Application of Clustering Methods for Analysing of TTCN-3 Test Data Quality

Diana Vega
Technical University of Berlin
Franklinstr. 28/29
vegadiana@cs.tu-berlin.de

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

Stefan Taranu
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31, Berlin
stefan-liviu.taranu@fokus.fraunhofer.de

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

Abstract

The use of the standardised testing notation, Testing and Test Control Notation (TTCN-3) [5] language has increased continuously over the last years. Many test suites of large sizes covering different domains exist. Therefore, it becomes important to provide the TTCN-3 community with methods and tools to evaluate the quality of tests.

This paper presents the idea of evaluating the quality of the test data stimuli by using a data clustering method and measuring the coverage related to data clusters. A cluster contains stimuli which are considered similar for the system under test (SUT) behaviour; that means that each stimuli within a cluster should provide similar results from the test point of view.

1 Introduction

Producing and identifying effective tests is a challenging task and a very debated subject in the testing research [12, 1, 10, 6]. The answer to the question, 'what makes a test suite good?' is relative and involves different views and characteristics [17]. An important role for test quality, if not the most important role, has the test data used to stimulate the SUT, i.e. stimuli, and observe its reaction. This paper presents the idea of evaluating the quality of the test data stimuli by using a data clustering method and measuring the coverage related to data clusters. Clustering is the data classification of similar data into different groups (clusters). Data clustering is a common method for domains where data sampling and processing is needed [8]. We apply this idea in order to create clusters for TTCN-3 types. From the testing point of view, a cluster should contain stimuli which are considered similar for the SUT behaviour, i.e. produce equivalent results for the test purpose. There are many options to split a type into clusters. Therefore, we consider important to create clusters for the most significant values (e.g. boundaries of integer ranges, dictionaries for charstring types). However, once the default clusters are created, the user has still the possibility to create further clusters.

The clustering method is applied for each type used to create stimuli during a test. Once the clusters have been created, the next step is to verify which clusters are actually covered by the test suite. We determine then the coverage as the number of covered clusters out of all possible clusters. For this purpose, the test data stimuli have to be identified and prepared for evaluation. Due to the high flexibility of the language, the stimuli templates may not contain concrete values but also variables, function calls, parameters, values which are known at runtime only, etc. Therefore, we use Constraint Programming (CP) - a software technology for effective solving problems - 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 analysed test behaviour. This method leads to a better computation of the data and, thus, to a better classification into clusters.

The maximal coverage is reached when all clusters are covered, i.e., at least one stimulus exists for each cluster. The coverage can be investigated for each particular type but also globally by computing the average coverage among all types.

2 Related Work

A framework for analysing different quality aspects of test specifications is provided in [17]. It proposes a qual-

978-0-7695-3372-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ICSEA.2008.44

ity model for test specifications and it is derived from the ISO/IEC 9126 [7] quality model. Various test metrics have been already developed measuring selected aspects [14, 3, 16], but they are still in their infancy or regard test quality aspects rather at programming language level than at test specific expectations.

Data variance is investigated by inspecting the *proximity of objects*. The survey in [8] 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 representation.

Cluster analysis is an analysis method for finding groups of clusters in a population of objects. The goal of cluster analysis is to partition a population into clusters in such a way that objects with similar attribute values are placed in the same cluster, while objects with dissimilar values are placed into different clusters. The similarity or dissimilarity is measured using *dissimilarity metrics*, as for instance the Euclidean Distance [13]. Another example is the semantical distance function on pairs of words or terms, entitled *Google distance* which has been suggested in [2].

3 TTCN-3 Test Data Selection

A TTCN-3 test suite consists of many testcases structured into modules. A testcase creates test components and connects them to the SUT over test ports. 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. Each testcase defines also a test behaviour which consists of program statements such as communication operations and data processing operations.

The TTCN-3 template mechanism provides the possibility to specify and structure test data. A template can describe concrete values or specify subsets of values of a given data type. Therefore, templates can be used to create both test stimuli and matching patterns for the SUT responses.

The first step in test data selection is a static analysis of TTCN-3 test suites, that is, no runtime values, elements, etc. are concerned, but the TTCN-3 test specification only. The target is the determination of the data set which forms the input space to be evaluated, i.e *TTCN-3 stimuli templates*. To accomplish that, we propose a *template subset* selection algorithm. To simplify the problem, we assume that all ports which belong to a TSI are message-based ports which can transport data to the SUT [1]. To easy the investigation, we also made the assumption that the behaviour of each testcase runs on *the main test component* (MTC) only, that is, no parallel test components are involved.

[1]This is in fact not a limitation, but a precondition that test data can be sent to the SUT via ports

3.1 Template Set Selection Algorithm

For every testcase, the TSI is indicated within the *system* clause. We consider the coverage at TSI level, for each port, and for each type *T* of data exchanged through the port. Therefore, it makes sense to select and determine the number of clusters regarding the SUT stimuli for a specific type using a specific port. The selection algorithm consists of the following steps:

1 determine all TSIs in a test suite
2 for each TSI, identify the set of all testcase having as *system* that TSI. The set is denoted $\{TCs_{TSI}\}$
3 identify all ports and all *in/inout* data types for each port
4 for each type *T*, and its associated port *pTSI*, initialise an empty set of templates $\{templates_{T,pTSI}\}$.
5 for each type *T*, and its associated port *pTSI*, determine in $\{TCs_{TSI}\}$ all *map* statements having as *system port* the name *pTSI*. Collect all port names found in map statements, i.e. ports on the main test component; let us denote the port set $\{MappedPorts_{pTSI}\}$.
6 for each element *p* in $\{MapedPorts_{pTSI}\}$, search in all test behaviours regarding $\{TCs_{TSI}\}$ after *p.send(message)* statements. Add the *message* to the set $\{templates_{T,pTSI}\}$.

The algorithm generates all possible $\{templates_{T,pTSI}\}$ sets, where each set consist of all messages of the *same type* sent over the *same port* of the *same TSI*.

3.2 Refinement of the Template Set using Constraint Programming

The TTCN-3 semantic permits various ways to assign values to templates or to fields of templates such as: direct values, i.e., integer, charstring values, variables, user-defined function calls, language specific function calls, testcase parameters, module parameters, expressions.

The degree of the completeness of a template definition impacts on the template set selection. A template reference can be used as stimuli in more than one testcase, but using different parameters. This results in many occurrences of the same template in the input space. Hence, to come up with a *template solver*, we investigated *symbolic execution* [9] technique. This technique 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)[11] have successfully adopted this technique to prove the correctness of a program.

Following this method, we built first the CFG associated to a testcase behaviour which contains a stimuli-message. Then a TTCN-3 symbolic template solver built with the help of CFG and constraint programming methods (CP) targets the domain reduction for a specific template field. The solution to the identified constraints upon template fields along an execution path, helps to individualise the template references which have many occurrences in the initial template set. We propose the following algorithm:

1. for each template from the template set identify the TTCN-3 behavioural 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 behavioural entity (e.g. testcase, function).
   (b) localise 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.

4 Coverage Computation Approach

The method to compute the test data coverage is based on the concept of test data clustering. The test data clusters are derived from the SUT specification and the coverage method computes how many clusters are covered by the test data created in the test specification. As illustrated in Figure 1, the SUT model is used to derive the data clusters. In TTCN-3 test specifications, the SUT can also be modelled within the test specification by means of ports and types of messages interchanged between the test system and SUT over those ports. This allows us to compute the data clusters using the test specification itself. Once the clusters have been identified, each stimulus, i.e., template, is investigated and placed in one cluster. The coverage is defined as the number of covered clusters out of all possible clusters.

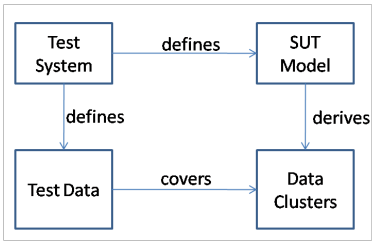

Figure 1. Cluster Derivation

According to [4], in clustering theory two main approaches are used: *partitioning* and *hierarchical*. The *partitioning* technique divides the data into k clusters, for a chosen k. The *hierarchical* approach is a stepwise process for generating clusters. Compared to *hierarchical* approach, the *partitioning* cluster analysis produces better high-quality clustering but it is computationally also more expensive. For this reason, we selected the first method. However, we plan to explore and apply the second method later on.

The clusters need to be created for each type of TTCN-3 template used as stimuli. For basic types the clusters are created from the type ranges derived from the SUT model. The clusters of structured types result as a combination of the clusters of the fields. However, the initial clusters are only default clusters, therefore, they can be further extended and refined.

As long as TTCN-3 supports 'omit' value for optional fields in structured types, the 'omit' is considered as separate cluster. This implies that the 'omit' value has to be a separate test stimuli in order to cover this cluster as well.

4.1 Clustering of Basic Types

Integer. The integer values used in test stimuli are defined as integer ranges. We distinguish between ranges including the '0' number and ranges not including the '0' number.

For a range $[a..b]$ not including '0' the default clusters are:

$[a](a..T](a+T..b-T)[b-T..b)[b]$

First of all, the margins of the ranges are considered as separate clusters as they need to be separate stimuli in the test system. Next, values from the sub-ranges around the margins are also important test stimuli. Therefore the $(a..T)$ and $(b-T..b)$ clusters are created, where 'T' is a threshold to configure the size of the cluster. This value can be arbitrary selected. For convenience, we introduced only one global threshold. However, the thresholds in the two ranges can be different. The rest of the range constitutes a separate

cluster.

If the '0' number is included in the range then additional clusters are generated around this number as follows:

$[a](a..T](a+T..0-T)(0-T..0)[0](0..0+T)[0+T..b)[b]$

The number '0' is considered a separate cluster since this value needs to be a separate stimuli. Also the values around '0' up to a threshold T are considered important, therefore, two additional clusters a created for these ranges.

Additionally, the cluster number can be extended by dividing the ranges into further subranges. This increase of the number of clusters will impact the coverage for that type by improving the level of detail.

The clusters of float types are created in a similar manner as for integer type.

Charstring. Charstring types are used in test specification to define meaningful words such as names, countries, cars, etc. Therefore we consider the dictionary of the possible meaningful words as being a separate cluster. However, this cluster includes all valid values. In order to cover also invalid values, further clusters need to be defined. Therefore, we consider three further clusters based on size: cluster of strings with minimal length, cluster of strings with maximal length and, finally, cluster of strings with length between minim and maxim.

The clusters of other string based types, e.g, bitstring, octetstring, hexstring, are created in a similar manner as for charstring type.

Boolean. The boolean types are separated into two clusters true and false as these values need to be tested by separate test stimuli.

4.2 Clustering of Structured Types

The clusters of structured types are created as combination of clusters of fields so that each cluster of a given field is combined with any cluster of another field. We provide as example the clustering of **record** type.

For the following record type:

type record r:= {nameType name, ageType age}, where,
type charstring nameType length 20;
type integer ageType (18..67);

there are 20 clusters possible: 4 clusters given by the charstring type multiplied by 5 clusters given by the integer type. We provide only the combinations between the dictionary_cluster of `name` charstring and the clusters of `age` integer: {dictionary_cluster,{18}}, {dictionary_cluster,(18..20]}, {dictionary_cluster,(20..65)}, {dictionary_cluster,[65..67)} and {dictionary_cluster,{67}}.

4.3 Cluster Coverage

The templates used as stimuli for the SUT are sorted out into clusters. If a template is a constraint one, then it may cover more than one cluster at the same time. For instance t := [0..100] can be in the cluster of '0', in (0..T) but also in (T..100).

The coverage is computed separately for each type as the number of covered clusters out of the total number of clusters. However, a global coverage number can be computed as the average of coverages of all types.

5 An Example

In this section, we show how to apply the introduced concepts to a TTCN-3 example. The SUT is a web service with an interface which supports queries in a hotel database, by specifying the *city*, availability on a *start date* during a number of *days*, the *range* for the price given as *minimum* and *maximum* price, the number of *stars* for the hotel and whether it offers *internet* access.

Data Types. The test suite defines the types presented in Listing 1. We defined two main TTCN-3 types: `ReqType`, for messages to be sent to SUT and `RespType` for messages to be received from the SUT. In a `ReqType`, a *city* is represented as a `charstring`, the *startDate* as a `record` structure having two fields - *day* as an `integer` from 1 to 31 and *month* as an `integer` from 1 to 12. The next field is an optional field, *priceRange* of `record` type having also two fields: *minimum* and *maximum* price as positive `floats` (in TTCN-3 this constraint is expressed as a range `(0.0..infinity)`). The last two fields of the `ReqType` are: *stars* as an `integer` between 0 and 5 and the optional field *hasInternet* of type `boolean`.

Listing 1. TTCN-3 Types Definition

```
1  // module parameters
2  modulepar charstring mpCity := "Berlin";
3  modulepar StarsType mpStars := 2;
4  modulepar boolean mpHasInternet := true;
5  // data types
6  type integer StarsType (0..5);
7  type integer PositiveIntegerType (0..infinity);
8  type record PriceRangeType {
9       float min (0.0..infinity),
10      float max (0.0..infinity)
11 }
12 type record DateType {
13      integer day (1..31),
14      integer month (1..12)
15 }
16 type float PriceType;
17 type record ReqType {
18      charstring city, DateType startDate,
19      PositiveIntegerType days,
20      PriceRangeType priceRange,
21      StarsType stars, boolean hasInternet
22 }
23 type record HotelType {
24      charstring city, PriceType price,
25      StarsType stars, boolean hasInternet
26 }
```

```
27 type record of HotelType RespType;
```

The response from the SUT, `RespType` represents a list of the available hotels which satisfy the query. Each hotel is described using the record `HotelType` containing the *city*, the *price*, *stars* and *hasInternet* fields.

Test Configuration. Listing 2 describes the *port* and *component* types that are used in the test configuration. We defined a main test component type, `MTCType`, and a system component type, `TSIType`. The communication is realised using message-based ports of type `MTCPortType` on the test side, and `TSIPortType` on the system side. Since our coverage computation method regards the TSI coverage, the analysed TTCN-3 type is `ReqType` because it is the only stimulus type and there is only one port *pTSI* of the TSI component.

Listing 2. TTCN-3 Test Configuration

```
1  type port MTCPortType message {
2    out ReqType;
3    in RespType;
4  }
5  type port TSIPortType message {
6    out RespType;
7    in ReqType;
8  }
9  type component TSIType{
10   port TSIPortType pTSI;
11   timer t := 10.0;
12 }
13 type component MTCType{
14   port MTCPortType pMTC;
15   timer t := 10.0;
16 }
```

Templates. For stimulating the SUT, three templates of type `ReqType` have been defined (see Listing 3). In TTCN-3, the sending templates have to be always completely specified, i.e. no usage of patterns or wildcards within their definition. As oracles used in establishing the the test verdicts, two templates of type `RespType` complete the template list. While the *resFoundTmpl* template embeds a non empty list of hotels that SUT may return as response, the *resNotFoundTmpl* defines a response encoding the lack of hotels satisfying the query.

Listing 3. TTCN-3 Templates

```
1  // Request Templates
2  template ReqType req3StarsTmpl:= {
3    city := mpCity,
4        startDate := {day:=1, month:= 6},
5        days := 4,
6        priceRange := {min:=0.0, max:=100.0},
7        stars := 3,     hasInternet := true
8  }
9  template ReqType req5StarsTmpl:={
10       city := mpCity,
11       startDate := {day:=1, month:=6},
12       days := 4,
13       priceRange := {min:=0.0, max:=100.0},
14       stars := 5,     hasInternet := true
15 }
16 template ReqType req6DaysTmpl:= {
17       city := mpCity,
18       startDate := {day:= 1, month:=2},
19       days := 6,
20       priceRange := {min:=0.0, max:=50.0},
21       stars := 5, hasInternet := true
22 }
23 // Response Templates
24 template RespType
25       resFoundTmpl := ? length (1..1000);
26 template RespType
27       resNotFoundTmpl := {}
28 }
```

Testcases. The test suite defines three testcases. The first testcase (Listing 4) tests if the SUT finds a hotel with 3 stars cheaper than 100 Euro. As a preamble, the SUT database contains at least one hotel with 3 stars and below than 100 Euro. The testcase sends the request template `req3StarsTmpl` and expects as result a non empty list of hotels. After sending the search request a timer is started in order to validate that the SUT responds in an acceptable amount of time.

Listing 4. Testcase 1

```
1  testcase t1() runs on MTCType system TSIType {
2        map (self:pMTC, system: pTSI);
3        p.send(req3StarsTmpl);
4        t.start;
5        alt {
6      [] p.receive(resNotFoundTmpl)
7           { setverdict(fail);}
8      [] p.receive(resFoundTmpl)
9           { setverdict(pass);}
10     [] t.timeout { setverdict(inconc);}
11   }
12 }
```

The second testcase (Listing 5) tests that the SUT finds a hotel cheaper than 100 Euro with an arbitrary number of stars. The test behaviour starts with the sending of `req5StarsTmpl` request which means that we search first for a hotel of five stars with a maximal price of 100 Euro. If no hotel is found, the test decreases the number of stars and tries again. This can repeat 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 test stops with verdict `pass`. The duration of each search operation is validated by a timer.

Listing 5. Testcase 2

```
1 testcase t2() runs on MTCType system TSIType  {
2   map (self:pMTC, system: pTSI);
3   var ReqType vReqA := req5StarsTmpl;
4   p.send(req5StarsTmpl);
5   t.start;
6   alt {
```

```
7      [] p.receive(resFoundTmpl)
8           { setverdict(pass);}
9        [] p.receive(resNotFoundTmpl) {
10             t.stop;
11             if(vReqA.stars >= 1) {
12                vReqA.stars := vReqA.stars -1;
13                p.send(vReqA);
14                t.start;
15                repeat;
16             }else { setverdict(fail);}
17       }
18       [] t.timeout { setverdict(inconc); }
19   }
20 }
```

The third testcase (Listing 6) tests that the SUT can find a hotel with a price below 50 Euro for 6 nights between 1.02-31.03 in Berlin. As a preamble, we assume that the SUT database contains a hotel available from 5.02 for 6 nights for 45 Euro. The test behaviour starts with the sending of `req6DaysTmpl` request which means that we search first for a hotel with a maximal price of 50 Euro available for 6 days, starting from 1.02. If no such hotel is found, the tests increases the starting date so that the upper limit of time interval as well as the availability for 6 nights are satisfied. In the test logic, this is expressed as a condition in the body of the second `alt` alternative:

```
if(vReqB.startDate.day < 25 and
vReqB.startDate.month <= 3
```

If in the meantime a hotel is found, then the tests stops with verdict `pass`. In the case when the requested start date reached the upper limit of a convenient time, the test decreases the number of stars and tries again. This can repeat 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 `pass`. The duration of each search operation is validated by a timer.

Listing 6. Testcase 3

```
1  testcase t3() runs on MTCType system TSIType {
2    map (self:p, system: p);
3        var ReqType vReqB:= req5StarsTmpl;
4        var RespType vResponse;
5        p.send(req6DaysTmpl);
6        t.start;
7
8        alt {
9          [] p.receive(resFoundTmpl)
10              { setverdict(pass);}
11         [] p.receive(resNotFoundTmpl) {
12              t.stop;
13            if(vReqB.startDate.day < 25 and
14                 vReqB.startDate.month <= 3)
15               {increment(vReqB.startDate);
16              }else {
17                 if(vReqB.stars >= 1) {
18                   vReqB.stars := vReqB.stars -1;
19                   vReqB.startDate := {1, 2};
20                      p.send(vReqB);
21                      t.start;
22                      repeat;
23                 }else { setverdict(fail); }
24              }
25         }
26         [] t.timeout
27              { setverdict(inconc); }
28       }//end alt
29 }
```

This test suite contains only one *template set* since the TSI has only one *port type* which accepts requests. The request are of the same *message type*. The template set consists of the templates that appear in `send` statements within the three test behaviours:

Set={*req2StarsTmpl, req5StarsTmpl, req6DaysTmpl, vReqA, vReqB*}.

The first three templates are defined such that all fields contain concrete values except the field *city* which is assigned the module parameter *mpCity*. This field has to be resolved by the template solver. The forth and fith messages in the set are variables which are defined locally in the second and in the third testcase. The variable vReqB appears in the *send* statement (line 20, Listing 6). The content of the variable is changed in line 3 (it is initialised with the template *req6DaysTmpl*), line 18 (the field *stars* is decreased) and line 19 (the field *startDate* is reset). Additionally, it also appears in the decision statement on the lines 13-14 where the *startDate* checked and on the line 17 where the number of stars is compared with value 1. All these constraints need to be solved.

For each testcase a CFG is built. We note that only one path exists from the starting point in the graph to the *send* statement. This means that no variation of the initial *template set* occurs as long as only one path to the send statement exists. Next, the path conditions are identified and the constraints on the templates are computed. For instance, the variable *vReqB* has associated the explicit constraint *vReqB.startDate* = {*1, 2*}. Additionally, if we consider the repetition of the testcase behaviour (possible due to the `repeat` statement) the number of *stars* can be decreased. Another constraint characterising the variable *vReqB* is *vReqB.stars == 4*. This is an implicit contraint which is computed by decreasing the initial value of the field *stars*. The initial value of the field *stars* is inherited from the template *req5StarsTmpl* (line 3, Listing 6).

The next step regards the clustering computation. The required number of clusters to be covered is determined starting with the type `ReqType` definition. It is a record based type, therefore, the number of clusters is deduced as the product of number of clusters associated to each leaf in the tree structure of the type. The `ReqType` contains 8 leaves, hence the product is of maximum 8 numbers. For example, the leaf-type `day`, described as `integer day (1..31)`, has five clusters: {*1*}*(1..2)*(2..11)[11..12)*{*12*}. The threshold of the boundaries has been taken as 1.

Figure 2 shows the leaf-clusters for the type `ReqType`.

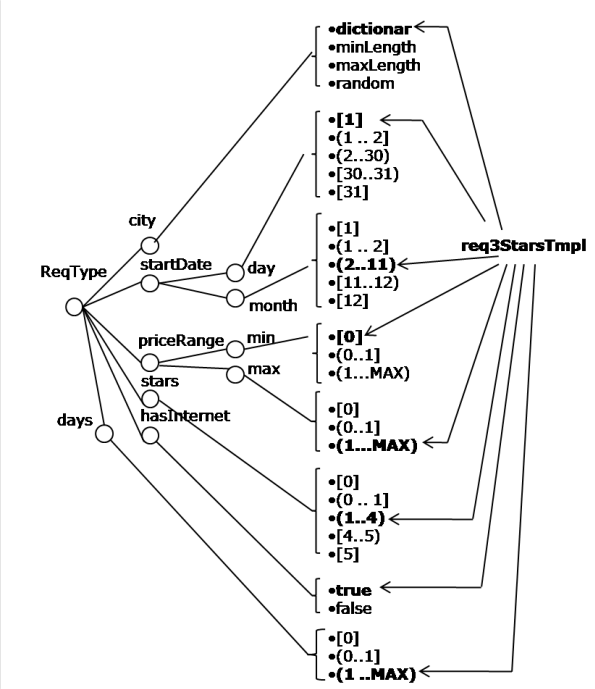

Figure 2. Coverage of ReqType Clusters

They have been derived from the basic type definitions, e.g. type integer `StarsType` (0..5). Additionally, the figure shows that 7 leaf-clusters are covered by `req3StarsTmpl` template. This means that only one cluster of the `ReqType` clusters is covered by the `req3StarsTmpl` template.

6 Prototype Implementation

To carry out automatically the coverage computation of a given TTCN-3 test suite, we built a framework based on the TTworkbench [15] product, an Eclipse based IDE that offers an environment for specifying and executing TTCN-3 tests. We present the main features of our framework.

Coverage Computation. The main trigger action *"Compute Coverage"* applied to a module in a TTCN-3 project, has as a first result the appearance of a new view *"TTCN-3 Data Variance View"*(Figure 3). It displays in a tree format the identified TSIs and their associated ports and types. Each node is accompanied by a number representing the template set size. The number has been obtained by applying the algorithm for initial template set selection and the refinement algorithm by using constraints programming.

Leaves selection. Given a complex TTCN-3 structured type, one may not be interested in all the fields in a clustering process of a template set. Instead, it is preferred to identify clusters considering relevant leaves only. We captured this feature in our framework as well. Figure 4 shows

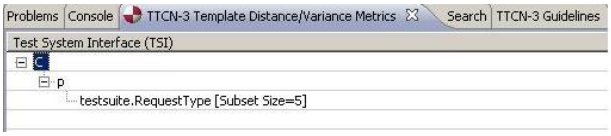

Figure 3. TTCN-3 Data Variance View

what the wizard displays in the case of a record type. If a leaf is not of interest, the linked check-box has to be simply unchecked. By default all of them are checked.

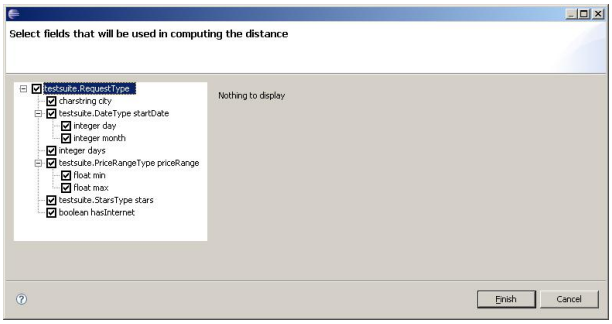

Figure 4. TTCN-3 Type Tree

Cluster visualisation. If one field is retained, then, according to its basic type definition, e.g. boolean, integer, charstring, a set of automatically derived clusters is generated and displayed in the right panel of the wizard. Figure 5 and Figure 6 show which clusters have been generated for the fields of type boolean and integer, as they have been defined in the Section 5.

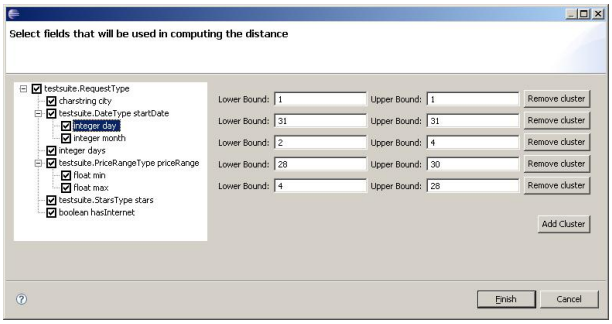

Figure 5. Automatically generated clusters for an integer-based type

Link to dictionary. For charstring types, a link to a user-defined dictionary is supported. All charstring templates found in the dictionary belong to that cluster.

New cluster creation. For basic types additional clusters can be created by using *"Add Cluster"* button or removed by using *"Remove Cluster"* button. The user has then to specify them concretely, e.g. for numeric type to introduce the limits of the range. Based on the specified *leaf-clusters*, the clusters for the whole structured type are

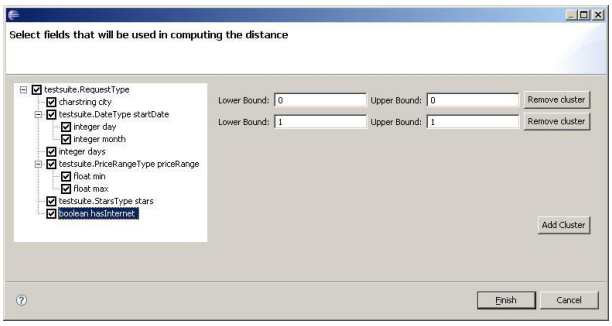

Figure 6. Automatically generated clusters for a boolean type

then computed (at *"Finish"* action in the wizard).

Coverage visualisation. In the end, the tools counts the coverage of the template set over the established clusters. The *Port-Type Coverage* is computed and displayed.

CFG visualisation. Additionally, the implementation supports also a graphical representation of the CFG's logical structure. While GUIs help more for debugging purposes, e.g. to identify in a visual manner the paths among the graph, the logical structure of the CFG brings more advantages along the template set computation process.

7 Conclusions

In this paper we investigated the use of a clustering method for TTCN-3 stimuli templates for measuring the coverage of test data. Throughout an example we shown how the proposed algorithms can be applied. They regard 1) the collection of the test data input space used to stimulate the SUT and using the TTCN-3 specification only and 2) how the identified templates cover a set of clusters derived from the underlying TTCN-3 stimuli types. The concepts have been integrated in TTworkbench [15] in order to automate the static analysis process. We also intent to apply them for larger systems.

As a next step, we plan to improve the clustering method by adding priorities for clusters and improving the coverage formula. However, the main target remains the test specification optimisation by automatically generating of the uncovered test data clusters.

References

[1] R. T. Alexander, J. Offutt, and J. M. Bieman. Fault Detection Capabilities of Coupling-based OO Testing. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 207, Washington, DC, USA, 2002. IEEE Computer Society. ISSN 0-8186-1763-3. IEEE Computer Society.

[2] R. L. Cilibrasi and P. M. B. Vitanyi. The Google Similarity Distance. *IEEE Trans. on Knowl. and Data Eng. IEEE Educational Activities Department*, 19(3):370–383, 2007. ISSN 1041-4347.

[3] D. 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.

[4] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, Washington, DC, USA, 2001. IEEE Computer Society.

[5] European Telecommunications Standards Institute (ETSI). European Standard (ES) 201 873-1 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2007. Sophia-Antipolis, France.

[6] M. Grindal, B. Lindstrom, J. Offutt, and S. F. Andler. An Evaluation of Combination Testing Strategies. *Kluwer's Empirical Software Engineering*, 11(4):583–611, December 2006.

[7] 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.

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

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

[10] B. Korel and P. J. Schroeder. Maintaining the Quality of Black-Box Testing. `http://www.stsc.hill.af.mil/crosstalk/2001/05/korel.html`, May 2001.

[11] NASA. Java PathFinder (JPF). `http://javapathfinder.sourceforge.net/`.

[12] P. Netisopakul, L. White, J. Morris, and D. Hoffman. Data Coverage Testing of Programs for Container Classes. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 183, Washington, DC, USA, 2002. IEEE Computer Society. IEEE Computer Society. ISSN 0-8186-1763-3.

[13] NIST. Euclidean Distance. `http://www.nist.gov/dads/HTML/euclidndstnc.html`, 2004.

[14] 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.

[15] T. Technologies. TTworkbench: an Eclipse based TTCN-3 IDE. `www.testingtech.de`.

[16] 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.

[17] 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, March 2007.