Fuzzy Logic Based Objective Function Construction for Evolutionary Test Generation

Andrea G. B. Tettamanzi

Università degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione
Via Bramante 65, I-26013 Crema (CR), Italy
andrea.tettamanzi@unimi.it

Abstract. The test case generation problem can be stated as an optimization problem whereby the closeness of test cases to violating the postcondition of a formal specification is maximized, subject to satisfying its precondition. This is usually implemented by constructing an objective function which provides a real-valued estimate of how distant all of the constraints are from being violated, and then trying to minimize it. A problem with this approach is that such objective functions may contain plateaux, which make their minimization hard. We propose a similar approach, grounded on fuzzy logic, which uses, instead of a "distance from violation" objective function, a fuzzy degree of proximity to postcondition violation and produces plateaux-free objective functions by construction. The approach is illustrated with the help of a case study on the functional (black-box) testing of computer programs.

Keywords. Functional (black-box) testing, `MIX`, evolutionary testing

1 Introduction

The use of evolutionary algorithms for the automatic generation of test data has received increasing attention in recent years [1].

The main motivation for the work presented in this paper has been to gain some insights on functional testing in a simplified environment. Since the author is using Knuth's `MIX` machine to introduce elementary programming techniques in a 1st-year class, the choice of that machine as a case study for functional testing was natural.

The main goal of this paper is to propose a method for the design of a fitness function for evolutionary approaches to test generations, which takes inspiration from concepts arising in fuzzy set theory.

A brief overview of the `MIX` machine is provided in Section 2. Functional testing requires a formal specification of the program being tested: an *ad hoc* specification language for `MIX` programs is introduced in Section 3. Section 4 is a gentle introduction to fuzzy logic and some concepts arising in it which provided an inspiration for a method for designing objective functions for evolutionary test generation, which is proposed in Section 5.

Dagstuhl Seminar Proceedings 08351
Evolutionary Test Generation
http://drops.dagstuhl.de/opus/volltexte/2009/2016

2 The MIX Machine

Donald E. Knuth introduced the MIX machine, a fictitious computer, in the first volume of his celebrated masterwork *The Art of Computer Programming* [2].

The basic unit of information is a 6-bit *byte*, which can represent numbers between 0 and 63. A computer *word* consists of five bytes and a sign (+ or −). The memory of the MIX machine consists of 4000 words.

The MIX machine has ten registers: the accumulator (rA) and the extension (rX), both consisting of five bytes and a sign; six index registers (rI1–rI6), each holding two bytes and a sign; the location counter (*, the address of the next instruction to be executed), and the jump register (rJ, which saves the previous value of * when a jump is taken), both holding two bytes (their sign is always +). Together, rA and rX form the extended accumulator rAX, consisting of ten bytes and rA's sign.

Besides the memory and the registers, the MIX machine has an *overflow toggle* ("on" or "off"), a *comparison indicator* (with three possible values: Less, Equal, or Greater), and several I/O devices.

MIX instructions can address partial fields of memory words. The five bytes and sign of a word are numbered as follows:

0	1	2	3	4	5
±	0 . . . 63	0 . . . 63	0 . . . 63	0 . . . 63	0 . . . 63

.

The allowed fields are those that are adjacent in a word, and they are represented by $(l : r)$, where l is the number of the left-hand part and r is the number of the right-hand part of the field. Thus, $(0 : 0)$ is the sign only, $(0 : 5)$ is the whole word, and $(4 : 4)$ is byte #4 only.

A platform-independent, open-source MIX simulator and development environment is MixIDE, originally written by the author for teaching his first-year class on *Computer Programming* at the University of Milan.

3 A Specification Language for MIX Programs

Functional testing requires a formal specification of the program being tested.

Instead of using a full-fledged, general-purpose formal specification language such as Z notation [3] or CASL [4], it is more convenient to devise a simpler, streamlined specification language for MIX programs.

Since there is no explicit concept of *type* when programming the MIX machine, as it is the case in general for assembly or machine level programming, the only entities a specification needs to handle are integers, which, depending on the context, may represent memory locations (addresses), contents of registers or of (fields of) memory words, and field specifications.

The minimal requirements for a language to be suitable to specify MIX programs are the following:

```
   ⟨SYMBOL⟩ ::= any C-style identifier
   ⟨NUMBER⟩ ::= any integer literal
   ⟨STRING⟩ ::= a 5-character string surrounded by double quotes (")
 ⟨REGISTER⟩ ::= rA | rX | rAX | rI1 | ... | rI6 | rJ | *
       ⟨OP⟩ ::= + | − | × | / | : | mod | ↑
     ⟨EXPR⟩ ::= ⟨SYMBOL⟩ | ⟨NUMBER⟩ | ⟨STRING⟩ | ⟨REGISTER⟩ |
                (⟨EXPR⟩) | ⟨EXPR⟩(⟨FSPEC⟩) | ⟨EXPR⟩ ⟨OP⟩ ⟨EXPR⟩
    ⟨FSPEC⟩ ::= any ⟨EXPR⟩ whose value is in {0, ..., 45}
  ⟨BINDING⟩ ::= ⟨SYMBOL⟩ ← ⟨EXPR⟩
      ⟨CMP⟩ ::= = | ≠ | < | > | ≤ | ≥
     ⟨ARGS⟩ ::= ⟨EXPR⟩ | ⟨EXPR⟩, ⟨ARGS⟩
     ⟨PRED⟩ ::= ⟨EXPR⟩ ⟨CMP⟩ ⟨EXPR⟩ | ⟨SYMBOL⟩(⟨ARGS⟩)
      ⟨WFF⟩ ::= ⟨PRED⟩ | ⟨BINDING⟩ | (⟨WFF⟩) | ⟨WFF⟩ ⟨CONN⟩ ⟨WFF⟩
     ⟨VARS⟩ ::= ⟨SYMBOL⟩ | ⟨SYMBOL⟩, ⟨VARS⟩
      ⟨DEF⟩ ::= ⟨SYMBOL⟩(⟨VARS⟩) ← ⟨WFF⟩ . | ⟨BINDING⟩ .
  ⟨PRECOND⟩ ::= ⟨WFF⟩ ?
 ⟨POSTCOND⟩ ::= ⟨WFF⟩ !
     ⟨SPEC⟩ ::= ⟨DEF⟩* ⟨PRECOND⟩ ⟨POSTCOND⟩
```

Fig. 1. A BNF grammar of the MIX specification language.

1. evaluate integer expressions constructed with the usual arithmetical operators like $+$, $-$, $\times$, $/$, etc.;
2. compare integer values with $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, which constitute the elementary predicates of the language;
3. define and evaluate new predicates based on more elementary predicates, by combining them with the $\neg$, $\wedge$, and $\vee$ logical connectives;
4. bind integer values to variable symbols, both locally (in a predicate definition) and globally;
5. state pre- and post-conditions for a program.

3.1 Syntax

The BNF grammar of the MIX specification language (MSL) is shown in Fig. 1.

The : operator is defined, for all $n, m \in \mathbb{N}$, as $n : m = 8n + m$, and is used to construct specifications of fields (⟨FSPEC⟩) of a byte-addressable MIX word: an ⟨FSPEC⟩ of $0 : 0$ specifies the sign of a word, of $0 : 2$ the address part of a word (i.e., the two most significant bytes with sign), of $0 : 5$ the whole word content with sign, of $1 : 5$ the whole word content whithout sign (i.e., its absolute value), of $5 : 5$ the least significant byte, etc.

Given an expression (⟨EXPR⟩) whose value is $0 \leq n < 4000$ and an ⟨FSPEC⟩ F, $n(F)$ is the content of the field specified by F of the word at location n in the MIX memory. This provides a way to write predicates about the contents of the MIX memory before and after execution of a program.

A ⟨BINDING⟩ allows one to assign values to global or local variables, depending on the nature of the variable symbol to which it is applied. Variables are

global if they are bound in a pre-condition (⟨PRECOND⟩), in a post-condition (⟨POSTCOND⟩), or in a definition (⟨DEF⟩) of the form "⟨BINDING⟩."; variables are local to a ⟨DEF⟩ if they appear in the list of the arguments of the defined predicate or if they are explicitly bound in the well-formed formula (⟨WFF⟩) which constitutes the body of a predicate definition.

3.2 Example: Triangle Classification

As a first example of how to use the MSL, we specify a triangle classification program, a benchmark used in many testing papers [1]. Assuming three non-zero, non-negative integer lengths of the sides of a triangle, stored in words 1, 2, and 3 of the `MIX` memory, the program decides if the triangle is isosceles, equilateral, scalene, or invalid (not a triangle), and writes its response in word 4, in the form of 5-character labels `ISOSC`, `EQLAT`, `SCALN`, and `NOTRG`.

The above informal specification can be formalized in MSL as follows:

$a \leftarrow 1. \quad b \leftarrow 2. \quad c \leftarrow 3. \quad \mathit{type} \leftarrow 4.$
$\text{Triangle}(x, y, z) \leftarrow x + y > z \wedge x + z > y \wedge y + z > x.$
$\text{Equilateral}(x, y, z) \leftarrow x = y \wedge y = z.$
$\text{Isosceles}(x, y, z) \leftarrow \neg\text{Equilateral}(x, y, z) \wedge (x = y \vee y = z \vee x = z).$
$a(0:5) > 0 \wedge b(0:5) > 0 \wedge b(0:5) > 0?$
$(\neg\text{Triangle}(a(0:5), b(0:5), c(0:5)) \wedge \mathit{type}(0:5) = \text{"}\texttt{NOTRG}\text{"}) \vee$
$\quad \text{Triangle}(a(0:5), b(0:5), c(0:5) \wedge ($
$\quad\quad (\text{Equilateral}(a(0:5), b(0:5), c(0:5)) \wedge \mathit{type}(0:5) = \text{"}\texttt{EQLAT}\text{"}) \vee$
$\quad\quad (\text{Isosceles}(a(0:5), b(0:5), c(0:5)) \wedge \mathit{type}(0:5) = \text{"}\texttt{ISOSC}\text{"}) \vee$
$\quad\quad (\text{Triangle}(a(0:5), b(0:5), c(0:5) \wedge \mathit{type}(0:5) = \text{"}\texttt{SCALN}\text{"})))!$

3.3 Example: Dynamic Memory Allocation

To demonstrate the expressive power of the MSL, we now tackle the specification of a much harder program which, given a linked list of free memory blocks between locations 1000 and 2999 (the *heap*), whose first element is pointed to by the address part of word 999, and the required size s of a block in `rI1` (index register 1),allocates a memory block of size s within the free memory blocks, and removes the allocated block from the heap. The base word of each individual free block on the heap contains its size in field $(4:5)$ and a pointer to the next free block in field $(0:2)$.

This can be formalized in MSL as follows:

$\mathit{heap} \leftarrow 999.$
$\mathit{next} \leftarrow 0:2. \quad \mathit{size} \leftarrow 4:5$
$\texttt{LOC}(x) \leftarrow x \geq 0 \wedge x < 4000.$
$\text{List}(x) \leftarrow \texttt{LOC}(x) \wedge (x = 0 \vee \text{List}(x(\mathit{next}))).$
$\text{NonOverlapping}(x, \mathit{xsize}, y, \mathit{ysize}) \leftarrow x = y \vee (x < y \wedge y \geq x + \mathit{xsize})$
$\qquad\qquad \vee (x > y \wedge x \geq y + \mathit{ysize}).$
$\text{Disjoint}(x, \mathit{xsize}, l) \leftarrow \text{List}(l) \wedge \text{NonOverlapping}(x, \mathit{xsize}, l) \wedge$

$$\text{Disjoint}(x, \mathit{xsize}, l(\mathit{next}), l(\mathit{size})).$$
$$\text{ValidHeap}(x) \leftarrow \text{List}(x) \wedge (x = 0 \vee (x > 0 \wedge \text{Disjoint}(x, x(\mathit{size}), x) \wedge \text{ValidHeap}(x(\mathit{next})))).$$
$$s \leftarrow \mathtt{rI1} \wedge s > 0 \wedge s < 2000 \wedge \mathit{heap}(\mathit{next}) > 0 \wedge \text{ValidHeap}(\mathit{heap}(\mathit{next}))?$$
$$b \leftarrow \mathtt{rI1} \wedge \text{ValidHeap}(\mathit{heap}(\mathit{next})) \wedge \mathtt{LOC}(b) \wedge b \geq 1000 \wedge b + s < 3000 \wedge \text{Disjoint}(b, s, \mathit{heap}(\mathit{next}))!$$

The $\mathtt{LOC}(x)$ predicate checks if x is a valid MIX memory location; $\text{List}(x)$ recursively checks that x is a list; $\text{NonOverlapping}(x, \mathit{xsize}, y, \mathit{ysize})$ checks that the memory block of size *xsize* starting at location x does not overlap with the memory block of size *ysize* starting at location y; $\text{Disjoint}(x, \mathit{xsize}, l)$ recursively checks that the memory block of size *xsize* starting at location x does not overlap with any of the blocks of list l; finally, $\text{ValidHeap}(x)$ checks that x is a valid linked list of free, non-overlapping memory blocks.

4 Fuzzy Logic

Fuzzy logic was initiated by Lotfi Zadeh with his seminal work on fuzzy sets [5]. Fuzzy set theory provides a mathematical framework for representing and treating vagueness, imprecision, lack of information, and partial truth.

Very often, we lack complete information in solving real world problems. This can be due to several causes. First of all, human expertise is of a qualitative type, hard to translate into exact numbers and formulas. Our understanding of any process is largely based on imprecise, "approximate" reasoning. However, imprecision does not prevent us from performing sucessfully very hard tasks, such as driving cars, improvising on a chord progression, or trading financial instruments. Furthermore, the main vehicle of human expertise is natural language, which is in its own right ambiguous and vague, while at the same time being the most powerful communication tool ever invented.

4.1 Fuzzy Sets

Fuzzy sets are a generalization of classical sets obtained by replacing the characteristic function of a set A, χ_A which takes up values in $\{0, 1\}$ ($\chi_A(x) = 1$ iff $x \in A$, $\chi_A(x) = 0$ otherwise) with a *membership function* μ_A, which can take up any value in $[0, 1]$. The value $\mu_A(x)$ is the membership degree of element x in A, i.e., the degree to which x belongs in A.

A fuzzy set is completely defined by its membership function. Therefore, it is useful to define a few terms describing various features of this function, summarized in Figure 2. Given a fuzzy set A, its *core* is the (conventional) set of all elements x such that $\mu_A(x) = 1$; its *support* is the set of all x such that $\mu_A(x) > 0$. A fuzzy set is *normal* if its core is nonempty. The set of all elements x of A such that $\mu_A(x) \geq \alpha$, for a given $\alpha \in (0, 1]$, is called the α-cut of A, denoted A_α.

If a fuzzy set is completely defined by its membership function, the question arises of how the shape of this function is determined. From an engineering

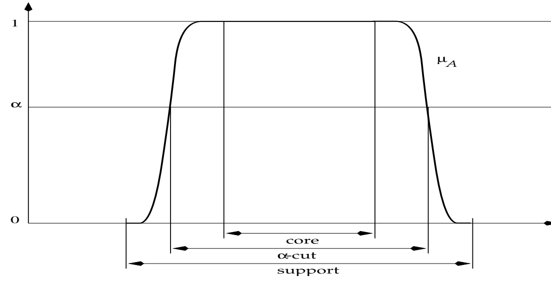

Fig. 2. Core, support, and α-cuts of a set A of the real line, having membership function μ_A.

point of view, the definition of the ranges, quantities, and entities relevant to a system is a crucial design step. In fuzzy systems all entities that come into play are defined in terms of fuzzy sets, that is, of their membership functions. The determination of membership functions is then correctly viewed as a problem of design. As such, it can be left to the sensibility of a human expert or more objective techniques can be employed. Alternatively, optimal membership function assignment, of course relative to a number of design goals that have to be clearly stated, such as robustness, system performance, etc., can be estimated by means of a machine learning or optimization method. In particular, evolutionary algorithms have been employed with success to this aim. This is the approach we follow in this chapter.

4.2 Operations on Fuzzy Sets

The usual set-theoretic operations of union, intersection, and complement can be defined as a generalization of their counterparts on classical sets by introducing two families of operators, called triangular norms and triangular co-norms. In practice, it is usual to employ the min norm for intersection and the max co-norm for union. Given two fuzzy sets A and B, and an element x,

$$\mu_{A\cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}; \quad (1)$$

$$\mu_{A\cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}; \quad (2)$$

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x). \quad (3)$$

Another pair of norm and co-norm of interest for this application is the *product* norm and the so-called *probabilistic sum* co-norm:

$$\mu_{A\cup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x); \quad (4)$$

$$\mu_{A\cap B}(x) = \mu_A(x)\mu_B(x); \quad (5)$$

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x). \quad (6)$$

4.3 Fuzzy Propositions and Predicates

In classical logic, a given proposition can fall in either of two sets: the set of all true propositions and the set of all false propositions, which is the complement of the former. In fuzzy logic, the set of true proposition and its complement, the set of false propositions, are fuzzy. The degree to which a given proposition P belongs to the set of true propositions is its degree of truth, $T(P)$.

The logical connectives of negation, disjunction, and conjunction can be defined for fuzzy logic based on its set-theoretic foundation, as follows:

$$\text{Negation } T(\neg P) = 1 - T(P); \tag{7}$$

$$\text{Disjunction } T(P \vee Q) = T(P) \bigtriangledown T(Q); \tag{8}$$

$$\text{Conjunction } T(P \wedge Q) = T(P) \bigtriangleup T(Q), \tag{9}$$

where $\bigtriangleup$ and $\bigtriangledown$ represent any pair of corresponding triangular norm and co-norm.

Much in the same way, a one-to-one mapping can be established as well between fuzzy sets and fuzzy predicates. In classical logic, a predicate of an element of the universe of discourse defines the set of elements for which that predicate is true and its complement, the set of elements for which that predicate is not true. Once again, in fuzzy logic, these sets are fuzzy and the degree of truth of a predicate of an element is given by the degree to which that element is in the set associated with that predicate.

5 Objective Function

The test case generation problem can be stated as an optimization problem as follows:

maximize *the closeness to violating* $\langle$POSTCOND$\rangle$
subject to $\langle$PRECOND$\rangle$

The way the above informal statement has been implemented by other authors, most notably Jones and colleagues [6], building on ideas first set forth by Miller and Spooner [7] and later extended by Korel and Tracey [8], is by constructing an objective function which provides a real-valued estimate of how distant all of the contraints are from being violated, and then trying to minimize it. These can be called *distance-based* approaches.

Distance-based approaches tend to contain areas of plateaux, which make their minimization hard. We propose a similar approach, inspired to fuzzy set theory [5], which uses, instead of a "distance from violation" objective function, a fuzzy degree of proximity to post-condition violation. This fuzzy degree can be regarded as a membership function

$$\delta : \langle \text{WFF} \rangle \to [0, 1], \tag{10}$$

of the set of postconditions which are violated by a given test case.

Function δ may be defined on the six comparison predicates and then extended to compound predicates. The fundamental definition is, for all $m, n \in \mathbb{Z}$,

$$\delta(m < n) = \frac{1}{1 + e^{n-m-\frac{1}{2}}}. \tag{11}$$

Of course, $\delta(m > n) = \delta(n < m)$; then, we observe that, for all $m, n \in \mathbb{Z}$, $m \leq n \Leftrightarrow m < n + 1$. Therefore,

$$\delta(m \leq n) = \delta(m < n + 1) = \frac{1}{1 + e^{n-m+\frac{1}{2}}}. \tag{12}$$

Function δ may be extended to compound predicates by defining, for all $\phi, \psi \in \langle \text{WFF} \rangle$,

$$\delta(\neg\phi) = 1 - \delta(\phi), \tag{13}$$

$$\delta(\phi \vee \psi) = \delta_1(\phi) \bigtriangleup \delta_1(\psi), \tag{14}$$

$$\delta(\phi \wedge \psi) = \delta_1(\phi) \bigtriangledown \delta_1(\psi). \tag{15}$$

We consider two alternative choices for $\bigtriangleup$ and $\bigtriangledown$:

1. for all $x, y \in [0, 1]$, $x \bigtriangleup y \equiv \min\{x, y\}$ and $x \bigtriangledown y \equiv \max\{x, y\}$;
2. for all $x, y \in [0, 1]$, $x \bigtriangleup y \equiv xy$ and $x \bigtriangledown y \equiv x + y - xy$.

Depending on the alternative used, the extended function will be denoted δ_1 and δ_2 respectively. For the disjunction and conjunction of predicates, we have

$$\delta_1(\phi \vee \psi) = \min\{\delta_1(\phi), \delta_1(\psi)\}, \tag{16}$$

$$\delta_1(\phi \wedge \psi) = \max\{\delta_1(\phi), \delta_1(\psi)\}, \tag{17}$$

or, alternatively,

$$\delta_2(\phi \vee \psi) = \delta_2(\phi)\delta_2(\psi), \tag{18}$$

$$\delta_2(\phi \wedge \psi) = \delta_2(\phi) + \delta_2(\psi) - \delta_2(\phi)\delta_2(\psi). \tag{19}$$

We now can define $\delta(m \neq n)$ by observing that $m \neq n \Leftrightarrow m > n \vee m < n$; therefore,

$$\delta(m \neq n) = \delta(m > n) \bigtriangleup \delta(m < n), \tag{20}$$

which translates into

$$\delta_1(m \neq n) = \min\{\delta(m > n), \delta(m < n)\} \tag{21}$$

or, alternatively,

$$\delta_2(m \neq n) = \delta(m > n)\delta(m < n) = \frac{1}{1 + e^{m-n-\frac{1}{2}} + e^{n-m-\frac{1}{2}} + e}. \tag{22}$$

Similarly, since $m = n \Leftrightarrow m \geq n \wedge m \leq n$,

$$\delta(m = n) = \delta(m \geq n) \bigtriangledown \delta(m \leq n), \tag{23}$$

which translated into

$$\delta_1(m = n) = \max\{\delta(m \geq n), \delta(m \leq n)\} \tag{24}$$

or, alternatively,

$$\delta_2(m = n) = \delta(m \geq n) + \delta(m \leq n) - \delta(m \geq n)\delta(m \leq n). \tag{25}$$

It is easy to verify that, no matter which extension is chosen,

$$\delta(m = n) = 1 - \delta(m \neq n), \tag{26}$$

$$\delta(m < n) = 1 - \delta(m \geq n), \tag{27}$$

$$\delta(m > n) = 1 - \delta(m \leq n); \tag{28}$$

Finally, $\delta_1(n \neq n) = 1/(1+e^{-\frac{1}{2}}) \approx 0.6225$ and $\delta_2(n \neq n) = 1/(1+2e^{-\frac{1}{2}}+e) \approx 0.2028$; both are the maximum that $\delta_1(m \neq n)$ and $\delta_2(m \neq n)$ can respectively attain for all $m, n \in \mathbb{Z}$, as shown in Fig. 3.

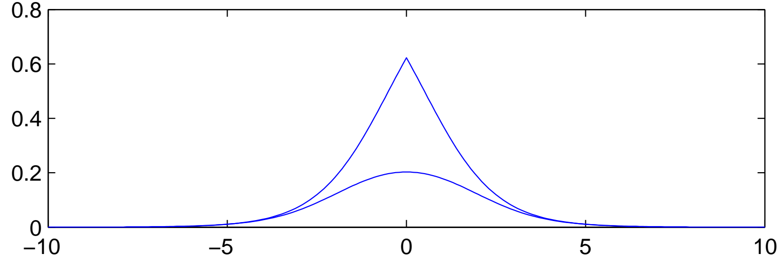

Fig. 3. A comparison of $\delta_1(m \neq n)$ and $\delta_2(m \neq n)$, plotted with respect to $m-n$ (x axis). The graph of $\delta_1(m \neq n)$ has a sharp peak for $m = n$ and always lies above the graph of $\delta_2(m \neq n)$.

It is interesting to study the behavior of δ when applied to predicate $P = \text{Triangle}(x, y, z))$, defined in Section 3.2. The general form of $\delta(P)$ is

$$\begin{aligned}\delta(P) &= \delta(x + y > z) \bigtriangledown \delta(x + z > y) \bigtriangledown \delta(y + z > x) \\ &= \delta(z < x + y) \bigtriangledown \delta(y < x + z) \bigtriangledown \delta(x < y + z) \\ &= \frac{1}{1 + e^{x+y-z-\frac{1}{2}}} \bigtriangledown \frac{1}{1 + e^{x+z-y-\frac{1}{2}}} \bigtriangledown \frac{1}{1 + e^{y+z-x-\frac{1}{2}}},\end{aligned}$$

which yields

$$\delta_1(P) = \frac{1}{1 + \min\left\{e^{x+y-z-\frac{1}{2}}, e^{x+z-y-\frac{1}{2}}, e^{y+z-x-\frac{1}{2}}\right\}} \tag{29}$$

and

$$\delta_2(P) = \alpha + \beta + \gamma - \alpha\beta - \alpha\gamma - \beta\gamma + \alpha\beta\gamma, \tag{30}$$

where

$$\alpha = \frac{1}{1 + e^{x+y-z-\frac{1}{2}}}, \quad \beta = \frac{1}{1 + e^{x+z-y-\frac{1}{2}}}, \quad \text{and} \quad \gamma = \frac{1}{1 + e^{y+z-x-\frac{1}{2}}}.$$

The graphs for $\delta_1(P)$ and $\delta_2(P)$ are shown in Fig. 4.

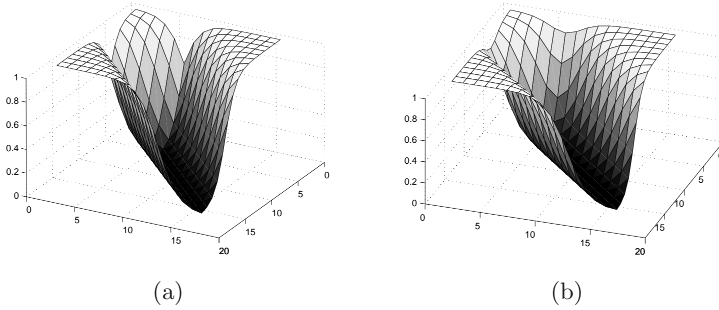

Fig. 4. Graphs of $\delta_1(\text{Triangle}(x, y, 5))$ (a) and $\delta_2(\text{Triangle}(x, y, 5))$ (b) for $x, y \in \{0, 15\}$.

6 Conclusions

A novel, principled way to define the objective function for search-based test case generation inspired by fuzzy set theory, grounded on the theory of triangular norm, and rooted in previous work on evolutionary test generation has been proposed, whose main advantage is a contribution to the elimination of plateaux from the fitness landscape. That should lead to improved performance.

Although the proposal is motivated by an application to the functional testing of `MIX` programs, the method presented can be readily adapted to functional testing of executable programs compiled for any real-world microprocessor.

References

1. McMinn, P.: Search-based software test data generation: a survey: Research articles. Software Testing, Verification & Reliability **14** (2004) 105–156
2. Knuth, D.E.: The Art of Computer Programming — 3rd ed. (3 volumes). Addison Wesley, Reading, Massachusets (1997)
3. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)

4. Mosses, P.: CASL Reference Manual. Springer, Berlin (2004)
5. Zadeh, L.A.: Fuzzy sets. Information and Control **8** (1965) 338–353
6. Jones, B.F., Sthamer, H., Yang, X., Eyres, D.E.: The automatic generation of software test data sets using adaptive search techniques. In: Proceedings of the 3rd International Conference on Software Quality Management, Seville, Spain (1995) 435–444
7. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering **SE-2** (1976) 223–226
8. Tracey, N.J.: A Search-Based Automated Test-Data Generation Framework for Safety Critical Software. PhD thesis, University of York, York, UK (2000)