TTCN-3 Test Data Analyser using Constraint Programming

Diana Vega
Technical University Berlin
Franklinstr. 28-29,
D-10623 Berlin, Germany
vegadiana@cs.tu-berlin.de

George Din
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
D-10589 Berlin, Germany
george.din@fokus.fraunhofer.de

Ina Schieferdecker
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
D-10589 Berlin, Germany
schieferdecker@fokus.fraunhofer.de

Abstract

This paper presents the idea of analyzing and refining the test data variance computation as a method to determine the quality of TTCN-3 tests. TTCN-3 as the only internationally standardized testing language is beeing used extensively by industry. A large amount of functional test specifications are written in this language. For this reason, a sensible issue is the quality analysis of these specifications.

Data variance characterizes the test data distribution over the test system interface. TTCN-3 template distance[1] *constitutes the criterion to quantify the variance: "similar" and "different enough" test data.*

Due to the high flexibility of the language, the stimuli templates to be compared may not only contain concrete values but also variables, function calls, parameters, values which are known at runtime only, etc. Therefore, we combine Constraint Programming (CP) with static analysis of TTCN-3 test suites targeting a realistic template solving. We name template constraint an expression whose value (or domain restrictions) can only be determined by looking at the execution paths in the analyzed test behavior.

This method leads to a better computation of the data distance and, thus, to a better refinement of similarity classes. The approach is illustrated by an example which shows how data varies when the distance is computed taking into account the template constraints.

[1]We call template all messages which appear in send statements. However, they can be variables, inline templates, templates returned by a function call, etc.

1 Introduction

With the advances of the complexity and dimensions of the test specifications used in industry, assessing the test quality, producing and identifying effective tests is a challenging task and debated subject in the research. In this respect, a framework for analysing different quality aspects of test specifications is provided in [19]. It proposes a quality model for test specifications derived from the ISO/IEC 9126 [6] quality model. Various test metrics have been already developed measuring selected aspects [12, 17, 18], but they are still in their infancy or regard test quality aspects rather at programming language level than at test specific expectations.

In a previous work [16], an approach of how to evaluate *test effectiveness*, the external quality aspect of a test specification, has been presented and applied to the *Testing and Test Control Notation* (TTCN-3) [4]. According to [19], three sub-characteristics describe the test effectiveness: *test coverage*, *test correctness* and *fault-revealing capability*. In practice, the test correctness and the fault-revealing capabilities are very hard if not even impossible to determine. However, the test coverage which is the measure of test completeness on different levels can be obtained easier, e.g. the extent to which a test specification covers system requirements, system model, system code, etc. Hence, test coverage can be analysed with the help of different metrics. In white-box (structural) testing, testing code metrics such as statement, branch coverage, etc. indicate the degree to which the test specification covers the system code. In many cases, only the system interfaces are available (e.g. black-box testing), either provided as interface specification, doc-

978-0-7695-3331-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICSEng.2008.74

umentation or formally described. With the advances of model-based system development, a model of the system is provided as well. Consequently, black-box (functional) system model coverage metrics are in use and usually adopted from code coverage metrics, e.g.: state coverage, transition coverage and alike.

TTCN-3 is based on concepts, which are independent of any syntax such as test cases, test components, test ports. A test case creates test components and connects them to the SUT. The communication with the SUT takes place through well-defined communication ports and an explicit test system interface (TSI), which defines the boundaries of the test system.

The TTCN-3 template mechanism provides the possibility to specify, organize and structure test data in a very comfortable way. A template either describes a concrete value or specifies a subset of values of a given data type or signature. Therefore, templates can be used to define concrete values to be transmitted or to describe conditions to be matched by received values.

This paper discusses a metric already introduced in [16] called TTCN-3 *template distance*. We aim at improving the limitations of the original definition and reduce some of the assumptions made initially. The distance has been defined between two different TTCN-3 templates of the same type using the following formula:

$distance_{T'} : T' \times T' \rightarrow [0..1]$
for $v_1, v_2 \neq omit : distance_{T'}(v_1, v_2) = \{ {}^{0 \leq d \leq 1 \text{ for } v_1 \neq v_2}_{0 \text{ otherwise}}$

where d is computed according to Table 1 and Table 2

$distance_{T'}(v, omit) = \{ {}^{0 \text{ for } v = omit}_{1 \text{ otherwise}}$

The distance metrics (yielding values between 0 and 1) for basic types are defined as depicted in Table 1. The distance metrics for structured types (yielding values between 0 and 1 or the uncomparable value) are defined in Table 2. These two tables have been taken over from our previous paper.

* o-DED = One-Dimensional Euclidian Distance
** HD = Hamming Distance

Special cases such as *omit* value should be treated independently of the general case of concrete values templates. We considered the distance between every basic type and omit value to be 1 (the maximum). The distance between omit and omit is considered 0.

On this basis, the *quantitative similarity* of the test inputs with respect to a specific *message-based port* that belongs to the test system interface (TSI) can be derived. Hence, the

Table 1. Distance Metrics for Values of Basic TTCN-3 Types

Basic Type	Distance based on	Definition of distance d for values x and y
Integer	o-DED*	$d(x,y) = \frac{\lvert x-y \rvert}{sizeof(Integer)}$
Float	o-DED	$d(x,y) = \frac{\lvert x-y \rvert}{sizeof(Float)}$
Boolean	Ineq	$d(x,y) = \{ {}^{0 \text{ for } x=y}_{1 \text{ otherwise}}$
Bitstring	HD**	number of positions for which the bits are different (the shorter bitstring is extended into the longer bitstring by filling it with leading '0'B) divided by the longer length: $d(x,y) = \frac{\eth(x,y)}{maxlength(x,y)}$ with $\eth(x,y) =$ number of i where $x_i \neq y_i$
Hexstring	HD	same but with leading '0'H
Octetstring	HD	same but with leading '0'O
Charstring	HD	same but with leading " " (spaces)
Universal Charstring	HD	same but with leading " " (spaces)

coverage at the TSI level may be assessed by means of a *semantic similarity* derived by using a partitioning method of the stimuli space. To carry out quantitative estimates, i.e. come to concrete values of the distance metrics, a template solver method is required. This is mostly applicable for the cases where complex tree-like structured templates (e.g. record types) define the set of stimuli for a specific port, of a specific TTCN-3 type. In the introduced approach, these templates are subject of distance computation. Due to the recursive definition of the template distance for structured types (e.g. for record types we consider n-Dimensional Euclidian Distance [10]), only the concrete values corresponding to the fields of basic types are necessary.

* NDED = N-Dimensional Euclidian Distance
** HD = Hamming Distance

However, TTCN-3 semantic permits various ways to assign values to templates or to fields of templates:

- direct values, i.e. integer, charstring values, etc.
- other variables
- user-defined function calls
- language specific function calls, e.g. *valueof*
- test case parameters
- module parameters

Table 2. Distance Metrics for Values of Structured TTCN-3 Types

Structured Type	Distance based on	Definition of distance d for values x and y
Record	N-DED*	$d(x,y) = \frac{\sqrt{\sum_{i=1}^{n}(d(x_i,y_i))^2}}{n}$
Record of	HD**	$d(x,y) = \frac{\sum_{i=1}^{n}\mathfrak{d}(x,y)}{maxlength(x,y)}$ with $\mathfrak{d}(x,y) =$ number of i where $d(x_i,y_i) > \frac{1}{3}$ and where the record sequence is extended into the longer record sequence by filling it with leading *omit*
Set	N-DED	same as for record
Set of	HD	same as for record of
Enumerated	Ineq	$d(x,y) = \frac{\lvert\mathfrak{n}(x)-\mathfrak{n}(y)\rvert}{n}$ where $\mathfrak{n}$ is the sequentially numbered index of the enumeration
Union	-	$d(x,y) = d(\mathfrak{v}(x),\mathfrak{v}(y)) = \begin{cases} 1 \text{ for } \mathfrak{v}(x)=\mathfrak{v}(y) \\ 0 \text{ otherwise} \end{cases}$

- expressions from simple ones to combination of the afore possibilities

The metrics computation in a TTCN-3 test suite is achieved along a static analysis process, that is, no runtime values, elements, etc. are concerned, but the TTCN-3 test specification only. To come up with a number in the interval $[0..1]$ as a result of the calculation of the distance metric between two messages instantiated using one of the listed choices, a *template distance* algorithm has been provided. In the framework presented in [16] a set of limitations and assumptions regarding the template solving have been made. As an example, the maximum value 1 was attributed to the distance, when one of its message could not have been identified[2], i.e. not a concrete value.

Obviously, these unreliable numbers will be propagated further on affecting the partitioning sensitivity. Hence, we applied known methods already imposed in white-box testing, to TTCN-3 behaviors. One of these is *symbolic execution* [8] which assumes that instead of supplying the normal inputs to a program (e.g. numbers), one supplies symbols representing arbitrary values within a specific domain or constrained values. In white-box testing, the symbolic execution method uses the *control flow graph* (CFG) of a program, which is a graph abstraction of the program, and symbolically executes the program by selecting only one execution path from the CFG. Tools such as the well-known Java PathFinder (JPF)[9] have successfully adopted this technique to prove the correctness of a program. An example of this application is to check if every execution of the program is feasible and does not lead to error states or other violations, i.e. each path of its control flow graph is verified. Following this method, we built first the CFG associated to a testcase whose behavior encapsulates a selected stimuli-message involved in the distance calculation. Then a TTCN-3 symbolic values solver built with the help of CFG and constraint programming methods (CP) targets the domain reduction for a specific template field. The identified constraints upon template fields are further propagated to the distance formula.

This paper is structured as follows. After reviewing the related work in Section 2, the TTCN-3 data variance computation method based on constrained templates and constraint propagation to template distances is discussed in Section 3. An example is given in Section 4 and details of our implementation are highlighted in Section 5. Our conclusions and the discussion of the next steps end the paper.

2 Related work

The contribution of this paper is motivated by the need of refinement of the TTCN-3 distance metrics introduced in [16]. The method described in that paper suffers from several limitations which occurred as a result of a poor static analysis approach along the computation algorithm. Hence, we were motivated to investigate further domains such as Symbolic Execution, Constraint Programming and to analyse their possible integration with the existing work. Consequently, we introduce next the basic notions regarding these fields and how they have been employed so far in the testing domain.

Similarity Measures

Data variance in general is investigated by inspecting the *proximity of objects*. The survey in [7] presents how clustering facilitates the grouping of a given collection into meaningful clusters (similar data points). Measurement of the proximity (similarity) between data points is accomplished through well defined partition clustering algorithms. Example of direct utility of this field is statistical theory and machine learning where the first step is pattern/data repre-

[2]Please note, that for a tree-like structured template, the *identification* refers to either leaf nodes which are attributed "not identifiable" basic values or to children non-leaf nodes, depending on level of recursiveness in the distance calculation

sentation. In our study [16] this maps directly to the identification of the type and set of TTCN-3 templates forming the space to be clustered. The next step is the definition of the pattern/data proximity measures. As an example, Euclidean distance [10] is the most referenced data proximity metric; semantical distance function on pairs of words or terms, entitled *Google distance* has been suggested in [3].

Symbolic Execution

Symbolic execution is a static analysis technique. In software testing it was pioneered by King (1976) [8]. The main idea of symbolic execution of a given program is to exercise the program with abstract (symbolic) inputs rather than with concrete ones. All computations of the program affecting the inputs are not resolved to concrete results, but are rather kept on an abstract level by using symbolic expressions. This implies that the program under consideration is not actually executed, its execution is rather "simulated" step by step. If a branching statement is encountered, each of the possible branches is visited according to the chosen strategy (depth-first, breadth-first, or others).

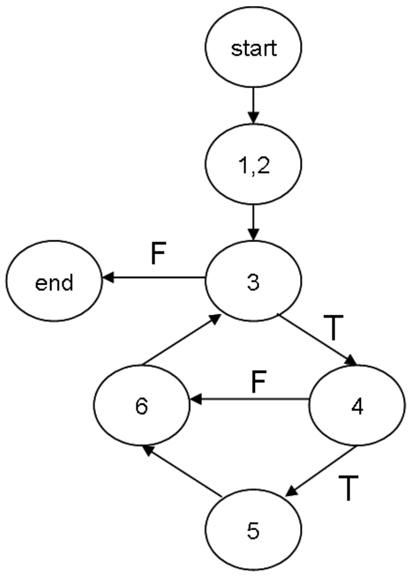

Figure 1. Control Flow Graph for the Listing 1

Listing 1. Code Exemple

```
1  a = read (b)
2  c = 0
3  while (pow(a,2) > 1) {
4      if (a > c)
5          c = c + a
6      a = a - 2
7  }
```

The example in Listing 1 shows a short sequence of code instructions (including a *loop* and an *if*). The associated control flow graph is given in Figure 1. In this graph several paths can be identified, e.g.:

$p1 : (start) \rightarrow (1,2) \rightarrow (3) \rightarrow (end)$

$p2 : (start) \rightarrow (1,2) \rightarrow (3) \rightarrow (4) \rightarrow (6) \rightarrow (3) \rightarrow (end)$

To each path, a *path condition* which defines a constraint on a (set of) variable may be associated. For instance, given the path $p : (start) \rightarrow (1,2) \rightarrow (3) \rightarrow (end)$, the related path condition is $|a| <= 1$. For each path (a set of statements), the symbolic execution delivers a set of constraints (referred to as the constraint system) which a concrete input must satisfy so that the path to the statement can be traversed.

Constraint Programming

The concept of Constraint Programming (CP), detailed in [1] has been subject of research in *Artificial Intelligence*. Basically, a Constraint Satisfaction Problem (CSP) is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. The task is to assign a value to each variable satisfying all the constraints. Formally, we can describe it as $\langle X, D, C\rangle$ where:

- $X = \{x_1, x_2, \dots x_n\}$, the set of variables with domains
- $D = \{D_1, D_2, \dots D_n\}$, domains for each variable
- $C = \{c_1, c_2, \dots\}$, constraints restricting the values that the variables can simultaneously take

A constraint is an expression that specifies some property of a set of variables. For example, the constraint $x \geq 1 \&\& x \leq 10$ specifies that the value of the variable x is between 1 and 10. Another familiar example constraint is from the *eight queens problem*; it specifies that a chess board contain eight queens such that no queen attack each other. *Constraint solving* refers to the finding solutions to constraints, i.e. assignments to the variables that make the constraint expression true. For example, $x = 3$ is one solution for the first example constraint. Constraint solver often use sophisticated local and global search techniques to find one, several or all solutions to a given constraint.

Many researches ([2],[5],[20]) have explored the constraint solving technique and its applicability to testing. The main idea relies on generating test-inputs from a desired property of the input domain expressed as a constraint, potentially resulted automatically from the SUT model or code by using constraint solving techniques.

3 Distance Computation Method based on Constraints

Our approach relies on the fact that TTCN-3 language is similar to a programming language. Hence, symbolic execution (SE) can be applied to its behavioral entities encapsulated in the test specifications.

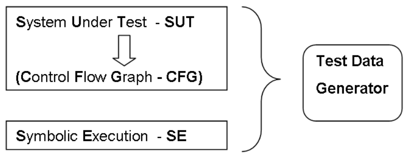

Figure 2. Symbolic Execution and Testing in current approaches

We put in contrast in Figures 2 and 3 the existing approach of applying SE versus our method. In the existing approaches, the target is test data generation. We changed the perspective, and, instead of supplying the SUT code to the SE, we serve the SE with a TTCN-3 test behavior. This way we aim at investigating the data variability in greater detail than in the previous approach.

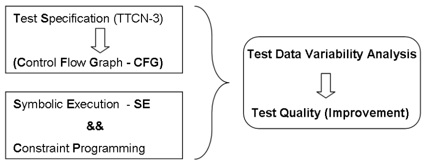
Test Specification (TTCN-3)
(Control Flow Graph - CFG)
Symbolic Execution - SE
&&
Constraint Programming
Test Data Variability Analysis
Test Quality (Improvement)

Figure 3. The Combination of SE & CP with Test Specifications

We recall that the data variability is computed on top of a template distance, which in turn is based on a template solving method. We apply SE and CP to improve the template solving method. The algorithm consists of the following steps:

1. start with the set of templates which stimulate the same port belonging to the test system interface (TSI) and having the same type

2. for each template from the template set identify the TTCN-3 behavioral entity (e.g. testcase, function) where the interested stimuli message, called *target template* occurs, i.e. a statement *p.send(¡message¿)*.

   (a) build the control flow graph (CFG) for that specific behavioral entity (e.g. testcase, function).

   (b) localize the *target template* (encapsulated in *p.send(¡message¿)* statements) to a corresponding node in CFG; this node is called *target node*.

   (c) determine the *path conditions* for each path that connects the *CFG head* with the *target node*.

      i. identify the decisions blocks and derive the constraints

      ii. build a variable symbol table on a top down manner which stores for each variable the constrained domain

      iii. apply the constraints on the variables directly/indirectly involved in the structure of the target template

   (d) expand the templates set with variations of the target template as resulted from the different constraints on different paths.

3. apply the coverage algorithm defined in [16] on the resulting template set

As result, the distance between two templates is calculated as the composition of the constraints associated to each encountered variable in the template construction.

Formally, we represent it as follows. Given two message stimuli (i.e. templates) $tmpl_1$ and $tmpl_2$:

– $tmpl_1$ with the domain $D_1 = \{D1_1, D1_2, \ldots D1_n\}$ corresponding to the elements of basic types composing the template structure and with the associated constraints $C_1 = \{c1_1, c1_2, \ldots c1_n\}$

– $tmpl_2$ with the domain $D_2 = \{D2_1, D2_2, \ldots D2_m\}$ corresponding to the elements of basic types composing the template structure and with the associated constraints $C_2 = \{c2_1, c2_2, \ldots c2_m\}$

$\Rightarrow$ the distance $d(tmpl_1, tmpl_2)$ is characterized by a new constraint derived from the two constraints C_1 and C_2. The derivation is based on the distance formula provided in [16].

The CFG Construction

For each TTCN-3 behavior entity, a CFG is constructed as a representation of all paths that might be traversed during test execution, by using a directed graph notation. The construction of the CFG can be seen as a mapping of the TTCN-3 statements to graph nodes.

Table 3. CFG Block Types

CFG block	Description
entry block	block through which all control flow enters the graph
exit block	statement block block through which all control flow leaves the graph
simple block	has one "in" edge and one "out" edge
decision start block	block which originates two or more edges
decision end block	block which marks the end of a decisional block in a program
back block	an edge that points to an ancestor in a depth-first (DFS) traversal of the graph
loop block	the entry point of the loop

At the building of the CFG we considered the types of nodes presented in Table 3

In the Table 4 we present the mappings rules considered in our method. The left column represent the TTCN-3 statements which are mapped to their corresponding CFG blocks in the second column. All listed statements regard statements which may occur in a testcase. With respect to limitations, for the moment statements such as defaults, altstep etc are not yet considered. However, they are not difficult to realize.

The TTCN-3 Template Constraint Solver

The ranges (the domain) a variable/template has at a specific node in a CFG are determined by updating its list of constraints along a specific path. The list of path conditions is obtained by analysing each node block belonging to that path and by inspecting the associated TTCN-3 statement. Usually, the constraints on variables appear in statements like:

- **assignment statements**: The constraint given to the left part of the assignment (e.g. a variable) can be a either a concrete value or a composition of constrains resulting by evaluating the right side of the assignment.
- **if-conditions**: The constraint applies to variables involved in the if-conditions.
- **alt guard conditions**. The constraint applies to the variables involved in the guard conditions.

Table 4. TTCN-3 Statements and their Control Flow Graph corresponding blocks

TTCN-3 Statement	CFG block
variable declaration (e.g. var integer x;)	simple block
variable declaration with assignment (e.g. var integer x:=3;)	simple block
assignment operation (e.g. x:=3;)	simple block
execution statements (e.g. log;)	simple block
if/alt statement	decision start block and decision end block; in-between further blocks are possible; the number of edges leaving the start block is equal with the if/alt branches [3]
while/for	loop block and back block
repeat	back block

- **alt conditions**: The constraint applies to the variable specified in the *receive* statement in order to take the value of the received message.
- **condition statements in loops**

The constraints of a variable may be propagated onto other variables which are involved in expressions with it. Therefore, it is necessary to analyse each particular statement and keep track of all detected constraints.

4 An example

In this section, we show how to apply the introduced concepts to a small TTCN-3 example. The SUT is a web service with an interface which supports queries in a hotel database, by specifying some parameters as *city*, *min/max price* and the number of *stars* for the hotel.

The test suite defines the types presented in Listing 2. A *city* is represented as an integer with values from 1 to 10 (i.e. each city has a number), the *price* is defined as integer from 0 to 500 and the number of stars of a hotel is also an integer from 0 to 5. Next, we defined two more types: *RequestType*, for messages to be sent to SUT and a *ResponseType* for messages to be received from the SUT. A request contains the *city* where the hotel should be searched, the range of prices given as *min* and *max* prices and the number of stars for the hotel. The response contains a boolean field called *found* to indicate whether a hotel has been found or not, and the price of the hotel. When more than one hotel satisfying the query is found, then the lower price is returned.

Listing 2. TTCN-3 Types Definition

```
modulepar CityType mpCity := 1;

type integer CityType (1..10);
type integer StarsType (0..5);
type integer PriceType (0..500);

type record RequestType {
  CityType city, PriceType min,
  PriceType max, StarsType stars }

type record ResponseType {
  boolean found, PriceType price }

template ReqType reqTwoStarsTempl:={
  city := mpCity, min := 0,
  max := 100, stars := 2 }

template ReqType reqFiveStarsTempl:={
  city := mpCity, min := 0,
  max := 100, stars := 5 }

template RespType respFoundTempl:={
  found := true, price := ? }

template RespType respNotFoundTempl:={
  found := false, price := ? }

type port P message {
   out ReqType;
   in RespType;
}
type component C {
   port P p;
   timer t := 10.0;
}
```

The test suite defines two testcases presented in Listing 3 and Listing 4. The first testcase tests if the SUT finds a hotel with two stars cheaper than 100 euro. The testcase sends the request template *reqTwoStarsTempl* and expects as result that the SUT returns a hotel with these constraints. After sending the search request a timer is started in order to validate that the SUT responds in an acceptable amount of time.

Listing 3. Testcase 1

```
testcase t1() runs on C system C {
  map (self:p, system: p);
  p.send(reqTwoStarsTempl);
  t.start;
  alt {
    [] p.receive(respFoundTempl)
    { setverdict(pass); }
    [] p.receive(respNotFoundTempl)
    { setverdict(fail); }
    [] t.timeout { setverdict(inconc); }
  }
}
```

The second testcase tests that the SUT can find a hotel cheaper than 100 euros with an arbitrary number of stars. The test behavior starts with the sending of *reqFiveStarsTempl* request which means that we search first for a hotel of five stars with a maximal price of 100 euros. If no such hotel is found, the tests decreases the number of stars and tries again. This can repeate until the number of stars is equal to 0 when the test stops with verdict fail. If in the meantime a hotel is found then the tests stops with verdict fail. The duration of each search operation is validated by a timer.

Listing 4. Testcase 2

```
1  testcase t2() runs on C system C {
2    map (self:p, system: p);
3    var ReqType vRequest :=
4         reqFiveStarsTempl;
5    p.send(reqFiveStarsTempl);
6    t.start;
7    alt {
8      [] p.receive(respFoundTempl)
9      { setverdict(pass); }
10     [] p.receive(respNotFoundTempl)
11     { t.stop;
12       if(vRequest.stars >= 1) {
13         vRequest.stars:=vRequest.stars -1;
14         p.send(vRequest);
15         t.start;
16         repeat;
17       }
18       else { setverdict(fail); }
19     }
20     [] t.timeout { setverdict(inconc); }
21   }
22 }
```

We apply now the afore described algorithm to this example by starting with the *template set* identification; in this case only one set exists since we used the same *port type* to send requests, the same *message type* to the SUT and the same TSI. The template set consists of the following templates:

(1) Set={*reqTwoStarsTempl, reqFiveStarsTempl, vRequest*}

The first two templates are defined such that all fields contain concrete values except the field *city* which is assigned the module parameter *mpCity*. This field has then to be resolved by the template solver. The third message in the set is a variable which is defined locally in the testcase. This variable appears in the *p.send* statement (line 13, Listing 4). The content of the variable is alterated in line 2 (it is initialized with the template *reqFiveStarsTempl*) and line 12 (the field *stars* is decreased) in Listing 4. Additionally, it also appears in the decision statement on the line 11 where the number of stars is compared with value 1.

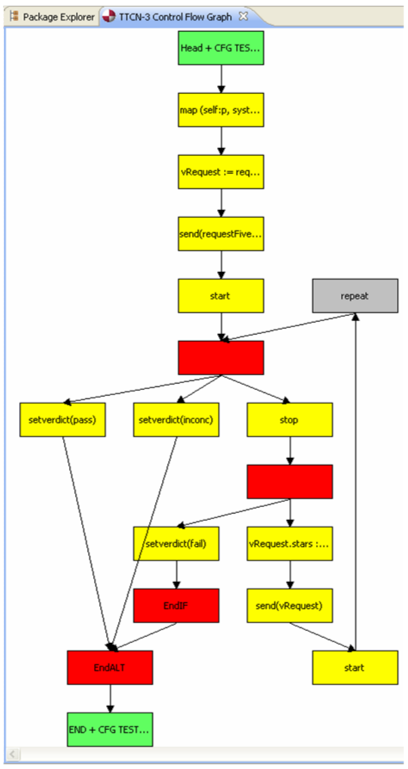

Figure 4. The CFG of testcase tc2

The next step is the distance computation which applies to each pair of templates in the template set. The formulas we need are:

(2) $d(x, y) = \frac{|x-y|}{sizeof(IntegerDomain)}$

(3) $d(x, y) = \frac{\sqrt{\sum_{i=1}^{n}(d(x_i,y_i))^2}}{n}$

Formula (2) is used to compute the distance between integer based fields while formula (3) is used to compute the distance between record based messages.

In the initial approach the distance that involves unresolvable values is assigned directly value 1 (maximal distance). This means:

(4) $d(reqTwoStarsTempl, reqFiveStarsTempl) = 0.125$,
which is lower than the $\frac{1}{3}$, the choosen threshold.

(5) $d(reqTwoStarsTempl, vRequest) = 1$, since the vRequest is local testcase variable which has been not resolved.

(6) $d(reqFiveStarsTempl, vRequest) = 1$, again, since the vRequest is local testcase variable which has been not resolved.

According to the distances computed in (4)(5)(6) we obtained two partitions:

(7) $P_1 = \{reqTwoStarsTempl, reqFiveStarsTempl\}, P_2 = \{vRequest\}$

This leads to the conclusion that the coverage is $\frac{2}{3}$. By applying the improved method, all templates and their fields can be entirely solved. The $d(reqTwoStarsTempl, reqFiveStarsTempl)$ remains the same as before since all fields are known. But the distances involving vRequest look different now since we have to take into account the possible paths and for each path compute the cumulated constraint. This will result into an expandation of the vRequest into as many variations of it as many paths exist in the CFG. The CFG for the testcase tc2 is shown in Figure 4 (*vReq* appears in testcase *tc2*).

For simplification we considered the depth level of the loops as 1. The paths in the example are:

path1: $headCFG \rightarrow map(\ldots \rightarrow vReq := \ldots \rightarrow send(\ldots \rightarrow start \ldots \rightarrow alt \rightarrow setverdict(pass) \rightarrow endAlt \rightarrow endCFG$

path2: $headCFG \rightarrow map(\ldots \rightarrow vReq := \ldots \rightarrow send(\ldots \rightarrow start \ldots \rightarrow alt \rightarrow setverdict(inconc) \rightarrow endAlt \rightarrow endCFG$

path3: $headCFG \rightarrow map(\ldots \rightarrow vReq := \ldots \rightarrow send(\ldots \rightarrow start \ldots \rightarrow alt \rightarrow stop \rightarrow IF \rightarrow setverdict(fail) \rightarrow EndIF \rightarrow endAlt \rightarrow endCFG$

path4: $headCFG \rightarrow map(\ldots \rightarrow vReq := \ldots \rightarrow send(\ldots \rightarrow start \ldots \rightarrow alt \rightarrow stop \rightarrow IF \rightarrow vReq.stars := \ldots \rightarrow send \ldots \rightarrow start \rightarrow repeat \rightarrow Alt \rightarrow setverdict(pass) \rightarrow endAlt \rightarrow endCFG$

path5: $headCFG \rightarrow map(\ldots \rightarrow vReq := \ldots \rightarrow send(\ldots \rightarrow start \ldots \rightarrow alt \rightarrow stop \rightarrow IF \rightarrow vReq.stars := \ldots \rightarrow send \ldots \rightarrow start \rightarrow repeat \rightarrow Alt \rightarrow setverdict(inconc) \rightarrow endAlt \rightarrow endCFG$

path6: $headCFG \rightarrow map(\ldots \rightarrow vReq := \ldots \rightarrow send(\ldots \rightarrow start \ldots \rightarrow alt \rightarrow stop \rightarrow IF \rightarrow vReq.stars := \ldots \rightarrow send \ldots \rightarrow start \rightarrow repeat \rightarrow Alt \rightarrow stop \rightarrow IF \rightarrow setverdict(fail) \rightarrow EndIf \rightarrow endAlt \rightarrow endCFG$

The template set will be expanded with the variations of vReq for each path: $vReq(C_{path_i})$, where i=1..6 and C_{path_i} is the constraint regarding the path condition i upon vReq variable. For instance, the constraint for C_{path_6} is $\{vReq := reqFiveStarsTempl, vReq.stars >= 1, vReq.stars := vReq.stars - 1\}$.

This means, the resulting template set is:

$Set = \{reqTwoStarsTempl, reqFiveStarsTempl, vReq(C_{path_i})\}$

However, the constraints along the paths 1..3 do not change the initial value of vReq (which is reqFiveS-

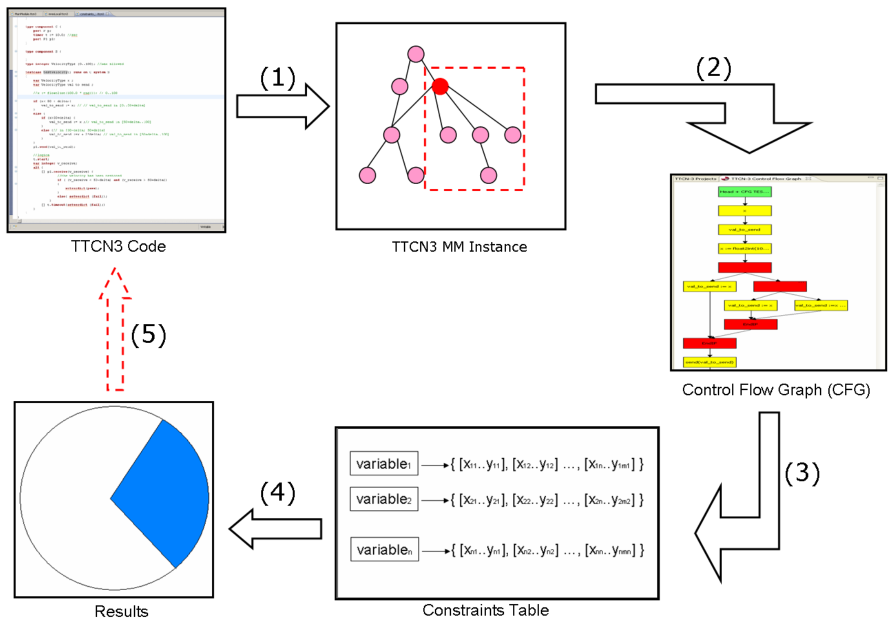
(1)
(2)
(3)
(4)
(5)
TTCN3 Code
TTCN3 MM Instance
Control Flow Graph (CFG)
Constraints Table
Results
variable1 → { [x11..y11], [x12..y12] …, [x1n..y1m1] }
variable2 → { [x21..y21], [x22..y22] …, [x2n..y2m2] }
variablen → { [xn1..yn1], [xn2..yn2] …, [xnn..ynmn] }

Figure 5. The Blocks Chain of the Implementation

tarsTempl) which implies that the distances between them and reqFiveStarsTempl is 0. Similarly, the distances between the three templates and the reqTwoStarsTempl is 0.125, according to formula (4). All these distances are below the threshold, therefore they belong to the same partition.

For the path 4..6 the field *vReq.stars* will become 4 due to constraint $vReq.stars := vReq.stars - 1$. Hence, the distance between
$vReq(C_{path_i}$ and *reqTwoStarsTempl, reqFiveStarsTempl* will be $\frac{1}{12}$, respectively $\frac{1}{24}$. Both of them are lower than the threshold. Thus, all templates in the new template set belong to the same partition.

In conclusion, the method based on constraint solving lead to a better sensitivity of the partitions. This reveals in fact that the data stimuli in the example is actually not that good as found with the first method.

5 Implementation

In order to implement the introduced concepts, we extended the framework presented in our previous paper. That framework is based on the TTworkbench [15] product, an Eclipse-based IDE that offers an environment for specifying and executing TTCN-3 tests. One of the main features provided by this tool is the metamodel for the TTCN-3 language.

Figure 5 summarizes the steps we follow to compute to come out with the data variability for a TTCN-3 test suite. Each step has a number which is refered in the following explanations.

The first step is to use the TTworkbench metamodel feature to create a metamodel instance out of the TTCN-3 test suite. To understand how this mechanism works we refer to [11]. The instance of the metamodel is used in the second step to create the CFG according to the algorithm presented in this paper. Additionally, the implementation supports also a graphical representation of the CFG's logical structure; the viewer is triggered for a specific selected testcase in the editor. The most important step is to translate the variables and the constraints to a CSP specific notation. We experimented two Java based CSP libraries [13, 14]. We decided to use jOpt [14] due to a larger flexibility in terms of variables types, associated operations and constraint propagation. In the fourth step, the CSP's symbols table is used to resolve the templates and continue with the algorithm presented in the approach.

Though not yet implemented, in the fifth step we forsee the posibility to generate test data out of the data partition information.

6 Conclusions and outlook

After reviewing the main approaches in the field and the existing technologies such as Symbolic Execution and Constraint Solving Problems, we endeavor their applicability to TTCN-3 test specification targeting the test effectivness analysis, particularly test coverage. The results show that the method presented in this paper delivers much better results than the method introduced before in [16].

We demonstrated how the sensitivity of data partition is affected when the distance metric takes into account the set of the constraints upon a message and a specific test execution path. Throughout an example we shown the contrast between the results obtained with the two methods but several limitations have still to be treated (e.g. variable assignement with function calls).

In the end we gave details concerning the improvement of our existing prototype tool to measure the data variability for TTCN-3 test specification on top of a distance metric. The extension regards a better strategy in resolving constraints upon variables in a test path execution according to the CFG in static manner. We also demonstrated how CSP tools have been succesfuly integrated with a TTCN-3 platform.

In future work, the analysis will become more effective with the addition of a dedicated part targeting generation of missing test data as they result from the partitioning method. Both symbolic execution and constraint programming can be computationally very expensive. Complexity analysis will definitelly improve the quality of the approach. Additionally, another aspect of test effectiveness, namely test correctness, can be approached with the same technology: test execution path feasibility, association of a verdict to each path, etc.

References

[1] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[2] J. R. Calame, N. Ioustinova, J. van de Pol, and N. Sidorova. Data abstraction and constraint solving for conformance testing. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 541–548, Washington, DC, USA, 2005. IEEE Computer Society.

[3] R. L. Cilibrasi and P. M. B. Vitányi. The Google Similarity Distance. *IEEE Transactions on Knowledge and Data Enginering*, 19(03):370–383, 2007.

[4] ETSI. Etsi standard es 201 873-1 v3.1.1 (2005-06): The testing and test control notation version 3; part 1: Ttcn-3 core language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2005.

[5] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998.

[6] ISO/IEC. ISO/IEC Standard No. 9126: Software engineering – Product quality; Parts 1–4. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 2001-2004.

[7] A. Jain, M. Murty, and P. Flynn. Data Clustering: A Review. *ACM Computing Surveys*, Vol. 31, No. 3:pp. 264–323, 1999.

[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[9] NASA. Java pathfinder JPF. http://javapathfinder.sourceforge.net/, 2007.

[10] NIST. Euclidean distance. http://www.nist.gov/dads/HTML/euclidndstnc.html, 2004.

[11] I. Schieferdecker and G. Din. A meta-model for ttcn-3. pages 366–379, 2004.

[12] H. M. Sneed. Measuring the Effectiveness of Software Testing. In S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, editors, *Proceedings of SOQUA 2004 and TECOS 2004*, volume 58 of *Lecture Notes in Informatics (LNI)*. Gesellschaft für Informatik, 2004.

[13] SourceForge. choco. `http://choco-solver.net/index.php?title=Main_Page`.

[14] SourceForge. jOpt. `http://jopt.sourceforge.net/index.php`.

[15] TestingTechnologies. TTworkbench: an Eclipse based TTCN-3 IDE. `www.testingtech.de/products/ttwb_intro.php`.

[16] D. Vega, I. Schieferdecker, and G. Din. Test Data Variance as a Test Quality Measure: Exemplified for TTCN-3. In *TestCom/FATES*, pages 351–364, 2007.

[17] D.-E. Vega and I. Schieferdecker. Towards quality of TTCN-3 tests. In *Proceedings of SAM'06 – Fifth Workshop on System Analysis and Modelling (formerly SDL and MSC Workshop), May 31st-June 2nd 2006, University of Kaiserslautern, Kaiserslautern, Germany, 2006*, 2006.

[18] B. Zeiss, H. Neukirchen, J. Grabowski, D. Evans, and P. Baker. Refactoring and Metrics for TTCN-3 Test Suites. In R. Gotzhein and R. Reed, editors, *System Analysis and Modeling: Language Profiles*, volume 4320 of *Lecture Notes in Computer Science*. Springer, 2006.

[19] B. Zeiß, D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski. Applying the ISO 9126 Quality Model to Test Specifications Exemplified for TTCN-3 Test Specifications. In *Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI). Copyright Gesellschaft für Informatik*. Köllen Verlag, Bonn, Mar. 2007.

[20] J. Zhang, C. Xu, and X. Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. *sefm*, 00:242–250, 2004.