RELIABILITY OF PROGRAMS SPECIFIED WITH EQUATIONAL SPECIFICATIONS

by

BORISLAV NIKOLIK

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 1998

"Reliability of Programs Specified with Equational Specifications," a dissertation prepared by Borislav Nikolik in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Dr. Dick Hamlet, Co-chair of the Examining Committee

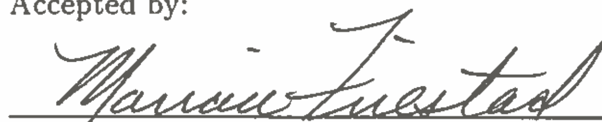
Dr. Zary Segall, Co-chair of the Examining Committee

11/01/98
Date

Committee in charge:

Dr. Dick Hamlet, Co-chair
Dr. Zary Segall, Co-chair
Dr. Amr Sabry
Dr. Chris Wilson
Dr. Shlomo Libeskind

Accepted by:



Dean of the Graduate School

An Abstract of the Dissertation of

Borislav Nikolik for the degree of Doctor of Philosophy

in the Department of Computer and Information Science to be taken December 1998

Title: RELIABILITY OF PROGRAMS SPECIFIED WITH EQUATIONAL SPECIFICATIONS

Approved: ______________________________

Dr. Dick Hamlet, Co-Chair

Approved: ______________________________

Dr. Zary Segall, Co-Chair

Ultrareliability is desirable (and sometimes a demand of regulatory authorities) for safety-critical applications, such as commercial flight-control programs, medical applications, nuclear reactor control programs, etc. A method is proposed, called the Term Redundancy Method (TRM), for obtaining ultrareliable programs through specification-based testing. Current specification-based testing schemes need a prohibitively large number of testcases for estimating ultrareliability. They assume availability of an accurate program-usage distribution prior to testing, and they assume

the availability of a test oracle. It is shown how to obtain ultrareliable programs (probability of failure near zero) with a *practical* number of testcases, *without* accurate usage distribution, and *without* a test oracle.

TRM applies to the class of decision Abstract Data Type (ADT) programs specified with unconditional equational specifications. TRM is restricted to programs that do not exceed certain efficiency constraints in generating testcases.

The effectiveness of TRM in failure detection and recovery is demonstrated on formulas from the aircraft collision avoidance system TCAS.

CURRICULUM VITA

NAME OF AUTHOR: Borislav Nikolik

PLACE OF BIRTH: Skopje, Macedonia

DATE OF BIRTH: August 26, 1969

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Portland State University

DEGREES AWARDED:

Doctor of Philosophy, 1998, University of Oregon
Master of Science, 1993, Portland State University
Bachelor of Science, 1992, Portland State University

AREAS OF SPECIAL INTEREST:

Software Testing
Software Reliability

PROFESSIONAL EXPERIENCE:

Software Consultant
College Instructor

ACKNOWLEDGEMENTS

I thank my wife Daniela for her emotional, linguistic, and technical support. I thank my advisor, Dick Hamlet, for sparking my interest in the area of Software Testing and Reliability, as well as for his support over the years.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION . . . 1

II. RELIABILITY BACKGROUND . . . 8

Reliability Overview . . . 8
Reliability Theory . . . 12
Reliability Modeling . . . 17
Fault Tolerance . . . 23
Result Checking/Data Diversity . . . 27
ADT Program Reliability . . . 29

III. TERM REDUNDANCY METHOD . . . 32

TRM Description . . . 32
Definitions and Notation . . . 37
Testing Phase . . . 42
Self-Checking Phase . . . 46
Scope of Term Redundancy Method . . . 51
Work Related to TRM . . . 57

IV. TRM AND BOOLEAN ADTS . . . 62

TCAS Boolean ADT . . . 63
TCAS Experiment . . . 65

V. SUMMARY AND FUTURE WORK . . . 71

APPENDIX

CLP(R) IMPLEMENTATION OF TCAS EXPERIMENT . . . 74

BIBLIOGRAPHY . . . 80

LIST OF FIGURES

Figure | Page

1. Stack-of-Boolean-Values ADT Program 4
2. C Fault and Failure Example 12
3. Self-Checking of a Multiplication Procedure 27
4. Stack-of-Boolean-Values ADT Program 32
5. Stack of Boolean Values TRS 36
6. Term Distribution Dependent on an if Statement 37
7. RBTR Algorithm 44
8. TRM Testing and Checking Algorithms 48
9. TCAS Specifications 64
10. Boolean ADT Specification 64
11. Exhaustive Mutated r_7 Rules 66
12. Boolean Bi-Directional TRS 68
13. TRM Effectiveness 69
14. Correct BTRS Axioms 74
15. Mutated BBTRS Axioms 75
16. CLP(R) Implementation of TCAS Specification 76
17. Illustration Driver Code 77

CHAPTER I

INTRODUCTION

Many safety-critical systems, such as medical, weapon, nuclear reactor, and avionics systems, depend on software for their functioning. Software failures, which are deviances of required and actual software outputs, in these systems can have severe consequences for people and the environment, including loss of human life and environmental pollution. For example, the radiation therapy machine Therac-25 has overdosed several patients, due to software faults in the control system [47]. It is important that such systems function ultrareliably; i.e., that the probability of software failure on a randomly chosen input is close to zero. In fact, some government regulatory authorities, such as FAA, demand ultrareliability for commercial flight-control programs [47, 60]. Therefore, one needs to be able to assess the reliability of the software under consideration in order to determine if the software is indeed ultrareliable.

Existing theory for assessing the reliability of software does not view software failures as deterministic events, but rather as random events, allowing the software to be characterized by random variables, such as Mean Time to Failure (MTTF) or Mean Time Between Failures (MTBF). Software reliability involves two kinds of models: reliability growth and reliability [51].

Reliability growth models the process of repeated debugging, failure detec-

tion by testing, and fault removal, until an acceptable reliability level is obtained. Reliability growth predicts the reliability of future program versions obtained from previous ones by fault removal.

Reliability, on the other hand, predicts MTTF or MTBF for the current version. Testing uncovers no failures, and reliability estimates are made based on the number of testcases executed. (If a failure occurs during testing, the program is corrected, and the testing process begins again.)

Unfortunately, in the reliability model it is impractical to show by straightforward sampling that software is ultrareliable, because prohibitively many testcases are needed, and the predictions depend critically on the availability of accurate usage information for the program under study.

Butler and Finelli define the "ultrareliable" region as program failure probabilities of 10^{-8} (per run) and below, and they present a very convincing argument that it is impractical to gain failure intensities in this region by testing, because too many testcases are needed [19]. Currently, it is considered that the practical reliability region is program failure probabilities of 10^{-4} and greater [51].

Whenever a statistical prediction is involved, one needs to assure that the sample on which the prediction is based is representative in order for the prediction to be accurate [32]. A representative random input sample for a program is a set of program inputs that are drawn from the same input distribution as the inputs drawn by the users of the program. In some cases, this distribution is known prior to program release, such as for jet engine controllers; however, this distribution is usually not known for new software, or for experimental software where the distribution changes based on the experiment being conducted. Therefore, it is of

critical importance that reliability estimates do not depend on a particular input distribution; i.e., that the estimates are accurate for an arbitrary input distribution.

Random testing involves determining if the program under test failed on a particular testcase or not. An effective procedure for evaluating outcomes of testcases is called a test oracle. Random testing for ultrareliability without a test oracle is infeasible due to the large number of testcases that need to be executed in order to estimate ultrareliability. Usually, test oracles are unavailable.

Many safety critical systems, such as military systems, involve Abstract Data Type (ADT) programs (coded mainly in ADA). ADTs can be specified by equational specifications, which can be transformed into Term Rewriting Systems (TRSs). A method for obtaining ultrareliability of ADTs specified with TRSs might encourage the development of many more such systems. Thus, a method for obtaining ultrareliable ADTs would be of value.

A method is proposed, called the Term Redundancy Method (TRM), that can obtain ultrareliability with a practical number of testcases for decision ADT programs specified with equational specifications. TRM's ultrareliability predictions hold true regardless of the input distribution chosen by the user of the software. TRM does not need a test oracle.

A brief description of TRM is given here. For a detailed, complete, and formal treatment of TRM see chapter III.

Programs employing ADTs involve ADT terms in their code that need to be evaluated as the program executes. For example, a program might be a C `main` function which at execution time needs a result of a term. Such programs can be viewed as consisting of parts that are responsible for obtaining inputs from

```
main(){
    ...
    s = empty;
    read(b);
    b = top(push(s,b));
    print(b);
    }
```

FIGURE 1. Stack-of-Boolean-Values ADT Program

the environment, parts that give outputs back to the environment, and parts that evaluate the terms (evaluators).

For example, consider the C `main` program of figure 1. The `read` statement is the part of the program that obtains a value for `b`, which becomes a part of the term `top(push(s,b))`. For presentation purposes, suppose the `read` statement reads in 0 for `b`, giving raise to the grounded term `top(push(empty,0))` at the assignment statement. The (perhaps incorrect) implementation of the `top`, `push`, and `empty` procedures together with the mechanism for calling them as directed by the term `top(push(empty,0))` is the part of the program that evaluates the term (evaluator). The `print` statement is the part of the program that outputs the value of the term back to the environment. Evaluators may unfortunately evaluate some terms incorrectly. For example, a Stack-of-boolean-values evaluator might give a value of 1 (instead of the correct value 0) when given the term:

```
top(push(empty,0))
```

as input. TRM forces such incorrect evaluation to occur rarely; i.e., TRM forces the evaluation of terms to be ultrareliable. Next, it is described how TRM obtains ultrareliable evaluations.

An input is given to a `main` program, which in turn gives rise to a ground term (call it the original term) that needs to be evaluated. Instead of evaluating only the original term, TRM generates multiple random equivalent terms from the equational specification of the ADT. TRM then gives these random terms (including the original term) to the evaluator, and returns the majority of evaluator outputs as the result of the original term. The `main` program then continues operation with this majority value and eventually produces an output (if it terminates).

The majority values can be trusted to the extent of the amount of pre-testing performed on the evaluator; i.e., more testing gives a higher confidence that the majority values can be trusted. Thus, TRM consists of two phases: (a) estimating the failure rate of an evaluator on a randomly chosen term through testing (called the testing phase), and (b) generating and executing the evaluator on a set of random equivalent terms, which comprise a self-check (called the self-checking phase), and performing a majority vote on the results of the executions. In phase (a), a statistical theory is used to obtain estimates on the failure rate of an evaluator. In phase (b), when a user requests a result of a term c at execution time, TRM generates a self-check $\{c_1, c_2, ..., c_n\}$ using a TRS, by applying random rewrite rules to c, such that all of the random terms $c, c_1, c_2, ..., c_n$ are equivalent according to the TRS. Then, the evaluator is executed on each term in the check. TRM uses majority vote on the execution results to decide on the result that it returns to the `main` program. The terms $\{c_1, c_2, ..., c_n\}$ that TRM generates are failure-independent terms. The idea of giving programs multiple random inputs was introduced by Blum [13] and Ammann [2].

TRM obtains ultrareliability with a practical number of testcases. For exam-

ple, if the probability of failure of an evaluator on a randomly chosen term from the equivalence class of c is 10^{-4}, then the probability of failure agreement of three failure-independent executions on three randomly chosen terms from the equivalence class of c is 10^{-12}.

TRM's predictions do not depend on a particular user distribution over terms, because TRM samples from the equivalence class of c from an appropriate fixed distribution that is the same in both phases. If testing in phase (a) shows a particular failure rate based on a particular distribution over terms, the same failure rate has to be observed in phase (b) for each term chosen according to the same distribution over terms as in (a). TRM assures that in phase (b) the terms for checking are chosen according to the distribution in (a).

TRM applies to decision term-evaluators, specified with unconditional equational specifications [44]. Informally, a decision term-evaluator is a program that takes as inputs a large set of terms, and it classifies these terms into a small number of output classes. For example, a Stack-of-boolean-values evaluator is a decision term-evaluator since it evaluates every term to either 1 or 0. This restriction of TRM to decision evaluators is necessary to assure that a practical number of terms are used in the testing phase.

An example of a conditional equation is

```
if not empty(S) then push(pop(S)) = S,
```

where the condition `not empty(S)` guards against the exception case, and the pop operation removes an element only from a stack that is not empty. In theory, TRM could be applied to conditional equational specifications by generating equivalent terms through conditional rewriting [44]. In particular, random conditional rewrit-

ing would be used to generate random equivalent terms. However, in practice, such a rewriting would introduce additional complications, which are not present in the unconditional case, so TRM in this work is applied only to unconditional equational specifications.

TRM does not apply to anything more then a set of unconditional equations $\{t_1 = s_1, ..., t_n = s_n\}$. For example, TRM would not apply to specifications which allow equational specifications to be associated to operational pre/post conditions of program interfaces through abstraction functions. That is, TRM is strictly a "black-box" method, which is not concerned with the internal details of evaluators, such as element arrangement in a stack, a particular implementation strategy of the stack, or a particular evaluation strategy.

CHAPTER II

RELIABILITY BACKGROUND

Software quality involves various aspects, such as reliability, functionality, usability, performance, serviceability, capability, installability and maintenability. The most important aspect of software quality is considered to be reliability, since it quantifies software failures [58].

Reliability Overview

Software reliability concepts have been adapted from hardware reliability. Both software and hardware reliability involve random testing. In this section, a general overview of software reliability concepts, an overview of random testing, as well as the differences between software and hardware reliability, are given.

Software Reliability

Software reliability is usually defined as the probability of failure-free software operation for a specified period of time in a specified environment (time domain definition) [51]. Software reliability is also defined (data domain definition) as the probability of failure-free software operation on a random program input selected from a specified environment. TRM uses the data domain definition.

Discrete-time (data domain) system reliability treats executions of programs as discrete events. Let p be the probability of a system failure at a randomly chosen

input point. Let $R_d(k)$ be the probability that no system failure occurred during an execution comprising k input point selections. Then the discrete-time system reliability is (assuming failure independence)

$$R_d(k) = (1 - p)^k.$$

Fundamental to reliability are the notions of *failure* and *environment*. Failure means that the program in its functioning has not met user requirements in some way. Failures are of a dynamic nature; i.e., the program has to be executing for a failure to occur. Reliability represents a user-oriented view of software quality. It relates to program operation rather than to the design of the program. Therefore, in general reliability is affected by the way the program is operated; i.e., by the program environment. The environment is described by the *operational profile* for the program. An operational profile is the set of program inputs that the program can take, along with the probabilities with which they will occur during the execution of the program. It may not always be practical to determine all the probabilities for all the inputs, because of the potentially large number of them. Furthermore, the operational profile of a program may not be known at all. Thus, it is of importance that reliability estimates do not depend on an operational profile.

Random testing is used to estimate the reliability of software.

Random Testing

Reliability is established by random testing: the input domain of the software is sampled according to its operational profile, and the program is executed using the random sample. The random testing involved in this process is a form of black-box

(or specification-based) testing, where the random samples are generated from the specification of the program's external behavior. This form of testing is different from white-box (structural or coverage) testing, which involves covering program internal structures, such as statements.

As can be seen from the discrete-time system reliability formula above, in order to estimate ultrareliability, where $R_d(k) > 1 - 10^{-9}$, more than 10^9 testcases are needed. Unfortunately, currently only less than 10^4 is considered practical.

Testing requires means to evaluate the outcome of each testcase (random sample) in order to determine if a failure occurred or not. A *test oracle* is a computable procedure for deciding if the output of a program matches its specified output. Since random testing for ultrareliability involves a large number of testcases, testing without an efficient oracle is infeasible. Currently, evaluating outcomes of testcases is mainly manual. In most cases, human visual inspection of the output and the specification (usually written in a natural language) determines if a failure occurred.

Thus, random testing for ultrareliability currently needs an impractical number of testcases, the usually unavailable test oracle, and an operational profile.

Software vs. Hardware Reliability

Software reliability has been adapted from hardware reliability [67]. The principal difference between software and hardware is that the source of failures in software is design faults, whereas the main source of failures in hardware has been wearout (and other physical causes) [58]. Software does not wear out, and design reliability has not been studied in hardware systems, mainly because hardware design faults are insignificant compared to physical deterioration faults.

Not only is the source of faults (design for software and wearout for hardware) different, but the environments in which hardware and software reliabilities are studied is different as well. The environment for hardware involves parameters such as temperature, pressure, acceleration, etc., while the environment for software involves inputs to the software (operational profile). The operational profile has profound influence on software reliability. In particular, the software can be incorrect in the sense of the conventional program correctness theory, but still have reliability of one. This could happen when the operational profile of the program is such that it does not allow selection of inputs that cause the program to fail (probability of chosing these inputs is zero).

The crucial differences between hardware and software have prompted many to question the validity of applying hardware reliability to software. Some justification for using hardware reliability on software could be given in terms of a particular random variable: MTTF, which is the most common random variable that characterizes failure in time. MTTF is the sum of interfailure times divided by the number of failures. A bigger MTTF means that on average the software is going to operate longer without failure. Is MTTF a random variable for software? The question is a consequence of the common argument which says that the software is either correct in which case its reliability is one, or it is incorrect in which case its reliability is zero. For MTTF to be a random variable, there have to be some random fluctuations present in the software and/or in the software environment, which would make interfailure times subject to uncertainty. The random fluctuations in software come from the uncertainty of the execution conditions that will actually occur during the use of software. One cannot, for example, predict what inputs are going to be

presented to a space shuttle's sensors at some arbitrary time in space. Therefore, it seems reasonable to apply hardware reliability to software.

Reliability Theory

Reliability theory defines and relates basic concepts such as failure, fault, failure intensity, mean time to failure and reliability [67]. Essential to the definition

```
void main(){
  int i;
  scanf("%d",&i);
  i=i*2;
  printf("%d",i);
}
```

FIGURE 2. C Fault and Failure Example

of reliability is *failure*. Failure is a departure of the external results of a program operation from *requirements*. Requirements are statements about the intended external operation of the program. Failure is distinct from but related to the notion of a *fault*, which is a defect in the program text. For example, if the requirement of the C program given in figure 2 is to output $i+2$, the program has a fault in the statement $i = i * 2$, and a failure would occur when `scanf` reads in any value different than 2.

Reliability theory considers software failures to form a random process, which is nothing more than a set of random variables characterized in time. Each value of a random variable is assigned a probability. The function that assigns these probabilities is called the *probability density function*. When the random variable represents failures, the probability density function is called the *failure density function*. Next,

the notions of *reliability* and *failure density functions* are given more formally, and the relations between them are given as well. The following continuous time-domain reliability formulas are taken from [67].

Let T be a random variable representing the failure time of software. The probability that the software will fail by time t is

$$F(t) = P[T \leq t] = \int_0^t f(x)dx,$$

where $F(t)$ and $f(t)$ are the cumulative distribution function and the failure density function, respectively. The probability that the software is going to survive until time t is:

$$R(t) = P[T > t] = 1 - F(t) = \int_t^\infty f(x)dx,$$

where $R(t)$ is the *reliability function*. *Failure rate* is the conditional probability that a failure occurs in the interval $[t_1, t_2]$, given that failure has not occurred before t_1. That is,

$$\frac{P[t_1 \leq T < t_2 | T > t_1]}{t_2 - t_1}$$

$$= \frac{P[t_1 \leq T < t_2]}{(t_2 - t_1)P[T > t_1]}$$

$$= \frac{F(t_2) - F(t_1)}{(t_2 - t_1)R(t_1)}.$$

The *failure intensity* $z(t)$ is the derivative of the failure rate. If we let the interval

be $[t, \Delta t]$, and let $\Delta t \to 0$, the failure intensity at time t is

$$z(t) = \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)}$$

$$= f(t)/R(t).$$

The failure intensity is the failure rate at time t, given that the software survives up to t.

Failure Intensity

All the formulas in the previous section depend on the failure density function. The usual assumption of conventional software reliability theory is that the failure density function is an exponential distribution for software [68] i.e.,

$$f(t) = \lambda e^{-\lambda t}, \lambda > 0.$$

(This distribution is a special case of the Poisson distribution where the number of events, in our case failures, experienced in the time interval $[0, t]$ is one.) Using the formulas of the previous section, we arrive at the following cumulative distribution function

$$F(t) = 1 - e^{-\lambda t},$$

where λ is a constant. The continuous-time reliability function is

$$R(t) = e^{-\lambda t},$$

and the failure intensity is

$$z(t) = \lambda.$$

When failures follow an exponential law then programs have constant failure intensity λ; i.e., failure is equally likely to occur at any time. The average lifetime of the software (MTTF) is $1/\lambda$.

Lyu [51] shows the relation between discrete (data domain) and continuous (time domain) time system reliability as follows. Let t_e be the execution duration associated with an input selection (assuming t_e is same for every input selection). The time elapsed since the start of the execution is $t = kt_e$. Let p/t_e have a finite limit λ as t_e approaches zero. Then,

$$R(t) = \lim_{t_e \to 0} R_d(k) = e^{-\lambda t},$$

which is exactly the time-domain reliability with a failure rate λ. ($R_d(k)$ is the probability that no system failure occurred during an execution comprising k input point selections.)

Operational Profile

The reliability formulas above do not take into account the fact that reliability depends not only on the failure density function, but also on the way the software is used. The formulas do not reflect this intuition, because they are *hardware* reliability formulas. Hardware failures are not dependent on the way a hardware component is used, because the simplicity of hardware circuitry allows one to assume that design faults are practically nonexistent, and the failures result primarily from random

fluctuations in the component, which are independent of the way the component is used. For software, the situation is different since only design faults are responsible for failures.

Failure is a function of the way software is used since the software could be such that some inputs may cause failures to occur, while other inputs may not. Since different users may use the program differently, some users may observe more failures than others. For example, users make different operating system calls than system administrators do. As a result, the operating system may have quite different reliability levels in these two cases. As a concrete example, consider the faulty C program given in figure 2, whose requirement is to print $i + 2$. If 90% of the time users input 3, and 10% of the time they input 2, the probability of failure for the C program is 9/10. However if users have 2 as an input 90% of the time, and they have 3 as an input value 10% of the time, the probability of failure for the C program is only 1/10. Next, the notion of "the way software is used" is made more precise.

Let $\mathcal{P}$ be a program and D be $\mathcal{P}$'s input domain. The frequency of selecting inputs from D defines a probability density function over D, expressing the probability that a particular input is going to be selected as input to $\mathcal{P}$. This frequency defines $\mathcal{P}$'s usage. Let $d(x)$ be the *input probability density function*, reflecting the frequency of input selection from $\mathcal{D}$. In general, $d(x)$ will weight some inputs more heavily than others, reflecting the actual usage expected of $\mathcal{P}$. Given a discrete density function $d(x)$, the *operational distribution* $F(x)$ is the cumulative probability that x will occur in actual use

$$F(x) = \sum_{-\infty}^{x} d(y).$$

In practice, the best that can be obtained, in general, is a crude approximation to $d(x)$ called the *operational profile* [57]. The function $d(x)$ is approximated in cases where the usage information is not available. Usage information may be unavailable for new software that has never been used before and as such has no usage history. Another example of unavailable usage information is experimental software, where each experiment is different form the previous one, and requires different inputs from the rest of the experiments. Among others, Miller [55, 54] points out that it is difficult to ensure that the distribution from which testcases are selected is identical to the operational distribution.

Reliability Modeling

In this section, the two distinct kinds of reliability models: *reliability growth* and *reliability* [64] are considered. Reliability is a limiting case of reliability growth where the number of observed failures is zero.

Reliability Growth Models

Software goes through a testing and debugging cycle until a release quality is obtained. Models that describe this process are called *reliability growth* models [58]. Reliability growth models reflect the intuition that on average, as faults are removed from the software, failure intensity drops and consequently reliability increases.

Let successive failures of a program obey a nonhomogeneous Poisson process. Let $N(t)$ be the cumulative number of failures observed during time interval $[0, t]$. Let $H(t) = E[N(t)]$ be the mean value of $N(t)$, and $h(t) = dH(t)/dt$ be the failure intensity. Reliability growth occurs when $h(t)$ is nonincreasing. Reliability can grow

on some interval $[0, T]$, despite local fluctuation of $h(t)$ on $[0, T]$, if the expected number of failures in any initial interval (of the form $[0, t]$) is no lower than the expected number of failures in any interval of the same length occurring later (of the form $[x, x + t]$).

Reliability may fluctuate between growth and decrease on $[0, T]$ because the fault removal process may not be perfect (one that may not always remove faults or/and one that may introduce new ones). Changes in the operational profile of a program may change reliability as well. Furthermore, an imperfect fault removal could increase reliability w.r.t. one operational profile, but decrease reliability w.r.t. another one.

Reliability growth is analyzed through the use of *trend tests* [64]. Trend tests are used to determine if the system undergoes reliability growth or reliability decrease. The most commonly used trend test is the MTTF test consisting of calculating $\tau(i)$ the arithmetic mean of the first i observed interfailure times t_j: $\tau(i) = (1/i)\sum_{j=1}^{i} t_j$. When $\tau(i)$ forms an increasing sequence, reliability growth is deduced.

The two most commonly used reliability growth models consider failures to obey a nonhomogeneous Poisson process [58]. The two variations of the model are the *basic model* and the *logarithmic Poisson model*. The failure intensity assumed for the basic model is

$$\lambda(\mu) = \lambda_0(1 - \mu/v_0),$$

where λ_0 is the initial failure intensity, μ is the average or expected number of failures experienced at a given point in time, and v_0 is the total number of failures that would occur in infinite time. The failure intensity assumed for the logarithmic

model is

$$\lambda(\mu) = \lambda_0 e^{-\beta\mu},$$

where β is the failure intensity decay parameter. β represents the relative change of failure intensity per failure experienced. Changes in failure intensity may occur only after a (possibly imperfect) fault removal process is conducted.

Both of the models allow imperfect fault removal by increasing the value of v_0 and decreasing the value of β. The difference between the two models is that the basic model is used for programs with uniform operational profiles, and the logarithmic for a highly nonuniform profiles. This is so because the failure intensity decreases very slowly for the logarithmic Poisson process, which results from infrequent executions of some inputs that cause failures. On the other hand, the failure intensity in the basic model is a linear function, so removal of faults results in a proportional drop of the failure intensity.

Reliability growth models are divided into those with finite failures and those with infinite failures. Most published models assume a finite number of failures in infinite time. For example, the basic model assumes a finite number of failures in infinite time. The logarithmic Poisson model assumes infinite failures in infinite time. Each model in a category is characterized by *type* and *class* attributes. The type attribute gives the distribution of the number of failures experienced by time t. The class attribute gives the functional form of the failure intensity in terms of time.

The two types of models in the finite failure category are the binomial-type model, which assumes perfect fault removal at each failure experienced, and the Poisson-type model, which assumes that the total number of failures is a random

variable with a given mean value. That is, the Poisson-type model, unlike the binomial-type model, assumes imperfect fault removal.

Important classes in the binomial-type model are the exponential and the Weibull class; i.e., the failure intensities have exponential and Weibull distribution respectively.

Reliability growth models are essentially empirical fitting of curves to observed data [67]. For the basic and the logarithmic Poisson models, the values of three parameters need to be estimated: λ_0, v_0 and β. Failure data is gathered, the parameters estimated, and a particular failure intensity function selected from a family of functions by instantiating the parameters [58].

The main models that belong to the class of finite failures with exponential failure intensity are Jelinski-Moranda model [56], NHPP model [28], Schneidewind's model, Musa's basic execution time model [58], and the Hyperexponential model. The main models in the class of finite failures with Weibull and Gama failure time are Weibull model [58], and the S-shaped reliability growth model [51]. Some of the models in the infinite failure category are Duane's model, Geometric model, and the Musa-Okumoto logarithmic poisson model [58]. The best known Bayesian model is Littlewood-Verrall reliability growth model [51], which accounts for increase in reliability if no failures occur during operation, and it reflects the fact that different faults have different impact on the reliability of the program. The numerous models are compared and evaluated in [58].

The Reliability Model

Conventional reliability theory relates a number of successful tests, failure intensities and confidence bounds [68]. Let $\mathcal{P}$ be a program and D be the input domain of $\mathcal{P}$. A *testset* T is a subset of D. An element of T is called a *testcase*. Let $\mathcal{S}$ be $\mathcal{P}$'s specification. A testcase $x \in T$ is *successful* if $[\mathcal{P}](x) = \mathcal{S}(x)$ (where $[\mathcal{P}]$ denotes the function computed by $\mathcal{P}$).

Suppose a distribution $d(x)$ is given over $\mathcal{D}$, constant failure intensity θ, and n testcases drawn from D according to $d(x)$. Suppose all n testcases are successful. The probability that a testcase fails is θ, $1-\theta$ is the probability of success. Given that the n testcases are independent, the probability that they all succeed is $(1-\theta)^n$. The upper confidence bound α on θ is defined as the probability that the failure intensity of $\mathcal{P}$ is below θ. The confidence bound is related to the testset size n and the failure intensity by (call this formula the *confidence bound* formula)

$$\alpha \leq (1-\theta)^n,$$

which is obtained from the binomial formula as a special case in which the number of failures is 0. If one needs the failure probability to be less than θ, and one has run n successful testcases, the probability that an unacceptable product would pass the test is no higher then α. This means that $1-\alpha$ is the confidence bound on θ expressing the probability that θ exceeds the correct value for $\mathcal{P}$. One continues testing without failure to make α acceptably low [60]. The confidence bound formula allows statements like "1 million test points gives 99% confidence that the failure intensity of $\mathcal{P}$ is below 4.6×10^{-6}, a MTTF of about $220,000$ runs"(example taken

from [33]).

The testing process described above is called hypothesis testing [60, 65]. The statement "the failure rate of $\mathcal{P}$ is below θ" is the hypothesis to be tested in the case above. α is the probability that the alternate (wrong) hypothesis is accepted (known technically as Type II error [65]; the Type I error — rejecting the right hypothesis — is not considered in this work).

Conventional reliability practice involves sampling the input domain of the software according to the expected way of use, executing the software on the samples and estimating the reliability of the software based on the outcomes of the samples. Next some problems of conventional reliability are discussed.

Musa [57] has given heuristics for constructing operational profiles; however, the expected use of the software is in general not known prior to the testing process, since obtaining precise profiles requires predicting the future use of the program. Availability of an operational profile prior to the testing process is crucial, since predictions based on non-representative samples have no statistical meaning.

Obtaining high levels of reliability is not practical since in general it requires prohibitively large number of test samples, as can be witnessed from the confidence bound formula. For example, the confidence bound formula requires more than 10^9 test samples for predicting ultrareliability with a confidence bound higher than 90%. Butler and Finelli [19] present a very convincing argument that it is impractical to gain failure intensities in the ultrareliable region by testing.

A fundamental assumption of software testing is availability of means to evaluate testcase outcomes. Usually, the success or failure of a testcase is evaluated through a visual examination by a programmer, with a slower but more reliable ver-

sion of the program under investigation, or with an independently developed version of the same program. Automatic oracles could in some cases be built from formal program documentation [61] and from executable specifications [5, 40]. Since ultra-reliability testing requires large number of samples, testing without an automatic oracle is infeasible.

Apparently unrelated test samples may be correlated by the program text. Statistical testing assumes that the objects under test are not correlated in any way. If the sample-independence assumption does not hold, the study is invalidated; testing a program on n correlated inputs is equivalent to repeating the same test n times. The problem of determining correlation between points in the input domain of software is, in general, undecidable since it requires solving the halting problem, so reliance on a precise operational profile is essential in obtaining meaningful estimates [34]. If the operational profile is correct then the correlation of test samples is irrelevant.

Fault Tolerance

Fault tolerant software can be found in various systems, such as railway systems, military and civilian aircrafts, and space shuttles. Fault tolerance has mainly been used for dealing with hardware faults [64, 50, 46], however fault tolerance has been used to tolerate software faults as well [71, 53, 47]. The goal of tolerating software faults (software fault tolerance) is to increase the reliability of systems, such as the ones above, by allowing software to continue with correct operation in presence of software faults. The reasoning behind considering software fault tolerance is that complex software cannot be practically shown to be fault free, since

the primary technique for showing fault absence, proof of program correctness, is currently considered not to be practical for complex software.

Fault tolerant software consists of two mechanisms: result-verification, which determines if the software is producing correct outputs, and error recovery, which attempts to correct the failures as detected by the self-checking mechanism.

Result Verification

The goal of Result Verification is to determine if the software produces a correct output at run time. Result verification is usually based on run-time comparison of results of different but functionally equivalent software versions. Majority and consensus voting schemes have been designed for this purpose [51]. Various other techniques for checking software outputs have been developed, such as self-checking and error-detecting codes. Self-checking involves inserting software redundancy in the program, including both instruction and data redundancy [71, 53]. In most cases, a complete check on the correctness of the output is infeasible and only the *reasonableness* of the output is checked [47]. The most commonly used checks are functional checking, control sequence checking, and data checking. These reasonableness checks are in general hard to formulate and the checks might be as error-prone as the software they are supposed to check [47].

Another result-verification technique is self-checking based on executable assertions [53, 4]. In the most crude case, to make a program self-checking is to insert after its code the assertion:

$$\text{if not } \mathit{assertion}(S_0, S_f) \text{ then error,}$$

where $\mathit{assertion}(S_0, S_f)$ is a predicate of the initial state S_0 and the final state S_f

that the computation has to satisfy. This solution is subject to memory, time and most importantly to complexity constraints [53]. In particular, $assertion(S_0, S_f)$ might be as complex as the program, and therefore as error-prone as the program itself.

Algorithm-based fault tolerance [37] uses error detecting and correcting codes for performing reliable computations with specific algorithms. This approach modifies algorithms to operate on encoded data, and to output encoded data as well.

Certification of computational results [30], is based on the idea of a certification trail — a trail of data left by an execution of the program. Another, simpler and faster program, solves the original problem using the trail information left by the program under check. The results of the two programs are compared and if they agree the results are accepted as correct; otherwise an error is reported. This method is similar to the one proposed by Babai [8], where the original algorithm is modified to output additional information called a "witness". These methods require encoding of the input with an error-correcting code.

Once a failure has been detected, run-time error recovery can take place. Error recovery takes the form of a backward or forward recovery. Backward recovery takes (rolls back) the system into a previously saved state, and the execution is resumed from that state. Forward recovery transitions the system into a new correct state, or masks the fault by compensating for the fault.

Basic Error-Recovery Techniques

The two basic error recovery techniques are N-version programming (which is an error detection technique as well) and recovery blocks. N-version programming

is considered a forward error recovery technique, and recovery blocks is a backward recovery technique.

N-Version Programming

In N-version programming, unrelated software teams implement the same specification, the versions are run against the same input, and a voting scheme choses a result [7, 6]. If the versions are failure independent, this approach answers the impractical ultrareliability problem given that each version is independently tested to obtain a certain reliability estimate [19].

Apart from the N-version approach requiring an increase in the number of programmers, maintenance costs, and additional parallel hardware, strong arguments against the independence of versions assumption have been given [45, 18, 17]. Experiments carried out in [45] show that programs based on different algorithms failed on the same inputs. The dependence of versions usually arise from the difficult parts of the problem solved, and not from the dependencies in the solution techniques [47].

Butler and Finelli [19] explore the possibility of showing failure independence of versions so that N-version programming can yield ultrareliability. They conclude that showing independence of versions by testing requires impractical number of test samples.

Recovery Blocks

The recovery blocks approach consists of executing N modules, and subjecting module outputs to an acceptance test [63]. The process starts by executing the first module and checking its output with an acceptance test. If the module does not pass the test, the system is rolled back into the state prior to execution of the

$$y = f(x - r_1, y - r_2) + f(x - r_1, r_2) + f(r_1, y - r_2) + f(r_1, r_2)$$

$$y_c = [\mathcal{P}](x - r_1, y - r_2) + [\mathcal{P}](x - r_1, r_2) + [\mathcal{P}](r_1, y - r_2) + [\mathcal{P}](r_1, r_2)$$

FIGURE 3. Self-Checking of a Multiplication Procedure

first module, and then the next module is executed. If none of the modules passes the test, the system fails. Apart from the inefficiency of this method, it relies on a highly reliable acceptance test; i.e., an oracle.

The hope with introducing distinct but functionally equivalent units in the system is that these redundancies are easy to systematically introduce, and that these redundancies increase the reliability of the system. The main problem is that redundancies may not result in obtaining ultrareliable software (or increase in reliability at all) when the redundancies cause correlated failures. Showing that redundancies give raise to statistically independent failures is infeasible by testing [19]. Furthermore, the introduction of redundancies has mainly been ad hoc, resulting in considerable human effort in constructing redundancies for specific programs [51].

Result Checking/Data Diversity

Result checking improves software reliability by executing the program under study multiple times with inputs that are randomized over the program's input domain [15]. Consider the following multiplication example given in [12]. Let $\mathcal{P}$ be a multiplication procedure over a finite field. Let $[\mathcal{P}]$ denote the function computed by the procedure $\mathcal{P}$. Suppose that $\mathcal{P}$ keeps no internal state information from execution to execution; i.e., the result of $[\mathcal{P}](x, y)$ depends on x and y only. Suppose that $\mathcal{P}$

fails only on 1/100 of total pairs $< x, y >$. Let $f(x,y)$ be the true multiplication function over the same finite field. Consider the *expanded* formulas given in figure 3, where r_1, r_2 are randomly chosen from a uniform distribution, causing each call to $\mathcal{P}$ to be on a uniformly distributed pairs of inputs. The first expanded formula, after simple algebraic manipulation, collapses to $y = f(x,y)$ ($+$ and $-$ are the usual addition and subtraction functions). If $f = [\mathcal{P}]$ and $+$ and $-$ are the correct addition and subtraction operators, then $y_c = [\mathcal{P}](x,y)$ as well. The probability of failure for y_c in the expanded formula is 4/100 (because there are four calls to $\mathcal{P}$). By computing several outputs $y_c^1, ..., y_c^n$ using new r_1, r_2 for each y_c^i, and choosing the majority of results as a final result, the chance of failure can be made arbitrarily small. This method solves the impractical ultrareliability problem and the operational profile problem. Clearly, 1/100 failure rate could be estimated by practical testing. An operational profile is not needs since the expanded formula involves a uniform distribution of inputs to $\mathcal{P}$ regardless of $\mathcal{P}$'s operational profile, that is, independent of x, y instantiations.

Of critical importance to this method is the relation among outputs based on the random inputs; i.e., the expanded formula in figure 3. Currently, this usually non-trivial relation is in general left to the developer/tester to discover. This relation has been already established for a few well-known problems, such as matrix multiplication, polynomial multiplication, etc. However, no general method has been given for establishing this relation for a general class of problems, such as for the class of all computable functions. Result Checking does not provide a test oracle. Result checking is considered in more detail, and compared to TRM in section "Work Related to TRM."

ADT Program Reliability

An *ADT* is defined as a set of values and a set of operations over these values [39]. ADTs are the basis for the "information-hiding" design philosophy that supports software engineering activities such as maintenance and reuse [59]. Information hiding is an important technique for dealing with the complexity involved in constructing large software systems. ADTs are supported in a number of languages, such as C++, SIMULA, CLU, Java and ADA.

The behavior of an ADT can be characterized by equational specifications [29], which have been used to specify non-trivial problems. In particular, Goguen [39] specifies a database system for an airport scheduler in OBJ-T. Equational specifications have been used to specify programming environments, which include interactive tools such as syntax-directed editors, debuggers, interpreters, code generators, and prettyprinters [43, 9]. Bergstra gives a complete equational specification of the "ToolBus," a coordination architecture that contains concurrency and time [10]. Because of their formal nature, equational specifications are very suitable for rigorous analysis and manipulation.

The following methods consider correctness and testing of ADTs. Theories of ADT correctness have been given in [24, 31, 39, 26]. The problem of correctness of a specification w.r.t. some mathematical model is considered as a problem of finding an isomorphism from the specification to the mathematical model [39]. Correctness of specifications without regard to a particular mathematical model consists of showing that a specification is sufficiently-complete and consistent [31].

Testing of ADTs has been investigated in [25, 40, 23, 29, 27, 52]. The axioms of an axiomatic specification are used to test the consistency of an axiomatic spec-

ification with an implementation [25]. Axioms are compiled into driver programs that make calls to implementation functions with data supplied by a user. This approach addresses the test evaluation problem: the axioms are test oracles.

Rewriting is used as a test oracle by rewriting a term to obtain its expected result [40, 29]. Testcases can be obtained by syntactic manipulation of the axioms [40], rather than by rewriting. Marre deals with generating testcases automatically based on traces [27].

Out of the ADT testing methods, the closest to TRM is the one proposed by Frankl and implemented in ASTOOT [23]. Frankl proposes that each testcase consists of a tuple of sequences of terms, coupled with tags indicating if these terms, when executed, should put the implementation under test into equivalent states or return equivalent results. That is, Frankl notes that it is important to consider the semantic part of the axioms, since different grounding of arguments in a term lead to different abstract states of the specification, resulting in a more comprehensive coverage of states. For example, given a Stack-of-Integers TRS, Frankl would use rewriting to generate the following testcases:

```
(1) (top(push(empty,56)),top(push(push(empty,23),56)), equivalent)
(2) (top(push(empty,56)),top(push(push(empty,78),10)), not-equivalent)
```

Both testcases consists of two terms each, and two tags indicating the expected outcome of the execution of these testcases against the implementation. In testcase (1), the implementation has to be in an equivalent state after the execution of the implementation on the two testcases. In testcase (2), the implementation has to

be in a non-equivalent state after the execution of the implementation on the two testcases. The two testcases could be executed against the Stack-of-Integers implementation, and the resulting states or outputs of the implementation could be checked for equivalence.

The benefits of ASTOOT are the automatic generation of testcases by rewriting and the automatic verification of test results by comparing implementation results. However, since Frankl is concerned with providing a testing methodology, the ASTOOT does not offer any benefits relating to the reliability of the implementation.

CHAPTER III

TERM REDUNDANCY METHOD

In this chapter, TRM is described formally, the analogy between TRM and the related method of Result Checking is established, and the scope of TRM is given. TRM is illustrated on a simple example involving a Stack-of-Boolean-values ADT. (In chapter IV, TRM is illustrated on a more comprehensive example.)

TRM Description

Programs employing Abstract Data Types (ADTs) involve ADT terms (terms in the logic sense) in their code that need to be evaluated as the program executes. For example, consider the C `main` program of figure 4. The `main` program involves the term `top(push(empty,0))`, that needs to be evaluated when program control reaches the `print` statement. This grounded term is obtained from the term `top(push(s,b))` by replacing `s` with `empty`, and `b` by 0. In general, programs involve ungrounded terms, that get grounded with actual values, as `main` reads in

```
main(){
    ...
    s = empty;
    b = 0;
    print(top(push(s,b)));
   }
```

FIGURE 4. Stack-of-Boolean-Values ADT Program

values from the environment. The program that evaluates terms is called an evaluator. In the example of figure 4, the evaluator consists of the implementation of the `top` and `push` procedures together with the mechanism for calling `push` and `top` as directed by terms, such as `top(push(empty,0))`. When a grounded term occurs during program execution, that term needs to be evaluated so that execution of the program can proceed. Thus, ADT programs involve grounded terms that may need to be evaluated ultrareliably. TRM is aimed at obtaining ultrareliable evaluations (probability of giving an incorrect value for a random term is less then 10^{-9}). Of particular interest to TRM is the case in which an evaluator correctly evaluates most of the terms, but fails on "isolated" terms, which testing is very unlikely to "hit". For example, an implementation of a Stack-of-Boolean values might fail only on

```
top(push(pop(push(empty,1)),0))
```

by evaluating it incorrectly to 1, but succeed on every other term.

Equational specifications are expressive enough to specify any ADT, and ADTs are powerful enough to compute any computable function [21]. Equational specifications can be turned into rewrite rules, effectively obtaining a computational model known as a Term Rewriting System (TRS). This conversion is trivial – it involves a syntactic replacement of $=$ by $\rightarrow$. A TRS is essentially an executable (but very inefficient) equational specification. TRM uses the TRS in order to obtain ultrareliable evaluations of ADT terms. In particular, instead of evaluating only the term given to the evaluator (the original term), TRM uses the original term and the TRS specification of the ADT to generate multiple random terms (all equivalent to the original term according to the TRS). TRM then gives these random terms (including the original term) to the evaluator, and returns the majority of evaluator outputs

(if defined) as the result of the original term. This phase of TRM is called the self-checking phase. The `main` program then continues operation with this majority value and eventually produces an output (if it terminates).

The majority values are trusted to the extent of the amount of testing performed prior to the self-checking phase; i.e., more testing gives a higher confidence that the majority value can be trusted. The phase that established this confidence is called the testing phase.

In the testing phase, statistical testing was performed prior to self-checking in order to obtain estimates on the failure rate of the evaluator. This failure rate is not the same as that of evaluating a term by TRM, since TRM involves multiple calls to the evaluator. The failure rate of the evaluator obtained in the testing phase is used in the self-checking phase. TRM assures failure independence, since the distribution of terms in the testing an self-checking phase is the same. For example, suppose that the testing phase estimates probability of failure of an evaluator on a randomly chosen term as less than 10^{-4}. Then the probability of failure agreement of three random evaluator executions on three randomly chosen terms in the self-checking phase is less than 10^{-12}, if the three terms are chosen from the same distribution used in the testing phase. Thus, TRM uses the failure rate estimated in the testing phase, in order to obtain ultrareliable term evaluations in the self-checking phase by sampling from the same distribution in both phases.

In order to obtain ultrareliable term evaluations, TRM needs to sample from the same distribution in both the testing and self-checking phase. Thus, it is of crucial importance that a sound procedure for sampling over terms in both phases is given. How are random terms generated in the self-checking phase? Consider the

TRS rules of figure 5, which is a part of a specification of a Stack-of-Boolean-Values (SBV) ADT. The variable `s` is a SBV variable, and the variable `b` is a Boolean variable. Informally, the $\rightarrow$ symbol means that any term that matches [44] the left-hand side of $\rightarrow$ can be replaced by the right-hand side of $\rightarrow$, but not the other way around. A *Bi-directional TRS* (BTRS) is obtained from the TRS in figure 5 by replacing $\rightarrow$ with $\leftrightarrow$. Informally, the symbol $\leftrightarrow$ means that any term that matches the left-hand side of $\leftrightarrow$ can be replaced by the right-hand side of $\leftrightarrow$, and *vice versa* (details are given in section "Definitions and Notation."

For example, given $t_1 =$ `top(push(empty,0))`, TRM might generate the term

$t_2 =$ `top(push(pop(push(empty,0)),0))`,

in a random fashion as follows. Suppose that a random choice was made to make one replacement to t_1. Let r_1 be chosen randomly from $\{r_1, r_2\}$, which is the set of possible rules that could be applied to t_1. The Bi-directional rule r_1 could be applied to the term `empty` in the $\leftarrow$ direction to obtain either `pop(push(empty,0))` or `pop(push(empty,1))`. Let `pop(push(empty,0))` be chosen randomly from

{ `pop(push(empty,0))`, `pop(push(empty,1))` }.

Replacing `empty` in t_1 with `pop(push(empty,0))` gives t_2. Since the choice of rule as well as the choice of the number of replacements to be made was random, the resulting term t_2 is random over all possible terms that can be reached from t_1 by replacements. (The detailed algorithm for generating random terms from a BTRS is described in section "Testing Phase.") Thus, TRM generates random terms in the self-checking phase by chosing a random number of replacements, and applying a random choice of rules for each replacement to the original term.

$r_1 : pop(push(s, b)) \rightarrow s$
$r_2 : top(push(s, b)) \rightarrow b$
$r_3 : replace(s, b) \rightarrow push(pop(s), b)$

FIGURE 5. Stack of Boolean Values TRS

How are random terms generated in the testing phase? Consider the TRS rules of figure 5, where b is a Boolean variable holding 0 or 1. This specification gives either a value of 0 or 1 for each term. These two values are technically called normal forms. In fact, the specification classifies all the terms into two equivalence classes: the equivalence class of 0, and the equivalence class of 1. In the testing phase, TRM tests both classes individually, generating random testcases by applying $\leftrightarrow$ to 0 and 1 respectively, using the same sampling procedure as in the self-checking phase. Thus, TRM generates random testcases for each equivalence class in the testing phase by starting with a normal form, chosing a random number of replacements, and applying a random choice of rules for each replacement to the normal form.

ADT programs do not take terms as input values, rather they take values of program variables storing (say) sensor readings such as temperature, pressure, etc. These values become parts of terms in the code, in the sense that they replace variables in terms throughout the code as the ADT program executes. Based on these input values, the control of the program may never reach certain terms, and may reach others only with a set of specific program-variable values. Since the distribution of terms evaluated by an evaluator is directly related to the input values given to the program, it seems to be necessary to know the distribution of these input values prior to testing of the evaluator in order for the predictions over terms to be accurate. Fortunately, this distribution does not affect TRM's predictions, since when a term is reached with specific values replacing its variables, a self check is

```
main(){
  ...
  s = empty;
  read(b);
  if(b==0)
    print(top(push(s,b)));
  ...
}
```

FIGURE 6. Term Distribution Dependent on an if Statement

generated from it, whose terms are distributed randomly over the equivalence class of the specific term.

For example, consider the C code in figure 6. The print statement is only reached with $b = 0$, giving raise to the term `top(push(s,b))`. When this term is given to TRM, it generates a self-check from the equivalence class of `top(push(empty,0))`, making the distribution over b irrelevant to TRM's predictions. Even if the user requests the same term c every time, by always selecting the same value for the input variables such as 0 for b in the example, TRM's predictions hold, since the result of c is obtained through terms randomly chosen from the equivalence class of c. Since the requested result of every term is obtained through the results of terms randomly chosen from the equivalence partition of the requested term, the distribution of the requested terms does not affect TRM's predictions.

Definitions and Notation

TRM requires an equational specifications of a decision problem, turned into a TRS, and an implementation of the TRS.

Term Rewriting Systems

A *TRS* is a pair $<\Sigma, R>$ of a signature Σ and a set of rewrite rules R [44]. Let $Ter(\Sigma)$ denote the set of terms over Σ. *Ground terms* are terms with no variables. The set of all grounded terms is denoted by $Ter_g(\Sigma)$. A *rewrite rule* is a pair $<t, s>$, where $t, s \in Ter(\Sigma)$. Rewrite rules are written as $r : t \rightarrow s$, where r is an identification of the rule $t \rightarrow s$. Two conditions must be met:

i) the left-hand side t is not a variable and

ii) the variables occurring in the right-hand side s occur in t.

A *substitution* ϕ assigns terms (values) to the variables in a term. s^{ϕ} denotes the term obtained from s with substitution ϕ. A term with no variables is called a *grounded* term. $c_i \xrightarrow{r} c_j$ denotes a rewrite step. A *rewrite step* means matching [44] the left-hand side of the rule r with c_i and replacing c_i with the right-hand side of r. An expression that can be rewritten is called a *redex*. The relation $\xrightarrow{*}$ is the transitive reflexive closure of $\rightarrow$. That is, $c_i \xrightarrow{*} c_j$ means $c_i = c_j$ or $c_i \rightarrow c_k$ and $c_k \xrightarrow{*} c_j$. $c_i \xrightarrow{*} c_j$ is a *reduction*, and c_j is *reduced* from c_i. c_j is *reachable* from c_i if there exists a reduction from c_i to c_j. A term is in a *normal form* if it cannot be reduced.

Example 1: Let Σ = {`top`,`push`, `pop`,`replace`,`empty`} with arities 1,2,1,2,0 respectively, and let R be the set of rewrite rules given in figure 5, where b ranges over the Boolean value set $\{0, 1\}$, and s ranges over the set SBV. Informally, the operation pop removes the top Boolean value from the stack s, push puts the Boolean value b on the top of the stack s, `top` returns the top element from the stack, and

`replace` changes the top element of s to b. The reduction

$$\texttt{top(pop(push(push(empty,1),0)))} \xrightarrow{r_1} \texttt{top(push(empty,1))}$$

is obtained by applying r_1 to the redex

$$\texttt{pop(push(push(empty,1),0))}.$$

The term `push(empty,1)` is a normal form.

A *non-constructor* is a function that maps the type of interest, such as a Stack-of-Boolean-values, to values that are outside of the type of interest, such as 0 and 1 in the Stack-of-Boolean-value case. For example, `top` is the only non-constructor in Example 1 because its codomain is $\{0,1\}$. A *non-constructor term* is a term that has as its left-most function a non-constructor. TRM considers only non-constructor terms. This is not a restriction on TRM since ADT programs give outputs only when non-constructor terms are evaluated. The set of grounded non-constructor terms is denoted by $Ter_{\overline{g}}(\Sigma)$.

The *result* of a term $c \in Ter_{\overline{g}}(\Sigma)$, denoted by $v(c)$, is the normal form of c, obtained by rewriting c. Let $c_i, c_j \in Ter_{\overline{g}}(\Sigma)$ be two terms. Then, $v(c_i) = v(c_j)$ if c_i and c_j rewrite to the same normal form. Two terms $c_i, c_j \in Ter_{\overline{g}}(\Sigma)$ are *equivalent* (or result-equivalent) iff $v(c_i) = v(c_j)$.

Example 2: Suppose $Ter_{\overline{g}}(\Sigma)$ is the set of grounded non-constructor terms built from `push`, `pop`, `empty`, and `top` over Boolean values, as in example 1. Let the terms $c_i, c_j, c_k \in Ter_{\overline{g}}(\Sigma)$ be:

c_i =`top(push(pop(push(empty,0)),1))`,

c_j =`top(push(empty,1))`, and

c_k =`top(push(empty,0))`.

Then, $v(c_i) = v(c_j)$, and $v(c_i) \neq v(c_k)$.

A TRS is *strongly terminating* if no infinite sequence of reduction steps exists. A TRS is *confluent* if for all reductions $c_i \xrightarrow{*} c_k$ and $c_i \xrightarrow{*} c_m$, there exists a term c_p and reductions $c_k \xrightarrow{*} c_p$ and $c_m \xrightarrow{*} c_p$. A TRS that is strongly terminating and confluent is called *complete* [11]. These definitions are needed since TRM could handle equational specifications that do not have these properties (see section "Specifications").

A *self-check* is a finite set of result-equivalent terms.

The *distance* between c_i and c_j is a subset of $N \cup \infty$ (∞ is greater than every element in N), defined as

$$d(c_i, c_j) = \begin{cases} \{n(c_j \xrightarrow{*} c_i)\} & \text{if } c_j \xrightarrow{*} c_i \\ \{\infty\} & \text{otherwise} \end{cases}$$

where *n(r)* is the number of rewrite steps in the reduction r. The distance set $d(c_i, c_j)$ contains the numbers of reduction steps in all the possible reductions from c_i to c_j.

Implementations

The symbol $\delta(c)$, $c \in Ter_{\overline{g}}(\Sigma)$, denotes the result given by an evaluator δ on input c. An evaluator is a program that returns a value obtained by calling the implementations of the functions that occur in a term. For presentation convenience, no distinction is made between an evaluator and the function computed by an evaluator. Let $c_i, c_j \in Ter_{\overline{g}}(\Sigma)$ be two terms. Then, $\delta(c_i) = \delta(c_j)$ if $\delta(c_i)$ and $\delta(c_j)$ give a syntactically identical value. It is assumed that an evaluator halts on all inputs.

An evaluator may not evaluate every term correctly. For presentation clarity, it is assumed that the codomains of v and δ are identical. In practice this is not true, and the comparison of v and δ involves the difficult mapping from the program codomain to the specification codomain [25]. A *failure* of an evaluator δ on a term $c \in Ter_{\overline{y}}(\Sigma)$ with respect to a TRS is a deviance of the actual and expected result of $\delta(c)$ i.e., $v(c) \neq \delta(c)$. That is, only input/output failures are considered. Other failure types are not considered, such as ones that cause the evaluator to be deleted from memory, or the evaluator not to terminate. *Failure rate* is the probability that an evaluator fails on a randomly chosen input from a particular input distribution.

A *decision evaluator* is an evaluator that involves classifying a set of inputs into a set of outputs, where the cardinality of the output set is small. An example of a decision evaluator is an operating system scheduler, where the input is a particular process and the output is a value in the set containing "running", "blocked", or "ready". Another example is a nuclear reactor shut-down system where the input set contains sensor readings such as temperature, and the output set contains "shut down" or "don't shut down". A more specific decision evaluator is the aircraft collision avoidance system called TCAS [47, 70], considered in detail in chapter IV.

The *majority* of results of $\delta(c_1), ..., \delta(c_m)$ is q if at *least* $m/2+1$ of $\delta(c_1), ..., \delta(c_m)$ equal q, and it is undefined otherwise.

Oracle is a procedure for determining if the outcome of a testcase is as specified; i.e., if $v(c) = \delta(c)$.

Testing Phase

In the testing phase, an evaluator is tested in isolation with terms sampled from each individual equivalence class of the decision specification. Random terms are generated from an equivalence class, the evaluator is executed on these terms, and the success or failure of the execution results is judged. Based on the number of successful testcases executed, the reliability of the evaluator on terms from an equivalence class is estimated. TRM uses a BTRS to generate random testcases.

Bi-directional TRS

TRM turns the equations in the equational specification of an ADT into a BTRS by a simple syntactic change. For example, the equation:

$$\texttt{pop(push(s,b))} = \texttt{s}$$

becomes

$$\texttt{pop(push(s,b))} \leftrightarrow \texttt{s}.$$

The $\leftrightarrow$ relation involves the $\leftarrow$ relation. $c_i \overset{r}{\leftarrow} c_j$ denotes a *reverse* reduction step using rule r. A reverse reduction step $c_i \overset{r}{\leftarrow} c_j$ means matching [44] the right-hand side of the rule r with c_j and replacing c_j with the left-hand side of r. Note that a reverse reduction step may introduce ungrounded variables in terms. $\overset{r}{\leftarrow}$ in $c_i \overset{r}{\leftarrow} c_j$ grounds all the ungrounded variables with randomly chosen values over the appropriate domains. The relation $\overset{*}{\leftarrow}$ is the transitive reflexive closure of $\leftarrow$. That is, $c_i \overset{*}{\leftarrow} c_j$ means $c_i = c_j$ or $c_k \leftarrow c_j$ and $c_i \overset{*}{\leftarrow} c_k$. The reduction $c_i \overset{*}{\leftarrow} c_j$ is called a reverse reduction. c_i is *reachable* from c_j if there exists a reverse reduction from c_j to c_i. The same terminology, such as a *redex*, a *substitution*, etc., that is

used for rewriting, is used for reverse rewriting as well. The relation $\leftrightarrow$ partitions $Ter_{\overline{g}}(\Sigma)$ into a set of result-equivalent classes Π. Two terms c_i, c_j are in the same class $\pi_k \in \Pi$ if $c_i \overset{*}{\rightarrow} c_j$ or $c_j \overset{*}{\leftarrow} c_i$.

Example 3: In Example 1,

$$\texttt{top(push(empty,1))} \leftrightarrow \texttt{top(push(pop(push(empty,0)),1))}$$

since

$$\texttt{pop(push(empty,0))} \overset{[1]}{\leftrightarrow} \texttt{empty},$$

and therefore both,

$$\texttt{top(push(empty,1))} \text{ and } \texttt{top(pop(push(empty,0),1))}$$

belong to the same equivalence class 1 also belongs to.

Bi-directional rewriting makes it possible to reach every term in an equivalence class from every other term in the equivalence class. If used in a random way, bi-directional rewriting can generate a set of random terms over an equivalence class, starting from any given term in the equivalence class.

Random Generation of Testcases

TRM uses *Random Bi-directional Term Rewriting* (RBTR) to generate random testcases from result-equivalent classes. Informally, RBTR is essentially the traditional term rewriting, except rewrites go in *either* direction a *random* number of times, and rules and subterms to be rewritten are chosen at random as well. This informal description is made precise in the RBTR algorithm, given in figure 7. Section "Scope of TRM" discusses values of k for which TRM is practical. The step

$RBTR_k(c_i)$
(1) do k times
(1a) pick a uniform random redex ρ_k in c_i
(1b) generate c_j from c_i by rewriting ρ_s one step
(1c) assign c_j to c_i
(2) return c_i

FIGURE 7. RBTR Algorithm

(1a) of the algorithm uniform-randomly decides which term to rewrite c_i to. At step (1b) of each iteration of the loop, c_i is rewritten to c_j that is one step away from c_i, such that c_j is uniform-randomly chosen from all the terms that can be reached from c_i in one rewrite step (either direction).

In the algorithm, the number k controls the number of loop executions, and determines the number of rewrites performed on c_i. It is obvious that $v(c_i) = v(RBTR_k(c_i))$, that is, $RBTR_k(c_i)$ generates equivalent terms. Furthermore, the rewrite rules are such that every term is reachable from every other term in the equivalence class. That is, the rewrite rules are terminating, and the rules do not under-specify the type under consideration. If the rules are non-terminating, there is always going to be a set of terms that are not going to be reachable from c_i. If the rules under-specify the type, there is going to be some c_j that is not going to be reachable from c_i.

To illustrate RBTR, consider the SBV specification from figure 5, which has two equivalence classes, with normal forms 0 and 1. The equivalence class of 0 is tested by generating random testcases making multiple calls to RBTR(0). For example, let $k = 2$. The set of redexes is $\{0\}$. At step (1a) of the algorithm in the first iteration of the do loop $\rho_s = 0$. c_j is $top(push(empty, 0)$ at step (1b), where

`empty` is a random grounding of `s`. In the second iteration of the loop, the set of redexes is: $\{empty, 0, top(push(empty, 0)))\}$. Let RBTR pick *empty*. c_j becomes $top(push(pop(push(empty, 1)), 0))$, where 1 is a random grounding of b. $RBTR_2(0)$ returns $top(push(pop(push(empty, 1)), 0))$ as a random term from the equivalence class of 0. Similarly, testcases from the equivalence class of 1 are generated by making multiple calls to $RBTR_(1)$.

Test Oracle

TRM executes the evaluator on a set of testcases generated by RBTR by calling the implementation procedures, in the example, `empty`, `top`, `push`, and `pop`. TRM also decides if the evaluator gives the expected outputs. In general, deciding if a test has passed or failed is currently (in practice) done by manual means. Even when formal specifications are available, and an effective pass/fail procedure is in place, testing may still be infeasible, because the expected result of every testcase needs to be computed, in order for it to be compared to the result given by the implementation. Thus, it is important that TRM does not need to compute the expected result of each testcase.

TRM does not need an oracle to determine if the outcome of a testcase is a pass or a failure. This is a result of the way in which TRM generates testcases. Let ν be a normal form for some class of terms. A testcase from the equivalence class of ν is a success if $\delta(RBTR(\nu)) = \nu$; otherwise, it is a failure. Therefore, no computation is needed to determine the expected outcome of evaluating a term. This fact adds to the overall efficiency of TRM in obtaining ultrareliability.

Self-Checking Phase

Self-checking involves generating a self-check from a term using RBTR, executing the evaluator on the self-check, and returning the majority of execution outcomes. Self-checking can obtain ultrareliability given that testing with a practical number of testcases has been performed prior to the checking phase.

Reliability Estimates

Let δ be an evaluator and $< \Sigma, R >$ be a TRS. First it is shown how to obtain statistical estimates on the failure probability of δ on an arbitrary term in an equivalence class π_i over $Ter_{\overline{g}}(\Sigma)$. Then, these estimates are used to state a probability that N executions of δ on random terms chosen from π_i will fail. These estimates are adjusted for RBTR in this section under "Reliability Adjustment."

Suppose the given are: constant failure rate θ of π_i, and n random terms drawn from π_i, executed on δ without failure. The probability that δ fails on a randomly chosen term from π_i is θ, and $1 - \theta$ that it will succeed. Given that the n terms are independent, the probability that δ succeeds on all the terms is $(1 - \theta)^n$. The confidence bound α on θ is defined as the probability that the failure rate of δ is below θ. The confidence bound is related to the testset size n and the failure rate θ by

$$\alpha \leq 1 - (1 - \theta)^n. \tag{III.1}$$

Formula III.1 could be used to estimate the confidence bound on the failure rate of δ on a majority of N random terms generated by RBTR. Suppose a successful test (no failures occurred during the test) of δ on n terms is conducted at test time. Half of N terms (majority) falsely agreeing at run-time gives a failure rate of at least

$(N/2n)$. Therefore, substituting $N/2n$ for θ in equation III.1 yields

$$\alpha \leq 1 - (1 - (\frac{N}{2n}))^n. \tag{III.2}$$

The meaning of the confidence bound is the probability that the failure rate is below $N/2n$ for a repetition of the test. For example, $1 - \alpha = 6.0 \times 10^{-8}$ with $N = 33$ and $n = 10^4$. As the example shows, TRM's ultrareliable predictions do *not* need the infeasible amount of testcases that are needed in [19]. Formula III.2 allows a tradeoff between amount of testing time (function of n) and execution time penalty (function of N) for a particular failure rate and confidence bound. That is, one has the flexibility of doing less testing, but executing the program more times when a result is requested.

Unfortunately, as the number of result-equivalent classes grows, TRM becomes impractical, because TRM tests each class independently. Therefore, an additional constraint is added to formula III.2 to obtain:

$$\alpha \leq 1 - (1 - (\frac{N}{2n}))^n \wedge Q * n < 10^4, \tag{III.3}$$

where Q is the number of equivalence partitions. The constant 10^4 is what is considered a practical number of testcases.

Self-Checking Algorithm

TRM conducts testing of equivalence classes through a procedure *Test*, given in figure 8, which is essentially an implementation of Formula III.2. N is the size of the self-check, R is a set of rewrite rules, α is a confidence bound, and Υ is a variable

```
Test (N, α, R, Υ)
      i := solve(α ≤ 1 − (1 − (N/2n)^n)
      for j := 1 to size(Υ) do
                  ν := Υ_j
                  do i times
                     C := RBTR_k(ν)
                     if (ν ≠ δ(C)) then return fail
                  endo
      endfor
return pass
end

Check (N, C, δ)
        M = ∅
        do N times
           M := M ∪ δ(RBTR_k(C))
        endo
        return majority(M)
end
```

FIGURE 8. TRM Testing and Checking Algorithms

that ranges over an indexed set of normal forms that characterize the equivalence classes. Υ_k picks the j-th element of set Υ. In the algorithm, n is the testset size (number of terms).

The function

$$solve(\alpha \leq 1 - (1 - (N/2n)^n)$$

returns the largest n that is a solution to

$$\alpha \leq 1 - (1 - (N/2n)^n.$$

If $Test(N, \alpha, R, \Upsilon)$ returns fail, then the failure is corrected in δ, and the test is repeated. If $Test(N, \alpha, R, \Upsilon)$ returns pass, then δ has the desired reliability, with confidence α.

TRM conducts self-checking through a procedure $Check(N, C, \delta)$. The procedure $Check(N, C, \delta)$ collects results of N random executions of δ in the set M, and it returns the majority of M. The confidence bound on the majority given by $Check(N, C, \delta)$ depends on α and N in the $Test(N, \alpha, R, \Upsilon)$.

Reliability Adjustment

The $RBTR_k(t)$ procedure generates non-uniform random terms from a set of terms that can be reached from t in k rewrite steps. Furthermore, the term distribution obtained using $RBTR_k(t)$ depends on t. That is, when $RBTR_k$ starts from different terms, the distribution obtained using $RBTR_k$ may vary. Therefore, the reliability estimates, as given in formulas III.2 and III.3 in this section under "Reliability Estimates," may not hold. This section shows how to adjust the reliability

estimates given by these two formulas, so that the readjusted formulas reflect the fact that RBTR samples non-uniformly.

$RBTR_k(t)$ samples some terms more often than others; however, every term is going to be sampled with some probability p_{min} or greater regardless of t. This probability can be calculated, and it can be used to adjust the size of a self-check in order to obtain ultrareliability.

For example, let $RBTR_k(\nu)$, where ν is a normal form, be used to generate 10^4 test terms. The evaluator probability of failre on those terms is roughly 10^{-4}. At self-checking time, $RBTR_k(t)$ might not sample terms with the same probability as $RBTR_k(\nu)$ did, and the product failure rate of $10^{-4} \times 10^{-4}$ might not hold for a self-check size of two. The worst case occurs when the $RBTR_k(t)$ choses terms that had chance of selection in the testing phase of p_{min} (say $p_{min} = 1/10$). That is, the self-checking terms had a low probability of being selected in the testing phase. In the example, out of the 10^4 terms, roughly 10^3 hit terms with selection probability p_{min} ($10^4 \times p_{min} = 10^3$), giving roughly a failure probability on those terms of 10^{-3}. A self-check of size three would give failure probability on three terms of $10^3 \times 10^3 \times 10^3$. Thus, ultrareliability is obtained by a self-check size of three, one more than in the case with no adjustement.

What is the least probability p_{min} with which $RBTR_k(t)$ is going to sample any term, and how is it used to adjust the self-check size?

Consider the RBTR algorithm from figure 7. At the end of each iteration of the do loop, $RBTR_k(t)$ would have chosen a term that has the same probability of being selected as all the other terms that can be reached from t in *one* step. Each term t has a number of terms i_t that rewrite *to* t in one $\rightarrow$ step. Each term

t has a number of terms o_t that rewrite *from* t in one $\rightarrow$ step. The fan-in of a term t, denoted by fi_t, is $i_t/(i_t + o_t)$, and the fan-out of a term t, denoted by fo_t, is $o_t/(i_t + o_t)$. Let f_{min} be the smallest fan-in or fan-out of any term in the term space covered by $RBTR_k(\nu)$. $RBTR_k(t)$, for any t, samples terms with probability greater than $(f_{min})^k$, that is $p_{min} \geq f_{min}^k$.

For example, in the SBV specification of figure 5, given rules r_1 and r_2, f_{min} is 1/3, because each term has i_t of 1 and o_t of 2, giving fan-in of 1/3 and a fan-out of 2/3. Given $k = 4$, $p_{min} \geq 1/100$.

Informally, the probability p_{min} gives the worst case sampling by RBTR. That is, there exists no term that RBTR can sample with probability less then p_{min}. In the worst case, RBTR would pick terms with probability of selection close to p_{min} in the self-checking phase. This worst case can be bounded, and an increase in the self-check size can correct the worst case, as described next.

A testset of size n, generated by $RBTR_k(\nu)$, is going to hit terms with probability of selection p_{min} at least $n \times p_{min}$ times. Let the probability of failure of a random term with a probability of selection p_{min} be p, based on $n \times p_{min}$ samples. The self-check size N is determined by $p^N < 10^{-8}$. For example, let $n = 10^4$ and let $p_{min} = 1/100$. At least 10^2 terms chosen by $RBTR_k(\nu)$ are of selection probability p_{min}. Then p is roughly 10^{-2}. In order to obtain probability of failure 10^{-8}, N needs to be at least 5.

Scope of Term Redundancy Method

TRM gives ultrareliable evaluation results with a practical number of testcases; however, in practice, the time to obtain these testcases might be prohibitively long

due to the large number of rewrites necessary to sample from the whole term space. In this section, constraints on the number of rewrite steps are given, which make TRM applicable in practice. Also, this section considers equational specification properties, such as termination, needed for TRM to function properly.

Efficiency Constraints

Unlike life testing [19] of evaluators, which would require testing an evaluator with an infeasible number of terms, TRM requires a practical number of testcases. However, care needs to be taken to assure that the actual generation of testcases is not prohibitively long. That is, in order for TRM to be used in practice, the number of bi-directional rewrites in the testing and self-checking phases needs to be bounded by a number that is considered practical. This constraint restricts the sample space w.r.t. term length (the number of operators involved in the terms). For example, if a rule introduces m operators in a term, then k applications of that rule (starting from a normal form) would result in a term of length km. The testing and the self-checking phases of TRM are carried out in such constrained term sample spaces. In the testing phase, $RBTR_k(\nu)$ is used to generate test cases, where k is a practical number of rewrite steps. In the self-checking phase, $RBTR_k(c_i)$ is used as well with k set to the same value as in the testing phase. However, TRM need to make sure that c_i is in the sample space, as well as that $RBTR_k(c_i)$ stays within the sample space, as determined by term length. If c_i is within the sample space, TRM self-checks c_i and it returns its result; otherwise, it returns "term is out of range, don't trust its result." Next, the above discussion is made more precise.

Suppose k rewrite steps are considered practical. Let ν_k denote the set of

terms that can be reached from the normal form ν in k steps. Let $max(\nu_k) \subset \nu_k$, be the set of terms that involves the maximum number of operations; that is, the maximum number of operations that a term could possibly have when k rewrite steps are applied to ν. For example, in the TRS of figure 5, 1_2 is

{`top(push(empty,1))`, `top(push(pop(push(empty,0)),1))`,
`top(push(pop(push(empty,1)),1))`,
`top(push(empty,top(push(empty,1))))` }.

The set $max(1_2)$ is

{`top(push(empty,top(push(empty,1))))`,`top(push(pop(push(empty,0)),1))`,
`top(push(pop(push(empty,1)),1))` }.

Let l be the number of operations in any $max(\nu_k)$ term (if more than one, all of them have the same l). In the example, $l = 4$. The count l is algorithmically obtained by scanning the list of rules and finding the rule(s) r_{max} that introduces the most operations, (say) q operations. Then, $l = qk$. In the example, if $k = 100$, then $r_{max} = \{r_1, r_2, r_3\}$, $q = 2$, and $l = 200$. Informally, if TRM starts from a normal form and applies $RBTR_{100}$ to it, the maximum length term that $RBTR_{100}$ could return would have 200 operations in it. TRM uses l in the following way. Let $o(c_i)$ be the number of operations in a term c_i. In the self-checking phase, when given a term c_i, TRM checks if $l < o(c_i)$. If so, TRM returns "term is out of range, don't trust its result." Otherwise, TRM self-checks c_i.

$RBTR_k(c_i)$ is a special case of $RBTR(c_i)$ described in section "Testing Phase." $RBTR(c_i)$, as shown in section "Testing Phase," returns a random term from the equivalence class of c_i. $RBTR_k(c_i)$ returns a random term from the set of terms that are at most k-distant from c_i.

The restriction of TRM to a small k dictates a trade off between the number of random rewrites, the time it takes to generate a self check, and the number of operations in a random term. Given a maximum practical time T to generate a random term c_i, and a time t for a random rewrite, the following simple relation exists between k, T, t, and l. The average number of rewrites is $k/2$, and the average number of operations in c_i is $l/2$. Then, $k \leq 2T/t$, and $l \leq (2T/t)q$, where q is the maximum number of operations that a rewrite step introduces in c_i.

Thus, TRM could be used in practice if the number of rewrites is restricted to some small number k, by testing and self-checking terms in the range of terms that k spans.

Specifications

TRM is intended to be used on evaluators whose behavior is captured by a set of equations. The only constraint on these equations is that they are terminating, that is, when the equations are turned into a traditional rewriting system, the TRS is strongly terminating. This constraint is necessary so that every term is reachable from every other term in an equivalence class. However, the TRS need not be confluent or consistent. Furthermore, the TRS need not (and due to complexity might not) be correct. As a result, obtaining results directly from these TRSs might not be possible (in the non-confluent and inconsistent cases), and/or the results might not be correct (in the case of an incorrect TRS).

Non-complete TRSs are essentially useless for obtaining results of terms directly, because in such systems there are multiple normal forms for the same term and/or one may never obtain a normal form because of infinite reductions, so one

cannot simply use the equational specification directly to obtain results. TRM can use a non-complete specification in both the testing and self-checking phases. TRM could generate terms through RBTR that are equivalent, and TRM could then check the consistency of the evaluator with the non-complete/incorrect specifications. However, if the evaluator is incomplete and/or incorrect in the same way the specification is, TRM would not be able to detect this problem. That is, it could happen that the specification has the same faults as the implementation; however, this should be unlikely due to the distinct nature of the specifications and the implementations.

An example will demonstrate how the termination and confluence difficulties in TRSs affect TRM.

Suppose the SBV of chapter III were specified as follows:

r_1 : `topx(replacex(s,b))` $\rightarrow$ `b`

r_2 : `replacex(s,b)` $\rightarrow$ `pushx(popx(s),b)`

The system is not confluent because

$$\texttt{topx(replacex(pushx(s,1),0))} \xrightarrow{r_1} 0$$

and

$$\texttt{topx(replacex(pushx(s,1),0))} \xrightarrow{r_2} \texttt{topx(pushx(popx(pushx(popx(s),1)),0))},$$

but

$$\texttt{topx(pushx(popx(pushx(popx(s),1)),0))} \not\rightarrow 0.$$

This specification suffers from underspecification, since there is no rule to bring the term

topx(pushx(popx(pushx(popx(s),1)),0))

to a normal form. Thus, if traditional rewriting is used on this system, there would be two results for the term topx(replacex(pushx(s,1),0)), but the two results cannot be shown to be equivalent by traditional rewriting using the non-confluent specification above. Therefore, one does not know if the system is consistent or not, and results given by such a system have no meaning.

Whereas non-confluent systems cause rewriting to produce multiple results for a term, non-terminating systems cause rewriting never to reach a normal form. Thus, in non-terminating systems, rewriting does not produce a value for some terms. For example, consider the SBV of chapter III, specified as follows:

r_1 : topy(replacey(s,b)) $\rightarrow$ topy(replacey(popy(pushy(s,b),b))

r_2 : popy(pushy(s,b)) $\rightarrow s$

r_3 : replacey(s,b)$\rightarrow$ pushy(popy(s),b)

This system is non-terminating since

replacey(pushy(empty,1),0) $\xrightarrow{r_1}$ relpacey(popy(pushy(pushy(empty,1),1)))

$\xrightarrow{r_1}$ ....

Deciding if a system if confluent and terminating is undecidable in general.

TRM deals with non-confluent specifications by simply ignoring the meaning of the equations when converted to BTRSs, and generating terms through RBTR. For example, RBTR could generate in the non-confluent case:

```
{topx(replacex(pushx(empty,1),0)),
topx(pushx(popx(pushx(popx(empty),1)),0)) }
```

In non-confluent systems, RBTR can generate random terms from the set of terms related by $\leftrightarrow$ starting from any term, because if RBTR is given any term, RBTR can reach every other term that is related to it by $\leftrightarrow$. Unfortunately, in non-terminating systems, RBTR cannot generate random terms from the set of terms related by $\leftrightarrow$ starting from any term, since the starting term given to RBTR could be such as to keep RBTR from reaching some terms that are related to it by $\leftrightarrow$.

Thus, TRM can be used when the TRS is non-confluent, inconsistent, and incorrect, as well as on correct TRSs.

Work Related to TRM

TRM has been inspired by the development of Result checking [13, 2]. TRM is related to the certification trail method [30] as well.

There is an interesting analogy of TRM to Blum's result checking. Consider the expanded multiplication formula from figure 3:

$$y_c = [\mathcal{P}](x - r_1, y - r_2) + [\mathcal{P}](x - r_1, r_2) + [\mathcal{P}](r_1, y - r_2) + [\mathcal{P}](r_1, r_2),$$

where r_1, r_2 are randomly chosen from a uniform distribution. The right-hand side of the expanded formula is a term over $\mathcal{P}, +, -$. This term gives rise to a set of grounded terms when the variables x, y, r_1, r_2 in the term get grounded to particular constant values. The "=" relation partitions the set of grounded terms into equivalence

classes. For example,

$$[\mathcal{P}](2-1,3-2)+[\mathcal{P}](2-1,2)+[\mathcal{P}](1,3-2)+[\mathcal{P}](1,2)$$

and

$$[\mathcal{P}](2-2,3-1)+[\mathcal{P}](2-2,1)+[\mathcal{P}](2,3-1)+[\mathcal{P}](2,1)$$

belong to the same class 6 belongs to. TRM partitions terms into equivalence classes as well, making the two methods the same in that respect. The departure of TRM and Blum's approach is that Blum does not restrict his method to any particular number of partitions. TRM on the other hand is restricted to a small number of partitions. His method reduces to computing the program on random inputs (terms) such as $< 2-1, 3-2 >$, etc. TRM executes the evaluator δ on random inputs (terms) as well. The difference is that Blum's inputs come from a uniform distribution over the *whole* input domain, while δ's inputs come from a uniform distribution over a set of *equivalent* terms (inputs). Since TRM starts with a particular term in the class, it could not be applied to an overall sample. The correspondence between result checking and TRM is thus easily established as follows. Blum executes the multiplication procedure multiple times at execution time on uniformly-random inputs. TRM executes δ on uniformly-random terms multiple times as well. The difference is that Blum does not restrict his method to decision problems as TRM does.

Blum suggests that the low reliability involving the 1/100 failure rate can be obtained by practical testing, but he does not specify the test distribution over which the testing was conducted in order to estimate the reliability of $\mathcal{P}$. If the testing were

conducted from a distribution different than uniform, his predictions would not hold since the expanded formula involves uniformly distributed inputs in the calls to $\mathcal{P}$. For example, suppose $\mathcal{P}$ was tested on inputs chosen according to the distribution which selects each of $< 1,1 >, < 2,2 >, < 3,3 >, < 4,4 >$ and $< 5,5 >$ with probability of 1/5. Suppose $\mathcal{P}$ was tested by drawing testcases from this distribution, and it showed failure rate of 1/5. In particular, let $\mathcal{P}$ fail on $< 1,1 >$ and let it succeed on the rest of the inputs, that is $f(1,1) \neq [\mathcal{P}](1,1)$ and $f(i,i) = [\mathcal{P}](i,i)$ for $i = 2,3,4,5$. Furthermore, for presentation convenience, suppose $\mathcal{P}$ fails on all other inputs in $\mathcal{P}$'s domain. (Testing according to the distribution above is not going to show this, of course). Since $\mathcal{P}$ is only correct on four input pairs, the probability of $\mathcal{P}$ being correct on a uniformly-randomly chosen input over the whole domain is almost 0. If r_1, r_2 in y_c are chosen from a uniform distribution, each $\mathcal{P}$ has a chance of failing of almost 1 and not 1/5 as predicted by testing, and the probability of failure of y_c is almost 1, and not 4/5. Thus, Blum has to test $\mathcal{P}$ according to a uniform distribution.

Self checking, as used by Blum, is applicable to certain clean mathematical functions [14], such as integer multiplication, modular multiplication, matrix multiplication, inverting matrices, computing the determinant of a matrix, computing the rank of a matrix, integer division, modular exponentiation, and polynomial multiplication. Rubinfeld [66] has shown that functions which satisfy robust functional equations are suitable for self checking. Veinstain [69] has shown that a large class of functions that satisfy polynomial functional equations are suitable for self checking. Lipton [49] shows that multivariate polynomials over finite fields are random self-reducible (a property that allows self checking). Rubinfeld [62] has shown that

polynomials are random self-reducible over more general domains such as rationals and non-commutative rings. Lipton shows how to efficiently check computations with polynomial-time verifiers. Blum [16] shows how self checking could be applied to processor arithmetic operations.

Data diversity [2, 1], which was proposed independently of result checking, could be looked at as an attempt to apply result checking more complex problems than those considered above. For example, Ammann [2] considers an antimissile missile launch decision program which involves differential equation solvers. Ammann also showed how to build an arbitrarily reliable *sine* function.

TRM is related to result checking and data diversity in that TRM generates multiple program inputs and checks if the expected relation (in our case equality) holds between program outputs. However, TRM applies to a different class of problems than the ones addressed by the self checking and data diversity approaches. TRM algorithmically derives the expected relation between program outputs form the equational specification alone, without any need for an outside assistance. Both self checking and data diversity rely on the user to supply this usually non-trivial and potentially error prone relation (a relation such as the formula 3. Furthermore, TRM does not need a test oracle, while a test oracle is needed in the self checking and the data diversity approaches.

The difference between the certification-trail method [30](as described in chapter II) and TRM is that the certification-trail method is code based, and TRM is specification based; i.e., the certification-trail method needs a particular encoding for each implementation of a particular specification whereas TRM is not tied to a particular implementation of the specification. Furthermore, the certification-trail

method is not probabilistic, and it involves executing the program only once.

CHAPTER IV

TRM AND BOOLEAN ADTS

This chapter illustrates TRM's effectiveness in failure detection and recovery. The following paragraphs outline the work, which is described in detail in the sections to follow.

To illustrate TRM's effectiveness, a set of Boolean formulas are taken from TCAS II, an aircraft collision-avoidance system. In a program implementing TCAS II, these formulas would need to be evaluated by routines that perform Boolean arithmetic. The Boolean evaluation can be specified by a set of rewrite rules. When the Boolean-term evaluator is given a TCAS II Boolean formula, TRM can be used to self-check the Boolean-term evaluator using the Boolean-term evaluator specification.

Instead of implementing a Boolean-term evaluator and introducing faults in the implementation for TRM to detect, this illustration introduces faults in the TCAS formulas themselves. Thus instead of selecting random equivalent terms for the possibly faulty evaluator to evaluate, possibly faulty terms are sent to a correct evaluator. The result should be the same: there is a possibility that that evaluator will return incorrect results.

The effectiveness of TRM is measured as the probability that the majority of values obtained by self-checking the Boolean-term evaluator implementation falsely agree.

TCAS Boolean ADT

Boolean-term Evaluator Inputs

Leveson et al. [48] use an AND-OR table representation of the conditions for state transitions to specify pats of TCAS II, an aircraft collision avoidance system. The state-transition conditions are Boolean formulas in five to 14 variables, with the average containing ten distinct variables. Weyuker et al. [70] use the Boolean formulas from TCAS II as a basis for evaluating the effectiveness of their method for automatically generating test data from Boolean specifications. They used ten of the larger Boolean TCAS II formulas in evaluating their method. TRM used the same ten Boolean formulas given in [70], and shown here in figure 9, as a basis for illustrating the effectiveness of TRM in failure detection and recovery. These formulas when grounded become inputs to a Boolean-term evaluator.

Each TCAS formula from figure 9 was grounded (values given for variables) once to obtain a total of ten grounded TCAS formulas. (The grounding was random, chosing uniformly from the set $\{0, 1\}$ for each varable in each TCAS formula.) Let the formulas from 9 be called *original formulas*, and let the original formulas with the random grounding be called *grounded* original formulas.

Boolean-term Evaluator Specifications

The Boolean-term evaluator is specified with the equational specification of figure 10. The original formulas represent various combinations of conditions that occur in the aircraft collision avoidance system. The equational specification of figure 10 gives meaning to these formulas. In particular, the equations assign a value of 0 or 1 to each of the ten formulas, depending on the particular grounding

1. $\overline{(ab)}(d\overline{e}\overline{f}+\overline{d}e\overline{f}+\overline{d}\overline{e}\overline{f})(ca(d+e)h+a(d+e)\overline{h}+b(e+f))$

2. $(a((c+d+e)g+af+c(f+g+h+i))+(a+b)(c+d+e)i)\overline{(ab)}\ \overline{(cd)}\ \overline{(ce)}\ \overline{(de)}\ \overline{(fg)}\ \overline{(fh)}\ \overline{(fi)}\ \overline{(gh)}\ \overline{(hi)}$

3. $(a(\overline{d}+\overline{e}+de\overline{(\overline{f}gh\overline{i}+\overline{g}hi)(\overline{f}glk+\overline{g}\overline{i}k)})+\overline{(\overline{f}gh\overline{i}+\overline{g}hi)}\ \overline{(\overline{f}glk+\overline{g}\overline{i}k)}(b+c\overline{m}+f))(a\overline{b}\overline{c}+\overline{a}b\overline{c}+\overline{a}\overline{b}c)$

4. $a(\overline{b}+\overline{c})d+e$

5. $a(\overline{b}+\overline{c}+bc\overline{(\overline{f}gh\overline{i}+\overline{g}hi)}\ \overline{(\overline{f}glk+\overline{g}\overline{i}k)})+f$

6. $(\overline{a}b+a\overline{b})\overline{(cd)}(f\overline{g}\overline{h}+\overline{f}g\overline{h}+\overline{f}\overline{g}\overline{h})\overline{(jk)}((ac+bd)e(f+(i(gj+hk))))$

7. $(\overline{a}b+a\overline{b})\overline{(cd)}\ \overline{(gh)}\ \overline{(jk)}((ac+bd)e(\overline{i}+\overline{g}\overline{k}+\overline{j}(\overline{h}+\overline{k})))$

8. $(\overline{a}b+a\overline{b})\overline{(cd)}\ \overline{(gh)}((ac+bd)e(fg+\overline{f}h))$

9. $\overline{(cd)}(\overline{e}f\overline{g}\ \ \overline{a}(bc+\overline{b}d))$

10. $a\overline{b}\overline{c}d\overline{e}f(g+\overline{g}(h+i))\overline{(jk\overline{j}l+m)}$

FIGURE 9. TCAS Specifications

$$
\begin{aligned}
&r_1 : 0 + x = x \\
&r_2 : 1 + x = 1 \\
&r_3 : 0 \cdot x = 0 \\
&r_4 : 1 \cdot x = x \\
&r_5 : \overline{0} = 1 \\
&r_6 : \overline{1} = 0 \\
&r_7 : (x + y)z = xz + yz \\
&r_8 : \overline{(x + y)} = \overline{x} \cdot \overline{y} \\
&r_9 : \overline{(xy)} = \overline{x} + \overline{y} \\
&r_{10} : (y + x) + z = y + (x + z) \\
&r_{11} : (yx)z = y(xz) \\
&r_{12} : y + (xz) = (y + x)(y + z)
\end{aligned}
$$

FIGURE 10. Boolean ADT Specification

of its variables.

Boolean-term Evaluator Implementation

In the `main` program implementing TCAS II, the grounded original formulas would need to be evaluated. In particular, as the `main` program executes, the original formulas become grounded, and would need to be evaluated. Let the program that would evaluate grounded TCAS formulas be called a TCAS evaluator. For the purposes of this illustration, the TCAS evaluator is a correct implementation of the Boolean ADT, specified with the equational specification of figure 10, and as such evaluates every grounded TCAS formula correctly. The TCAS evaluator is implemented as a correct TRS with the rules of figure 10.

TCAS Experiment

The Fault Model

The TRM effectiveness demonstration is based on the *Operator Reference Fault (ORF)* [20, 41, 42, 70, 22, 36], which involves replacing a Boolean operator with another one, or omitting a negation operator from a grounded original formula. ORF faults could plausibly arise in a variety of situations [70].

ORFs are systematically introduced into the grounded original formula by applying ORF-seeded specification rewrite rules to the grounded original formula. ORFs are exhaustively introduced in rules $r_7 - r_{12}$ of BBTRS. For example, rule r_7 gives raise to the set of mutated rules given in figure 11. That is, from each rule $r_7 - r_{12}$, a set of mutated rules was generated by single mutations; i.e, by mutating only one operator. There are a total of 27 mutated rules w.r.t. ORF and rules

$(xy)z \leftrightarrow xz + yz$
$(x + y) + z \leftrightarrow xz + yz$
$(x + y)z \leftrightarrow xzyz$
$(x + y)z \leftrightarrow x + z + yz$
$(x + y)z \leftrightarrow xz + y + z$

FIGURE 11. Exhaustive Mutated r_7 Rules

$r_7 - r_{12}$. Introducing ORF in the specification rewrite rules instead of in the TCAS evaluator effectively simulates faults in the TCAS evaluator. For example, when the mutated rule $(xy)z \leftrightarrow xz + yz$ is used to generate random equivalent terms, the application of this rule to a term effectively simulates a fault in the TCAS evaluator that is characterized as incorrect evaluation of $(xy)z$ to $xz + yz$.

Self-checks are generated by applying ORF mutated rules to grounded TCAS II formulas. Based on these faults, the correct Boolean evaluator may either obtain the correct Boolean value or it may not. Using a correct program on incorrect inputs is a convenient fault abstraction — Ammann [3] also uses fault abstraction for an experimental validation of his data redundancy technique on a differential equation solver.

In order for this illustration to match the TRM theory, the random formulas generated from a particular grounded TCAS formula have to be equivalent according to the Boolean ADT specification. What is the relation between the outputs of the TCAS evaluator when given random formulas generated by faulty rewrite rules? If the faults were in the TCAS evaluator instead of in the TCAS formulas, the rewrite rules would not need to be incorrect to simulate these evaluator faults, and the correct rewrite rules would always generate equivalent formulas. Therefore, effectively the faulty random formulas are equivalent — it is only the TCAS evaluator faults

(captured by incorrect rewrite rules) that may make these formulas non-equivalent.

Self-check Generation

For presentation purposes, TRM was restricted to generate self-checks of random TCAS formulas at distance *one* from the TCAS formulas given to TRM. For each of the grounded original formulas, TRM generates a set of ten random equivalent (to the grounded original formula) formulas using $RBTR_1$ on a set of mutated rules. In order to apply $RBTR_1$ to the grounded original formula, TRM needs a Boolean bi-directional TRS.

A *Boolean TRS* (BTRS) is a pair $\langle \Sigma, R \rangle$ of a signature Σ and a set of rewrite rules R, where $\Sigma = \{+, \cdot, -, 0, 1\}$. The binary operators $+$ and $\cdot$ represent the "or," and "and," Boolean operators, whereas the unary operator $\overline{b}$ represents the "not," Boolean operation applied to b. The constants 0 and 1 denote "false" and "true," respectively. R is obtained from the equations in figure 10 where "=" is replaced by "$\rightarrow$".

$Ter_g(\Sigma)$ denotes all the grounded Boolean terms, such as for instance $1 \cdot 1 + 0 \cdot \overline{1}$ and $(0 + 1)1$. The *result* of a term $c \in Ter_g(\Sigma)$, denoted by $v(c)$, is either 1 or 0, and is obtained by rewriting c to its normal form. Two terms $c_i, c_j \in Ter_g(\Sigma)$ are *equivalent* (or result-equivalent) iff $v(c_i) = v(c_j)$. v partitions $Ter_g(\Sigma)$ into two equivalence classes:

$$\{x \in Ter_g(\Sigma) | v(x) = 1\}$$

and

$$\{x \in Ter_g(\Sigma) | v(x) = 0\},$$

$$b_1 : 1 + 0 \leftrightarrow 1$$
$$b_2 : 1 + 1 \leftrightarrow 1$$
$$b_3 : 0 \cdot 1 \leftrightarrow 0$$
$$b_4 : 0 \cdot 0 \leftrightarrow 0$$
$$b_5 : 1 \cdot x \leftrightarrow x$$

FIGURE 12. Boolean Bi-Directional TRS

denoted by T and F, respectively. For instance, $0 \cdot 1 \in F$, $(1 \cdot 0) + \bar{1} \in T$, and $(1 \cdot 0) + 1 \in T$.

A *Boolean Bi-directional TRS* (BBTRS) is a pair $\langle \Sigma, R^b \rangle$ of a signature Σ and a set of rewrite rules R^b, where $\Sigma = \{+, \cdot, -, 0, 1\}$. R^b is obtained from the equations in figure 10, where "=" is replaced by "$\leftrightarrow$", and the rules from figure 12 are added, where x, y and z are Boolean variables. The rules r_2 and r_3 in BTRS have four corresponding rules b_1, b_2 and b_3, b_4 in BBTRS respectively. This is necessary since $1 + x \leftarrow 1$ and $0 \cdot x \leftarrow 0$ introduce ungrounded variables if present in BBTRS. Introducing additional rules to eliminate generation of ungrounded variables is possible in this case because there are only two situation to consider. With the additional rules in place, no rule in BBTRS introduces ungrounded variables, when applied in either direction. In general, when no such rules could be conveniently added, when BBTRS introduces an ungrounded variable, that variable is grounded.

Effectiveness Criteria

Based on the simulated faults, the results of the TCAS evaluator on random formulas from the self-check may or may not agree on the correct Boolean value. Furthermore, some results may agree on one value, and other results on another value. The worst situation — the case where neither failure detection nor correction

Spec.	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	Correct	Failure
1	13	21	20	15	15	13	83.50	1.49×10^{-8}
2	26	60	30	23	27	10	71.59	3.4×10^{-6}
3	30	50	47	25	55	26	99.64	3.6×10^{-25}
4	0	0	0	0	0	0	100	0
5	11	22	21	9	22	9	94.68	1.7×10^{-13}
6	21	21	18	17	31	19	92.75	3.7×10^{-12}
7	14	19	30	12	20	13	91.66	1.5×10^{-11}
8	11	10	19	10	17	10	93.50	6.5×10^{-13}
9	6	9	13	7	12	7	79.62	1.0×10^{-7}
10	10	12	12	4	20	3	98.36	1.6×10^{-29}

FIGURE 13. TRM Effectiveness

is possible — is when the majority of results agree on a false Boolean value. The effectiveness of TRM on TCAS is measured as the probability of the TCAS evaluator's results falsely agreeing when evaluating ten randomly generated formulas. The demonstration of effectiveness is implemented in CLP(R) [38, 35], and all the code necessary is given in the Appendix. CLP(R) was chosen for its convenience in manipulating rewrite rules.

Effectiveness Results

The effectiveness of TRM in detecting and correcting ORF in TCAS is measured as the probability that the majority of twenty uniformly randomly chosen formulas from the equivalence class of an grounded original formula agree on the incorrect Boolean value. The effectiveness results are summarized in figure 13. The column labeled "Spec" shows the TCAS formula used, and columns labeled $r_7 ... r_{12}$ give the number of possible mutant formulas that can be generated by all the possible ORF mutants of the rules $r_7 - r_{12}$.

The percentage of mutated formulas that evaluated to the same Boolean value as the grounded original formula (from which these mutated formulas were created) are given in the column labeled "Correct". The column labeled "Failure" shows the probability of ten uniformly-randomly chosen Boolean mutants agreeing on the false Boolean value (the one that is different from the grounded original formula from which these mutants were created). As the figure shows, most of the 10-failure agreements fall into the ultrareliable region.

These effectiveness results should be taken with caution since — even though very impressive — they are based on a simple fault model. Many faults may be simple ones, but there are many faults which involve multiple simple faults that interact in peculiar ways.

CHAPTER V

SUMMARY AND FUTURE WORK

A framework has been presented for reliability quantification of ADT programs specified with TRSs. In this framework, practical ultrareliable quantifications of ADTs are possible for the class of decision ADTs. The formal nature of equational specifications has been exploited, as well as the explicit equality in them, to systematically generate and execute redundancies in order to obtain ultrareliable computations. Unlike previous self-checking methods, the generation of the redundancies with TRM is algorithmic. Furthermore, TRM extends self-checking to any ADT program, making it applicable in principal to programs written in ADT languages such as ADA, C++, CLU, Java, etc. Therefore, TRM has a potential application to the rapidly growing object oriented technology. TRM can be used on complete and non-complete specifications.

The effectiveness of TRM has been illustrated on the aircraft collision avoidance system TCAS. The illustration has demonstrated how ultrareliability can be obtained in the presence of a particular class of faults in term evaluators.

TRM may be applied to programs specified with non-equational specifications, such as Hoare's input/output specifications. These specifications may be used to systematically derive equations, by generating equations and using the input/output specification to check if the equation holds. Consider for example input/output specification of n procedures $f_1, f_2, ..., f_n$ which take as inputs the vectors $\vec{x_1}, \vec{x_2}$

$, ..., \vec{x_n}$, written as:

$$S_1\{f_1(\vec{x_1})\}Q_1$$

$$S_2\{f_2(\vec{x_2})\}Q_2$$

$$...$$

$$S_n\{f_n(\vec{x_n})\}Q_n,$$

where S_i and Q_i are the pre- and post conditions of the procedure f_i. Some languages for expressing input/output specifications are Euclid, VDM, GYPSY, Z, etc. A generate-and-test procedure for constructing equations involving $f_1, ..., f_n$ could consist of combining $f_1, f_2, ..., f_n$ and their arguments into syntactically valid terms, equating terms, and deciding if the equalities hold by using the pre- and post-conditions of $f_1, ..., f_n$. In particular, let Ω be an input/output specification consisting of a set of n procedures paired with pre- and post-conditions given in the form above. Let Ξ be a set of equations. Initially, Ξ is empty. Let $Ter(\Sigma)$ be a set of terms over the signature Σ specifying a set of variables and the set of function symbols $f_1, ..., f_n$. For each term $t \in Ter(\Sigma)$ a pre- and post-condition, denoted by S_t and Q_t, could be constructed by syntactic manipulation of the pre- and post-conditions of the procedures involved in t. Suppose we have constructed $S_{t_1}\{t_1\}Q_{t_1}$ and $S_{t_2}\{t_2\}Q_{t_2}$, where $t_1, t_2 \in Ter(\Sigma)$. The equation $t_1 = t_2$ is in Ξ iff $S_{t_1} \Leftrightarrow S_{t_2} \wedge Q_{t_1} \Leftrightarrow Q_{t_2}$.

For example, consider the following specification:

$$true\{f_1(x)\}x = x' + 1$$

$$true\{f_2(x)\}x = x' - 1,$$

where x' is the value of x prior to the execution of the f_1, f_2. Let t_1 be $f_1(f_2(x))$ and t_2 be $f_1(f_2(f_1(f_2(x))))$. By manipulation of the pre- and post conditions of f_1 and f_2, we arrive at:

$$true\{f_1(f_2(x))\}x = x' - 1 + 1$$

$$true\{f_1(f_2(f_1(f_2(x))))\}x = x' - 1 + 1 - 1 + 1.$$

The equation $t_1 = t_2 \in \Xi$ since

$$(true \Leftrightarrow true) \wedge (x = x' - 1 + 1 \Leftrightarrow x = x' - 1 + 1 - 1 + 1).$$

It may be possible to derive equational specifications from other formal specifications in the manner described above. In particular, a syntactic manipulation of the formal specification is used to generate equations, and then the formal specification is used to prove that the equations hold.

APPENDIX

CLP(R) IMPLEMENTATION OF TCAS EXPERIMENT

The complete CLP(R) code, necessary to carry out the TCAS illustration from chapter IV, is given in this appendix. The TCAS illustration involves executing a correct Boolean evaluator on mutated TCAS formulas. The correct Boolean evaluator is implemented as a TRS with the correct BTRS axioms, as given in figure 14. The figure 14 also gives the operator precedence definitions. In particular, negation has a higher precedence than plus and minus.

In the TCAS illustration, faults are introduced in the TCAS formulas, by applying mutated Boolean rules to correct TCAS formulas. The mutated Boolean rules are given in figure 15. The figure 15 gives all the possible ORF-mutated rules for the associativity, distributivity, and the DeMorgan laws. The TCAS formulas are implemented as CLP(R) predicates, as shown in figure 16. The illustration starts by grounding the TCAS formulas of figure 16. This grounding is obtained

```
::- op(41,yfx,~+).
::- op(31,yfx,~*).
::- op(21,fx,~-).

%identity
rrule(0 ~+ X,X).
rrule(1 ~* X,X).

%negation
rrule(~-0,1).
rrule(~-1,0).

%zero
rrule(1 ~+ X,1).
rrule(0 ~* X,0).
```

FIGURE 14. Correct BTRS Axioms

```
%distributivity
%mrule((X ~+ Y)~*Z, X~*Z ~+ Y~*Z).
mrule((X ~* Y)~*Z, X~*Z ~+ Y~*Z).
mrule((X ~+ Y)~+Z, X~*Z ~+ Y~*Z).
mrule((X ~+ Y)~*Z, X~+Z ~+ Y~*Z).
mrule((X ~+ Y)~*Z, X~*Z ~* Y~*Z).
mrule((X ~+ Y)~*Z, X~*Z ~+ Y~+Z).

%mrule(Y~+(X~*Z),(Y~+X)~*(Y~+Z)).
mrule(Y~*(X~*Z),(Y~+X)~*(Y~+Z)).
mrule(Y~+(X~+Z),(Y~+X)~*(Y~+Z)).
mrule(Y~+(X~*Z),(Y~*X)~*(Y~+Z)).
mrule(Y~+(X~*Z),(Y~+X)~+(Y~+Z)).
mrule(Y~+(X~*Z),(Y~+X)~*(Y~*Z)).
```

```
%associativity
%mrule((Y~+X)~+Z,Y~+(X~+Z)).
mrule((Y~*X)~+Z,Y~+(X~+Z)).
mrule((Y~+X)~*Z,Y~+(X~+Z)).
mrule((Y~+X)~+Z,Y~*(X~+Z)).
mrule((Y~+X)~+Z,Y~+(X~*Z)).

%mrule((Y~*X)~*Z,Y~*(X~*Z)).
mrule((Y~+X)~*Z,Y~*(X~*Z)).
mrule((Y~*X)~+Z,Y~*(X~*Z)).
mrule((Y~*X)~*Z,Y~+(X~*Z)).
mrule((Y~*X)~*Z,Y~*(X~+Z)).
```

```
%DeMorgan
%mrule(~-(X~+Y),~-X~*(~-Y)).
mrule((X~+Y),~-X~*(~-Y)).
mrule(~-(X~*Y),~-X~*(~-Y)).
mrule(~-(X~+Y),X~*(~-Y)).
mrule(~-(X~+Y),~-X~+(~-Y)).
mrule(~-(X~+Y),~-X~*(Y)).

%mrule(~-(X~*Y),~-X~+(~-Y)).
mrule((X~*Y),~-X~+(~-Y)).
mrule(~-(X~+Y),~-X~+(~-Y)).
mrule(~-(X~*Y),X~+(~-Y)).
mrule(~-(X~*Y),~-X~*(~-Y)).
mrule(~-(X~*Y),~-X~+(Y)).
```

FIGURE 15. Mutated BBTRS Axioms

```
tcas(~-(A~*B)~*(D~*(~-E)~*(~-F)~+(~-D)~*E~*(~-F)~+
(~-D)~*(~-E)~*(~-F))~*(C~*A~*(D~+E)~*H~+A~*
(D~+E)~*(~-H)~+B~*(E~+F))).

tcas((A~+((C~+D~+E)~*G~+A~*F~+C~*(F~+G~+H~+I))~+(A~+B)~*
(C~+D~+E)~*I)~*~-(A~*B)~* ~-(C~*D)~* ~-(C~*E)~* ~-(D~*E)~*
~-(F~*G)~* ~-(F~*H)~*~-(F~*I)~* ~-(G~*H)~* ~-(H~*I)).

tcas((A~*(~-D~+ ~-E~+D~*E~*(~-(~-F~*G~*H~* ~-I ~+ ~-G~*H~*I))~*
(~-(~-F~*G~*L~*K~+(~-G)~*(~-I)~*K)))~+(~-(~-F~*G~*H~* ~-I~+
(~-G)~*H~*I))~*(~-(~-F~*G~*L~*K~+(~-G)~*(~-I)~*K))~*(B~+C~*
(~-M)~+F))~*(A~*(~-B)~*(~-C)~+(~-A)~*B~*(~-C)~+(~-A)~*
(~-B)~*C)).

tcas(a~*(~-b~+(~-c))~*d~+e).

tcas(A~*(~-B~+(~-C)~+B~*C~*(~-(~-F~*G~*H~*(~-I)~+(~-G)~*H~*I))
~*(~-(~-F~*G~*L~*K~+(~-G)~*(~-I)~*K)))~+F).

tcas((~-A~*B~+A~*(~-B))~*(~-(C~*D))~*(F~*(~-G)~*(~-H)~+(~-F)~*
G~*(~-H) ~+(~-F)~*(~-G)~*(~-H))~*(~-(J~*K))~*((A~*C~+B~*D)~*E~*
(F~+(I~*(G~*J~+H~*K))))).

tcas((~-A~*B ~+A~*(~-B))~*(~-(C~*D))~*(~-(G~*H))~*(~-(J~*K))~*
((A~*C~+B~*D)~*E~*(~-I~+(~-G)~*(~-K)~+(~-J)~*(~-H~+(~-K))))).

tcas((~-A~*B~+A~*(~-B))~*(~-(C~*D)) ~* (~-(G~*H))~*((A~*C~+B~*D)
~*E~*(F~*G~+(~-F)~*H))).

tcas(~-(C~*D)~*(~-E~*F~*(~-G)~*(~-A)~*(B~*C~+(~-B)~*C~+(~-B)~*D))).

tcas(A~*(~-B)~*(~-C)~*D~*(~-E)~*F~*(G~+(~-G)~*(H~+I))~*
(~-(J~*K~*(~-J)~*L~+M))).
```

FIGURE 16. CLP(R) Implementation of TCAS Specification

```
v(0,F) :- reduce(0,F).
v(1,F) :- reduce(1,F).

v(X~+Y,F):-
  v(X,X1),
  v(Y,Y1),
  reduce(X1~+Y1,F),!.

reduce(X,Z) :- rrule(X,Y), v(Y,Z).
reduce(X,X).
drive(Ci,CiResult,Cj,CjResult):-
   getTerm(Ci),
   v(Ci,CiResult),
   rewrite(Ci,Cj),
   v(Cj,CjResult).
gterm(Ci) :-
   var(Ci),
   rand(X),
   X>=0.5,
   Ci=1,!.
gterm(Ci) :- ground(Ci),!.

gterm(Ci) :-
   var(Ci),
   Ci=0,!.
rewrite(Term,ResTerm) :-
   mrule(Term,ResTerm).
rewrite(Term,ResTerm) :-
   Term =.. [Op,L,R],
   rewrite(R,RR),
   ResTerm =.. [Op,L,RR].
```

```
v(X~*Y,F):-
  v(X,X1),
  v(Y,Y1),
  reduce(X1~*Y1,F),!.

v(~-X,F) :-
  v(X,X1),
  reduce(~-X1,F),!.


getTerm(Ci) :-
  tcas(Ci),
  gterm(Ci).


gterm(Ci) :-
   Ci =.. [Op,L,R],
   gterm(L),
   gterm(R),
   Ci =.. [Op,L,R],!.
 gterm(Ci) :-
    Ci =.. [Op,R],
 gterm(R),
    Ci =.. [Op,R],!.

rewrite(Term,ResTerm) :-
    mrule(ResTerm,Term).
rewrite(Term,ResTerm) :-
    Term =.. [Op,R],
    rewrite(R,RR),
    ResTerm =.. [Op,RR].
```

```
rewrite(Term,ResTerm) :-
  Term =.. [Op,L,R],
  rewrite(L,LR),
  ResTerm =.. [Op,LR,R].
```

FIGURE 17. Illustration Driver Code

through the predicate `gterm(Ci)`, given in figure 17. The predicate `gterm(Ci)` parses a TCAS formula `Ci`, looking for a variable, which it randomly replaces with either 0 or 1. The grounded formulas are evaluated using the correct rules of figure 14. The predicate `v(Ter,Res)` recursively (and correctly) evaluates the term `Ter` to produce a Boolean value `Res`. The predicate `reduce(X,Y)` applies correct rewrite rules to `X` in order to reduce `X` to `Y`. These correct values of grounded TCAS formulas are needed in oder to compare the outcome of the TCAS evaluator on mutated and correct TCAS formulas.

The mutated formulas are obtained by rewriting grounded TCAS formulas in one step using the mutated rules of figure 15. The predicate `rewrite`, that obtains mutated TCAs formulas, is given in figure 17. The predicate `rewrite(Term,ResTerm)` parses the grounded TCAS formula `Term`, searches for a subterm of `Term` and a matching mutated rule, applies the mutated rule to the subterm, and returns the resulting mutated TCAS formula `ResTerm`. These mutated TCAS formulas are evaluated with a correct BTRS, and the value of this evaluation compared to the value of the correct evaluation of the original TCAS formula (the one from which the mutant was produced).

The predicate `drive(Ci,CiResult,Cj,CjResult)` drives the TCAS illustration. In particular, it calls `getTerm(Ci)`, which gets a grounded TCAS formula `Ci`. This formula is evaluated with a correct BTRS, using `v(Ci,CiResult)`. That is, `CiResult` is the correct value for the grounded TCAS formula `Ci`. The predicate `drive(Ci,CiResult,Cj,CjResult)` then calls `rewrite(Ci,Cj)`, which produces a mutant from `Ci` by applying one mutated rule one time to `Ci`. The mutant `Cj` is evaluated with a correct BTRS to obtain the value `CjResult`. For each TCAS

formula, the number of discrepancies of `Ci` and `Cj` are counted. The ratio of the number of discrepancies and total number of evaluations gives the "Correct" column in figure 13.

BIBLIOGRAPHY

[1] P. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *Digest FTCS-17*, pages 122–126, 1987.

[2] P. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37:418–425, 1988.

[3] P. Ammann, D.L. Lukes, and J.C. Knight. Applying data redundancy to differential equation solvers. *Annals of Software Engineering, to appear*, 1997.

[4] D. Andrews. Using executable assertions for testing and fault-tolerance. *FTCS-9*.

[5] S. Antoy and R. Hamlet. Self-checking against formal specifications. In *International Conference on Computing and Information*, pages 355–360, Toronto, 1992.

[6] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. on Soft. Eng.*, 11:1491–1501, 1985.

[7] A. Avizienis and J. Kelly. Fault tolerance by design diversity: concepts and experiments. *Computer*, 17:67–80, 1984.

[8] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking computations in polylogarithmic time. *Proc. 23rd ACM Symp. Theory of Computation*, pages 21–31, 1991.

[9] J. A. Bergstra, J. Heering, and P. Klint. Algebraic specifications. *ACM press frontier series*, 1989.

[10] J. A. Bergstra and P. Klint. The discrete time toolbus. *Technical Report P9502, University of Amsterdam*, pages 1–111, 1995.

[11] J. A. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32:323–362, 1986.

[12] M. Blum and S. Kannan. Designing programs that check their work. In *21st ACM Symposium of Theory of Computing*, pages 86–96, 1989.

[13] M. Blum and S. Kannan. Designing programs that check their work. *JACM*, 42:269–291, 1995.

[14] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47:549–595, 1993.

[15] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *35th Annual Symposium on Foundations of Computer Science*, pages 382–391, Santa Fe, NM, 1994.

[16] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Transactions on Computers*, 45:385–393, 1996.

[17] Susan Brilliant, J. C. Knight, and N. G. Leveson. The consistent comparison problem in n-version programming. *IEEE Trans. on Soft. Eng.*, 15, 1989.

[18] Susan Brilliant, J. C. Knight, and N.G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Trans. on Soft. Eng.*, 16, 1990.

[19] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. on Soft. Eng.*, pages 3–12, 1993.

[20] J.J. Chilenski and S.P. Miller. Applicability of modified conditional/decision coverage to software testing. *manuscript.*

[21] M. Davis, R. Sigal, and E. Weyuker. *Computability, Complexity, and Languages*, 1994.

[22] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–43, 1978.

[23] R.K. Dong and G. P. Frankl. The astoot approach to testing object-oriented programs. *ACM TOSEM*, 3, 1994.

[24] M. Ehrig and B. Mohr. Fundamentals of algebraic specifications 1. In *Springer-Verlag*, Berlin, 1985.

[25] J. Gannon, R. Hamlet, and P. McMullin. Data abstraction implementation, specification, and testing. *ACM Trans. Prog. Lang. and Systems*, 3:211–223, 1981.

[26] J. D. Gannon, R. G. Hamlet, and H. D. Mills. Theory of modules. *IEEE Trans. on Soft. Eng.*, 13, 1987.

[27] M.-C. Gaudel and B. Marre. Generation of test data from algebraic specifications. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 138–139, Banff, Canada, 1988.

[28] A.L. Goel and K. Okumoto. Time dependent error detection rate model for software and other performance measures. *IEEE Transactions on Reliability*, R-28:206–211, 1979.

[29] J.A Goguen and J.J. Tardo. An introduction to obj:a language for writing and testing algebraic program specifications. In *Proc. Specifications of reliable software conference*, pages 178–189, 1979.

[30] G.Sullivan, D. Wilson, and G.Masson. Certification of computational results. *IEEE Trans. on Soft. Eng.*, 44:833–847, 1995.

[31] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[32] D. Hamlet. Are we testing for true reliability? *IEEE Software*, pages 21–27, July 1992.

[33] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[34] R. G. Hamlet. Probable correctness theory. *Info. Proc. Letters*, 25:17–25, 1987.

[35] N. Heintze, J. Jaffar, S. Michailov, P. Stukey, and R. Yap. The clp(r) programmer's manual. *Monash University, Clayton*, 1987.

[36] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. on Soft. Eng.*, 8:371–379, 1982.

[37] K. Huang and J Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transaction on Computers*, 33:518-529, 1984.

[38] J. Jaffar and J-L. Lassez. Constraint logic programming. *POPL Munich*, 1987.

[39] J.A.Goguen, J.W. Thatcher, and E.G.Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *In R.T. Yeh, editor, Current Trends in Programming Methodology*, pages 80–149, 1978.

[40] P Jalote. Specifications and testing of abstract data types. In *COMPSAC*, pages 508–511, 1983.

[41] K.A.Foster. Sensitive test data for logic expressions. *ACM SIGSOFT Software Eng. Notes*, 9:120–126, 1984.

[42] K.C.Tai. Condition-based software testing strategies. *Proc. Compsac 14th Ann. Intl. Comput. Soft. and Applic. Conf.*, pages 564–569, 1990.

[43] P. Klint. A meta-environment for generating programming environments. *Lecture notes in Computer Science*, 490:105–124, 1991.

[44] J. W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, vol. II*, pages 1–112, Oxford University Press, 1992.

[45] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on Soft. Eng.*, 12:96–109, 1986.

[46] J. Laprie. Dependability, basic concepts and terminology. In *Springer-Verlag*, Vienna, 1992.

[47] N. G. Leveson. Safeware system safety and computers. In *Addison-Wesley*, 1995.

[48] N.G. Leveson, M.P.E. Heimdahl, H. Hilderth, and J.D. Reese. requirements specification for process-control systems. *Tech. Rep. 92-106 Dept. of Inform. and Comput. Sci., Univ. of Cal., Irvine*, 1992.

[49] R. Lipton. New directions in testing. *DIMACS Series on Discrete Mathematics and Theoretical computer science*, 2:191–202, 1991.

[50] B. Littlewood and L. Strigini. Assessment of ultra-high dependability for software-based systems. *CACM*, 36, 1993.

[51] M. R. Lyu. Handbook of software reliability engineering. *IEEE Computer Society Press*, 1995.

[52] B. Marre. Toward automatic test data selection using algebraic specification and logic programming. *Proc. of the 8th International Conference on Logic Programming*, pages 202–219, 1991.

[53] G. Metze and A. Mili. Self-checking programs: An axiomatization of program validation by executable assertions. *IEEE Trans. on Soft. Eng.*, pages 118–120, 1981.

[54] D.R. Miller. Making statistical inferences about software reliability. *NASA Contractor Rep. 4197*, 1988.

[55] D.R. Miller. The role of statistical modeling and inference in software quality assurance. *CSR Workshop on softawre certification*, 1988.

[56] P.L. Moranda and Z. Jelinski. Final report on software reliability study. *McDonnell Douglas Astronautics Company, MADC Report number 63921*, 1972.

[57] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE*, pages 14–32, 1993.

[58] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, NY, 1987.

[59] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM*, 15:1053–1058, 1972.

[60] D. L. Parnas, A. van Schouwen, and S. Kwan. Evaluation of safety-critical software. *Comm. of the ACM*, 33:638–648, 1990.

[61] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. *ISSTA*, 1994.

[62] P.Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Widgerson. Self-testing/correcting for polynomials and for approximate functions. *Proc. 23rd Symp. Theory of Comp.*, pages 32–42, 1991.

[63] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Soft. Eng.*, SE-1:220–232, 1975.

[64] B. Randell, J. Laprie, H. Kopetz, and B. Littlewood. Predictably dependable computing systems. In *Springer Verlag, ESPRIT Basic Research Series*, 1995.

[65] John A. Rice. Mathematical statistics and data analysis. *Wadsworth and Brooks/Cole, New York*, 1988.

[66] R. Rubinfeld. On the robustness of functional equations. *Proc. 35th Symp. Foundations of Computer Science*, pages 288–299, 1994.

[67] M. L. Shooman. *Software Engineering Design, Reliability, and Management.* McGraw-Hill, New York, NY, 1983.

[68] R. Thayer, M. Lipow, and E. Nelson. *Software Reliability.* North-Holland, New York, NY, 1978.

[69] F. Veinstain. Error detection and corection in numerical computations by algebraic methods. *Proc. 9th Int'l Symp. Applied Algebra, Algebraic Algorithms and Error-Detecting Codes*, pages 456–464, 1991.

[70] E. Weyuker, T. Gordia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Trans. on Soft. Eng.*, 20:353–363, 1994.

[71] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proc. Int'l Conf. on Reliability Software*, pages 450–457, 1975.