TXEM;CV}
  efids' = {   5  10,   1  12,   2   8  20,  20}
  nume' = {   2   2   3   0}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,TXEM;}
  efids' = {   5  10,   1  12,   2   8  20,}

Test Case 25 Valid  { Y A N G D O C U C V   Y A N G D O C U S D M E}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/  ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;YANG,DOCU,SDME}
  nume' = {   2   2   2   2}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;CV ,SDME}
  efids' = {   5  10,   1  12,   2   8,  20  20}
  nume' = {   2   2   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND;SDME}
  efids' = {   5  10,   1  12,   2   8,  20}

route 2   Fitness function:
            For all i, : 1 .. numn
            oldname? != names (i )

Test Case 26 Valid  { G A N Z X F U R P A F A I E N Y A X N G D L E S}
  report! = No Such Name

route 3   Fitness function:
            For all i, : 1 .. numd
            newdir? != dnames (i )

Test Case 27 Valid  { V Z R T A B E B Q K O T S Y G H N U M F I G N Z}
  report! = No Such Dir
```

The first seven test cases are generated for invalid inputs *oldname?*, *newdir?* and *newname?*. Test cases 10 to 25 are generated for valid inputs *oldname?*, *newdir?* and *newname?* to test the input predicates in schema *move0*. The test data are generated in the same way as in schema *link0*. The values of state variables in the naming system are also reset for each test case after the operation. The entries related to *newname?* are added to the naming system by function overriding using direct assignment. The entries related to *oldname?* are looked up from the two state typed existential quantifiers in

schema *move0* and the entries are removed by domain subtraction. Note that in test case 18 (i=4) the *oldname?* is YANG which is a directory name without an attached file entry name. Similar with operation 'unlink', ZTEST failed the search in the fourth existential quantifier for a file entry name, so the four test cases for *oldname?* = YANG (i=4) and *newdir?* = *dnames*(j) (j=1, 2, 3, 4) are abandoned because last(*oldname?*) is empty and the search for l to make the state existential quantifier ∃ m : 1 .. *numentry*(l) | *enames*(l)(m) = last(*oldname?*) true are failed. New test cases are generated afterwards (i=7). The last two test cases are for valid input of schema *NameError* and *DirError*, which are the same as in operation 'link'.

### 6.2.9. Operation 'create'

The operation 'create' creates an empty file in the storage system, refers it with a new channel in the channel system and associates a name to it in the naming system. There are two cases: creating a file with an existing file name and creating a file with a new name. If an existing name is created, the file it refers to is emptied and its previous contents are lost. In addition to the first case if the name does not exist in the naming system, a file id is generated which is not currently in use. The operation is described in the following schemas *create1* for a non-existing name and *create2* for an existing name. The schema *create1* takes two inputs *dir?* and *name?* as the directory and file name of the new file to be created while *create2* only takes one input *name?* as the full path name. Both schemas give a output *cid!* as the new channel id created.

*create1*

$\Delta SS$
$\Delta CS$
$\Delta NS$
*dir?* : DIR
*name?* : SYL
*cid!* : CID
*fid* : FID
*usedfids* : FIDS

---

$usedfids = cfids \cup nfids$
$\forall i : N \bullet fid \neq usedfids(i)$
$numfid' = numfid + 1$
$fids' = fids \oplus \{ numfid' \mapsto fid \}$
$files' = files \oplus \{ numfid' \mapsto <> \}$
$\forall j : 1 .. numcid \mid cid! \neq cids(j)$
$numcid' = numcid + 1$
$cids' = cids \oplus \{ numcid' \mapsto cid! \}$
$posns' = posns \oplus \{ numcid' \mapsto 0 \}$
$cfids' = cfids \oplus \{ numcid' \mapsto fid \}$
$\exists k : 1 .. numdir \mid dir? = dnames(k)$
$\forall n : 1 .. numentry(k) \mid name? \neq enames(k)(n)$
$numname' = numname + 1$
$nfids' = nfids \oplus \{ numname' \mapsto fid \}$
$names' = names \oplus \{ numname' \mapsto (dir? \frown name?) \}$
$numentry'(k) = numentry(k) + 1$
$enames'(k) = enames(k) \oplus \{ numentry'(k) \mapsto name? \}$
$efids'(k) = efids(k) \oplus \{ numentry'(k) \mapsto fid \}$

---

In schema *create1*, an intermediate variable *usedfids* is defined as the union of *cfids* and *nfids* to represent all file ids which are currently in use. A new *fid* for the file to be created is defined in the first universal quantifier as a file id that is not in use. A new empty file is added to the file storage system and its file id is set to *fid* by function overriding. The number of files is increased by one. A new channel id *cid!* is defined as a channel id which is not currently in use by the second universal quantifier. A new channel is opened in the channel system which its channel id is set to *cid!*, its position

is set to 0 and its channel file id is set to *fid* by function overriding. The number of channels is increased by one. After the creation of the new file in the storage and the channel system, a new file name needs to be associated to the file. The directory of the new file *dir?* is defined as one of the directory names in the naming system by the existential quantifier. The file entry name *name?* is defined as an unknown entry name under the *dir?* by the third universal quantifier. The new file is added to the naming system by adding new values to the state variables. The number of names is increased by one.

*create2*

$\Delta SS$
$\Delta CS$
$\Xi NS$
*name?* : NAME
*cid!* : CID
*fid* : FID

---

$\exists$ i : 1 .. *numname* | *name?* = *names*(i)
$\exists$ j : 1 .. *numfid* | *nfids*(i) = *fids*(j)
*files'* = *files* $\oplus$ { j $\mapsto$ < > }
$\forall$ m : 1 .. *numcid* | *cid!* $\neq$ *cids*(m)
*numcid'* = *numcid* + 1
*cids'* = *cids* $\oplus$ { *numcid'* $\mapsto$ *cid!* }
*posns'* = *posns* $\oplus$ { *numcid'* $\mapsto$ 0 }
*cfids'* = *cfids* $\oplus$ { *numcid'* $\mapsto$ *nfids*(i) }

In schema *create2*, the name of the file to be created *name?* is defined as one of the names in the naming system by the first existential quantifier. Having known the index i of *name?* in *names*, the associated file id *fids*(j) is given by the second existential quantifier as *nfids*(i) = *fids*(j). The contents of the file *files*(j) is emptied by the function overriding. As in schema *create1*, a new channel is opened whose channel id *cid!* is a

new channel id, whose position is set to 0 and whose file id is *nfids*(i). The number of channel ids is increased by one. The complete operation is defined as:

$$create \triangleq create1 \vee create2$$

Eighteen test cases are generated by ZTEST for inputs *dir?* and *name?*.

Test Cases

```
Test Case  1  Invalid { }
Test Case  2  Invalid {   A   I   J   N   s   H   c   O   e   r
                          v   }
Test Case  3  Invalid {   r   o   C   d   M   O   W   E   S   n
                          Q   U   L }
Test Case  4  Invalid {  204  120  206   27  247  491   0  204   34   36
                         127  487  }
Test Case  5  Invalid {  0.21 0.24 0.75 0.88  0.9 0.73 0.46 0.53 0.03 0.13
                         0.45 0.48 }
Test Case  6  Invalid {   @   @   @   @   @   @   @   @   @   @
                          @   @  }
Test Case  7  Invalid {   {   {   {   {   {   {   {   {   {   {
                          {   { }
Test Case  8  Valid   {   A   A   A   A   A   A   A   A   A   A
                          A   A}
Test Case  9  Valid   {   z   z   z   z   z   z   z   z   z   z
                          z   z }
```

```
usedfids = cfids &uni. nfids
 For all i, : 1 .. usedfids
 fid != usedfids (i )
numf' = numf + 1
files' (numf' )= ,
fids' = fids &fovr. numf' &map. fid ,
For all j, : 1 .. numc
 cid! = cids (j )
numc' = numc + 1
cids' = cids &fovr. numc' &map. c ,
posns' = posns &fovr. numc' &map. 0,
cfids' = cfids &fovr. numc' &map. fid ,
```

```
usedfids = {   1   3   12   5   20   2   8   10}
fid = {   31}
numf' = {   13}
files' = {   65   68   73   80   65   66  107,,,   5   87   190,,,,,,,
             63   68   63   88   65   66   10   98   129   100,   0, }
fids' = {   1   2   3   4   5   6   7   8   9   10   12   20   31}
cid! =    8
```

```
numc' = {    6}
cids' = {    1   2   3   4   5   8}
posns' = {   2   0   8   0   0   0}
cfids' = {   1   3   12   5   20   31}
```

route 1 Fitness function:

```
There exists k, : 1 .. numd
dir? = dnames (k )
For all n, : 1 .. nume (k )
name? != enames (k )(n )
numn' = numn + 1
nfids' = nfids &fovr. numn' &map. fid ,
names' = names &fovr. numn' &map. dir? &cat. name? ,
nume' = nume &fovr. k &map. nume (k )+ 1,
enames' (k )= enames' (k )&fovr. nume' (k )&map. name? ,
efids' (k )= efids' (k )&fovr. nume' (k )&map. fid
```

```
Test Case  10  Valid   { /              Y P V O}
  numn' = {    9}
  nfids' = {   1   2   3   5   8   10   12   20   31}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;VONG,DOCU,CV;/ ,  ,  ,YPVO}
  nume' = {    3   2   2   1}
  enames' = {YANG,TEMP,YPVO;ZT ,DOCU;ADD ,FIND;CV}
  efids' = {   5   10   31,   1   12,   2   8,   20}

Test Case  11  Valid   { Y A N G          N N F X}
  numn' = {    9}
  nfids' = {   1   2   3   5   8   10   12   20   31}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;FXNG,DOCU,CV;YANG,  ,  ,NNFX}
  nume' = {    2   3   2   1}
  enames' = {YANG,TEMP;ZT ,DOCU,NNFX;ADD ,FIND;CV}
  efids' = {   5   10,   1   12   31,   2   8,   20}

Test Case  12  Valid   { Y A N G Z T        C Z G S}
  numn' = {    9}
  nfids' = {   1   2   3   5   8   10   12   20   31}
  names' = {YANG,ZT ;YANG,ZT ,ADD ;/ ;YANG;YANG,ZT ,FIND;
            TEMP;YANG,DOCU;GSNG,DOCU,CV;YANG,ZT ,  ,CZGS}
  nume' = {    2   2   3   1}
  enames' = {YANG,TEMP;ZT ,DOCU;ADD ,FIND,CZGS;CV}
  efids' = {   5   10,   1   12,   2   8   31,   20}

Test Case  13  Valid   { Y A N G D O C U      X R Z I}
  numn' = {    9}
  nfids' = {   1   2   3   5   8   10   12   20   31}
```

```
  names' = {YANG,ZT  ;YANG,ZT  ,ADD ;/  ;YANG;YANG,ZT  ,FIND;
            TEMP;YANG,DOCU;ZING,DOCU,CV;YANG,DOCU,   ,XRZI}
  nume' = {   2   2   2   2}
  enames' = {YANG,TEMP;ZT  ,DOCU;ADD ,FIND;CV  ,XRZI}
  efids' = {   5   10,   1   12,   2   8,   20   31}

route 2   Fitness function:
            There exists i,  : 1 .. numn
            name? = names (i )
           There exists j,  : 1 .. numf
            nfids (i )= fids (j )
           files' (j )= ,
           For all m,  : 1 .. numc
            cid! = cids (m )
           numc' = numc + 1
           cids' = cids &fovr. numc' &map. c ,
           posns' = posns &fovr. numc' &map. 0,
           cfids' = cfids &fovr. numc' &map. nfids (i),

Test Case  14  Valid   { Y A N G Z T              }
  files' = {,,,   5   87   190,,,,,,,   63   68   63   88   65   66   10   98   129   100,
             0}
  cid! =    9
  numc' = {   6}
  cids' = {   1   2   3   4   5   9}
  posns' = {   2   0   8   0   0   0}
  cfids' = {   1   3   12   5   20   1}

Test Case  15  Valid   { Y A N G Z T   A D D       }
  files' = {   65   68   73   80   65   66   107,,,   5   87   190,,,,,,,
             63   68   63   88   65   66   10   98   129   100,   0}
  cid! =    8
  numc' = {   6}
  cids' = {   1   2   3   4   5   8}
  posns' = {   2   0   8   0   0   0}
  cfids' = {   1   3   12   5   20   2}

Test Case  16  Valid   { Y A N G              }
  files' = {   65   68   73   80   65   66   107,,,   5   87   190,,,,,,,
             63   68   63   88   65   66   10   98   129   100,   0}
  cid! =    8
  numc' = {   6}
  cids' = {   1   2   3   4   5   8}
  posns' = {   2   0   8   0   0   0}
  cfids' = {   1   3   12   5   20   5}

Test Case  17  Valid   { Y A N G D O C U          }
  files' = {   65   68   73   80   65   66   107,,,   5   87   190,,,,,,,,   0}
  cid! =    10
```

```
  numc' = {   6}
  cids' = {   1   2   3   4   5  10}
  posns' = {  2   0   8   0   0   0}
  cfids' = {  1   3  12   5  20  12}

Test Case 18 Valid  { Y A N G D O C U C V       }
  files' = {  65  68  73  80  65  66 107,,,   5  87 190,,,,,,,
              63  68  63  88  65  66  10  98 129 100,}
  cid! =   8
  numc' = {   6}
  cids' = {   1   2   3   4   5   8}
  posns' = {  2   0   8   0   0   0}
  cfids' = {  1   3  12   5  20  20}
```

The first seven test cases are generated for invalid inputs *dir?* and *name?* which are alphabetic strings. Test cases 10 to 13 are generated for valid inputs *dir?* and *name?* to test the input predicates in schema *create1*. These test cases test the four values of *dir?* from it's search domain in the input typed existential quantifier in schema *create1*. The test data are given by direct assignment. After the directory name *dir?*, a new entry name *name?* is generated in each test cases using GA from the fitness function of the input universal quantifier in schema *create1* as an unknown name under the *dir?*. In each test case, the values of state variables are reset after the operation by function overriding. Test case 14 to 18 are generated for valid inputs *name?* to test the input predicate in schema *create2*. The test data are given by direct assignment from the fitness function of the input existential quantifier in schema *create2*. These test cases test four boundary values and an intermediate value of *name?* from it's search domain. The second existential quantifier is of state type, no test cases are generated from it. ZTEST checks values of j until the j is found to make *nfids*(i) = *fids*(j) true, then *files*(j) is set to empty by function overriding. A new channel id *cid!* is generated at random in each test case from the fitness function of the output typed universal quantifier in schema *create2*, *cid!* is a unknown channel id in the channel system. Other state

variables in the channel system are also updated after the operation by function overriding.

### 6.2.10. Operation 'destroy'

The operation 'destroy' removes a file associated with a given file id from the storage system. The operation takes an input *fid?* as the file id of the file to be removed which must not be in use currently. The operation looks it up from the file ids in the storage system and removes associated values of the state variables in the storage system. The operation can be described in the following schemas *destroy1* and *FidError*.

*destroy1*

$\Delta SS$
$\Xi CS$
$\Xi NS$
*fid?* : FID
*usedfids* : FIDS

---

$usedfids = cfids \cup nfids$
$\exists i : 1 .. numfid \mid fid? = fids(i)$
$\forall j : 1 .. N \mid fid? \neq usedfids(j)$
$fids' = \{ i \} \lhd fids$
$files' = \{ i \} \lhd files$
$numfid' = numfid - 1$

In schema *destroy1*, the intermediate variable *usedfids* which represents the file ids currently in use is defined as the union of *cfids* and *nfids*. The input *fid?* is defined as a file id from the file storage system by the first existential quantifier. Meanwhile *fid?*

should also make the second input predicate, the universal quantifier $\forall j : 1 .. N \mid fid? \neq usedfids(j)$ true.

*FidError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
$fid?$ : FID
$report!$ : REPORT

---

$\forall i : 1 .. numfid \mid fid? \neq fids(i)$
$report!$ = No Such Fid

After knowing the index i, associated values of *fids* and *files* are removed by domain subtraction. The number of file ids is decreased by one. The complete operation is defined as:

$$destroy \triangleq destroy1 \vee FidError$$

Test cases generated by ZTEST are shown below.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 99 100}
Test Case 3 Invalid { I}
Test Case 4 Invalid { 0.35}
Test Case 5 Invalid { 101}
Test Case 6 Invalid { 0}
Test Case 7 Valid { 100}
Test Case 8 Valid { 1}

usedfids = cfids &uni. nfids
usedfids = { 1 3 12 5 20 2 8 10}

route 1 Fitness function:
There exists i, : 1 .. numf
fid? = fids (i )
For all j, : 1 .. usedfids

```
fid? != usedfids (j )
fids' = {i , } dsub fids
files' = {i , } dsub files
numf' = numf - 1
```

```
Test Case  9 Valid   {     1}
Invalid solution, test case abandoned.

Test Case  9 Valid   {     2}
Invalid solution, test case abandoned.

Test Case  9 Valid   {     4}
  fids' = {    1    2    3    5    6    7    8    9   10   12   20}
  files' = {   65   68   73   80   65   66  107,,,,,,,,
                 63   68   63   88   65   66   10   98  129  100,    0}
  numf' = {   11}

Test Case  10 Valid   {    12}
Invalid solution, test case abandoned.

Test Case  10 Valid   {    20}
Invalid solution, test case abandoned.
```

The first six test cases are generated for invalid input *fid?* which is a natural number in the range of [1, 100]. Only one valid test case is generated from the input predicates (*fid?* = 4) while other four cases are abandoned. The reason is that the input *fid?* is defined in the input typed existential quantifier within the domain of *fids* ($\exists$ i : 1 .. *numfid* | *fid?* = *fids*(i) ). ZTEST generates test cases for the four boundary values and an intermediate value of *fid?* from it's search domain. These five values of *fid?* are not guaranteed to satisfy the rest of predicates such as the input typed universal quantifier $\forall$ j : 1 .. N | *fid?* $\neq$ *usedfids*(j). The test case will be abandoned if one of the predicates is not satisfied.

There are two ways to solve the problem: extra test cases can be added manually afterwards, or makes the defined search domain of the input variables more clearly. The above schema *destroy1* is rewritten as:

*destroy0*

$\Delta SS$
$\Xi CS$
$\Xi NS$
*fid?* : FID
*usedfids, removable* : FIDS

---

*usedfids* = *cfids* $\cup$ *nfids*
*removable* = *fids* \ *usedfids*
$\exists$ i : 1 .. N | *fid?* = *removable*(i)
$\exists$ j : 1 .. *numfid* | *fids*(j) = *fid?*
*fids'* = { j } $\lhd$ *fids*
*files'* = { j } $\lhd$ *files*
*numfid'* = *numfid* - 1

---

where the intermediate variable *removable* is defined as the difference between *fids* and *usedfids*. The input variable *fid?* is in the search domain of *removable* where all members will satisfy the predicate *fid?* $\neq$ *usedfids*(j). Four valid test cases are generated by ZTEST from schema *destroy0*.

Test Cases

usedfids = cfids &uni. nfids
removable = fids &diff. usedfids

usedfids = { 1 3 12 5 20 2 8 10}
removable = { 4 6 7 9}

route 1 Fitness function:
There exists i, : 1 .. removable
fid? = removable (i )
There exists j, : 1 .. numf
fids (j )= fid?
fids' = {j , } dsub fids
files' = {j , } dsub files
numf' = numf - 1

Test Case 9 Valid { 4}
fids' = { 1 2 3 5 6 7 8 9 10 12 20}
files' = { 65 68 73 80 65 66 107,,,,,,,,
63 68 63 88 65 66 10 98 129 100, 0}
numf' = { 11}

```
Test Case  10 Valid   {    6}
  fids' = {   1   2   3   4   5   7   8   9  10  12  20}
  files' = {  65  68  73  80  65  66 107,,,  5  87 190,,,,,,
              63  68  63  88  65  66  10  98 129 100,   0}
  numf' = {   11}

Test Case  11 Valid   {    7}
  fids' = {   1   2   3   4   5   6   8   9  10  12  20}
  files' = {  65  68  73  80  65  66 107,,,  5  87 190,,,,,,
              63  68  63  88  65  66  10  98 129 100,   0}
  numf' = {   11}

Test Case  12 Valid   {    9}
  fids' = {   1   2   3   4   5   6   7   8  10  12  20}
  files' = {  65  68  73  80  65  66 107,,,  5  87 190,,,,,,
              63  68  63  88  65  66  10  98 129 100,   0}
  numf' = {   11}

 route 2   Fitness function:
                For all i, : 1 .. numf
                fid? != fids (i )

Test Case  13 Valid   {    41}
  report! = No Such Fid
```

Test cases 9 to 12 are generated for valid input *fid?* to test the input typed predicate in schema *destroy0*. The test data are given by direct assignment from the fitness function of the input existential quantifier. These test cases test the four values of *fid?* from it's search domain (*removable*). The second existential quantifier is of state type, no test data are generated from it. ZTEST checks value of j to find the j to make *fids(j) = fid?* true. The values of state variables in the storage system are updated after the operation in each test case by domain subtraction, and the number of files is reduced by one. The last test case is for valid input of schema *FidError*, *fid?* is generated using GA as a unknown file id in the file storage system.

6.2.11. Operation 'read'

The operation 'read' returns the data of a given length from a file in a given channel. The operation takes two inputs *cid?* and *length?* which are the channel id of the file to read and the length of the data to be read. The operation looks up the channel id of the file to read from the channel system and finds out the index of the file from the storage system. The beginning position of the data to be read is the current position of the channel plus one, and the last position of the data is the minimum value of the size of the file and the current position plus *length?*. The output *data!* holds the contents of the file from the beginning to the last position. The current position is updated after the operation. The operation is described in the following schemas *read0* and *CidErroe*.

*read0*

$\Xi SS$
$\Delta CS$
$\Xi NS$
*CID?* : CID
*length?* : N
*data!* : seq BYTE
*n1, n2* : N

---

$\exists$ i : 1 .. *numcid* | *cid?* = *cids*(i)
$\exists$ j : 1 .. *numfid* | *cfids*(i) = *fids*(j)
*n1* = *posns*(i) + 1
*n2* = min ( #files(j), *posns*(i) + *length* )
*data!* = { k : *n1* .. *n2* • *files*(j)(k) }
*posns'* = *posns* $\oplus$ { i $\mapsto$ *n2* }

In schema *resd0* the input *cid?* is defined as one of the channel ids in the channel system by the first existential quantifier. Having known the index i of *cid?* from the channel system, the index j of the associated file in the storage system can be defined by the

*CidError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*CID?* : CID
*report!* : REPORT

---

$\forall$ i : 1 .. *numcid* | *CID?* $\neq$ *cids*(i)
*report!* = No Such CID

---

second existential quantifier to make *cfids*(i) = *fids*(j) true. Two intermediate variables n1, n2 are introduced as the beginning and the last position of the output data in the original file. The output *data!* is given by the set comprehension as the contents of the file from n1 to n2. The current position of the channel is reset to n2 after the operation. The complete operation is defined as:

$$read \triangleq read0 \vee CidError$$

Fifteen test cases are generated by ZTEST for inputs *cid?* and *length?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 10}
Test Case 3 Invalid { 10 67 97}
Test Case 4 Invalid { J N}
Test Case 5 Invalid { 0.7 0.94}
Test Case 6 Invalid { 11 101}
Test Case 7 Invalid { 0 0}
Test Case 8 Valid { 10 100}
Test Case 9 Valid { 1 1}

route 1 Fitness function:
There exists i, : 1 .. numc
CID? = cids (i )
There exists j, : 1 .. numf
cfids (i )= fids (j )

```
                n1 = posns (i )+ 1
                n2 = &min. # files (j ), posns (i )+ length
                data = { k: n1 .. n2 &bul. files (j )(k ) }
                posns' = posns &fovr. i &map. n2

Test Case  10 Valid  {    1   24}
  n1 = {    3}
  n2 = {    7}
  data = {   73   80   65   66  107}
  posns' = {    7    0    8    0    0}

Test Case  11 Valid  {    2   84}
  n1 = {    1}
  n2 = {    0}
  data = {}
  posns' = {    2    0    8    0    0}

Test Case  12 Valid  {    3   40}
  n1 = {    9}
  n2 = {   10}
  data = {  129  100}
  posns' = {    2    0   10    0    0}

Test Case  13 Valid  {    4   32}
  n1 = {    1}
  n2 = {    0}
  data = {}
  posns' = {    2    0    8    0    0}

Test Case  14 Valid  {    5   46}
  n1 = {    1}
  n2 = {    1}
  data = {    0}
  posns' = {    2    0    8    0    1}

route 2   Fitness function:
                For all i, : 1 .. numc
                cid? != cids (i )

Test Case  15 Valid  {    6   60}
  report! = No Such Cid
```

The first seven test cases are generated for invalid inputs *cid?* and *length?*. The input *length?* is a natural number in the range of [1, 100], there are no further constraints on it. Test cases 10 to 14 are generated for valid input *cid?* to test the input typed predicate in

schema *read0*. The test data are given by direct assignment from the fitness function of the input existential quantifier in schema *read0*. These test cases test the four boundary values and an intermediate value of *cid?* from it's search domain. In each test case, the index j of the file associated with *cid?* in the file storage system *files*(j) is looked up by the state typed existential quantifier in schema *read0*. ZTEST checks values of j until the j is found to make *cfids*(i) = *fids*(j) true. The output *data!* is set to the associated values by set comprehension. The current position of the channel is updated after the operation. The last test case is for valid input of schema *CidError*, cid? is a unknown channel id in the channel system generated using GA from the universal quantifier in schema *CidError*.

### 6.2.12. Operation 'write'

The operation 'write' writes a given piece of data into a file in a given channel. The operation takes two inputs *cid?* and *data?* which are the channel id of the file to write and the data to be written. The operation looks up the channel id of the file to write from the channel system and finds out the index of the file from the storage system. The contents of the file is set to the original contents from the beginning to the current position. The input data *data?* is written to the file after the current position. The current position is updated after the operation. The operation is described in the following schemas *write0* and *CidError*.

In schema *write0* the input *cid?* is defined as one of the channel ids in the channel system by the first existential quantifier. As in schema *read0*, the index j of the associated file in the storage system can be defined by the second existential quantifier

to make *cfids*(i) = *fids*(j) true. A intermediate variable n3 is used to hold the value of the current position in the channel. The intermediate variable *file* is used to represent the file

*write0*

$\Delta SS$
$\Delta CS$
$\Xi NS$
*cid?* : CID
*data?* : DATA
*file* : FILE
*n1, n2, n3, length* : N

---

$\exists$ i : 1 .. *numcid* | *cid?* = *cids*(i)
$\exists$ j : 1 .. *numfid* | *cfids*(i) = *fids*(j)
*n3* = *posns*(i)
*file* = { k : 1 .. *n3* • *files*(j)(k) }
*length* = #*data?*
*n1* = *posns*(i) + 1
*n2* = *posns*(i) + *length*
{ l : *n1* .. *n2* • *file*(l) } = { m : 1 .. *length* • *data?*(m) }
*files'* = *files* $\oplus$ { j $\mapsto$ *file* }
*posns'* = *posns* $\oplus$ { i $\mapsto$ *n2* }

---

*CidError*

$\Xi SS$
$\Xi CS$
$\Xi NS$
*cid?* : CID
*report!* : REPORT

---

$\forall$ i : 1 .. *numcid* | *cid?* $\neq$ *cids*(i)
*report!* = No Such Cid

---

to write. The contents of the file is set to the original contents from 1 to n3 by the first set comprehension. The second set comprehension writes the input *data?* into the file

following the current position n3. The *files* in the storage system is updated so does the current position of the channel. The complete operation is defined as:

$$write \triangleq write0 \vee CidError$$

Fifteen test cases are generated by ZTEST for inputs *cid?* and *data?*.

Test Cases

Test Case 1 Invalid { }
Test Case 2 Invalid { 10 254 170 247 164 200 118 205 76 12 }
Test Case 3 Invalid { 8 142 228 77 111 245 213 47 80 60 44 10}
Test Case 4 Invalid { F y y l r o C d M O W}
Test Case 5 Invalid { 0.17 0.42 0.69 0.16 0.53 0.64 0.62 0.0 0.78 0.26 0.46}
Test Case 6 Invalid { 11 256 256 256 256 256 256 256 256 256 256}
Test Case 7 Invalid { 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1}
Test Case 8 Valid { 10 255 255 255 255 255 255 255 255 255 255 }
Test Case 9 Valid { 1 0 0 0 0 0 0 0 0 0 0 }

route 1 Fitness function:

There exists i, : 1 .. numc
cid? = cids (i )
There exists j, : 1 .. numf
cfids (i )= fids (j )
n3 = posns (i )
file = { k: 1 .. n3 &bul. files (j )(k ) }
length = # data?
n1 = posns (i )+ 1
n2 = posns (i )+ length
{ l: n1 .. n2 &bul. file (l ) } = { m: 1 .. length &bul. data? (m ) }
files' = files &fovr. j &map. file
posns' = posns &fovr. i &map. n2

Test Case 10 Valid { 1 202 113 139 242 221 231 99 40 75 235}

n3 = { 2}
file = { 65 68}
length = { 10}
n1 = { 3}

```
n2 = {   12}
file = {   65   68  202  113  139  242  221  231   99   40   75  235}
files' = {   65   68  202  113  139  242  221  231   99   40   75  235,,,
          5   87  190,,,,,,,   63   68   63   88   65   66   10   98  129  100,    0}
posns' = {   12    0    8    0    0}

Test Case  11  Valid  {    2  214  202  144  188   19   87  141   65  224
                        230}
n3 = {    0}
file = {}
length = {   10}
n1 = {    1}
n2 = {   10}
file = {  214  202  144  188   19   87  141   65  224  230}
files' = {   65   68   73   80   65   66  107,,  214  202  144  188   19   87  141
         65  224  230,    5   87  190,,,,,,,   63   68   63   88   65   66   10   98
        129  100,    0}
posns' = {    2   10    8    0    0}

Test Case  12  Valid  {    3  193  215   10   27  195  148  158  157  202
                        60}
n3 = {    8}
file = {   63   68   63   88   65   66   10   98}
length = {   10}
n1 = {    9}
n2 = {   18}
file = {   63   68   63   88   65   66   10   98  193  215   10   27  195  148
          158  157  202   60}
files' = {   65   68   73   80   65   66  107,,,    5   87  190,,,,,,,   63   68   63
  88   65   66   10   98  193  215   10   27  195  148  158  157  202   60,    0}
posns' = {    2    0   18    0    0}

Test Case  13  Valid  {    4  130  156    9  102   79   46   75  122  228
                        120}
n3 = {    0}
file = {}
length = {   10}
n1 = {    1}
n2 = {   10}
file = {  130  156    9  102   79   46   75  122  228  120}
files' = {   65   68   73   80   65   66  107,,,    5   87  190,  130  156    9  102
          79   46   75  122  228  120,,,,,,   63   68   63   88   65   66   10   98
          129  100,    0}
posns' = {    2    0    8   10    0}

Test Case  14  Valid  {    5  119   60  250  160   34  117   59  102  244
                        186}
n3 = {    0}
file = {}
```

```
length = {   10}
n1 = {    1}
n2 = {   10}
file = {  119   60  250  160   34  117   59  102  244  186}
files' = {   65   68   73   80   65   66  107,,,   5   87  190,,,,,,,   63   68   63
      88   65   66   10   98  129  100,  119   60  250  160   34  117   59  102
      244  186}
posns' = {   2   0   8   0  10}
```

route 2 Fitness function:

For all i, : 1 .. numc
cid? != cids (i )

Test Case 15 Valid { 10 41 159 168 173 167 143 1 6 30
70}
report! = No Such Cid

The first seven test cases are generated for invalid inputs *cid?* and *data?*. The input *data?* is defined as the sequence of BYTE of the length 10, and there are no further constraints on it. Test cases 10 to 14 are generated for valid input *cid?* to test the input typed predicate in schema *write0*. The test data are given by direct assignment from the fitness function of the input existential quantifier in schema *write0*. These test cases test the four boundary values and an intermediate value of *cid?* from it's search domain. In each test case an intermediate variable *file* is given to hold the contents of the original file first, and then append the input *data?* after it by set comprehension. The *file* is put back to the file storage system, and the current position of the *file* is updated using function overriding. The last test case is for valid input of schema *CidError*, which is the same as in operation 'read'.

# CHAPTER 7. DISCUSSION

## 7.1. Completeness of Test Suites

For the completeness of the test suites, every aspect of the behaviour of the system under test should be covered by at least one test case. But it is impractical to check all the behaviour by testing all possible values and combinations of inputs. Hence, a test case suite is limited to test a small subset of all possible inputs. A quality test case suite will cover the right subset of inputs with the highest probability of finding the most errors. In practice, test cases should cover extreme values and an intermediate value of input variables [14] [46]. The guidance for testing safety critical software in Defence Standard 00-55[46] suggested that all numerical inputs have been set to their minimum, maximum and an intermediate value, all loops executed 0, 1, an intermediate number and maximum times. The test cases generated by ZTEST test the boundary values (minimum-1, minimum, maximum, maximum+1) and a typical intermediate value from every input variable. For iteration structures, the test cases cover boundary values (minimum, minimum+1, maximum-1, maximum) and an intermediate value of the iteration variable. Compared with the above guideline, this approach is adequate enough to test software system for general cases which are not safety critical.

By using ZTEST, the test case suite generated automatically covers the invalid inputs of invalid data types and invalid number of inputs as well as the boundary values for both valid and invalid inputs. Every objective function from all branches of the Z specification is covered by at least one test case, and there may be three test cases for each objective function in the case of high quality tests. For the objective functions that

are inside an iteration structure, five test cases are generated which cover the four boundary values and an intermediate value of the iteration variable. In some special software systems, such as safety critical systems, certain values from the input domain may be of special interest. Each one of this kind of critical input values should be tested by a specific test case. The test case will be generated automatically by ZTEST if the critical input value is on the boundary of the search domain ( minimum-1, minimum, maximum, maximum+1); or if the critical input value is related to a boundary value of the iteration variable in an iteration structure.

If the critical input value is in neither of the above two categories, the special test cases need to be added manually to the test suite generated by ZTEST. In order to let ZTEST generate the specific test case automatically, in some cases the input search domain can be divided into several sub-domains when writing the input format of the Z specification. For example, the iteration structure in Z is:

$$\exists\, i : 1 \,..\, number \mid input? = state(i)$$
$$\ldots\ldots$$

The input format of Z is:

$$\&exi.$$
$$i : 1 \,..\, number$$
$$\&cbar.\ input? = state(i)$$
$$\ldots\ldots$$

Test cases generated by ZTEST automatically cover the values of i = 1, 2, 4, 199 and 200 for *number* equals to 200. If the value i = 10 need to be tested specifically, then the test case can be added manually or the input format of Z specification can be written into:

$$\&exi.$$
$$i : 1 \,..\, 10$$
$$\&cbar.\ input? = state(i)$$

```
.....
&exi.
        i : 11 .. number
        &cbar. input? = state(i)
.....
```

Then the test cases generated by ZTEST will cover the test case i = 10. This kind of subdivision is only for the purpose of automatically generating test data, it should not affect the original design from the Z specification.

## 7.2. High Quality Test Data

High quality tests are test cases that have higher probability to reveal errors in software systems. It has been proved by experience that test cases which explore boundary conditions have a higher payoff than others[14]. Input and output values on the domain boundary belong to boundary conditions, as well as boundary iteration values in iteration structures.

ZTEST generates test cases to check the valid and invalid boundary values for every input variable. For the objective functions of integer type, high quality test cases can be derived in ZTEST to check the valid and invalid output boundary conditions. A test data set consisting of three test cases is generated in high quality tests for each objective function by using the successor and predecessor.

For the objective function x? + y? <= 88, the three test cases are:

```
Test Case 1   Valid   { 51  37 }     ---- x? + y? = 88
Test Case 2   Valid   { 43  44 }     ---- x? + y? = 87
Test Case 3   Invalid {  8  81 }     ---- x? + y? = 89
```

These three test cases certainly have more possibility to detect potential errors than a test case that is not with the boundary condition.

On the other hand, the search domain needs to be clearly defined in Z specification, that is the Z specification must give the precise description of the functionality in the software system. Otherwise the test cases produced by ZTEST will not have high quality for testing the software system.

An example is the Z schema for operation 'destroy' in Chapter 6.2.10. The operation has an input *fid?* which belongs to *fids*; the operation removes the *fid?* from *fids* so it must not be inside *usedfid* that is a sub-domain of *fids* as shown in Fig 16.

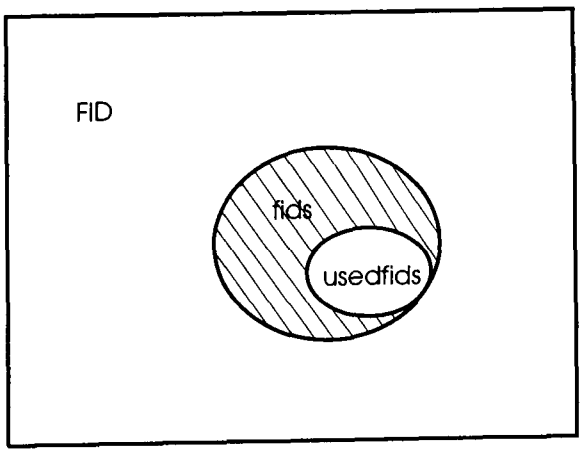



Fig. 16. Search domain of fid?.

In schema 'destroy1', the input objective functions are given as:

$$\exists i : 1 .. numfid \mid fid? = fids(i)$$

$$\forall j : 1 .. N \mid fid? \neq usedfids(j)$$

The input variable *fid?* is first defined by the existential quantifier as one of the members in *fids*, then a post condition (the universal quantifier) is followed to give the criterion of *fid?* is not inside *usedfids*. ZTEST generates test cases from the first objective function to check the four boundary values and an intermediate value in the search domain of *fids*. Because the input search domain is not defined very clearly, these test data produced in the domain of *fids* are not guaranteed being outside the sub-domain of *usedfids*. In the example of 'destroy1' in Chapter 6.2.10, four out of five test cases produced by ZTEST are abandoned due to the failure of the second objective function that is the universal quantifier.

In schema 'destroy0', a sub-domain is defined first as *removable* which is the difference between *fids* and *usedfids* as shown by the shaded area in Fig 16. The schema then is written as:

$$
\begin{aligned}
&\mathit{removable} = \mathit{fids} \setminus \mathit{usedfids} \\
&\exists\, i : 1 \,..\, N \mid \mathit{fid?} = \mathit{removable}(i) \\
&\quad \exists\, j : 1 \,..\, \mathit{numfid} \mid \mathit{fids}(j) = \mathit{fid?}
\end{aligned}
$$

In the first existential quantifier, the input variable *fid?* is clearly defined as one of the members in *removable*. The post condition in the following existential quantifier is only used to pick up the related value of *fid?* from *fids* which will never fail because *removable* is a sub-domain of *fids*. Four valid test cases are generated by ZTEST in schema 'destroy0' and no test cases are abandoned.

## 7.3. Algorithms of Test Data Generation

Evolutionary methods such as Genetic Algorithms and Simulated Annealing are used to generate test data sets by ZTEST. As described in Chapter 3, each algorithm of test data

generation has its advantages and disadvantages depending on the nature of objective functions.

Direct assignment can be used to produce test data from the objective functions that there is only one variable on the left side, and there are only constants or variables with known values on the right side of the objective function. ZTEST does the calculation on the right side of the objective function, including numeric expressions and set operations, and assigns the value to the variable on the left side of the objective function. Direct assignment is the quickest and safest way to get the solution from the objective functions, and it is always the first method to be used in ZTEST to generate test data if it is possible.

Test data have to be generated using a test data generator if direct assignment is not applicable for the objective function. The easiest test data generator is random generation, or random testing. Random generation randomly generates test data from the search domain until the value is found that will satisfy the objective function. Different from other random testing, only the values that satisfy the objective functions will be kept as test data in ZTEST. The high speed of data generation is the advantage of random testing. The disadvantage is it may need a long time or even fail to find the solution if the solution is be given by a complex objective function in a small sub-domain of the search domain. This is because the probability is very low for a random number to be generated inside a small sub-domain of the search domain. Random testing is suitable for simple objective functions with their solution being inside a large sub-domain of the search domain.

For the evolutionary algorithms such as GA and SA, the initial guesses of solution which are generated randomly are evaluated by the algorithms against the objective functions. The better guesses are selected to be used to form the next pool of guesses. Using this kind of deterministic algorithms will increase the speed of convergence to the optimum solution even for complicated objective functions. The disadvantage of the algorithms is the longer time in data generation because of the extra calculations in the algorithms. Evolutionary algorithms are suitable for more complicated objective functions that direct assignment and random testing can not handle efficiently.

For input typed objective functions such as

$$\exists\, i : 1 \,..\, num1 \mid name? = names(i) \bullet$$
$$\forall\, j : 1 \,..\, num2 \mid name? \neq entrynames(i)(j)$$
$$\ldots\ldots$$

the search domain can be complicated as shown in the shaded area of Fig. 17.

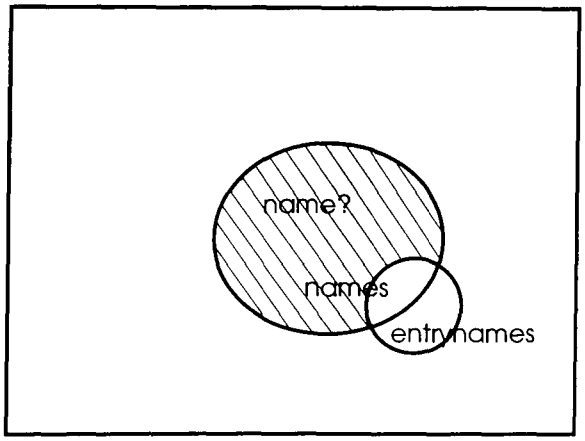



Fig. 17. Search domain of name?.

Genetic Algorithms are chosen as the default method to generate test data sets from input objective functions because they are robust and stable in converging to the solution, but not necessarily the global optimum. Other algorithms available in ZTEST to generate test data sets are Simulated Annealing and Parallel Annealing.

Functional testing tests inputs against the related outputs. Test cases are derived only from input typed objective functions and test data are generated only for input variables. Other typed objective functions are only been used as the pre and post conditions for the input typed objective functions during test data generation, as discussed in Chapter 5.1.

For state typed objective functions, the purpose is not to generate test data from them but to find any one value of the state variable to make the predicates true. For state typed universal quantifiers the objective functions can be complicated, but the search domain usually is quite large (outside a certain domain). For the state typed universal quantifiers such as

$$\forall j : 1 .. number \bullet fid \neq usedfids(j)$$

.....

the search domain is shown in the shaded area of Fig. 18.

Because the solution is easy to find, random generation is adequate to generate the test data from state typed universal quantifier objective functions and is used in ZTEST.

In the case of state typed existential quantifiers such as

$$\exists i : 1 .. number \bullet fid = fids(i)$$

.....

the objective function is simple and the search domain is limited within the domain of the state variable as shown in the shaded area of Fig. 19. Exhaustive testing is used in ZTEST to evaluate values from the search domain (members from fids in the above example) until the solution for state typed existential quantifier is found to make the predicates true.

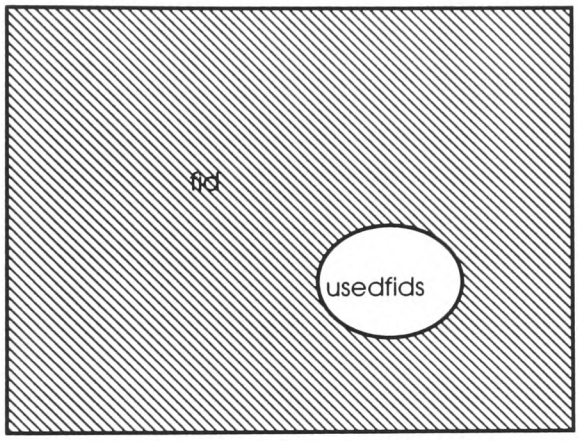



Fig. 18. Search domain of fid in the universal quantifier.

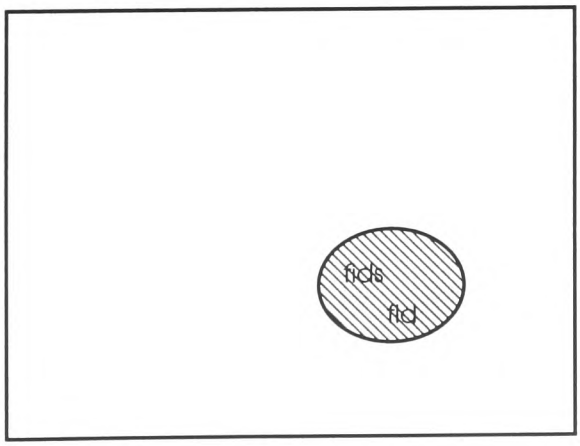



Fig. 19. Search domain of fid in the existential quantifier.

For both of the state typed existential quantifiers and universal quantifiers, evolutionary algorithms can be easily adapted to meet the requirements of more complicated objective functions.

If the process of GA search converges slowly during test data generation, a portion of random selected guesses with lower fitnesses are added deliberately to maintain the diversity of the population. In case the test data generator (GA or SA) failed to find the solution for the objective function in a run. The generator will be started again automatically in ZTEST for up to a maximum number of runs (MAXFAIL) to continue the search. In most of the cases the solution can be generated. In some extreme cases when the solution is not found after the maximum number of runs, ZTEST will switch the objective function automatically to generate a complementary invalid test case.

## 7.4. Comparison with Related Work

Software testing is a very important part in the life cycle of software development. The manual derivation of test cases and construction of test data sets need a large amount of work in a software project. It is a big advantage to automate the testing process. Research work has been done in the area of testing from formal specifications, and the area of automatic test data generation.

The main criteria for judging the effectiveness of an automatic software testing method are:

1. Effectiveness

Test effectiveness is a metric that relates to the degree of coverage of same attribute of the software. For some of the software testing methods, there are classes of test data which it is impossible to generate. A good testing method can generate test cases that have a high degree of functionality coverage for the software system under test. The test data generation method can automatically generate test data for every test cases derived by the testing method.

2. Efficiency

The testing method and the test data generation method work fast and have high level of automation. The methods are inexpensive in terms of computer resources for software and hardware. The test data generation algorithm can converge to the solution quickly. The methods are also inexpensive in terms of human resources. By using the testing method test engineers do not need to check large volumes of test data.

3. Applicability

The testing method is applicable to a wide range of software systems with different properties and in different scales from simple software systems to large complex software systems. The test data generation method can cope with a wider variety types of objective functions even for complicated objective functions. An automatic software testing tool can be implemented based on the methods.

4. Robustness

The testing method requires little pre-processing of the software system under test. The testing method and the test data generation method can work even under extreme and exceptional circumstances without failure, such as complicated iteration structures. The testing method and the test data generation method are robust, little human involvement is needed during the testing process.

Hierons [36] introduced an algorithm that rewrites the Z specifications into a form, partitions the input domain and derives test cases from the subdomains. Using the algorithm, a Z specification is rewritten and simplified by flattening the specification into first-order predicate logic and set theory including breaking up of set expressions. For example, a set comprehension

$$x \in \{ y : \mathrm{Y} \mid p(y) \bullet f(y) \}$$

is broken into

$$\exists\, y \bullet p(y) \wedge (x = f(y))$$

The next step of rewriting Z specifications is to remove all the existential and universal quantifiers, so the Z specification is decomposed into a conjunction and disjunction form. If the subdomains defined by the set of preconditions *Pi* are disjoint, the behaviour on all the states that satisfy a particular *Pi* is the same post condition *Qi*. Thus the set of *Pi* represent the subdomains of the partition of the input domain. Test cases can be derived from the partition. It is also possible to derive a finite state automaton model from the rewritten Z specification.

There is a similarity between ZTEST and the algorithm introduced by Hierons for deriving test cases from Z specifications according to the partition of input domain. Like Hierons's algorithm, test cases are derived from the partition of input domain in ZTEST. The subdomains of the partition are defined from the functions described in Z specifications which are conjunctions of Z expressions.

In contrast to Hierons's approach, Z specifications need not be rewritten in ZTEST. For example, the set comprehension

$$X = \{ y : Y \mid p(y) \bullet f(y) \}$$

and the existential quantifier

$$\exists i : N \mid q(i) \bullet f(i)$$

are both considered as iteration structures in ZTEST. ZTEST treats an iteration structure as one expression in a Z specification. Test cases are derived from the expressions by generating objective functions for each test case. Objective functions in ZTEST are conjunctions of simple expressions including the ones decomposed from the iteration structures. The only pre-process for Z specifications is writing them into the input format that can be read by ZTEST as input.

Because derivation of test cases from Z specifications is almost an automatic process in ZTEST, it is more efficient and more robust than Hierons's algorithm.

The Z specification example of partitioning into subdomains given by Hierons was:

*calculate*

$a?$ : R
$b?$ : R
$c!$ : R

---

$a? \geq 0$
$b? \geq 0$
$\exists p_1, p_2 : \mathrm{R} \bullet (((a? \geq 50) \wedge (p_1 = 0.95)) \vee ((a? < 50) \wedge (p_1 = 1))) \wedge$
$(((2 * a? + b? \geq 1000) \wedge (p_2 = 0.9)) \vee ((2 * a? + b? < 1000) \wedge (p_2 = 1))) \wedge$
$c! = p_1 * p_2 * (2 * a? + b?)$

In the existential quantifier, $p_1$ and $p_2$ are intermediate variables to represent the two discount rates for calculating the price $c!$ of purchasing the two kinds of products $a?$ and $b?$. For each of $p_1$ and $p_2$ there are only two possible values ($p_1 = 0.95$ or $p_1 = 1$; $p_2 = 0.9$ or $p_2 = 1$). The Z specification was rewritten into simple expressions linked by conjunction and disjunction combining the possible values of $p_1$ and $p_2$. General existential quantifiers such as:

$$\exists \mathrm{i} : 1 \,..\, hwm \mid q(\mathrm{i}) \bullet f(\mathrm{i})$$

were both covered by Hierons's approach. In this case the intermediate variable i has the possible values from 1 to *hwm*, which represent an iteration structure.

One of the advantages of ZTEST over Hierons's algorithm is that ZTEST can handle this kind of existential quantifiers by treating them as iteration structures. ZTEST can also generate test cases automatically to test four boundary values and an intermediate value from the domain of the iteration (intermediate ) variable.

ZTEST can also handle the existential quantifiers when the domain of the iteration variable is the whole domain of a sequence and the existential quantifiers when the domain of the iteration variable is not defined, as described in Chapter 4.3.

Another advantage of ZTEST over Hierons's algorithm is that ZTEST is an automatic testing tool that derives test cases automatically from Z specifications using evolutionary algorithms. No tool for test case derivation or test data generation has been constructed using Hierons's algorithm. Therefore ZTEST is more efficient and robust than Hierons's algorithm.

There are advantages in Hierons's algorithm. One of the advantages of Hierons's algorithm is that the algorithm is based on some general rules to rewrite Z specifications, and therefore it has wider applicability. These rewriting rules could be used for Z specifications with variables of any data types, but currently there is no support for real data type in ZTEST. On the other hand, Hierons's algorithm can be applied to high level Z specifications. In order to simplify the implementation of the automatic tool, ZTEST can only be applied to refined Z specifications whose sets are represented using sequences.

The other advantage of Hierons's algorithm is that the rewritten Z specifications can also be used to derive a finite state automaton model, while ZTEST does not have the feature.

Weyuker [61] proposed an algorithm for automatically generating test data from specifications that are Boolean formulas. A testing tool was developed to implement the algorithm. The tool was applied to twenty Boolean formulas from a specification to automatically generating test data sets.

The kind of specifications Weyuker used are Boolean formulas such as:

$$a(b\bar{c} + \bar{d})$$

The Boolean operations "and", "or" and "not" are represented by ".", "+" and "-", respectively. Usually "." is omitted. The possible values of a Boolean variable are "true", "false" and "don't care" that are represented by 1, 0 and $\phi$.

A Boolean formula can be represented by disjunctive normal form, also known as sum of products form. For the above example, the disjunctive normal form is:

$$ab\bar{c} + a\bar{d}$$

Weyuker's testing tool takes a Boolean formula specification as input, then converts the formula to its disjunctive normal form. The resulting formula is used to generate test cases.

Given a Boolean formula in disjunctive normal form containing m product-terms: $F = p_1 + p_2 + \ldots p_i + \ldots + p_m$, the ith product-term $p_i = ( l_{i1}\ l_{i2} \ldots l_{ik} )$ where $l_{ij}$ denotes the jth literal in the ith term. The points from the Boolean space that make $F$ true as well as $p_i$ true, but not make any other $p_j$ true is denoted by $U_i$. One point is selected as test data from each $U_i$.

In Weyuker's algorithm $p_{ij}$ is obtained by complementing the jth literal $l_{ij}$ of $p_i$. For example, if $p_i = abd$, then $p_{i3} = ab\bar{d}$ . The points that make F false but make $p_{ij}$ true are denoted by $N_{ij}$. One point is selected as test data from each $N_{ij}$. The points of test data are selected randomly from the Boolean space using a uniform distribution, so the algorithm is non-deterministic.

Weyuker's algorithm is supported by a tool to automatically convert a Boolean formula specification into its disjunctive normal form, and automatically generate test data sets from the formulas. ZTEST is equally robust and it automatically parses Z specifications to derive test cases, as well as automatically generating test data sets for these test cases.

Weyuker's algorithm is non-deterministic, and therefore the test data are more diverse than the test data generated by deterministic algorithms. The test strategy of ZTEST is a combination of deterministic assignment and non-deterministic test data generation algorithms depending on the type of objective functions. In the general case, ZTEST generates test data sets using non-deterministic evolutionary algorithms to keep the diversity of the test data. Meanwhile, direct assignment is used for some specific type of objective functions to achieve high efficiency in test data generation.

Weyuker's testing tool was applied to 20 Boolean formulas from a specification and generated test data sets successfully. ZTEST was applied to 48 Z schemas from 7 Z specifications. The size of Z specifications varied from one schema to twenty seven schemas. ZTEST automatically generated test data sets for all of the Z specifications with full functional coverage.

Weyuker's algorithm is suitable for the specification of Boolean formulas. The operations in the specification are limited to "and", "or" and "not", the only possible values of the Boolean formulas are "true", "false" and "don't care". The applicability of Boolean formulas as specifications for software systems is very limited. In comparison, ZTEST is applied to Z specifications that are mathematical formal specifications based on set theory. Z specifications provide more colourful features to describe a general software system, including various data types of variables, a family of set and logic operations and complicated iteration structures that are impossible for Boolean formulas. ZTEST has wider applicability than Weyuker's algorithm.

On the other hand, Weyuker's algorithm may have higher speed in test data generation due to the simplicity of data type and data structures.

Arkko [60] presented a testing system that can generate test cases from the specification of algebraic form. The testing system generates tests for the specification and tests for the implementation as well.

Arkko considered the algebraic specification to be based on the concept of data abstraction. The data types in the specification are defined in terms of the operations applicable to them. A specification consists of two parts: the syntactic part defines the data type operations and the semantic part defines the behaviour of the operations using axioms. The operations are divided into classes of non-observers and observers. Test cases for the specification are generated from the syntactic part by generating expressions of the tested data type and selecting appropriate values for the variables in

the expressions. The expressions are generated by combining the non-observer operations up to a certain level. The value of the variables in the expressions are either defined specifically by the user, or selected randomly by the testing system. For instance, a specified data type $t$ consists of non-observers $con1(\text{int}) \rightarrow t$ and $con2(t, \text{int}) \rightarrow t$. For each combination of the operations, two expressions are given in which *value1* and *value2* are used for the first argument of the data type $t$. The resulting expressions for the first level are:

*con1( i1 )*
*con2(value1, i2)*
*con2(value2, i3)*

The resulting expressions for the second level are:

*con1( i1 )*
*con2(value1, i2)*
*con2(value2, i3)*
*con2(con1( i1 ), i4)*
*con2(con2(value1, i2), i5)*
*con2(con2(value2, i3), i6)*

Together with the appropriate values of the variables, these expressions are used as the test cases to test the specification.

By giving the relations between the specification operations and the implementation that includes user implemented equality functions, the test cases are transformed into test cases to test the implementation. The axioms from the semantic part of the specification are used to add more test cases for the implementation at this stage.

The test cases for testing specifications are generated merely from the syntactic part of the specification in Arkko's system. There is a possibility that some of the axioms are not satisfied by any of the expressions generated. It means that the specification is tested

only partially because the semantic part of the specification is not used in generating test cases for testing the specification. In ZTEST, Z expressions in both the declaration part and the predicate part are involved in the process of deriving test cases; the test cases generated cover the full functionality of the Z specification. ZTEST is more effective in generating test cases for testing specifications.

In Arkko's system, the values of variables in the test case expressions are either user defined, or selected randomly by the system. In ZTEST, the initial values of the variables in Z are defined by the user and the subsequent values of the variables are generated using evolutionary algorithms. Evolutionary algorithms are more efficient to generate test data than random selection for more complex objective functions (expressions).

Arkko's system is applicable for specifications in algebraic form. There was no algorithm mentioned for dealing with complex data structures such as existential and universal quantifiers. The test system does not have the wide applicability of ZTEST.

One of the advantages of Arkko's test system is that it allows user participation for giving additional information and adding additional test cases. Another advantage of Arkko's test system is that the system can be used to generate test cases for testing the specification and test cases for testing the implementation. The test data sets generated by ZTEST are used for testing the specifications only. It will enhance the capability of ZTEST if these two features are added.

Gallagher [51] introduced a software testing system ADTEST for generating test data from programs developed in Ada83. The testing system takes the tested program source code, inserts instrumentation statements into the source code to return information to the test data generation system about the state of different variables, path predicates and about test coverage achieved. A set of test paths is then generated from the tested program source code to form the objective functions for the test data generation system. The test data generation problem is treated as a numerical optimisation problem. The classical optimisation method quasi-Newton method is used to generate test data from the objective functions.

One restriction of classical optimisation methods is that the objective function must be differentiated. A group of penalty functions are given by Gallagher to overcome the problem. For the objective function $f(\mathbf{x})$ including the set of non-linear constraints:

$$g_i(\mathbf{x}) \begin{Bmatrix} > \\ \geq \\ = \\ \neq \end{Bmatrix} 0, \quad i = 1, 2, ..., m$$

the penalty imposed objective function becomes:

$$f'(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n} G(g_i(\mathbf{x}), w_i)$$

where $\mathbf{w} = (w_1, w_2, ..., w_n)$ is a set of positive weighting factors, and the $G(g_i(\mathbf{x}), w_i)$ represents the ith penalty imposed constraint.

The penalty functions for the four operators are:

$$\text{For } g_i(\mathbf{x}) \geq 0, \qquad G(g_i(\mathbf{x}), w_i) = e^{-w_i(g_i(x)+\delta)}$$

For $g_i(\mathbf{x}) > 0$, $G(g_i(\mathbf{x}), w_i) = e^{-w_i(g_i(x))}$

For $g_i(\mathbf{x}) = 0$, $G(g_i(\mathbf{x}), w_i) = e^{-w_i|g_i(x)|-1}$

For $g_i(\mathbf{x}) \neq 0$, $G(g_i(\mathbf{x}), w_i) = e^{-w_i|g_i(x)|}$

Test data are generated by minimising the objective function $f'(\mathbf{x}, \mathbf{w})$ using the quasi-Newton method. The use of penalty functions ensures a continuous objective function therefore allowing derivative information to be used in the optimisation search.

By treating the test data generation process as a numerical optimisation process, Gallagher's testing system does not suffer from the difficulties found in symbolic execution systems such as input dependent loops and array references. In ZTEST, test data generation is also an optimisation process. The difference is that a classical optimisation method is used in Gallagher's testing system while evolutionary algorithms are employed in ZTEST. The other difference is that Gallagher's test system is for structural testing and ZTEST is for functional testing.

Unlike classical methods, evolutionary algorithms do not require the objective functions to be continuous. No matter what form the objective function is in, evolutionary algorithms can work provide a numerical measurement is given for the closeness of the tests to the optimal solution.

In Gallagher's testing system, it is possible for the penalty functions to exceed the maximum floating point value. In the case the optimisation method fails the search because the evaluation values from the objective functions are all truncated to the

maximum value giving a zero gradient. There is no such restriction for evolutionary algorithms, so ZTEST has wider applicability than Gallagher's test system for test data generation.

Experiments shown that Gallagher's testing system failed to generate test data when initial search points are very distant from the optimal solution. Sensible initial search points need to be chosen to ensure the success of the optimisation search. By contrast, genetic algorithms are not as sensitive as classical methods to the initial search points. Test data generation process in ZTEST starts from a randomly generated initial values, there is no requirement to choose specific initial values. With the further enhancement of automatic restart and complementary test cases for some extreme cases, it is unlikely ZTEST will fail to generate test data. ZTEST is more robust than Gallagher's testing system.

On the other hand, Gallagher's testing system is more flexible than ZTEST by allowing real data type and allowing the user to hand select a test path from the generated test paths.

Jones et al. [50] developed a library of genetic algorithms and applied them to test data generation for structural software testing. In the test data generation process, a control flow graph is generated manually for the program under test to form the fitness functions. Instrumentation statements are inserted into the tested program to calculate the fitness values from the fitness functions, and to check the branch coverage of the test data sets. Genetic algorithms are applied to the fitness functions to automatically generate test data.

The test data generated using Jones's technique has high quality by selecting the fitness functions to generate data close to a subdomain boundary where the chance of revealing an error is higher. For loops in the tested program, test data are generated corresponding to one, two and more than two iterations. The number of generations for genetic algorithms is set to a limit of between 100 and 2000. Uniform crossover with a probability of 0.5, mutation with a probability of the reciprocal of the bit string length and a weighted mutation of the five least significant bits are used for genetic algorithms.

Similar genetic algorithm is used in ZTEST as one of the test data generation method, but Jones's technique is for structural testing while ZTEST is for functional testing. A software system may be structurally correct but still has errors because the implementation may not reflect the user requirement correctly. Functional testing is based on the software specification which is a representation of the user requirement in the software system. Functional testing is an important step in the process of software development.

The advantage of Jones's technique is that it has a library of genetic algorithms including weighted mutation operation and grey code bit string representations that are not available in ZTEST.

One of the advantages of ZTEST is that crossover and mutation operations in GA are not applied to the fixed length of bit strings but only applied to part of the bit strings depending on the domain boundary values of the variables. By applying GA operators

up to the most significant bit of the bit string representing the maximum and minimum values of the variable, the tests in GA process are kept inside the valid domain, meanwhile the GA process becomes more efficient by ruling out the unnecessary operations.

The maximum generations of GA search is set to a fixed number between 100 and 2000 in Jones's technique. There is a more intelligent approach in ZTEST that if the solution is not found after certain number of generations, GA search is restarted automatically. ZTEST is also capable for automatically altering the objective functions to generate complementary test data in some extreme cases.

Genetic algorithm is not the only algorithm used in ZTEST. The test data generation strategy in ZTEST consists of a combination of deterministic and non-deterministic algorithms for high effectiveness and efficiency of test data generation from different kind of objective functions.

Another advantage of ZTEST over Jones's technique is that it is a fully automatic testing system capable not only for automatic test data generation, but also for automatic deriving test cases from Z specifications and automatically generating objective functions.

Most importantly, functional testing like ZTEST is preferred to structural testing like Jones's technique in the process of software development especially for large software systems.

## 7.5. Perspectives

By generating test data sets from Z specifications, ZTEST has wider applicability than the testing systems based on the specifications in algebraic or Boolean forms. As an automatic software testing system, ZTEST is more efficient and robust than the algorithms that are not supported by automatic testing tools. Having a test data generation strategy which is a combination of evolutionary algorithms and deterministic method, ZTEST is more flexible and robust than the testing techniques using classical optimisation methods or using only genetic algorithms for test data generation. Having been applied successfully to 7 different scaled Z specifications that include totally 48 Z schemas, ZTEST can be easily used for test data generation from Z specifications.

ZTEST can be improved from further work and adding more features. The coverage of Z constructs by ZTEST can be expanded to support real data type, and to support set variables and set operations in a more general way. The new features will make ZTEST applicable not only to refined Z schemas, but also to general high level Z specifications. Currently the initial values of the variables in Z are set by the user through an initialisation schema. It is useful to add the ability of generating these initial values automatically for the functions under test. It is also possible to enhance the functionality of ZTEST by adding the capability of allowing user participation in generating test cases. It is necessary to allow the user to add more test cases of special interest, to generate test cases for any user selected input values and to allow the user to chose a test path from the generated objective functions.

ZTEST generates test cases automatically from Z specifications. The test cases can be used by testing engineers to test the functionality of the software system described in Z. Further work can be done to make a more comprehensive testing tool that takes the Z specification as input, generates test cases and tests the software system in one go.

Test cases are generated automatically by ZTEST from Z specifications. Z specification is a formal language that can be used to precisely describe the functionality of computer systems. Therefore, ZTEST can be used for test data generation of software systems as well as hardware systems.

1. Test data sets have been automatically generated for both numerical and string data types to test the functionality of simple numerical procedures, the birthday book and the UNIX filing system from their Z specifications. Different structured properties of software systems are covered, such as arithmetic expressions, existential and universal quantifiers, set comprehension, union, intersection and difference, etc. A CASE tool ZTEST has been implemented to automatically generate test data sets. Unlike manual tests, it has the advantage of automatic test data generation.

2. Test cases can be derived from the functionality of the Z specifications automatically. The test data sets generated from the test cases check the behaviour of the software systems for both valid and invalid inputs. Test cases are generated for the boundary values of input search domain. For test cases that derived from the input variable iteration structures in Z such as the existential quantifiers, five sample values are tested from the input search domain, four extreme values and an intermediate value. Test cases can also be derived from nested iteration structures.

3. High quality testing can be achieved by analysing the system described in Z specifications. For input variables of integer type, high quality test data sets can be generated on the search domain boundary and on each side of the boundary for both valid and invalid tests. To generate test data sets that can test the system more thoroughly, the search domain of the input variables must be clearly defined.

4. Adaptive methods such as Genetic Algorithms and Simulated Annealing are used successfully to generate test data sets from the test cases. GA is chosen as the default

test data generator of ZTEST for input objective functions. Random testing is chosen to find the solution for state variables in the universal quantifiers. Direct assignment is used if it is possible to make ZTEST system more efficient.

5. Z specification is a formal language that can be used to precisely describe the functionality of computer systems. Therefore, the test data generation method can be used widely for test data generation of software systems. It will be very useful to the systems developed from Z specifications.

## References


[ 1] J. B. Wordsworth, "Software Development with Z: a Practical Approach to Formal Methods in Software Engineering", Addison-Wesley, 1992.

[ 2] B. Beizer, "Software Testing Techniques", Second Edition, Van Nostrand Reinhold, 1990.

[ 3] O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, "Structured Programming", Academic Press 1972.

[ 4] T. Higashino, G. V. Bochmann, "Automatic Analysis and Test Case Derivation for a Restricted Class of LOTOS Expressions with Data Parameters", IEEE Trans. Software Eng. Vol. 20, No. 1, P29-42 January. 1994.

[ 5] N. D. North, "Test Generation for the Triangle Problem", NPL Report, National Physical Laboratory, UK. 1989.

[ 6] G. Laycock, "Formal Specification and Testing: a Case Study", Software Testing, Verification and Reliability, Vol. 2 P7-23 1992.

[ 7] A. J. J. Dick, P. J. Krause, J Cozens, "Computer Aided Transformation of Z into Prolog", Proc. 4th Annual Z Users Meeting, Oxford Univ. Computing Lab. 1989.

[ 8] M. M. West, B. M. Eaglestone, "Software Development: Two Approaches to Animation of Z Specifications Using Prolog", Software Engineering Journal, P264-276 July 1992.

[ 9] J. M. Spivey, "An Introduction to Z and Formal Specifications", Software Engineering Journal, 4(1), 40-50, 1989.

[10] I. Hayes, "Specification Case Studies", Prentice Hall, 1993.

[11] M. A. McMorran, J. E. Nicholls, "Z User Manual", IBM UK Laboratories Technical Report, 1989.

[12] J. M. Spivey, "The Z Notation: a Reference Manual, Second Edition", Prentice Hall, 1992.

[13] D. Lightfoot, "Formal Specification Using Z", Macmillan, 1991.

[14] G. J. Myers, "The Art of Software Testing", Wiley Interscience, 1979.

[15] J. Abbott, "Software Testing Techniques", NCC Publications, 1986.

[16] M. Morrow, "Genetic Algorithms", Dr. Dobb's Journal, P26-32 April 1991.

[17] L. Davis, M. Steenstrup, "Genetic algorithms and simulated annealing", Pitman, London, P1-11 1989.

[18] L. B. Booker, D. E. Goldberg, J. H. Holland, "Classifier Systems and Genetic Algorithms", The Univ. of Michigan, Cognitive Science and Machine Intelligence Lab. Technical Report No. 8, April 1987.

[19] G. E. Liepins, U. R. Hilliard, "Genetic Algorithms: Foundations and Applications", Annals of Mathematics and Artificial Intelligence, Vol. 21, pp. 31-57, 1989.

[20] K. A. De Jong, W. M. Spears, "An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms", Internat. Workshop parallel problem solving from nature, Univ. of Dortmund, pp. 38-47, 1990.

[21] V. J. Rayward-Smith, J. C. W. Debuse, "Generalised Adaptive Search Techniques", Proceedings of ACEDC'94, PEDC, University of Plymouth, UK. pp. 141-145, 1994.

[22] K. V. Price, "Genetic Annealing", Dr. Dobb's Journal, pp.127-132, Oct. 1994.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Recipes in C, The Art of Scientific Computing", Second Edition, Cambridge University Press, 1992.

[24] X. Yang, "The Automatic Generation of Software Test Data from Z Specifications", Research Project Report, CS-95-2, University of Glamorgan, UK. Feb. 1995.

[25] B. F. Jones, H.-H. Sthamer, X. Yang, D. E. Eyres, "The Automatic Generation of Software Test Data Sets Using Adaptive Search Techniques", Software Quality Management III, Vol. 2, Computational Mechanics Publications, pp. 435-444, April 1995.

[26] X. Yang, B.F. Jones, D. E. Eyres, "Automatic Test Generation from Mathematical Software Specifications Using Genetic Algorithms", Proceedings of ACEDC'96, University of Plymouth, UK. pp. 313-315, March 1996.

[27] L. J. Osborne, B. E. Gillett, "A Comparison of Two Simulated Annealing Algorithms Applied to the Directed Steiner Problem on Networks", ORSA Journal on Computing, Vol. 3, No. 3, pp. 213-225, Summer 1991.

[28] J. W. Green, K. J. Supowit, "Simulated Annealing without Rejected Moves", IEEE Trans. On Computer-Aided Design, Vol. CAD-5, No. 1, pp. 221-228, Jan. 1986.

[29] Chiang Tzuu-Shuh, Chow YunshYong, "On the Convergence Rate of Annealing Processes", SIAM Journal on Control and Optimisation, Vol. 26, No. 6, pp. 1455-1470, Nov. 1988.

[30] Bruce Hajek, "Cooling Schedules for Optimal Annealing", Mathematics of Operations Research, Vol. 13, No. 2, pp. 311-329, May 1988.

[31] Mario P. Vecchi, Scott Kirkpatrick, "Global Wiring by Simulated Annealing", IEEE Trans. On Computer-Aided Design, Vol. CAD-2, No. 4, pp. 215-222, Oct. 1983.

[32] R. D. Knott, P. J. Krause, "On the Derivation of an Effective Animation: Telephone Network Case Study", Report No. A1.3 of Alvey Project SE/065, Univ. of Surrey, Nov. 1988.

[33] C. Morgan, B. Sufrin, "Specification of the Unix Filing System", Specification Case Studies, pp. 45-78, Prentice Hall, 1993.

[34] D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System", Communications of the ACM, Vol. 17, No. 7, pp. 365-375, July 1974.

[35] T. C. Royer, "Software Testing Management", Prentice Hall, 1993.

[36] R. M. Hierons, "Testing from a Z Specification", Software Testing, Verification and Reliability, Vol. 7, pp. 19-33, 1997.

[37] B. Strulo, "The Test Generation Framework and Method", PROST-Objects, BT7004.0.20.34, Issue 2.0, 29 Jan. 1996.

[38] "Zylva", http://public.logica.com/~prost/ch3.htm.

[39] J. Jacky, "Specifying a Safety-Critical Control System in Z", IEEE Trans. On Software Engineering, Vol. 21, No. 2, pp. 99-106, Feb. 1995.

[40] H. M. Hörcher, J. Peleska, "The Role of Formal Specifications in Software Test", Technical Report, DST Deutsche System-Technik GmbH, Edisonstr.3, D-24145 Kiel, Germany, Oct. 1994.

[41] C. W. Johnson, "Using Z to Support the Design of Interactive Safety-Critical System", Software Engineering Journal, pp. 49-60, March 1995.

[42] M. Mullerburg, "Systematic Stepwise Testing: a Method for Testing Large Complex System", Software Quality Management III, Vol. 2, Computational Mechanics Publications, pp. 391-402, April 1995.

[43] J. Bowen, S. Stepney, R. Barden, "Annotated Z Bibliography", Information and Software Technology, Vol. 37, No. 5-6, pp. 317-332, 1995.

[44] "Formaliser", http://public.logica.com/~formaliser

[45] H. B. Zadch, S. Stcpncy, "ZEST – Z Extended with Structuring: A User's Guide", PROST-Objects, BT7001.0.20.13, Issue 2.0, 22 Jan. 1996.

[46] "Defence Standards 00-55, Part 1, Section 5, 37. Testing and Integration", http://www.seasys.demon.co.uk/0055h/Part1/Section5

[47] P. Black, "Automatic generation of Tests Based on Formal Specification", http://hissa.ncsl.nist.gov/~black/FTG/formtestgen.html

[48] X. Yang, "Automatic Software Test Data Generation from Z Specifications Using Genetic Algorithms", Research Project Report, CS-96-7, University of Glamorgan, UK. Jan. 1996.

[49] P. Bennett, "Experience in Engineering Quality into Software", Software Engineering Journal, March 1996, Vol. 11. No 2. pp. 95-98.

[50] B. F. Jones, H. -H. Sthamer, D. E. Eyres, "Automatic Structural Testing Using Genetic Algorithms", Software Engineering Journal, Sept. 1996, Vol. 11. No 5. pp. 299-306.

[51] M. J. Gallagher, V. L. Narasimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems", IEEE Trans. On Software Engineering, August 1997, Vol. 23. No 8. pp. 473-484.

[52] J. W. Duran, S. C. Ntafos, "An Evaluation of Random Testing", IEEE Trans. On Software Engineering, July 1984, Vol. 10. No 4. pp. 438-443.

[53] D. C. Ince, S. Hekmatpour, " An Empirical Evaluation of Random Testing", The Computer Journal, 1986, Vol. 29. No 4. pp. 380.

[54] B. Littlewood, D. Wright, "Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software", IEEE Trans. On Software Engineering, Nov. 1997, Vol. 23. No 11. pp. 673-683.

[55] C. Chung, T. K. Shih, C. Wang, "Object-Oriented Software Testing and Metric in Z Specification", Information Science, 1997, Vol.98, No.1-4, pp. 175-202.

[56] P. Stocks, D. Carrington, "A Framework for Specification-Based Testing", IEEE Trans. On Software Engineering, Nov. 1996, Vol. 22. No 11. pp. 777-792.

[57] M. Neil, G. Ostrolenk, M. Tobin, M. Southworth, "Lessons from Using Z to Specify a Software Tool", IEEE Trans. On Software Engineering, Jan. 1998, Vol. 24. No 1. pp. 15-23.

[58] J. Savir, "Random Pattern Testability of Memory Control Logic", IEEE Trans. On Computers, March 1998, Vol. 47. No 3. pp. 305-312.

[59] S. Chen, S. Mills, "A Binary Process Model for Random Testing", IEEE Trans. On Software Engineering, March 1996, Vol. 22. No 3. pp. 218-223.

```
&sname. AddError
&bsdec.
 name? : NAME
 reply! : MESSAGE
&esdec.
&bspre.
 &exi.
  i : 1 .. hwm
  &bul. name? = names(i)
 &eexi.
 reply! = Name already known
&espre.
```

2.3. The Input of Z Schema Card List

```
&sname. CardList
&bsdec.
 Initial
 today? : DATE
 cardlist! : &seq. NAME
 ncard! : &Int.
&esdec.
&bspre.
 &exi.
   k : 1 .. hwm
   &bul.  today? = dates(k)
   &setc.
    j : 1 .. ncard!
    &bul. cardlist!(j)
   &esetc.
   = &setc.
        i : 1 .. hwm
        &cbar.  dates(i) = today?
        &bul. names(i)
    &esetc.
 &eexi.
&espre.

&sname. CardError
&bsdec.
 today? : DATE
 reply! : MESSAGE
&esdec.
&bspre.
 &all.
  i : 1 .. hwm
  &bul. today? &neq. dates(i)
 &eall.
```

```
  reply! = No Card Today
 &espre.
```

2.4. The Input of Z schema Open

```
 &sname. Open0
 &bsdec.
  Initial
  cid! : CID
  name? : NAME
  c : CID
  cids' : &seq. CID
  cfids' : FIDS
  posns' : &seq. &Nat.
  numc' : &Nat.
 &esdec.
 &bspre.
  numc' = numc + 1
  &all.
   i : 1 .. numc
   &bul. c &neq. cids(i)
  &eall.
  &exi.
   j : 1 .. numn
   &cbar. name? = names(j)
   &bul. cfids' = cfids &fovr. &lset. numc' &map. nfids(j) &rset.
  &eexi.
  cid! = c
  cids' = cids &fovr. &lset. numc' &map. c &rset.
  posns' = posns &fovr. &lset. numc' &map. 0 &rset.
 &espre.

 &sname. NameError
 &bsdec.
  name? : NAME
  report! : REPORT
 &esdec.
 &bspre.
  &all.
   i : 1 .. numn
   &bul. name? &neq. names(i)
  &eall.
  report! = No Such Name
 &espre.

Open &sdef. &lsch. Open0 &or. NameError &rsch.
```

2.5. The Input of Z schema Seek

```
&sname. Seek0
&bsdec.
 Initial
 cid? : CID
 newposn? : &Nat.
 cids' : &seq. CID
 cfids' : FIDS
 posns' : &seq. &Nat.
 numc' : &Nat.
&esdec.
&bspre.
 &exi.
  i : 1 .. numc
  &&cbar. cid? = cids(i)
  &bul. posns' = posns &fovr. &lset. i &map. newposn? &rset.
 &eexi.
 cids' = cids
 cfids' = cfids
 numc' = numc
&espre.

&sname. CidError
&bsdec.
 cid? : CID
 report! : REPORT
&esdec.
&bspre.
 &all.
  i : 1 .. numc
  &bul. cid? &neq. cids(i)
 &eall.
 report! = No Such Cid
&espre.

Seek &sdef. &lsch. Seek0 &or. CidError &rsch.
```