Automated Black-Box Testing of Functional Correctness using Function Approximation

[Extended Abstract]

Karl Meinke
Department of Numerical Analysis and Computer Science
Royal Institute of Technology
100-44 Stockholm, Sweden
January 14, 2004
karlm@nada.kth.se

ABSTRACT

We consider black-box testing of functional correctness as a special case of a satisfiability or constraint solving problem. We introduce a general method for solving this problem based on function approximation. We then describe some practical results obtained for an automated testing algorithm using approximation by piecewise polynomial functions.

Keywords: black-box testing, constraint solving, coverage problem, formal specifications, functional testing, satisfiability testing.

1. INTRODUCTION.

Black-box functional testing of a software system involves generating and executing a set of test cases (i.e. inputs) and judging each outcome (the oracle step) using only a set of functional requirements. This approach is usually justified on the grounds that structurally based ("glass-box") testing methods, such as exhaustive path exploration, have a superlinear time complexity with respect to the size of the system under test (SUT). Thus glass-box testing methods do not scale well to large systems.

For functional test automation, some kind of formal and machine recognisable requirements language is necessary. Without any loss of generality, we can assume that first-order predicate logic is an adequate formal requirement language. (See for example [6].)

We shall consider software systems whose functional correctness can be modeled by pre and postconditions *pre* and *post* which are first-order predicate formulas. A *successful black-box test* of a system S, with respect to a functional specification *pre* and *post* is an assignment α of values to the input variables $\overline{x}$ of S which satisfies the precondition *pre*, such that S terminates given α and the resulting assignment ω to the output variables $\overline{y}$ of S (together with α) satisfies the negated postcondition $\neg post$. Black-box functional testing of S is therefore the search for successful tests with respect to a functional specification $\{pre\}S\{post\}$ of S.

We will consider a particular algorithmic approach to this search problem. As we have seen, software testing involves solving the pair of constraints *pre* and $\neg post$. However, a naive application of classical constraint solving methods does not answer all the questions we are interested in. An important issue is the choice of a stopping criterion for any search based testing algorithm. Quite simply, how much testing is enough? This problem is well known in the testing community as the *coverage problem*. We will consider a measure of coverage, based on convergence of functional approximations, which reflects the underlying complexity of the SUT.

In the classical theory of function approximation, some general class F of functions is shown to be approximable by a much more limited and tractable subclass $A \subseteq F$ of functions. Well known classes of approximating functions include polynomials, trigonometric functions, sigmoidal functions, radial basis functions, wavelets, etc. We can apply function approximation theory to testing in the following way. After executing some finite number n of test cases on an SUT S, we will have constructed n test input assignments $\alpha_1, \ldots, \alpha_n$. Assuming termination in each case, we will have computed with S a corresponding series of n output assignments, $\omega_1, \ldots, \omega_n$. Suppose that we have still not found a successful test case and wish to continue testing. Instead of throwing away the negative results obtained so far, we could try to use them in some intelligent way to construct a new (and hopefully better) test case α_{n+1}. At the very least, α_{n+1} should differ from $\alpha_1, \ldots, \alpha_n$. A more sophisticated approach than random choice of α_{n+1} is to combine $\alpha_1, \ldots, \alpha_n$ and $\omega_1, \ldots, \omega_n$ into some sort of approximate model m_n of the SUT S. From this approximate model, we may be able to construct a "best guess" of where a successful test case might be found, and choose α_{n+1} to be this estimate. Now even if α_{n+1} fails as a test, it can be used together with the output ω_{n+1} to produce a more refined approximate model m_{n+1}. By choosing α_1 at random and constructing each α_{n+1} from m_n, we obtain an iterative test case generation algorithm. In a sense, we

ISSTA 2004 Boston, Mass. USA

approach the testing problem as a kind of learning problem about the underlying system SUT S. One advantage of the approximation approach is that a stopping criterion can be formulated as a convergence criterion on approximation.

Our approach opens up a large family of interesting algorithms based on different kinds of approximations. We will illustrate the approach, and a simple case study, for a particular class of approximations, namely piecewise polynomial functions.

The approach to testing as a computational learning (CL) problem has been pursued by several authors, starting with pioneering work of Budd and Angluin [3] and Weyuker's early work (e.g. [13]). The CL approach seeks to infer a program from test results, and use the inferred program to establish test adequacy. It therefore has some similar principles to our approach. However, functional requirements are implicit in the CL approach, and convergence within an approximation space is not used to define adequacy. Nor is the automatic generation and optimisation of test cases considered.

Other approaches to learning-based testing include [2], which uses classical constraint solving algorithms and a glass box approach, but do not use approximation theory. Theoretical research on the complexity of testing measured by the complexity of learning using Vapnik-Chervonenkis (VC) dimension is [11] and [12], though this research does not yet seem to have led to the design of practical testing algorithms.

The coverage problem for black-box testing has been extensively analysed in work on mutation testing, starting with [5] and [7]. In practise, mutation testing is limited by theoretical problems, such as the recursive unsolvability of the equivalence problem for programs (see e.g. [9]). Mutation testing is concerned with quantitative adequacy measures of a specific test set, and not numerical or probabilistic results about coverage as such. Furthermore, it does it provide any feedback about how to improve an inadequate test set, as our approximation approach does.

The organisation of this paper is as follows. In Section 2 we formalise some basic definitions of black-box functional testing. In Section 3 we describe our algorithmic approach. In Section 4 we present a case study for testing a simple numerical algorithm. Finally, in Section 5 we draw some conclusions and discuss some issues for further research.

The pre-requisites of our paper are some familiarity with functional requirements specification using formal methods and logic. Technically, we make as few assumptions as possible. In particular, we introduce the concept of a functional specification formalised as a pair of pre and postconditions over a first-order language from first principles. A more detailed introduction can be found in [1] or [8].

2. LOGICAL FOUNDATIONS OF FUNCTIONAL BLACK-BOX TESTING.

In this section we review some basic technical definitions necessary to formalise functional black-box testing within the framework of the theory of program correctness. The principle concept to be defined is that of the *success or failure of a black-box test* for functional correctness.

We begin by briefly fixing our notation for the syntax and semantics of first-order logic over a relational signature Σ.

2.1. Definition. A *relational signature* Σ is an indexed family of disjoint sets

$$\Sigma = \langle\ \Sigma_n,\ \Sigma_m^{'}\ \mid\ n \in \mathbf{N},\ m \in \mathbf{N}^+\ \rangle.$$

Each element $c \in \Sigma_0$ is a *constant symbol*. For any $n \in \mathbf{N}^+$ each element $f \in \Sigma_n$ is a *function symbol* of arity n, and each element $r \in \Sigma_n^{'}$ is a *relation symbol* of arity n.

The semantics of a relational signature Σ is given by a Σ-structure.

2.2. Definition. Let Σ be a relational signature. By a Σ *structure* we mean pair $A = (A,\ \Sigma^A)$, consisting of a non-empty set A termed the *domain* of A, and an indexed collection

$$\Sigma^A = \langle\ \Sigma_n^A,\ \Sigma_m^{'\ A}\ \mid\ n \in \mathbf{N},\ m \in \mathbf{N}^+\ \rangle$$

of families of constants, functions and relations over A. Thus $\Sigma_0^A = \langle c_A \mid c \in \Sigma_0 \rangle$, where $c_A \in A$ is a *constant* that interprets c in A. For each $n \in \mathbf{N}^+$, $\Sigma_n^A = \langle f_A \mid f \in \Sigma_n \rangle$, where $f_A \colon A^n \to A$ is an n-ary *function* which interprets f. Also $\Sigma_n^{'\ A} = \langle r_A \mid r \in \Sigma_n^{'} \rangle$, where $r_A \subseteq A^n$ is an n-ary *relation* which interprets r. We let $Mod(\Sigma)$ denote the class of all Σ structures.

Usually we are interested in computation, correctness and testing over one particular Σ structure A that we consider to be the standard semantics of Σ. For example A may be the ring $\mathbb{Z}$ of integers or the ring $\mathbb{Q}$ of rationals.

Recall the definition of first-order *terms* and *formulas*.

2.3. Definition. Let Σ be a relational signature, and let X be a set of variable symbols disjoint from Σ_0. We define the set $T(\Sigma,\ X)$ of all *terms* over Σ and X, inductively. Each constant symbol $c \in \Sigma_0$ is a term $c \in T(\Sigma,\ X)$. Each variable symbol $x \in X$ is a term $x \in T(\Sigma,\ X)$. For any $n \geq 1$, if $t_i \in T(\Sigma,\ X)$ is a term for $i = 1,\ \ldots,\ n$ and $f \in \Sigma_n$ is an n-ary function symbol then $f(t_1,\ \ldots,\ t_n) \in T(\Sigma,\ X)$ is a term. We let $T(\Sigma) = T(\Sigma,\ \emptyset)$ denote the set of all variable free or *ground terms*.

If a term t contains the variables $x_1,\ \ldots,\ x_n$ then we may write $t[x_1,\ \ldots,\ x_n]$. If $\alpha : X \to A$ is any assignment then $\overline{\alpha} : T(\Sigma,\ X) \to A$ denotes the *term evaluation mapping*.

2.4. Definition. Let Σ be a relational signature, and let X be a set of variable symbols which is disjoint from Σ_0. An *atomic formula* over Σ and X is a formula of the form

$$r(\ t_1,\ \ldots,\ t_n\ )$$

where, for any $n \geq 1$ and for $i = 1,\ \ldots,\ n$, $t_i \in T(\Sigma,\ X)$ is a term and $r \in \Sigma_n^{'}$ is an n-ary relation symbol. The set $L(\Sigma,\ X)$ of all *first-order formulas* over Σ and X is defined to be the smallest set containing all atomic formulas over Σ and X which is closed under the propositional connectives $\wedge,\ \vee,\ \rightarrow,\ \neg$ and the quantifiers $\forall,\ \exists$.

Recall the distinction between free and bound variables in a formula ϕ. If ϕ contains the free variables $x_1,\ \ldots,\ x_n$ we may write $\phi[x_1,\ \ldots,\ x_n]$ to indicate this.

Given a relational structure $A \in Mod(\Sigma)$ and any assignment $\alpha : X \to A$, we write $A,\ \alpha \models \phi$ if ϕ is true in A under α. We write $A \models \phi$ if $A,\ \alpha \models \phi$ for every $\alpha : X \to A$, i.e. ϕ is *valid* in A.

Next we recall how first-order formulas are used to define pre and postconditions for a program S within the frame-

work of the theory of program correctness. This is quite straightforward, except that we will need to distinguish between the values of program variables before execution and after termination of S.

2.5. Definition. Let X be a set of variable symbols. We define the set of variable symbols $X' = \{\ x' \mid x \in X\ \}$ so that X' is disjoint from X. A variable $x \in X$ is termed a *prevariable* while the variable $x' \in X'$ is termed the corresponding *postvariable*.

If $\alpha : \{\ x_1, \ \ldots, \ x_n\ \} \to A$ is any prevariable assignment, then α induces a postvariable assignment

$$\alpha' : \{\ x'_1, \ \ldots, \ x'_n\ \} \to A$$

defined by $\alpha'(x') = \alpha(x)$.

A *precondition* over Σ and X is a formula $\phi \in L(\Sigma,\ X)$ containing only prevariables. A *postcondition* over Σ and X is a formula $\phi \in L(\Sigma,\ X \cup X')$ which may contain both pre and postvariables.

In order to formally define the semantics of pre and postconditions, we require a semantics for programs. We let $Prog(\Sigma,\ X)$ denote an arbitrary programming language which has a signature Σ and set X of variables as an interface. By this we mean that in any semantics for $Prog(\Sigma,\ X)$ the variables of X serve as program input/output variables, and range over values from some Σ structure A, which is the underlying semantics of Σ as a computational data type. Taking a black-box view of testing, we will deliberately ignore the syntactic structure and operational semantics of $Prog(\Sigma,\ X)$. If $S \in Prog(\Sigma,\ X)$ has the interface

$$\{\ x_1, \ \ldots, \ x_n\ \} \subseteq X$$

then consistent with our notation for terms and formulas, we will indicate this by writing $S[x_1, \ \ldots, \ x_n]$.

Every program $S[x_1, \ \ldots, \ x_n] \in Prog(\Sigma,\ X)$ is assumed to have a simple *deterministic transformational action* on its inputs. Thus given an initial assignment

$$\alpha : \{\ x_1, \ \ldots, \ x_n\ \} \to A,$$

if S terminates on the input α, then the output of S is assumed to be recovered from among the final values assigned to $x_1, \ \ldots, \ x_n$ after termination, and is uniquely determined by α. This assumption includes most basic forms of software, but precludes many kinds of real-time and reactive systems which may not terminate.

2.6. Definition. Let $Prog(\Sigma,\ X)$ be a programming language with a relational signature Σ and variable set X as interface. By a *semantic mapping* for $Prog(\Sigma,\ X)$ with respect to a Σ structure A we mean a mapping of $Prog(\Sigma,\ X)$ programs into partial state transition functions

$$[\![\ .\ ]\!] : Prog(\Sigma,\ X) \to [\ [X \to A] \rightsquigarrow [X \to A]\ ].$$

For any program $S \in Prog(\Sigma,\ X)$, $[\![\ S\ ]\!] : [X \to A] \rightsquigarrow [X \to A]$ is a partial function from intial states to final states. Intuitively, for any initial state, $\alpha : X \to A$, either S fails to terminate on α, and $[\![\ S\ ]\!](\alpha)$ is undefined (in which case we write $[\![\ S\ ]\!](\alpha) \uparrow$) or else S terminates, and $[\![\ S\ ]\!](\alpha) : X \to A$ is the state of the variables of S after termination. In the latter case we write $[\![\ S\ ]\!](\alpha) \downarrow = \beta$ to indicate that the l.h.s. is defined and equal to β.

By an abuse of notation we may also use $[\![\ S[x_1, \ \ldots, \ x_n]\ ]\!]$ to denote an n-ary partial function, where it is understood that

$$[\![\ S\ ]\!](a_1, \ \ldots, \ a_n) = [\![\ S\ ]\!](\overline{a})$$

and for each $s \in S$ and $x \in X_s$, $\overline{a} : X \to A$ is given by $\overline{a}(x) = a_n$ if $x = x_n$, otherwise $\overline{a}(x) = a_s$ for some fixed, arbitrary $a_s \in A_s$.

Let us collect together the definitions introduced so far to formalise the concept of test success and failure.

2.7. Definition. Let Σ be a relational signature, X be any set of variable symbols, and $Prog(\Sigma,\ X)$ be any programming language with Σ and X as an interface. Let $A \in Mod(\Sigma)$ be any Σ structure.

(i) A *functional specification* $\{p\}S\{q\}$ is a triple consisting of a precondition $p \in L(\Sigma,\ X)$, a program $S \in Prog(\Sigma,\ X)$ and a postcondition $q \in L(\Sigma,\ X \cup X')$.

(ii) A functional specification $\{p\}S\{q\}$ is said to be *valid or correct in A under partial correctness semantics* if, for every assignment $\alpha : X \to A$, if $A,\ \alpha \models p$ and $[\![\ S\ ]\!](\alpha) \downarrow = \beta$ for some $\beta : X \to A$ then $A,\ \alpha \cup \beta' \models\ q$. If $\{p\}S\{q\}$ is valid in A under partial correctness semantics we write

$$A \models \{p\}S\{q\}.$$

(iii) Let $S[x_1, \ \ldots, \ x_n] \in Prog(\Sigma,\ X)$ be any program. Let $p \in L(\Sigma,\ X)$ be a precondition and $q \in L(\Sigma,\ X \cup X')$ be a postcondition. For any $\alpha : \{\ x_1, \ \ldots, \ x_n\ \} \to A$ we say that S *fails the test* α of $\{p\}S\{q\}$ if $A,\ \alpha \models p$ and there exists $\beta : \{\ x_1, \ \ldots, \ x_n\ \} \to A$ such that

$$[\![\ S\ ]\!](\alpha) \downarrow = \beta$$

and $A,\ \alpha \cup \beta' \not\models\ q$. We say that S *passes* the test α of $\{p\}S\{q\}$ if S does not fail α.

The following result is obvious, but concisely summarises the relationship between program testing and program correctness.

2.8. Proposition. Let Σ be a relational signature, X be a set of variables and $Prog(\Sigma,\ X)$ be a programming language with Σ and X as interface. Let $A \in Mod(\Sigma)$ be any Σ structure. Given a program $S[x_1, \ \ldots, \ x_n] \in Prog(\Sigma,\ X)$, a precondition $p \in L(\Sigma,\ X)$, and a postcondition $q \in L(\Sigma,\ X \cup X')$, then

$$A \not\models \{p\}S\{q\}.$$

if, and only if, S fails some test $\alpha : \{\ x_1, \ \ldots, \ x_n\ \} \to A$ of $\{p\}S\{q\}$.

Proof. Immediate from Definitions 2.7.(ii) and 2.7.(iii).

Thus we can say abstractly that program testing consists of the search for counterexamples to program correctness. In the case that S fails to terminate on an input α, a search algorithm for successful tests would potentially loop forever executing S. However, we can include a non-functional performance requirement on S, that S must terminate within some time bounded by a function $f(\alpha)$. When this time bound is exceeded we can flag a failure of a performance requirement, rather than a functional requirement.

3. APPROXIMATION-BASED TESTING.

As we have seen in Section 1, the testing problem for a software system $S[x_1, \ldots, x_n]$ with respect to a functional specification $\{pre\}S\{post\}$ can be viewed as a search or satisfiability problem to find a prevariable assignment $\alpha : \{ x_1, \ldots, x_n \} \to A$ for S such that S terminates and the resulting postvariable assignment $\omega : \{ x'_1, \ldots, x'_n \} \to A$ obtained after termination (possibly together with α) satisfies $pre \wedge \neg post$. In this section we will show how methods of function approximation can be used to solve this problem, while also providing information about the degree of coverage obtained after n executed tests.

3.1 Overview of the Approach.

We take a bottom up approach to satisfying a formula $\Phi = pre \wedge \neg post$. This means that we begin by analysing the literals of the form $r(t_1, ..., t_n)$ or $\neg r(t_1, ..., t_n)$ at all leaves of the parse tree for Φ. Here r is a relation symbol from the data type signature Σ and $t_1, ..., t_n$ are terms over Σ and sets X, X' of pre and postvariables. The satisfiability problem for r or its complement $\neg r$, is entirely determined by the semantics of the data type $A \in Mod(\Sigma)$, over which we compute. In this report, for concreteness, we focus on the basic arithmetic relations $<$, $\leq$ and $=$ over the signature Σ^{ring} of rings and the standard interpretation of this signature over the rings $\mathbb{Z}$ and $\mathbb{R}$ of integers and reals respectively. To simplify our account, we will ignore the discrepancy between $\mathbb{R}$ and an actual floating point implementation, as well as problems of over and underflow for both arithmetic systems. Such issues, while leading to errors, fall outside the scope of the functional correctness statements we have in mind. We will deal with $\mathbb{Z}$ by embedding the integers into $\mathbb{R}$ and retracting back possible satisfying solutions to $\mathbb{Z}$ in an appropriate way. This gives us access to many powerful approximation methods from analysis for both the integer and floating point data types.

To illustrate our approach consider, within the overall context of testing S, the problem of trying to satisfy a single literal such as

$$t_1 \leq t_2,$$

with assignments $\alpha : \{ x_1, \ldots, x_n \} \to \mathbb{Z}$ and $\omega : \{ x'_1, \ldots, x'_n \} \to \mathbb{Z}$. Recall from Section 2 that ω is entirely determined by α and the semantics of S, so that we are in fact simply searching for a satisfying assignment α. Now in practical testing applications, n may easily be rather large, and so we must begin by finding a way to control the dimensionality of the search space for α.

A natural approach is to restrict attention to subsets of the prevariables

$$Y = \{ x_{i(1)}, ..., x_{i(d)} \} \subseteq \{ x_1, ..., x_n \},$$

for successively increasing values $d = 1, 2, ..., n$. For a specific choice of search variables Y, we can arbitrarily choose an assignment to its complement $\overline{Y} = \{ x_1, ..., x_n \} - Y$ say $\beta : \overline{Y} \to \mathbb{Z}$ which is consistent with pre. We can then search the reduced d-dimensional subspace for a satisfying assignment $\alpha : Y \to \mathbb{Z}$. That is to say, the union $\alpha \cup \beta : \{ x_1, ..., x_n \} \to \mathbb{Z}$ should satisfy the formula $t_1 \leq t_2$.

For $d = 1$, this approach is similar to *orthogonal array testing* in the literature, e.g. [10]. Both approaches are motivated by the hypothesis that successful tests will often be found for quite low values of d, e.g. $d = 1, 2$, or 3 given the correct choice of Y.

With the literal $t_1 \leq t_2$ we can associate an n-ary partial function $U(t_1,\ t_2) : \mathbb{Z}^n \rightsquigarrow \mathbb{Z}$ given by

$$U(t_1,\ t_2) = \lambda\ x_1,\ ...,\ x_n\ .\ \overline{\omega}(t_1)[x_1, ..., x_n] -$$

$$\overline{\omega}(t_2)[x_1, ..., x_n],$$

where

$$\omega = [\![\ S\ ]\!](x_1, ..., x_n),$$

and $\overline{\omega}$ is the resulting term evaluation mapping. If postvariables occur in t_1 or t_2, then due to the black box nature of the SUT S, the exact relationship between a prevariable assignment α to $\{ x_1, \ldots, x_n \}$ and the resulting postvariable assignment ω to $\{ x'_1, \ldots, x'_n \}$ remains unknown. Hence, even though we know the structure of t_1 and t_2, the behaviour of $U(t_1,\ t_2)$ remains unknown. Instantiating the variables of $\overline{Y}$ using the assignment β we obtain a d-dimensional section of this function $U(t_1,\ t_2,\ \beta) : \mathbb{Z}^d \rightsquigarrow \mathbb{Z}$ given by

$$U(t_1,\ t_2,\ \beta) =$$

$$\lambda\ x_{i(1)},\ ...,\ x_{i(d)}\ .\ \overline{\omega}(\ \overline{\beta}(t_1)[x_{i(1)},\ ...,\ x_{i(d)}]\ ) -$$

$$\overline{\omega}(\ \overline{\beta}(t_2)[x_{i(1)},\ ...,\ x_{i(d)}]\ ),$$

where

$$\omega = [\![\ S\ ]\!](\ \overline{\beta}(x_1),\ \ldots,\ \overline{\beta}(x_n)\ ).$$

whose behaviour is therefore also unknown. However, all zeros of $U(t_1,\ t_2,\ \beta)$ are associated with solutions to the original constraint $t_1 \leq t_2$, and these are bounded by $d-1$ dimensional surfaces. Now after n prevariable assignments

$$\alpha_1 \cup \beta,\ \ldots,\ \alpha_n \cup \beta : \{ x_1,\ \ldots,\ x_n \} \to \mathbb{Z}$$

have been executed as tests on S, with termination of S in each case, we have n postvariable assignments

$$\omega_1,\ \ldots,\ \omega_n : \{ x'_1,\ \ldots,\ x'_n \} \to \mathbb{Z}$$

and n assignments to the search prevariables $\alpha_1, ..., \alpha_n : Y \to \mathbb{Z}$ from which n values $v_1,\ \ldots,\ v_n$ of $U(t_1,\ t_2,\ \beta)$ can be calculated, namely

$$v_j = U(t_1,\ t_2,\ \beta)(\ \alpha_j(x_{i(1)}),\ \ldots,\ \alpha_j(x_{i(d)})\ )\ ,\ 1 \leq j \leq n.$$

We can use the points $v_1,\ \ldots,\ v_n$ to construct a function $f_n : \mathbb{R}^d \to \mathbb{R}$ which approximates the unknown partial function $U(t_1,\ t_2,\ \beta) : \mathbb{Z}^d \rightsquigarrow \mathbb{Z}$ in an appropriate approximation space. (This simultaneously embeds the problem into $\mathbb{R}$.) The zeros of the known function f_n may then provide estimates of the zeros of the unknown function $U(t_1,\ t_2,\ \beta)$. These zeros of f_n may be identified either numerically or algebraically from the definition of f_n. Any zero point for f_n can then be chosen as the next prevariable assignment $\alpha_{n+1} : Y \to \mathbb{Z}$, to test S by suitable retraction back into $\mathbb{Z}$ (for example by rounding). The SUT S is tested using the combined prevariable assignment $\alpha_{n+1} \cup \beta : \{ x_1,\ \ldots,\ x_n \} \to \mathbb{Z}$. If S terminates on this input, then from the resulting assignment to postvariables, after the termination of S, we can calculate the value

$$v_{n+1} = U(t_1,\ t_2,\ \beta)(\ \alpha_{n+1}(x_{i(1)}),\ \ldots,\ \alpha_{n+1}(x_{i(d)})\ )$$

and can observe whether v_{n+1} is indeed equal to zero. If this is the case then $\alpha_{n+1} \cup \beta$ was a successful test for S.

However, even if v_{n+1} is not equal to zero, v_{n+1} can be used to refine the approximation f_n to a better approximation f_{n+1} of $U(t_1,\ t_2,\ \beta)$ which might yield more accurate estimates of the zero points of $U(t_1,\ t_2,\ \beta)$. If the sequence of approximations $f_n,\ f_{n+1},\ \ldots$ converges to $U(t_1,\ t_2,\ \beta)$, then the probability of finding an error in S, if one exists, must gradually increase.

By choosing α_0 at random, and each α_{n+1} from f_n, we obtain an iterative algorithm for generating test cases for S with respect to the testing condition $t_1 \leq t_2$. Furthermore, observations about the convergence of the sequence $f_1,\ f_2,\ \ldots$ within the chosen approximation space give us insight into the degree of coverage obtained.

The arithmetic relations $=$ and $<$ can be handled in a similar fashion to $\leq$, in terms of zero points or zero crossings respectively.

Obviously, a central question in our whole approach is the choice of appropriate approximation spaces for the f_i, and the efficient representation and manipulation of such functions. These questions need to be investigated both theoretically and empirically. Some important details are missing from this brief account of the approximation method, but these will be added in Sections 3.2 and 3.3.

3.2 Piecewise Polynomial Approximation.

The well known Weierstrass Approximation Theorem (see for example [4]) establishes that any continuous function $f : \mathbb{R} \to \mathbb{R}$ over a closed interval $[m, n]$ can be approximated by a sequence $p_1,\ p_2,\ \ldots : \mathbb{R} \to \mathbb{R}$ of polynomial functions, i.e. f is the limit of the p_i in an appropriate metric topology on real valued continuous functions. The ease with which polynomials can be represented, evaluated, integrated, differentiated and solved algebraically (up to quartics) makes them an obvious candidate for experimentation with the approach described in 3.1. However, since discontinuity is commonplace in computable functions, we should immediately generalise the approach to piecewise polynomial approximation.

For simplicity, we will reconsider the situation of 3.1, under the assumption that $d = 1$ and the variable subset Y is a singleton, $Y = \{\ x_\lambda\ \}$, for some $1 \leq \lambda \leq n$. This allows us to efficiently solve the roots of any polynomial p in a single variable up to fourth degree in a purely algebraic fashion.

An efficient implicit method of representing a piecewise polynomial function $p : \mathbb{R} \to \mathbb{R}$ is by using *divided differences*. Using this approach, we can avoid explicitly calculating the coefficients of any polynomial piece until they are needed. The representation method can be described as follows.

Assume that we wish to approximate $U(t_1,\ t_2,\ \beta)$ using polynomial pieces of degree at most k. As was observed in Section 4.1, we can assume that after executing n test intputs

$$\alpha_1 \cup \beta,\ \ldots,\ \alpha_n \cup \beta : \{\ x_1,\ \ldots,\ x_n\ \} \to \mathbb{Z}$$

on S, with termination of S in each case, we know n values

$$v_j = U(t_1,\ t_2,\ \beta)(\ \alpha_j(x_\lambda)\ )\ ,\ 1 \leq j \leq n$$

for the unknown function $U(t_1,\ t_2,\ \beta)$. Now let

$$r : \{\ 1,\ \ldots n\ \} \to \{\ 1,\ \ldots n\ \}$$

be a permutation which ranks the values $\alpha_j(x_\lambda)$ into strictly ascending order, i.e.

$$\alpha_{r(1)}(x_\lambda) < \alpha_{r(2)}(x_\lambda) < \ldots < \alpha_{r(n)}(x_\lambda)$$

(We assume that we do not test the same input value twice!) For each $1 \leq j \leq n$ we define a $k+1$-tuple

$$\Delta^j = (\ \Delta^j_1,\ \ldots,\ \Delta^j_{k+1}\ ) \in \mathbb{R}^{k+1}$$

of divided differences defined by

$$\Delta^j_1 = v_{r(j+1)} - v_{r(j)}\ /\ \alpha_{r(j+1)}(x_\lambda) - \alpha_{r(j)}(x_\lambda)$$

and for $1 \leq m \leq k$,

$$\Delta^j_{m+1} = \Delta^{j+1}_m - \Delta^j_m\ /\ \alpha_{r(j+m+1)}(x_\lambda) - \alpha_{r(j)}(x_\lambda).$$

Obviously, in the upper end region for $n-k \leq j \leq n$ certain values of Δ^j_i are undefined, and these are simply marked and then ignored in a concrete data structure.

Now for each $1 \leq j \leq n$ and for each $0 \leq i \leq k$, we can associate a polynomial piece $p^j_i : \mathbb{R} \to \mathbb{R}$ of degree i,

$$p^j_i(y) = a^{(j,\ i)}_i y^i + a^{(j,\ i-1)}_{i-1} y^{i-1} + \ldots + a^{(j,\ i)}_0$$

with the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i)}(x_\lambda)]$. For example: when $i = 0$, p^j_0 is a constant polynomial with

$$a^{(j,\ 0)}_0 = \Delta^j_0;$$

when $i = 1$ then p^j_1 is a linear polynomial with

$$a^{(j,\ 1)}_1 = \Delta^j_1,$$

and

$$a^{(j,\ 2)}_0 = v_{r(j)} - (a^{(j,\ 1)}_1 * \alpha_{r(j)}(x_\lambda));$$

when $i = 2$ then p^j_2 is a quadratic polynomial with

$$a^{(j,\ 2)}_2 = \Delta^j_2,$$

$$a^{(j,\ 2)}_1 = \Delta^j_1 - (a^{(j,\ 2)}_2 * (\alpha_{r(j)}(x_\lambda) + (\alpha_{r(j+1)}(x_\lambda))),$$

$$a^{(j,\ 2)}_0 = v_{r(j)} - ((a^{(j,\ 2)}_1 + (a^{(j,\ 2)}_2 * \alpha_{r(j)}(x_\lambda))) * \alpha_{r(j)}(x_\lambda)).$$

We leave the reader to generalise this definition to higher orders as a useful exercise. Considering the i-th degree polynomial p^{j+1}_i obtained over the shifted interval

$$[\alpha_{r(j+1)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$$

then if Δ^j_{i+1} is close to zero then p^i_j is close to p^{j+1}_i in the function space, i.e. as Δ^j_{i+1} converges to zero then p^j_i converges to p^j_{i+1}. Thus Δ^j_{i+1} is a measure of the "goodness of fit" of p^j_i to $U(t_1,\ t_2,\ \beta)$ over the extended interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$.

We must initialise the divided difference model by constructing an initial $k+1$ length list

$$\Delta = \Delta_1,\ \ldots,\ \Delta_{k+1}$$

of $k+1$-tuples. For this we require at least $k+1$ initial pre and postvariable assignments to calculate values of $U(t_1,\ t_2,\ \beta)$ from $k+1$ tests of S. It is appropriate to choose $\alpha_{r(1)}(x_\lambda)$ and $\alpha_{r(k+1)}(x_\lambda)$ to be maximum and minimum points over the desired range of testing. These might for instance correspond to the maximum positive and minimum negative machine representable integers. The remaining initialisation points $\alpha_{r(2)}(x_\lambda),\ \ldots,\ \alpha_{r(k)}(x_\lambda)$ might then be

evenly spaced over this interval. Another possibility is to distribute $\alpha_{r(1)}(x_\lambda),\ \ldots,\ \alpha_{r(k+1)}(x_\lambda)$ over the input space according to some expected probability distribution of inputs that reflects some hypothesis about usage patterns of the SUT S. Though since k is typically small there are better ways to integrate such probability distributions. This subject is beyond the scope of the present paper however.

In general, after executing n additional tests of S, we obtain a list $\Delta(n)$ of length $n+k+1$

$$\Delta(n) = \Delta(n)_1,\ \Delta(n)_2,\ \ldots,\ \Delta(n)_{n+k+1}$$

where for $1 \leq j \leq (n+k+1)$, $\Delta(n)_{r(j)}$ is the $k+1$-tuple of divided differences associated with the input/output pair $(\alpha_{r(j)},\ v_{r(j)})$. Thus in the case that $d=1$, the model size grows in a tractable linear fashion with the number of executed tests.

Suppose now that after initialisation and executing n additional tests, we have still not obtained a succesful test of S and wish to choose a new prevariable assignment $\alpha_{n+1} : \{\ x_\lambda\ \} \to \mathbb{Z}$ in order to construct one further test $\alpha_{n+1} \cup \beta$. Surveying the whole list $\Delta(n)$ of differences (though see the above remark about upper endpoints of $\Delta(n)$), we effectively observe $k.(n+k+1)$ polynomial pieces of degrees between 0 and k, together with, for each such polynomial piece p_i^j the measure Δ_{i+1}^j of its goodness of fit locally. Thus if we consider the global maxima and minima of all difference values,

$$Min(n) = min\{\ \Delta(n)_i^j : 1 \leq j \leq n\ \ and\ \ 1 \leq i \leq k+1\ \}$$

$$Max(n) = max\{\ \Delta(n)_i^j : 1 \leq j \leq n\ \ and\ \ 1 \leq i \leq k+1\ \}$$

then $Min(n)$ is associated with a polynomial piece of best fit, while $Max(n)$ is associated with a polynomial piece of worst fit. Typically $Max(n)$ arises as a difference over an input interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+1)}(x_\lambda)]$ containing a substantial discontinuity. If test points must be added closer and closer to this discontinuity then we will observe certain difference values diverging to $\pm\infty$ and this must be dealt with in a practical implementation, e.g. by introducing a maximum resolution between consecutive input values and some exception handling.

Considering $Min(n)$, associated say with the difference $\Delta(n)_{i+1}^j$ we may calculate the coefficients of the corresponding ith degree polynomial piece p_i^j. We consider zero points of p_i^j within the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$ only. This interval represents the domain of validity of the approximating polynomial piece p_i^j. Outside this interval, solutions to p_i^j would be an extrapolation of this function to an interval where another polynomial piece might provide more accurate information. There will be at most i solutions to p_i^j within the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$ and any one (or all in succession) of these values can be chosen for the next test input α_{n+1}.

Independently of whether solutions occur within $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$, we need to decide on an acceptance criterion for the accuracy of the approximation p_i^j. It is probably pointless to keep testing S over the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$ if p_i^j is an accurate approximation to $U(t_1,\ t_2,\ \beta)$ over this interval and p_i^j has no zeros within the interval. In the case that p_i^j does contain zeros then these predictions can be used to test the accuracy of p_i^j. However, if p_i^j contains no zeros within the interval then we must choose at least one further test assignment $\alpha_{n+1} : \{\ x_\lambda\ \} \to \mathbb{Z}$ in the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$ and compare the predictions made using p_i^j for $U(t_1,\ t_2,\ \beta)$. In this case we may choose $\alpha_{n+1}(x_\lambda)$ randomly within the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$ or as a maximum or minimum point in this interval, or by some other strategy. Then we make a prediction for the value of v_{n+1} according to p_i^j, since this is no longer expected to be zero.

If the predicted and observed values of v_{n+1} agree to within some specified tolerance then we can with some confidence reject this interval as a region for further testing.

After executing the generated test input $\alpha_{n+1} \cup \beta$ on the SUT S and obtaining a new output value $v_{n+1} = U(t_1,\ t_2,\ \beta)(\ \alpha_{n+1}(x_\lambda)\ )$. We must check that: (a) v_{n+1} actually is equal to zero (if this was predicted), and (b) v_{n+1} is close enough (to within some tolerance value) to its predicted value using p_i^j, in order to accept the accuracy of approximation of p_i^j.

In the case that p_i^j was sufficiently accurate, we mark the interval

$$[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$$

as adequately modelled or approximated, and no further tests are chosen from this interval. In the case that p_i^j was not sufficiently accurate, we reconstruct a refined approximation model $\Delta(n+1)$ containing the new and old test results. This simply involves computing the new permutation $r : \{\ 1,\ \ldots,\ n+1\ \} \to \{\ 1,\ \ldots,\ n+1\ \}$ such that

$$\alpha_{r(1)}(x_\lambda) < \alpha_{r(2)}(x_\lambda) < \ldots < \alpha_{r(n+1)}(x_\lambda)$$

and re-calculating the divided difference vectors

$$\Delta(n+1)_{r(n+1)-1},\ \ldots,\ \Delta(n+1)_{r(n+1)-k}$$

locally. Then we have completed one iteration of the test procedure.

The case where $d \geq 2$ is more complex. In general we obtain a d-dimensional matrix of differences. To solve for zeros algebraically, we can for example, decompose d-ary polynomials into their 1-dimensional sections. The details are beyond the scope of this paper and will be described elsewhere.

3.3 Solving Boolean Combinations of Literals.

A practically useful specification language first emerges when we can combine literals over a data type using Boolean operations. How does the approximation method of Section 3.2 generalise to this case?

We will assume that in the system specification

$$\{pre\}S\{post\},$$

the precondition *pre* and postcondition *post* are quantifier free formulas. We begin by rewriting the corresponding testing condition $pre \wedge \neg post$ into disjunctive normal form (DNF). After this we rewrite all negative literals of the form

$$(a)\ \neg(t_1 \leq t_2),\quad (b)\ \neg(t_1 < t_2),\quad (c)\ \neg(t_1 = t_2)$$

as positive formulas of the form

$$(a')\ t_2 < t_1,\quad (b')\ t_2 \leq t_1,\quad (c')\ (t_1 < t_2) \vee (t_2 < t_1)$$

respectively. Notice that the disjunction (c') may disturb the DNF property in which case we must renormalise every conjunction in which it is introduced. Of course DNF normalisation may expand the formula size considerably, but

the complexity of our method will be affected by the number of disjuncts and not the total number of literals.

Let *TSpec* denote the resulting DNF normalised form of $pre \wedge \neg post$ in which all literals are positive. (In general it does not seem essential for the approximation method that all literals can be converted to positive form. But in this case study of the arithmetic data types it is clearly convenient.) Then *TSpec* has the form

$$TSpec = \phi_1 \vee \phi_2 \vee \ldots \vee \phi_l$$

where for $1 \leq i \leq l$, the disjunct ϕ_i is a conjunction of positive binary literals,

$$\phi_i = r_1^i(t_{(1,1)}^i,\ t_{(1,2)}^i)\ \wedge \ldots \wedge\ r_{l(i)}^i(t_{(l(i),1)}^i,\ t_{(l(i),2)}^i).$$

Now in principle we can analyse the satisfiability of each disjunct ϕ_i independently, and in parallel. Then for a particular disjunct ϕ_i we can analyse the satisfiability of each binary literal $r_j^i(t_{(j,1)}^i,\ t_{(j,2)}^i)$ for $1 \leq j \leq l(i)$ in sequence, using the approximation method described in Section 3.2. Note that if $\{\ x_1,\ \ldots,\ x_n\ \}$ represents the set of all prevariables occurring in *TSpec* then we choose

$$Y \subseteq \{\ x_1,\ \ldots,\ x_n\ \}$$

and its complement $\overline{Y}$ globally for all literals. We then choose a single global assignment $\beta : \overline{Y} \to \mathbb{Z}$ for investigating satisfiability of all literals.

However, the above naive approach is wasteful in a number of ways. Firstly, we must regard every execution of a test of the SUT S as costly in terms of computation time. The computational overhead of building approximation models and generating the next test case will typically be a small proportion of the total time spent executing tests on S, when S is a large system. Therefore, we must make the best use of the information gleaned from each test execution, and this requires updating the approximation models for every binary literal in every conjunct, simultaneously. In turn this requires that the approximation models for all literals are stored simultaneously!

Secondly, the number and size of data structures for approximation models (in this particular case, lists or arrays of divided differences) grows rather quickly, especially when $d > 1$ and this may be exacerbated by the conversion to DNF. Fortunately, in the arithmetic case that we consider, we can conflate the $l(i)$ divided difference structures required for the i-th conjunct, into just three divided difference structures. Thus, in this case, the method can be made linear in the number of disjuncts.

Consider again satisfiability testing of the i-th disjunct ϕ_i. Now ϕ_i is equivalent to a conjunction of three formulas, namely:

(i) a conjunction of all its inequalities

$$LE_i = t_{(j(1),1)}^i \leq t_{(j(1),2)}^i\ \wedge \ldots \wedge\ t_{(j(le_i),1)}^i \leq t_{(j(le_i),2)}^i,$$

(ii) a conjunction of all its strict inequalities

$$L_i = t_{(j'(1),1)}^i < t_{(j'(1),2)}^i\ \wedge \ldots \wedge\ t_{(j'(l_i),1)}^i < t_{(j'(l_i),2)}^i,$$

and

(iii) a conjunction of all its equations

$$E_i = t_{(j''(1),1)}^i = t_{(j''(1),2)}^i\ \wedge \ldots \wedge\ t_{(j''(e_i),1)}^i = t_{(j''(e_i),2)}^i.$$

So

$$\phi_i = LE_i \wedge L_i \wedge E_i.$$

Consider for example the conjunction LE_i of inequalities. Recalling their definition from Section 3.2, we can conflate the unknown functions

$$U(\ t_{(j(1),1)}^i,\ t_{(j(1),2)}^i,\ \beta\ ),\ \ldots,\ U(\ t_{(j(le_i),1)}^i,\ t_{(j(le_i),2)}^i,\ \beta\ )$$

into a single function $U(LE,\ i,\ \beta)$ defined by

$$U(LE,\ i,\ \beta)(x_{i(1)},\ \ldots,\ x_{i(d)}) = sup_{k \in \{\ 1,\ \ldots,\ le_i\ \}}($$

$$U(\ t_{(j(k),1)}^i,\ t_{(j(k),2)}^i,\ \beta\ )(x_{i(1)},\ \ldots,\ x_{i(d)})\ ).$$

Then zero points of $U(LE,\ i,\ \beta)$ correspond precisely to regions satisfying all inequalities of LE_i simultaneously. Similarly for L_i we construct a single conflated function

$$U(L,\ i,\ \beta)(x_{i(1)},\ \ldots,\ x_{i(d)}) = sup_{k \in \{\ 1,\ \ldots,\ l_i\ \}}($$

$$U(\ t_{(j'(k),1)}^i,\ t_{(j'(k),2)}^i,\ \beta\ )(x_{i(1)},\ \ldots,\ x_{i(d)})\ ).$$

Finally for E_i we construct a conflated function,

$$U(E,\ i,\ \beta)(x_{i(1)},\ \ldots,\ x_{i(d)}) = sup_{k \in \{\ 1,\ \ldots,\ e_i\ \}}($$

$$|\ U(\ t_{(j''(k),1)}^i,\ t_{(j''(k),2)}^i,\ \beta\ )(x_{i(1)},\ \ldots,\ x_{i(d)})\ |\ ).$$

Notice the use of absolute values of the underlying unknown functions in this last case.

A zero point of $U(LE,\ i,\ \beta)$ or $U(E,\ i,\ \beta)$, or a zero crossing of $U(L,\ i,\ \beta)$, corresponds to a solution of the inequalities, strict inequalities and equations of the disjunct ϕ_i, and these can by analysed independently. Each solution region for the inequalities LE_i can then be analysed for solutions to the strict inequalities L_i, and a solution region for both of these can finally be investigated for solutions to the equations E_i. Furthermore, only three divided difference arrays need be kept in store for each disjunct ϕ_i, one for the approximation of each conflated function. To improve their approximation accuracy, the three approximations associated with every other disjunct ϕ_j can also be updated after the execution of the SUT S on each new test case $\alpha_n \cup \beta$.

How should we choose which disjunct ϕ_i to analyse for satisfiability on any particular iteration, and how should we decide when to terminate this iterative testing process? The key to the solution can be found in the convergence properties of the approximations built by our method. If *TSpec* has l disjuncts, each consisting of a conjunction of inequalities, strict inequalities and equations, then after n tests have been executed we have $3.l$ lists of divided differences,

$$\Delta(n)_1^{LE},\ \Delta(n)_1^{L},\ \Delta(n)_1^{E}$$

$$\ldots$$

$$\Delta(n)_l^{LE},\ \Delta(n)_l^{L},\ \Delta(n)_l^{E}$$

for the inequalities, strict inequalties and equations respectively of *TSpec*.

There exist a variety of ways to measure convergence. One simple measure is as follows. Recall from Section 3.2 that a simple fixed tolerance value is used to accept the accuracy of approximation of a particular polynomial piece p_i^j. Once this value is reached, the interval $[\alpha_{r(j)}(x_\lambda),\ \alpha_{r(j+i+1)}(x_\lambda)]$, covered by the piece p_i^j is marked as adequately approximated and no further test values are chosen from the domain covered by this piece. Let $Coverage(n)_i^{LE}$ (respectively

$Coverage(n)_i^L$ and $Coverage(n)_i^E$) be the integral of the intervals covered by all polynomial pieces p_i^j in $\Delta(n)_1^{LE}$ (respectively in $\Delta(n)_1^L$ and $\Delta(n)_1^E$) such that p_i^j is an adequate approximation of $U(LE,\ i,\ \beta)$ (respectively of $U(L,\ i,\ \beta)$ and $U(E,\ i,\ \beta)$) over its associated interval. Then, for each $1 \leq i \leq l$, we can define the coverage value $Coverage(n)_i$ achieved for the i-th disjunct as the average sum of the coverages achieved for its inequalties, strict inequalities and equations

$$Coverage(n)_i =$$

$$(\ Coverage(n)_i^{LE} + Coverage(n)_i^L + Coverage(n)_i^E\ )/3.$$

One global strategy for testing then is to systematically reduce the worst case uncertainty. This corresponds to choosing for the $n+1$-th test, the disjunct ϕ_i which has the smallest coverage value $Coverage(n)_i$ after n tests. (In the case of several equivalent alternatives, for example during initialisation of the algorithm, one alternative can be chosen at random.)

Similarly a stopping criterion for the procedure under *worst case uncertainty* is to choose a value $\sigma \in \mathbb{R}$ and terminate the algorithm when $Coverage(n)_i \geq \sigma$ for all $1 \leq i \leq l$.

We should point out that a variety of other disjunct choice and stopping criteria can be envisaged, based on approximation and convergence concepts, which have yet to be evaluated.

3.4 A Complete Algorithm.

We can now summarise the informal presentation of the approximation method for black-box specification based testing given in Sections 3.2 and 3.3 with a more formal description of the algorithms for test execution, choice of next test and updating of coverage. As in Section 3.3, for simplicity, we deal only with the case where $d = 1$ and Y is a singleton set $Y = \{\ x_\lambda\ \}$ for some chosen prevariable x_λ. It should be clear how to generalise the algorithms for $d > 1$.

In the sequel, if l is a list data structure and $1 \leq i \leq length(l)$ then we write l_i for the i-th element of l. We also use a "C" style notation for array declarations where the length is determined dynamically at initialisation time. Thus if *ident* is an identifier and τ is a type then $ident[\ ] : \tau$ is an array of values of type τ whose length will be determined when *ident* is first initialised. If Y is a finite set of variable identifiers for a first order language then we assume the existence of a type $Assignment(Y, A)$ of bindings of Y to elements of a data type A. These may for example be implemented as arrays.

3.4.1. Algorithm.

Input:

(i) an SUT $S[x_1,\ \ldots,\ x_n] \in Prog(\Sigma^{ring},\ X)$,
(ii) a quantifier free precondition $pre \in L(\Sigma^{ring},\ X)$,
(iii) a quantifier free postcondition
$post \in L(\Sigma^{ring},\ X \cup X')$,
(iv) a time limit $\tau : [\{\ x_1,\ \ldots,\ x_n\ \} \to \mathbb{Z}] \to \mathbb{R}$,
(v) a coverage limit $limit \in \mathbb{R}$,
(vi) a search variable $x_\lambda \in \{\ x_1,\ \ldots,\ x_n\ \}$.

Output:

either:
(i) prevariable assignments $\alpha : \{\ x_\lambda\ \} \to \mathbb{Z}$,
$\beta : \{\ x_1,\ \ldots,\ x_n\ \} - \{\ x_\lambda\ \} \to \mathbb{Z}$ and
postvariable assignment $\omega : \{\ x'_1,\ \ldots,\ x'_n\ \} \to \mathbb{Z}$
such that $(\mathbb{Z},\ \alpha \cup \beta \cup \omega) \models\ pre \wedge \neg post$, or
(ii) error: execution time of S exceeds $\tau(\alpha \cup \beta)$, or
(iii) error: coverage reaches *limit* without success.

Algorithm PolyTest
Types
Predicate = { LE, L, E }
Constants
k : integer; // Maximum degree of approximation.
tol : real; // Error tolerance of each polynomial.
$\alpha_1,\ \ldots,\ \alpha_{k+1}$: Assignment($\{\ x_\lambda\ \}$, $\mathbb{Z}$);
// used to initialise divided difference structure Δ.
Variables
successful, timeout : boolean;
coverage, disj_coverage[ ], v_prediction : real;
pred_coverage[ ] : array [LE ... E] of real;
Δ[ ] : array [LE ... E] of list of array [1 ... k+1] of real;
modelled [ ] : array [LE ... E] of
list of array [1 ... k+1] of boolean;
v [ ] : array [LE ... E] of list of real;
α : Assignment($\{\ x_\lambda\ \}$, $\mathbb{Z}$);
β : Assignment($\{\ x_1,\ \ldots,\ x_n\ \}$ - $\{\ x_\lambda\ \}$, $\mathbb{Z}$);
ω : Assignment($\{\ x'_1,\ \ldots,\ x'_n\ \}$, $\mathbb{Z}$);
TCond : DNF formula in $L(\Sigma^{ring},\ X \cup X')$;

{
successful := false;
timeout := false;
coverage := 0;
TCond := PosDNF($pre \wedge \neg post$);
// Convert the test condition into positive DNF

β := Random(β, $\{\ x_1,\ \ldots,\ x_n\ \} - \{\ x_\lambda\ \}$, *pre*);
/* Make a random choice for β consistent with
pre by conventional constraint solving.*/

Δ := array [1 ... length(TCond)] of
array [LE ... E] of
list of array [1 ... k+1] of real;
// Create difference structure Δ
modelled := array [1 ... length(TCond)] of
array [LE ... E] of
list of array [1 ... k+1] of boolean;
// Create modelled.
v := array [1 ... length(TCond)] of
array [1 ... length(TCond)]
of array [LE ... E] of list of real;
// Create v.
for d = 1 to length(TCond) do
for p = LE to E do
InitialiseList(Δ[d][p], modelled[d][p],
v[d][p], α_1 , ... , α_{k+1}, S);
/* Initialise divided difference list,
modelled list
and value list for the d-th disjunct
and predicate p by executing
the $k+1$ initial input assignments
$\alpha_1,\ \ldots,\ \alpha_{k+1}$ on S.*/

while ($coverage < limit$
and not (successful or timeout)) do
{

```
        α := ChooseInput( );
        // chooses α : { x_λ } → ℤ so (ℤ, α ∪ β) ⊨ pre

        while ExecutionTime(S) < τ(α ∪ β)
        and not Terminated(S) do
            Execute( S, α ∪ β );
        /* Execute S on α ∪ β while
        monitoring execution time
        and termination.*/

        if ExecutionTime(S) < τ(α ∪ β) then
        {
            ω := [[ S ]](α)′;
            // Save the final state of S.
            coverage := UpdateCoverage( coverage );
            if (ℤ, α ∪ β ∪ ω) ⊨ ¬post then
            successful := true;
         };
        else timeout := true;

     }; // of while loop

    if successful then
        return ("successful with: " α, β, ω )
    else
        if timeout then return ( "error: timed out ")
        else return ("error: coverage reaches:", limit );
}
```

Note that the algorithm PolyTest terminates as soon as a prevariable assignment $\alpha \cup \beta$ is encountered such that the execution of S on $\alpha \cup \beta$ exceeds the allowed time bound $\tau(\alpha \cup \beta)$. This flags potential infinite loops for further investigation. It is also possible to allow PolyTest to proceed in such cases, for example by assigning a constant non-zero value in the corresponding position of the result array v.

It remains to define the algorithms for *ChooseInput* and *UpdateCoverage* using the piecewise polynomial approximation method described in Section 3.3.

3.4.2. Algorithm.

Output:

(i) An assignment $result : \{ x_\lambda \} \to \mathbb{Z}$ such that $(\mathbb{Z}, \alpha \cup \beta) \models pre$, and $\alpha \cup \beta$ is a potentially successful test case for S according to the approximation model.

Algorithm ChooseInput
Variables

```
d, p, j, i : real;
poly : polynomial with real coefficients:
solutions : set of real;

{
    d := least x ∈ { 1, ..., length(TCond) }
    s.t. coverage[x] is minimal;

    p := least x ∈ { LE, L, E }
        s.t. coverage[d][x] is minimal;

    j, i := least x, y ∈ { 1, ..., length(Δ[d][p]) }×
    { 1, ..., k + 1 } s.t. Δ[d][p]_x[y] is minimal
    and modelled[d][p]_x[y] = false ;

    /* Construct polynomial at this position
    in the difference structure Δ[d][p].*/
    poly := MakePolynomial( Δ[d][p]_j[i] );

    // Calculate solutions of the polynomial
    solutions := SolvePolynomial( poly );

    /* Now intersect solutions with
    the interval of the polynomial
    to see if any can be used as zero predictions.*/
    if solutions ∩[v[d][p]_j, v[d][p]_j] ≠ ∅
    {
        /* Then a zero exists in the interval of poly.
        So use this zero point for the next test input.*/
        result( x_λ ) :=
            least s ∈ solutions ∩[v[d][p]_j, v[d][p]_j];
        v_prediction := 0;
     }
    else
    {
        /* No zero point exists in the interval of poly.
        Choose a point within interval for next test
        according to some strategy (see Section 3.2)
        to evaluate the accuracy of approximation. */
        result( x_λ ) := Choose( y ∈ [v[d][p]_j, v_⌊d][p]_j] );
        v_prediction := Evaluate( poly, result( x_λ ) ) ;
     }
return( result );
 }
```

The function $Choose(S)$ chooses an element from the set S according to some strategy, e.g. randomly. The function $Evaluate(p, x)$ evaluates a polynomial p on an argument x.

3.4.3. Algorithm.

Output:

(i) A worst case coverage value *result* which is the minimum over all disjuncts ϕ_l of $TCond$ of the average coverage achieved by approximation up to tolerance *tol* for ϕ_l separately on its inequalities, strict inequalities and equations.

Algorithm UpdateCoverage
Variables

```
v : real:

{
for d = 1 to length(TCond) do
    for p = LE to E do
    {
        /* Evaluate the unknown function U for disjunct
        d and predicate p on the prevariable assignment
        α ∪ β and postvariable assignment ω.*/
        v := U( d, p, α, β, ω );

        if | v - v_prediction | < tol then
        {
            /* A new polynomial piece satisfying
            the approximation accuracy of tol
            has been identified in the difference model./*
```

```
            modeled[d][p]_j[i] := true ;

            coverage[d][p] :=
                coverage[d][p] + | v[d][p]_{j+i} - v[d][p]_j | ;
            coverage[d] :=
                ( coverage[d][LE] + coverage[d][L]
                + coverage[d][E] ) / 3;

            result :=
                min_{x∈{ 1, ..., length(TCond) }} coverage[x];
        }
        else
        {
            /* Insert the assignment α into the sorted
            list of prevariable assignments
            and return insertion position.*/
            pos := insert( α(x_λ), a[d][p] );
            v[d][p]_pos := insert(v);
            //Update Δ locally around pos
            UpdateDifferences( Δ[d][p]_pos ) , ... ,
                Δ[d][p]_{pos-k} );
        }
    }
return( result ):
}
```

Notice how updating the coverage value $coverage[d][p]$ by adding the integral of the domain of a sufficiently accurate polyomial piece simply consists of adding the length of its interval, in the case that $d = 1$.

4. A CASE STUDY.

It is beyond the scope of this extended abstract to give a full analysis of the empirical performance of the approximation based testing algorithms of Section 3. Indeed, this evaluation work is still in progress. However, it is appropriate to describe here a simple case study using an early and unoptimised version of our algorithms implemented in C++. In particular we can convey the complexity of the modeling process for a simple SUT.

Consider Algorithm 4.1, which presents a conventional implementation of Newtons method for finding the square root of a floating point number. A black-box specification for this algorithm is simply

$$number \geq 0 \quad \rightarrow \quad (\ |(root * root) - number| < epsilon \ )$$

where *epsilon* is a fixed error tolerance of choice.

We have introduced five different mutation errors by hand into the code, where these are numbered and commented out to reveal the correct code only. It is particularly interesting for us to consider floating point computations, since for this type of algorithm very few symbolic methods exist for code correctness analysis. Note that several of the more obvious mutation errors that could be introduced into Newton_Root lead to infinite loops at runtime. Errors 1-5 do not have this property (at least individually).

4.1. Algorithm.

```
// Algorithm Newton_Root
float number, root;
float error = 0.000001;
// float error = 0.01; // error 1
// float error = 0.1; // error 2

if ( number == 0)
{ root = 0; }
else
{
    root = 1.0;
    // root = 0.5; // error 3
    do
    {
        root = ( number / root + root ) / 2.0;
        // number = ( number / root + root ) / 2.0;
        // error 4
    }
    while ( fabs( ( number / root + root)) -1 ) >= error );
    // while ( ( number / root + root) -1 ) >= error );
    // error 5
}
```

Algorithm 4.1. was seeded with various combinations of the given mutation errors, compiled and executed within the approximation based test framework described in Section 3. Table 1 summarises the results obtained. In column 1, we list each individual mutation error, (rows 1-5), seven double mutation errors (rows 6-12), and five triple mutation errors (rows 13-17). In columns 2-4 we list the number of constant, linear and quadratic polynomial pieces required before convergence of approximation exceeded the threshold value. Column 5 lists the number of solution intervals found, where a solution interval is a range of values for the single input variable, which give rise to an error. Thus column 5 does not correspond to individual errors but rather sets of errors. Column 6 indicates the total number of inputs or test cases on which the SUT was executed before convergence of approximation was obtained.

In terms of time performance, the algorithm terminated in under 1 second for all cases considered. Recall from Section 3 that for $d = 1$ the time complexity of testing is linear in the number of test cases considered.

We can observe a number of interesting phenomena already in this simple case. The number of test cases examined before model convergence was achieved varied widely, between 8 and 1410. The variation in the number of solution intervals obtained is less significant, since this prototype implementation does not merge adjacent intervals.

Interestingly, the algorithm gives us some insight into the pathology of code errors when multiple mutation errors are introduced. We can see that error 4 totally dominates error 1 in the double mutation error on row 6. This makes tangible the situation where one error can mask another, and confirms that debugging must therefore be an iterative process. This phenomenon is visible elsewhere in the table. For example, errors 3 and 4 (row 8) cause an infinite loop, which is inherited by error combinations 1, 3 and 4 (row 13) and 2, 3 and 4 (row 14).

Error 4 also dominates error 5 when the two exist simultaneously (row 8). Among double mutation errors, only the combination of errors 3 and 5 (row 11) seems to exhibit characteristics not determined uniquely by one error component. These characteristics are inherited by row 15. So we see that these mutation errors seem to possess an interesting

Error No.	*Approximations used*			*Solution Intervals*	*Model Points*
	Const.	*Lin.*	*Quad.*		
1	40	4	39	21	65
2	168	5	134	132	187
3	0	0	1000	98	1410
4	1	0	5	5	8
5	14	4	15	18	23
1 & 4	1	0	5	5	8
2 & 4	1	0	5	5	8
3 & 4	*infinite loop*				
1 & 5	14	4	15	18	23
2 & 5	14	4	15	18	23
3 & 5	16	0	17	15	23
4 & 5	1	0	5	5	8
1 & 3 & 4	*infinite loop*				
2 & 3 & 4	*infinite loop*				
1 & 3 & 5	16	0	17	15	23
1 & 4 & 5	1	0	5	5	8
3 & 4 & 5	*infinite loop*				

Table 1: Complexity of Testing for Newton_Root

semi-lattice like structure under conjunction.

5. CONCLUSIONS.

In this report, we have shown how specification-based software testing can viewed as a satisfiability problem, and how this problem can be solved using techniques from function approximation which also yield a model of test coverage.

There is considerable scope for future research. In particular, the following questions seem important.

(i) What is the practical scope of our existing testing technique for low values of the approximation dimension d? What other methods can be used for dimension reduction?

(ii) What is the scalability of our approach to large scale systems and complex functional specifications?

(iii) How can algorithm PolyTest be extended from quantifier free formulas to general quantified formulas in an efficient way?

(iv) What are the best types of approximation function for relations chosen from other (e.g. non-numeric) data types? For example, in recent years, approximation techniques using wavelets have achieved significant success and we expect that Haar wavelets (for discrete data types such as Booleans, chars and integers) and Daubechies wavelets (for continuous data types such as floats) could have an important role to play. In general, one can say that functional analysis is rich in providing approximation spaces and useful theoretical results for designing algorithms.

We ackowledge the helpful comments of H. Johansson while this research was carried out. We also gratefully acknowledge the financial support of the Swedish Research Council for Engineering Sciences (TFR) under grant 2000-447.

5.1 References.

[1] J.W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.

[2] F. Bergadano, D. Gunetti, Testing by means of inductive program learning, *ACM Trans. Software Engineering and Methodology* 5 (2) 119-145, 1996.

[3] T.A. Budd, D. Angluin, Two notions of correctness and their relation to testing, *Acta Informatica* 18, 31-45, 1982.

[4] E.W. Cheney, *Approximation Theory*, American mathematical Society Chelsea Publishing, Providence, Rhode Island, 1966.

[5] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, *IEEE Computer* 11(4), 34-41, 1978.

[6] H.B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, 1972.

[7] R.G. Hamlet, Testing programs with the aid of a compiler, *IEEE Transactions on Software Engineering*, 3(4), 279-290, 1977.

[8] J. Loeckx and K. Sieber, *The Foundations of Program Verification*, John Wiley, Chichester, 1987.

[9] A.J. Offut, J. Pan, Automatically detecting equivalent mutants and infeasible paths, *The Journal of Software Testing, Verification and Reliability* 7 (3), 165-192, 1997.

[10] M.S. Phadke, Planning efficient software tests, *Crosstalk* 10 (10), 11-15, 1997.

[11] K. Romanik, Approximate testing and its relationship to learning, *Theoret. Comput. Sci.* 188, 79-99, 1997.

[12] K. Romanik and J.S. Vitter, Using Vapnik-Chervonenkis dimension to analyze the testing complexity of program segments, *Information and Computation* 128 (2), 87-108, 1996.

[13] E.J. Weyuker, Assessing test data adequacy through program inference, *ACM Trans. Program. Lang., Syst.*, 5 (4), 641-655, 1983.