Implementing heuristic testing selection

NICOLAE GOGA
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
THE NETHERLANDS

Abstract: [1] elaborates an abstract theory for test selection. The link between the theory and the practice was not in the focus of that paper. The current paper fills this gap by presenting ways of implementing the abstract formulas of test selection. Several examples of useful distances are given.

Key-Words: test selection, trace distances, public telephony systems, reactive systems, coverage

1 Introduction

[1] elaborates an abstract theory for test selection. The link between the theory and the practice was not in the focus of that paper. The current paper fills this gap by presenting ways of implementing the abstract formulas of test selection. Several examples of useful distances are given. These examples add plausibility to the claim that the theory is useful in practice. Moreover they are a source of inspiration for other testers which wants to use the theory of test selection presented in [1]. Also the work described in this paper is a step forward in using this selection theory in the TorX testing environment [2].

The implementation of the abstract theory of test selection is in the form of a C program. The C program is not connected to TorX. It is an independent module. The functions contained in this program form the basis of a new module which can be linked to TorX later. While describing the main features of this program, different ways of connecting it to TorX will be presented. Moreover we will indicate other possible connections to other test generation tools like Autolink [4]. This is one of the reasons for which we kept the module tool-independent.

We tried to keep the program simple to make its understanding easy, but general enough to be used in realistic application domains such as classical public telephony and to be connected later to TorX or other test generation tools. Keeping the program simple was possible by making it tool independent. In this way it does not inherit complicated data structures and algorithms that are specific for e.g. TorX.

In this paper we will not present all the functions of the C program in detail. In this paper, only the main algorithms of the program are presented. The program contains two modules named *Unfold* and *Distance* which are described in Sections 2 and 3. Section 4 outlines the conclusions.

2 The module unfold

The main function of the *Unfold* module is called *Unfold* like the name of the module itself. Let us consider an automaton of a specification. The objective of the *Unfold* function is to generate all the traces which do not cycle more than a cycle limit, l_c, through the states of the automaton. The set of traces generated by *Unfold* can be seen as the test purpose (automatically generated) which is used by TorX to derive test cases [6]. The theory of test purposes from [6] considers a set of traces as being the test purpose of TorX. The *ioco* test derivation algorithm from [5] which is implemented by TorX is modified slightly such that for each trace from the set a test case is generated. A module which will be integrated in TorX and which will work with

test purposes is under development at the University of Twente.

As we explained, the set of traces derived with *Unfold* can represent a test purpose objective for TorX. But not only for this tool such a set of traces can represent a test purpose. Let us take another test derivation tool, namely Autolink. Each trace from this set can be transformed into a correspondent MSC [3]. A function which will implement such a transformation can easily be incorporated in the *Unfold* module. This MSC can be the test purpose used by Autolink for deriving test cases. Such way of building test purposes can represent an alternative way to the ones provided by the Autolink tool. In this way the work presented here can give useful insights to other test related communities.

Example Consider the automaton from Figure 1. This automaton has two states labelled with 0 and 1. The initial state is 0. From 0, via a the state 1 can be reached. In this state, the transition b leads to the same state and the transition y to 0. Now let us fix the cycle limit to 1.

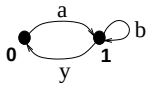

Fig.1 An automaton.

When applying the *Unfold* algorithm on this automaton for $l_c = 1$ the set of traces obtained is $\mathcal{F} = \{ayab, abyab\}$ (one should take $\mathcal{F}$ and its prefixes). The corresponding set of marked traces [1] is: $\{[\langle ay\rangle]^{1,0}a[\langle b\rangle]^{1,1}, [\langle a[\langle b\rangle]^{1,1}y\rangle]^{1,0}a[\langle b\rangle]^{1,1}\}$. We remind that on the first place of the power of a term $[\_]^{\cdot,\cdot}$ is the cycle number and on the second place is the state name. A look at the corresponding set of marked traces shows that none of these traces passes the limit.

3 Algorithms for distance computation

In this section we present the functions from the module *Distance*. As shown by its name, the functions defined here deal with distances.

In [1], for making test selection, we considered a distance between the labels of an automaton and another distance between the traces of an automaton. A formula for the label distance is not defined in the general theory from [1]. In the general definition, the label distance is considered to be a parameter of the trace distance. In our C program we consider it also to be one of the parameters of the trace distance. Therefore if one wants to use this program it is necessary to define one's own distance.

There are many ways to define a label distance by using different approaches. For example one can think of defining a label distance suitable for a boundary value analysis. Another label distance can be suitable for a random selection of a number of values. Considering the label distance as a parameter of the program leaves freedom to use the *Distance* module in a convenient way.

We will give some examples of how a label distance can be constructed. We will define a label distance for a network of telephones. Defining a label distance for such a telephone network will give useful insights for defining other label distances for other applications. Together with the label distance we will present also the *Reduction* heuristic applied to the labels of the telephone's transition system.

For a phone, there are situations in which signals have parameters which can take different values: for example the phone numbers or the messages exchanged. We will show how to define a label distance first for the phone numbers x and after that for the messages *mes*. For all other non-trivial cases the label distance is 1 because all the other labels are necessary for testing different behaviours of the phone (for identical labels the label distance is 0).

A phone number is formed by: 1) a country prefix; 2) a city prefix and 3) a local phone number. We assume that the goal is to test phone connections between different countries from the European Union and different cities from the Netherlands. We restrict the domain to a certain connected subset of the countries from the European Union: Austria (A), Belgium (B), Denmark (D), France (F), Germany (G), Italy (I), Luxembourg (L), the Netherlands (N), Portugal (P) and Spain (S). The names are given in alphabetical order. The prefixes of the countries are given in the following table.

Table 1

A	B	D	F	G	I	L	N	P	S
43	32	45	33	49	39	352	31	351	34

When replacing x with a phone number from one of these countries the digits of the country prefix will

be present on the first positions of the phone number (which is represented as a char-string). We reserved the first 5 characters for a prefix, because the longest prefix has 5 numbers (for Luxembourg or Portugal). For the rest of the countries the first digit is a space. For example, for France the prefix is represented as ' 0033'. The country prefixes are kept in the C program by the global variable *Prefix_Country*.

As we mentioned a phone number is built by three parts: the country prefix, the city prefix and the local number. This decomposition is reflected in the label distance in the following way: we built the label distance as a sum of atomic weighted distances. Let us consider two phone numbers x and x' which have the country prefixes *PCtr* and *PCtr'*; their city prefixes are given by *PC* and *PC'* and the local numbers are *PL* and *PL'*. Then

$$DlPhNr(x, x') = w_1 \times Dist_1(PCtr, PCtr') + w_2 \times$$

$$Dist_2(PC, PC') + w_3 \times Dist_3(PL, PL')$$

where w_1, w_2 and w_3 are the weights; $Dist_1, Dist_2$ and $Dist_3$ are the distances between countries prefixes, city prefixes and local numbers and *DlPhNr* represents the distance between two phone numbers. We will show the computation for each of the three distances below.

For the country prefixes we considered that a natural way for the occurrence of errors is when a call is routed through different telephone networks of the different countries. For example, taking the shortest path, between the Netherlands and France a phone call should go through the Belgian telephony system. Following this reasoning, we consider the number of border crossings a good candidate for computing the distance between two country prefixes. The numbers of border crossings between all pairs of countries are given in the following table.

Table 2

	A	B	D	F	G	I	L	N	P	S
A	0	2	2	2	1	1	2	2	4	3
B	2	0	2	1	1	2	1	1	3	2
D	2	2	0	2	1	3	2	2	4	3
F	2	1	2	0	1	1	1	2	2	1
G	1	1	1	1	0	2	1	1	3	2
I	1	2	3	1	2	0	2	3	3	2
L	2	1	2	1	1	2	0	2	3	2
N	2	1	2	2	1	3	2	0	4	3
P	4	3	4	2	3	3	3	4	0	1
S	3	2	3	1	2	2	2	3	1	0

The maximal number of border crossings between two countries is 4 (between Portugal and Austria or between Portugal and Netherlands). The distance $Dist_1$ between two country prefixes is given by the number of border crossings divided by 4 (we scaled the distance back to the range $[0, 1]$). The values from the table above are kept in the C program by the global variable *BorderCross*.

Next we considered cities from the Netherlands. There are three cities considered: Amsterdam, Eindhoven and Rotterdam. The prefixes of these cities are given in the following table:

Table 3

Amsterdam	Rotterdam	Eindhoven
20	10	40

These prefixes are stored in the C program by the global variable *PrefixCity*. The physical distances of the cities are given in kilometers in the following table.

Table 4

	Amsterdam	Rotterdam	Eindhoven
Amsterdam	0	72	119
Rotterdam	72	0	109
Eindhoven	119	109	0

The values from this table are stored in the global variable *DistCity*. The distance between the prefixes of two considered cities from the Netherlands is computed as the physical distance divided by 119. The distance is thus scaled back to $[0, 1]$ dividing by 119. For example, the distance between Amsterdam and Rotterdam is $\frac{72}{119}$ which is 0.60. Considering more cities from the Netherlands, the distances between their prefixes can be computed in a similar style. We assumed that within the telephony system of one country errors are more likely when the distance becomes larger. Between two cities from different countries the distance is considered to be maximal (1) because they are coming from different telephony systems, say two different clusters. For example, we want to test all the inhabitants of Germany. Testing all is impossible. Phoning from the Netherlands to two users in Germany does not make too much difference. In this case the label distance is mostly based on the distance between the country prefixes.

In the case of local numbers, we considered the numbers of the Technical University Eindhoven (TU/e for short) and Philips Research (Philips for short), both institutions located in Eindhoven. Their prefixes are given in the following table:

Table 5

TU/e	Philips
247	274

These prefixes are stored in the C program by the global variable *PrefixCompany*. The physical distance between the companies are given in the following table.

Table 6

	TU/e	Philips
TU/e	0	1
Philips	1	0

These values are stored in the C program in the global variable *DistCompany*. The distance between two local phone numbers is computed as the physical distance between them. Considering more local phone numbers, the distance can be computed in a similar style using the physical distance between the users. Of course, as in the case of the cities, the distance between two local numbers from two different cities is one.

For completing the picture we should say that when defining a phone number in the C program, we used the character '*' which means 'any'. For example the phone numbers ' 0033*' and ' 0045*' means a phone number from France and Denmark for which the city prefixes and the local numbers are not specified. The label distance computations between phone numbers are made in the C program by the Function *DlPhoneNumber*. For example, using ' 0033*' and ' 0045*' which represent two phone numbers from France and Denmark the returned value is:

```
DlPhoneNumber(' 0033*', ' 0045*') = 0.6
```

This computation is made as follows. The weights considered in the C program are: $w_1 = 0.8$, $w_2 = 0.18$ and $w_3 = 0.02$. The values of the weights are stored in the C program by the variables *ScaleCountry*, *ScaleCity* and *ScaleCompany*. These values reflect the assumption that the most important thing is to check the phone communication between the countries, after that between cities and after that between local numbers. For our example, the distance between the city prefixes and local numbers in this case is one. The computation is:

$$0.80 \times \frac{2}{4} + 0.18 \times 1 + 0.02 \times 1 = 0.60$$

We recall that the number 2 from the formula above represents the number of border crossings between France and Denmark. We illustrated the construction of the label distance for the phone numbers. This label distance can be used for performing a *Reduction* heuristic with a given ε approximation on the complete set of phone number labels (which might contain millions of phone numbers) by computing an ε-cover of the whole set. Because the same thing can be done also for the set of messages, first we will define a label distance for messages.

Let us assume that the phones have an SMS (Short Message Service) functionality. SMS is an asynchronous mechanism for the delivery of short messages between mobile telephones. The size of a short message is small. In this context, signals such as *conv_in(mes)/pn_conv(mes)* and *np_conv(mes)/conv_out(mes)* represent the sending and the reception of a short message by the user. It will be shown how to construct a label distance in a way different from the one used for the phone numbers. For messages, it can be assumed that the transmission between telephones of a short message can cause a mutilation of the message. Testing a selected set of messages will indicate whether the transmission is erroneous or not. For making this selection, a label distance is to be defined. Let us also assume that a user can choose from a limited set of words: 'Hello' translated in 6 languages. We use 'Hello' in English; 'Salut' in French; 'Hola' in Spanish; 'Ciao' in Italian; 'Hallo' in Dutch; 'Dav' in Danish.

For defining a label distance between two messages, first the messages are ordered according to how many users make use of each language: first we credited English to take first place (although some of the readers of this paper will disagree with this point of view); the second was French and the third Spanish; the fourth was Italian; the fifth was Dutch and the sixth was Danish. This ordering is not based on any statistics (any ordering can be chosen). Each message from the six gets a corresponding weight which reflects its importance.

The weights of the messages are given in the following table.

Table 7

Hello	Salut	Hola	Ciao	Hallo	Dav
1	0.9	0.8	0.7	0.6	0.5

These values are stored in the C program by the variable *WeightMes*. Now the label distance between two messages is computed as the difference between their weights. For example the label distance between *Hello* and *Hallo* is computed as $\mid 1 - 0.6 \mid = 0.4$. The function which computes the label distance between two messages is called *DlMes* in the C program.

The way in which the distance between messages is computed corresponds to an *enumeration* approach for *Reduction.* In this approach the values are ordered after some rules and the first n values are selected. The values of the messages are ordered such that the label distance starts to decrease for messages which are on remote positions. This label distance is suitable for an enumeration approach because selecting the first n labels will give a good covering of the whole set. In a similar style, in other applications label distances can be built.

For the phone number case, the label distance uses physical distances (expressed in kilometers or in numbers of border crossings). In this case the *Reduction* can be made by giving an ε value and computing the corresponding ε-cover of the whole set. The same thing can be done in the case of the messages. In the C program, the function *Selection* gives for a given ε approximation the subset of labels with a minimum cardinality which is an ε-cover of a whole set of labels.

Example Applying this function to the set of all phone numbers and taking $\varepsilon = 0.325$ (the choice of this value is explained below) leads to the following 0.325-cover.

```
number( 0043 *)/pn_number( 0043*)
number( 0049*)/pn_number( 0049*)
number(00351*)/pn_number(00351*)
```

These three prefixes correspond to the following countries: ‘ 0043’ for Austria, ‘ 0049’ for Germany and ‘00351’ for Portugal. The rest of the countries cluster around one of these three: Italy has one border crossing with Austria; Belgium, Denmark, France, Luxembourg and Netherlands have one border crossing with Germany and Spain has one border crossing with Portugal. The approximation of 0.325 tolerates an error produced by a border crossing. The more borders are crossed by a call, the more likely it is that an error occurs. Testing the communication between Germany and Portugal will increase confidence in the communication between Germany and France, between France and Spain and between Spain and Portugal. Choosing for example to test the communication between Denmark and Spain will lead to the testing of the communication between Denmark and Germany, between Germany and France and between France and Spain. Because the communication between Germany and France and between France and Spain was already tested when testing the communication between Germany and Portugal, the error which can occur is related to the border crossing between Germany and Denmark. A reasoning similar to this one can also be made for other pairs of countries. In this way the whole network of countries is almost tested. Because there is a tolerance for an error produced by a border cross and because the communications between the cities and the local numbers are not tested, this explains why there is still a possibility of errors which leads to a value of 0.325 of the ε approximation.

As explained in the beginning of this section the label distance is used for the computation of the distance between traces. A similar function as *Selection* can be made for regression testing (traditional regression testing is about running all tests over and over again). In regression testing, a set of tests is selected to be reused later for testing IUTs. In the case of TorX the tests are execution traces. Now, using the trace distance d and an ε value, an ε-cover of tests can be computed. The ε-cover represents the most important behaviours from the whole set of execution traces which can be re-used later for testing IUTs.

4 Conclusions

This paper presents a C program which implements the theoretical work from [1] for test selection. This represents a kernel which later can be connected to TorX. The program is kept simple but general enough to make its use in real applications possible. Its use is illustrated by examples taken from the application domain of telephony.

The program is divided in two modules called *Unfold* and *Distance*. The module *Unfold* implements the *Cycling* heuristic while the module *Distance* deals with distance computations.

The set of traces derived with *Unfold* can represent a test purpose objective for TorX. But not only for this tool such a set of traces can represent a test purpose. Let us take another test derivation tool, namely Autolink. Each trace from this set can be transformed into a corresponding MSC. A function which will implement such transformation can easily be incorporated in the *Unfold* module. This MSC can be the test purpose used by Autolink for deriving test cases. Such a way of building test purposes can represent an alternative way to the ones provided by the Autolink tool.

In the *Distance* module a label distance for a phone specification is worked out in detail. This gives useful insights of how similar label distances can be built for other applications. For the phone specification, the label distance was built for two categories of labels: the phone numbers and the messages exchanged.

The way in which we computed the label distance between messages corresponds to an enumeration approach for *Reduction*. In this approach the values are ordered after some rules and the first n values are selected. The values of the messages are ordered such that the label distance starts to decrease for messages which are on remote positions. Selecting the first n labels will give a good coverage of the whole set.

For phone numbers, the label distance was built by using physical distances. In this case the *Reduction* can be made by given an ε value and compute the corresponding ε-cover of the whole set. In the C program this is made by the function *Selection* which computes for a given ε approximation the subset of labels with a minimum cardinality which is an ε-cover of a whole set of labels. Other approaches (different from the one presented in this paper) can also be considered for constructing suitable label distances, such as the boundary value analysis.

The label distance is used for the computation of the distance d between traces. A similar function as *Selection* can be made for regression testing. In regression testing, a set of tests is selected to be re-used later for testing IUTs. In the case of TorX the tests are execution traces. Now, using the trace distance d and an ε value, an ε-cover of tests can be computed. The ε-cover represents the most important behaviours from the whole set of execution traces which can be re-used later for testing IUTs. As it can be seen, the function *Selection* can be used not only for implementing the *Reduction* heuristic but also to give useful insights in the regression testing.

Of course, really connecting this program to TorX is a first possible continuation of this work. Further research should also investigate the efficiency of the algorithms constructed and the implementation of other elements of the theory of test selection.

References:

[1] Feijs, L.M.G., N. Goga, S. Mauw and J. Tretmans, Test selection, trace distance and heuristics, *Testing of Communicating Systems*, I. Schieferdecker and H. König and A. Wolisz (Ed), pp. 267–282, Publisher: Kluwer Academic Publishers, Germany, 2002

[2] Goga, N., Comparing TorX, Autolink, TGV and UIO Test Algorithms, *SDL2001: Meeting UML*, Publisher: Springer, Denmark, 2001

[3] ITU-T, *Recommendation Z.120: Message Sequence Charts (MSC)*, Publisher: ITU-T, Geneve, 1993

[4] Schmitt M., B. Koch, J. Grabowski and D. Hogrefe, A Tool for the Automatic and Semi-Automatic Test Generation, In: *Formale Beschreibungstechniken für verteilte Systeme*, A. Wolisz, I. Schieferdecker, and A. Rennoch (Ed), Publisher: GMD-Studien, Germany, 1997

[5] Tretmans, J., Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts & Tools, Volume No 17(3)*, pp. 103–120, 1996

[6] Vries, R.G. de, Towards Formal Test Purposes. In: *Formal Approaches to Testing of Software 2001 (FATES'01)*, J. Tretmans (Ed), pp. 61 - 76, Publisher: University of Aarhus, Denmark, 2001