An Optimization of the TorX Test Generation Algorithm

Nicolae Goga

In this paper we will discuss the process of automatic test derivation from formal specification. The process will be illustrated in the TORX algorithm. We will present an optimization of TORX. The extension of the algorithm with explicit probabilities leads to improvements in the tests generated with respect to the chances of finding errors in the implementation.

Introduction

Testing plays an important role in the process of detecting the errors of the system implementations. Today, more and more energy is concentrated in building up testing systems which can produce better results in detecting an faulty implementation.

The approach by which a set of behaviours are transformed in tests can lead to the hidden errors not being detected (the method is not based on any theory). A more optimal approach is to use the formal specification and an algorithm for test derivation for obtaining tests which will be able to detect more errors as in the previous approach. This ability is justified because the formal specification which expresses the requirements on which the implementation should work define also what is an error for it. So all behaviours are expressed in the specification and theoretically the test derivation is capable to detect all the errors of the implementation.

There are two ways of test derivation: the manual one and the automatic one. The manual process of test derivation is time consuming and sub–optimal. The automatic test derivation process gains more and more interests. There is much effort in building up theory foundation and tools in this area. One example is the project *Cote–de–Resyste* (CdR) formed by a consortium of Dutch research groups from academia and industry. The tool for automatic test derivation developed by the CdR project was baptized TORX (see [5]).

The TORX tool tries to be an open system and to interconnect its system with a wide range of related tools. With ToRX, several case studies have already been performed (see e.g. [2] and Tretman'ss article in this XOOTIC MAGAZINE).

The TORX test generation tool is based on the *ioco* theory developed at the University of Twente. In the heart of the theory is the *ioco* relation, which formally expresses the assumptions about stimulation and observation during testing. An algorithm for deriving a sound and complete test suite with respect to this relation forms the center of the TORX test generation tool. This algorithm is incorporated in such a way that it can be used both for on–the–fly testing (test generation and test execution are combined in one phase) and batch–oriented testing (test generation and test execution are separated phases).

This algorithm is non-deterministic in the sense that in every state where the system can do both an input and an output a choice must be made between these two. In practice a random generator was used to resolve this non-determinism, which resulted in an equal distribution of chances.

Practical experiments showed that in most cases this equal distribution served very well, but in some cases we encountered an anomalous situation. A case study, concerning an elevator, indicated that the derived test suite was not optimal. Analysis showed that the test suite mostly contained rather uniform test cases with respect to the ratio of inputs and outputs. Thereby neglecting a collection of unbalanced behaviours which were very interesting for this particular case study. The natural solution to this problem is to extend the test derivation algorithm with explicit probabilities.

This research on the role of probabilities in test derivation is also inspired by our experiments, performed with the SDT tool set from Telelogic (see [4]), on testing the *conference protocol* (see [3]). This case study also showed that a poor test suite may result when simply selecting at random between inputs and outputs.

This paper is structured as follows. We start with an explanation of the TORX test derivation algorithm. Then a section follows in which we discuss the proposed modification. We summarize our findings in the final section of the article.

Acknowledgements

We thank the members of the CdR project for their co-operation and support. In particular we thank Jan Tretmans, Sjouke Mauw and Loe Feijs for the stimulating discussions and Axel Belinfante for implementing our ideas in the TORX tools.

The TORX algorithm

Before explaining the TORX algorithm, we will present in more details the test derivation process. This process is represented in Figure 1.

In this process, the specification is the input of the test generation algorithm. The specification describes the actions that the system is allowed to do. Using it, the algorithm produces test cases which are taken by the tester system and executed against the Implementation Under Test (IUT). The tester and the IUT exchange stimuli and responses. If one of the executions leads to an error the verdict will be Fail. If no error is discovered the verdict is Pass. If the tester gives feedback to the test generation algorithm which will be used for building up the test case, the test derivation is called on–the–fly (test generation and test execution are combined in one phase); in any other case it is called batch–oriented (test generation and test execution are separated phases).

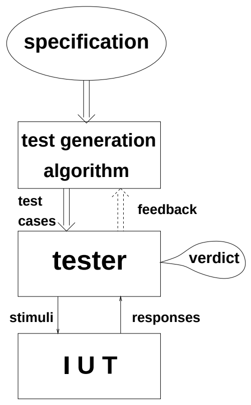

Figure 1: Automatic test generation

The TORX test generation algorithm is at the heart of the TORX architecture. The algorithm has a sound theoretical base, known as the *ioco* theory.

In this theory the behaviours of the implementation system (physical, real object) are tested by using the specification system (mathematical model of the system). The behaviours of these systems are modelled by labelled transition systems, systems which are formed by: 1) a countable, non-empty set of states; 2) a countable, non-empty set of observable actions; 3) the set of transitions; 4) the initial state. Futhermore, a special type of transition systems, the input–output transition systems, are used. In these systems the set of actions can be partitioned in a set of input actions L_I and a set of output actions L_U.

Example

For a good understanding let us take the following example: the input–output transition system for a simple candy machine (Figure 2). The label set of this automaton is the union of the set of inputs $L_I = \{but_i\}$ and of the set of outputs $L_U = \{null, liq_u, choc_u\}$ (for this system the set of outputs is extended with the $null$ output which denotes the absence of outputs). After pushing the button but_i, the machine will produce liquorice (liq_u) or nothing ($null$). When the button but_i is pushed again the candy machine will produce liquorice or chocolate ($choc_u$). If nothing was produced and the button is pressed, the machine will provide only the chocolate. After the chocolate or the liquorice is given, pushing the button will give no response ($null$ output).

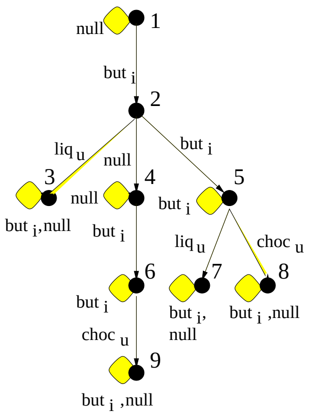

Figure 2: The specification of a candy machine.

In *ioco* theory, such systems as the system from Figure 2 in which the set of outputs is extended with the $null$ output are called suspension automaton.

One of the main ingredients of the TORX algorithm is the correctness relation. Informally, an implementation is a correct implementation with respect to the specification s and implementation relation **ioco**$_{\mathcal{F}}$ if for every trace from $\mathcal{F}$ the set of possible outputs the implementation can generate after performing the trace is specified by the specification.

The correctness of an implementation with respect to a specification is checked by executing test cases (which specifies a behaviour of the implementation under test). A test case is seen as a finite labelled transition system which contains the terminal states Pass and Fail. An intermediate state of the test case should contain either one input or a set of outputs. The set of outputs is extended with the output θ which means the observation of a refusal (detection of the absence of actions).

When executing a test case against an implementation the test case can give a Pass verdict if the implementation satisfies the behaviour specified by the test cases or a Fail verdict if the implementation does not satisfy the behaviour

A test suite is a set of test cases. The conformance relation used between an implementation i and a specification s is $ioco_F$. In the ideal case, the implementation should pass the test suite (completeness) if and only if the implementation conforms. In practice, because the test suite can be very large, completeness is relaxed to the detection of non–conformance (soundness). Exhaustiveness of a test suite means that the test suites can only assure conformance but it can also reject conforming implementation. If an implementation passes a test suite, than the implementation conforms with the specification with respect with the conformance relation. Hoever this does not mean that every conforming implementation passes that test suite. For deriving tests the following specification of an algorithm is presented in [1]:

The specification of the test derivation algorithm
Let S be the suspension automaton of a specification and let F be a set of traces included in the set of traces of S; then a test case t is obtained by a finite number of recursive applications of one of the following three nondeterministic choices:

1. *terminate the test case*

   t $=$ Pass;
2. *supply an input for the implementation*

   take an input a such that there exist in F a trace which contains the input a. Remove from F all the traces which does not contain the input a at the current position and go into the new state of the specification (the state reached with the a input).

   t $= a; t_a$

   where t_a is obtained by applying the algorithm recursivly to the new F and for the new state of the specification.
3. *check the next output of the implementation*

   t = $\sum x$;Fail

   if output x is not produced by the specification and it is present in a trace from F and F contains the empty trace;

   + $\sum x$;Pass

   if output x is not produced by the specification and it is present in a trace from F but F does not contain the empty trace;

   + $\sum x; t_x$

   if output x is produced by the specification; t_x is obtained by applying the algorithm forward for the new F (from F are eliminated all the traces which do not contain the output x) and for the new state of the specification (the state reached with the x output).

The summation $\sum$ means choice. In the implementation of the algorithm initially F equals all the traces off the specification.

The algorithm has three *Choices*. In every moment it can choose to supply an input a from the set of inputs L_I or to observe all the outputs ($L_U \cup \{\theta\}$) or to finish. When it finishes, the verdict is Pass, that is, no error is detected. After supplying an input, the input becomes part of the test case and the algorithm is applied recursively for building the test case. When it checks the outputs, if the current output is present in $out(S)$, that output will also become part of the test case and the algorithm will be applied recursively. If the output is not present in $out(S)$ the algorithm finishes in almost all the cases with a Fail verdict (if the empty trace is considered an element of F). If the empty trace is not in F then the verdict will be Pass.

This algorithm satisfies the following properties (for a proof see [1]):

Theorem 1 1. A test case obtained with this algorithm is finite and sound with respect to $ioco_{\mathcal{F}}$.
2. The set of all possible test cases that can be obtained with the algorithm is exhaustive.

For a good understanding of the algorithm let us apply it on the suspension automaton for the candy machine from Figure 2.

The implementation of this algorithm in the TorX architecture usually generates the test cases on–the–fly. To simplify our explanation below we will use a batch oriented approach. The set F equals the set $traces(\text{candy})$.

A possible execution sequence of the algorithm on this automaton is:

- First *Choice 2* (*select an input*) ($S = S_1$, $F = traces(S_1)$):
  $t = but_i; t_1$;
- To obtain t_1 the algorithm chooses *Choice 2*($S = S_2$, $F = traces(S_2)$):
  $t_1 = but_i; t_2$;
- Now *Choice 3* is selected (*check the output*) for computing t_2 ($S = S_5$, $F = traces(S_5)$, $\epsilon \in F$):
  $t_2 = liq_u; t_{21} + choc_u; t_{22} + \theta;$ Fail;
- For liq_u the algorithm finishes (*Choice 1*) ($S = S_7$, $F = traces(S_7)$):
  $t_{21} =$ Pass;
- For $choc_u$ the algorithm again checks the output (*Choice 3*):
  $t_{22} = liq_u;$ Fail $+ choc_u;$ Fail $+ \theta; t_{31}$ ($S = S_8$, $F = traces(S_8)$, $\epsilon \in F$);
- If θ action is produced, it chooses *Choice 1* ($S = S_8$, $F = traces(S_8)$):
  $t_{31} =$ Pass.

The resulting test is shown in Figure 3. Recall that the output θ means the observation of a refusal. We see that $but_i but_i liq_u$ is correct behaviour. We can also see that $but_i but_i choc_u choc_u$ is incorrect behaviour.

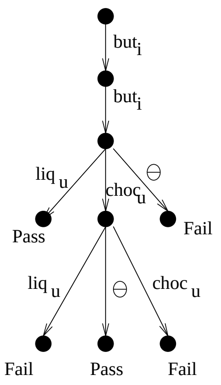

Figure 3: The Test generated for the candy machines.

The optimization of the TorX algorithm

Our optimization of the TorX algorithm introduces global probabilities p_1, p_2 and p_3 to the three choices of the algorithm. To get started, we assume that the probabilities p_1, p_2 and p_3 are globals by which we mean that they do not depend on the specific moment of generation. Furthermore, we have:

$$p_1 + p_2 + p_3 = 1, p_1 \neq 0, p_2 \neq 0, p_3 \neq 0$$

The modified TorX algorithm now reads as follows:

- Choose *Choice 1* (*terminate the test case*) with probability p_1;
- Choose *Choice 2* (*supply an input for the implementation*) with probability p_2; Select every input with the same probability;

- Choose *Choice 3* (*check the next output of the implementation *) with probability p_3;

An important observation is that the extended algorithm still produces the same test cases. We only control the chance of a trace to occur. This means that it keeps the properties of the old algorithm (Theorem 1): a generated test-case is finite, sound and the union of all tests is exhaustive.

After having extended the algorithm with probabilities, the question which will arise is: what value we should give to these probabilities?

The answer of this question is related with the introductory problem of ratio between inputs and outputs. Given a required ratio between the inputs and the outputs in a test trace what values should the probabilities of sending an input and receiving an output have?

After some complicated computations (see [8]) we arrived to a formula which maximizes the probability to arrive at the end of one given trace as function of the trace ratio between inputs and outputs. We will illustrate the computation in the following example

Example

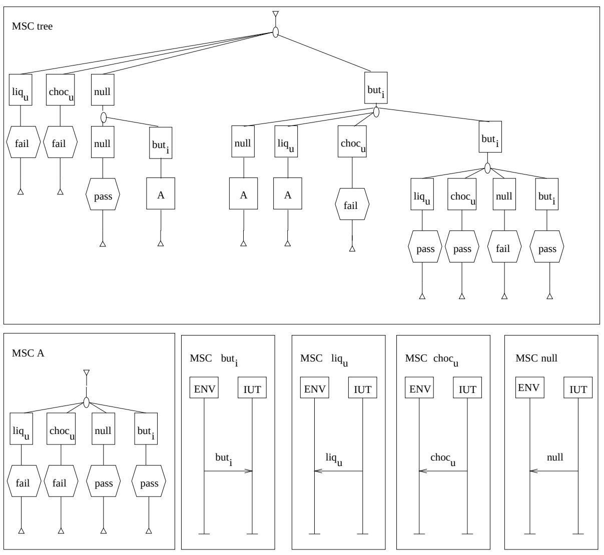

Figure 4: Tests derived from candy machine represented in an HMSC.

Let us consider all execution traces of the tests generated from the candy machine with a length less than or equal to three. These traces are represented in the HMSC (see [3]) from Figure 4. We use HMSC (High level Message Sequence Chart) to represent the test cases because this is a convenient technique which supports reusing parts of the diagram

In the HMSC the *Fail* traces $\{null\ liq_u, null\ choc_u, null\ null\ liq_u, null\ null\ choc_u\}$ are not represented because in conformance with our observation, only choosing to check the outputs will not lead to interesting test cases (so for the sake of the simplicity we excluded them). Our example works even if these traces are present in the set of *Fail* traces considered.

The set of all the *Fail* traces are represented in Figure 5. In this figure, also the ratio between the number of inputs in that trace and the number of outputs is represented. So for example the trace $but_i\ null\ liq_u$ has one input and two outputs so the ratio is $\frac{1}{2}$; the same procedure is applied to every trace in the set.

In this set of *Fail* traces there are two traces with a ratio between inputs and outputs of $\frac{0}{1}$, six with a ratio $\frac{1}{2}$, one with ratio $\frac{1}{1}$ and one with ratio $\frac{2}{1}$. It is clear that the number of traces with ratio $\frac{1}{2}$ is the biggest and we will choose it to be the ratio between inputs and outputs ($\frac{n}{m} = \frac{1}{2}$). For computing the new configuration of the probabilities we choose $p_1 = 0$ if the length of the trace is less than three and $p_1 = 1$ if the length is equal to three. The new probabilities configuration is computed as in the followings

$p_2 = \frac{\frac{1}{2}}{\frac{1}{2}+1} \times (1-0) = \frac{1}{3} \approx 0.33$

and

$p_3 = \frac{1}{\frac{1}{2}+1} \times (1-0) = \frac{2}{3} \approx 0.67$

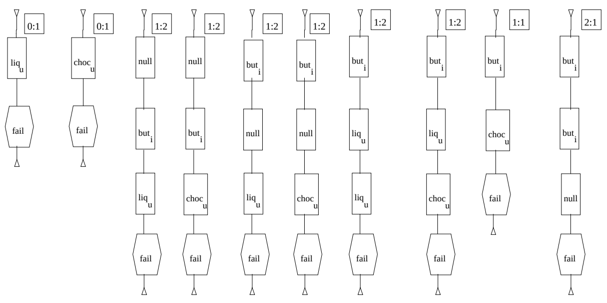

Figure 5: *Fail* traces represented in HMSC.

The old configuration of the TORX algorithm of (p_2, p_3) was $(0.5, 0.5)$ the new one is $(0.33, 0.67)$. For computing the probability of getting a *Fail* when the algorithm runs one time against an erroneous implementation (which has all the *Fail* traces from the set) first the probability of every individual *Fail* trace should be computed. A graphical representation for the computation of the probability of the trace $(but_i\ null\ liq_u)$ is given in Figure 6 for the old and the new configuration of (p_2, p_3).

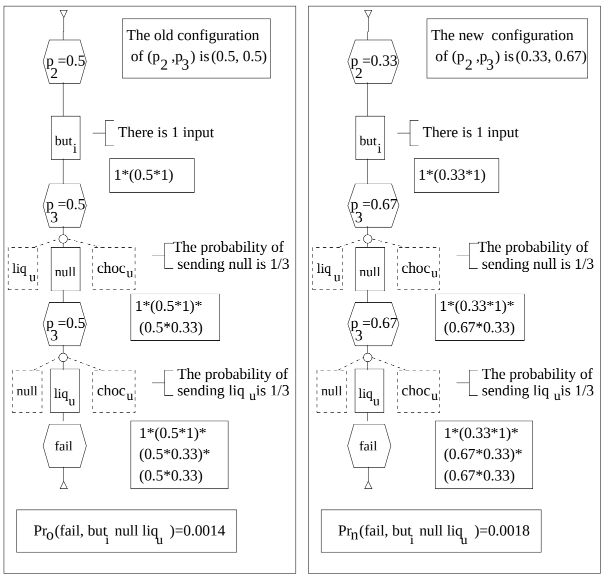
The old configuration of (p2 ,p3) is (0.5, 0.5)
p2 =0.5
but i
There is 1 input
1*(0.5*1)
p3 =0.5
liq u
null
choc u
The probability of sending null is 1/3
1*(0.5*1)* (0.5*0.33)
p3 =0.5
null
liq u
choc u
The probability of sending liq u is 1/3
fail
1*(0.5*1)* (0.5*0.33)* (0.5*0.33)
Pro(fail, but i null liq u)=0.0014
The new configuration of (p2 ,p3) is (0.33, 0.67)
p2 =0.33
but i
There is 1 input
1*(0.33*1)
p3 =0.67
liq u
null
choc u
The probability of sending null is 1/3
1*(0.33*1)* (0.67*0.33)
p3 =0.67
null
liq u
choc u
The probability of sending liq u is 1/3
fail
1*(0.33*1)* (0.67*0.33)* (0.67*0.33)
Prn(fail, but i null liq u)=0.0018

Figure 6: The probability of generating and executing the trace $but_i\ null\ liq_u$.

After performing the trace but_i, the IUT can send three outputs $null$, liq_u, $choc_u$, so the probability of sending one (for example $null$) is 0.33. In the same way the probability of sending liq_u is also 0.33. So, the probability of generating and executing the trace $but_i\ null\ liq_u$ is 0.0014 for the old configuration of the probabilities and 0.0018 for the new one. In a similar way the probabilities for every individual trace which ends in a *Fail* are computed.

It is not entirely trivial to see that optimizing for each individual *Fail* trace leads to a better error detection capability for the suite as a whole. In order to show that this is the case, we made some further calculation in the context of this example. The general claims about better error detection capability are outside the scope of the present paper.

The probability $Pr($*Fail*, TorX, $1)$ of getting a *Fail* verdict when the TorX algorithm runs once against the IUT is obtained by summing the probabilities of every individual *Fail* trace; so for the old configuration this probability is $Pr_{old}($*Fail*, TorX, $1\ ) = 0.51$ and for the new configuration it is $Pr_{new}($ *Fail*, TorX, $1\ ) = 0.64$. This simple case clearly demonstrates that a modification of the probabilities can lead to a higher chance of discovering an erroneous implementation in the same amount of algorithm runs. This is also clear from the graph in Figure 7 in which the probability of getting a *Fail* ($Pr($ *Fail*, TorX, $n)$) in function of the number n of test generation–executions is expressed (for the old and for the new probabilities configuration).

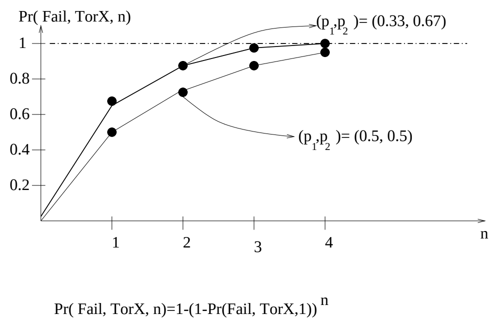

Figure 7: The probability of getting a *Fail* as function of the number of test generation–execution.

Conclusions

In this paper we gave a short description of the automatic test derivation process, an informal description of the *ioco* theory and we proposed to modify the TorX test derivation algorithm such that the probabilities of the non-deterministic alternatives are made explicit.

We argued that in some cases the generated test suite can be optimized by adapting the values of these probabilities. Case studies gave evidence that assuming an equal distribution of chances, the TorX algorithm will sometimes yield relatively few really interesting test cases. Our calculations on the toy example of the candy machine also showed that an appropriate choice of the probabilities improves the chance to detect errors in the implementation.

An important question is, of course, whether there are heuristics which help in selecting appropriate values for the probabilities. In the case studies which we performed, clearly the ratio between the number of inputs and the number of outputs in a test trace influenced the quality of the test cases. Therefore, we derived in this paper the optimal values for the probabilities in the algorithm given some prefered ratio between the number of inputs and outputs.

The proposed modification of the TORX algorithm has already been implemented. Futher research could investigate the impact of this work on the ongoing series of case studies performed in the CdR project.

An important follow-up of the current research is the extension of the testing theory from [7] in more ways with probabilities. In particular the study of the probabilistic coverage seems promising.

References

[1] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[2] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12th Int. Workshop on Testing of Communicating Systems, pages 179–196. Kluwer Academic Publishers, 1999.

[3] S. Mauw, M.A. Reniers. High-level Message Sequence Charts In A. Cavalli and A. Sarma, editors, SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the Eighth SDL Forum, pages 291-306, Evry, France, September 1997.

[4] B. Koch, J. Grabowski, D. Hogrefe, M. Schmitt. Autolink - A Tool for Automatic Test Generation from SDL Specifications. IEEE International Workshop on Industrial Strength Formal Specification Techniques,(WIFT98), Boca Raton, Florida, Oct. 21-23, 1998.

[5] J. Tretmans, A. Belinfante. Automatic testing with formal methods. In EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis and Review, Barcelona, Spain, November 8-12, 1999. EuroStar Conferences, Galway, Ireland. Also: Technical Report TR-CTIT-17, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[6] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu. CADP (caesar/aldebaran development package): A protocol validation and verification toolbox. In R. Alur and T.A. Henziner, editors, Computer Aided Verification CAV'96. Lecture Notes in Computer Science 1102, Springer–Verlag, 1996.

[7] L. Heerink, J. Tretmans. Formal methods in conformance testing: a probabilistic refinement. International Workshop in Testing and Comunication System '96.

[8] L.M.G. Feijs, N. Goga, S. Mauw Probabilities in the TORX test derivation algorithm SAM'2000, Grenoble, France.