# Probabilities in the TorX test derivation algorithm

L.M.G. Feijs, N. Goga, S. Mauw
Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
feijs@win.tue.nl, goga@win.tue.nl, sjouke@win.tue.nl

**Abstract**

We propose to extend the TorX algorithm for automatic test derivation with explicit probabilities. Using these probabilities, the generated test suite can be tuned and optimised with respect to the chances of finding errors in the implementation. The main result of this paper is a theorem that shows that the optimal balance between giving stimuli and checking responses is determined by the ration of inputs and outputs along a typical test trace. A simulation experiment demonstrates that this gives rise to an improved error detection capability.

**Keywords:** testing, test generation, probabilities, tools.

## 1 Introduction

The part of the software development process where the application of formal methods is expected to have considerable impact in the near future is the phase of testing. Manual derivation and execution of test cases leads to an expensive, time consuming and suboptimal testing process. We think that problem areas such as regression testing and conformance testing will benefit from a more formal approach. In the work presented in this paper, the central concepts of SDL and MSC play an important role: inputs, outputs and event sequences (although the precise syntax of SDL and MSC is not needed for the theoretical analysis given here). Moreover we will employ HMSC which we found convenient to represent the test suite of our example.

The formal underpinning of the testing process gains more and more interest, as witnessed by the development of theoretical foundations for testing and corresponding tool support. In 1998 a consortium of Dutch research groups from academia and industry founded the *Côte-de-Resyste* project (CdR, for short) to join efforts in the formalization and automatization of the testing practice.

The CdR project aims at developing theory, methodology and tools which support a formal approach towards testing. The development of the CdR tools is embedded in the design of an open tool environment, baptized TorX. This tool environment allows for an easy integration of a wide range of third party tools which support the spectrum from (automatic) test generation to (automatic) test execution (such as the SDT tools [KGHS98], Lotos tools [FKG96], Verilog tools). For a detailed description of the CdR project and the TorX tools, we refer to [Tret99].

Of course, tools developed within the project can also be linked to the TorX tool architecture. The tool development within the CdR project currently concentrates on

building a tool for automatic test generation. Several case studies have already been performed with this tool (see e.g. [BFVT99]).

The TorX test generation tool is based on the *ioco* theory (input/output conformance) developed at the University of Twente ([Tret96]). 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 [KGHS98]), on testing the *conference protocol* (see [BFVT99]). This case study also showed that a poor test suite may result when simply selecting at random between inputs and outputs.

These are the main motivations for the research presented in this paper. We will study the impact of parameterizing the TorX test derivation algorithm with the probabilities of selecting between inputs and outputs. Furthermore, we will derive the optimal values for these probabilities given a desired ratio between inputs and outputs in the test cases.

This paper is structured as follows. In Section 2 we explain the *ioco* theory and the TorX test derivation algorithm. The proposed modification is presented in Section 3. Here we also calculate optimal values for the probabilities and we analyse a simple example. Our findings are summarized in Section 4.

### Acknowledgements

We thank the members of the CdR project for their co-operation and support. In particular we thank Jan Tretmans for the stimulating discussions, Axel Belinfante for implementing our ideas in the TorX tools, and Jan Feenstra for proof reading our paper.

## 2 Technical preliminaries

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. Below, we will give a brief summary of this theory. For a full description of the *ioco* theory see [Tret96].

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. A labelled transition system is defined as follows.

**Definition 2.1** A labelled transition system is a quadruple $\langle S, L, \rightarrow, s_0 \rangle$, where

- $S$ is a (countable) non empty set of states;
- $L$ is a (countable) non empty set of observable actions;
- $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ is a set of transitions;
- $s_0 \in S$ is the initial state.

The universe of labelled transition systems over $L$ is denoted by $\mathcal{LTS}(L)$.

A labelled transition system is represented in a standard way as a graph or by a process–algebraic behaviour expression.

The special action $\tau \notin L$ denotes an unobservable action. A trace $\sigma$ is a sequence of observable actions ($\sigma \in L^*$) and $\Rightarrow$ means the observable transition between states ($s \stackrel{\sigma}{\Rightarrow} s'$ indicates that $s'$ can be reached from state $s$ after performing the actions from trace $\sigma$). The empty trace is denoted by $\epsilon$. In some cases the transition system will not be distinguished from its initial state (or the state in which it is). Furthermore we will use $s \stackrel{a}{\rightarrow}$ (or $s \stackrel{\sigma}{\Rightarrow}$) to denote $\exists s' : s \stackrel{a}{\rightarrow} s'$ (or $\exists s' : s \stackrel{\sigma}{\Rightarrow} s'$).

**Definition 2.2** Consider a labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ and let $s \in S$, $\sigma \in L^*$ and $A \subseteq L$.

1) $traces(s) =_{\text{def}} \{\sigma \in L^* \mid s \stackrel{\sigma}{\Rightarrow}\}$ (the set of traces from $s$);

2) $init(s) =_{\text{def}} \{\mu \in L \cup \{\tau\} \mid s \stackrel{\mu}{\rightarrow}\}$ (the set of initial actions of a state);

3) $s$ **after** $\sigma =_{\text{def}} \{s' \in S \mid s \stackrel{\sigma}{\Rightarrow} s'\}$ (the set of reachable states after $\sigma \in L^*$).

A failure trace is a trace in which both actions and refusals, represented by a set of refused actions, occur. For this, the transition relation $\rightarrow$ is extended with refusal transitions (self–loop transitions labelled with a set of actions $A \subseteq L$, expressing that all actions in $A$ can be refused) and $\Rightarrow$ is extended analogously to $\stackrel{\varphi}{\Rightarrow}$, with $\varphi \in (L \cup \mathcal{P}(L))^*$ ($\varphi$ is a trace which leads to a state of the system in which all the actions from a set $A \subseteq L$ can be refused).

**Definition 2.3** Let $p \in \mathcal{LTS}(L)$; then we define the failure traces of $p$ as follows.

$Ftraces(p) =_{\text{def}} \{\varphi \in (L \cup \mathcal{P}(L))^* \mid p \stackrel{\varphi}{\Rightarrow}\}$.

A special type of transition systems, the input–output transition systems, is 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 $L_U$. The universe of such systems is denoted by $\mathcal{IOTS}(L_I, L_U)$. Because $\mathcal{IOTS}(L_I, L_U) = \mathcal{LTS}(L_I \cup L_U)$, these input–output transition systems are also labelled transition systems.

For modelling the absence of outputs in a state (a quiescent state) a special action *null* ($\delta$ in the notation of [Tret96], $null \notin L$ ) is introduced and the transformation of the automaton into a *suspension automaton* is used. Formally a state $s$ is quiescent (denoted as $null(s)$) if $\forall a \in L_U \cup \{\tau\}$, $\nexists s' : s \stackrel{a}{\rightarrow} s'$.

**Definition 2.4** Let $p \in \mathcal{LTS}(L)$ $(L = L_I \cup L_U)$. Then the *suspension automaton* of $p$ is the labelled transition system $\langle S_{null}, L_{null}, \rightarrow_{null}, q_0 \rangle \in \mathcal{LTS}(L_{null})$, where

- $S_{null} = \mathcal{P}(S) \setminus \{\emptyset\}$;
- $L_{null} =_{\text{def}} L \cup \{null\}$;
- $\rightarrow_{null} =_{\text{def}} \{q \xrightarrow{a}_{null} q' \mid q, q' \in S_{null}, a \in L, q' = \cup_{s \in q}\{s' \in S \mid s \xrightarrow{a} s'\}, \exists s \in q, s' \in q', s \xrightarrow{a} s'\} \cup \{q \xrightarrow{null}_{null} q' \mid q' = \{s \in q \mid null(s)\}, q \neq \emptyset, q' \neq \emptyset\}$;
- $q_0 =_{\text{def}} \{s_0\}$.

The suspension traces of $p \in \mathcal{LTS}(L)$ ($p$ is a normal automaton) are: $Straces(p) =_{\text{def}} Ftraces(p) \cap (L \cup \{L_U\})^*$ (for $L_U$ occuring in a suspension trace, we write $null$).

Informally, an implementation $i$ is a correct implementation with respect to the specification $s$ and implementation relation $\mathbf{ioco}_{\mathcal{F}}$ if for every trace $\sigma \in \mathcal{F}$ each output or absence of outputs, that the implementation $i$ can perform after having performed sequence $\sigma$, is specified by $s$.

**Definition 2.5** Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$ ( $i$ and $p$ are normal automata), and $\mathcal{F} \subseteq L^*_{null}$, then:
$i$ $\mathbf{ioco}_{\mathcal{F}}$ $s =_{\text{def}} \forall \sigma \in \mathcal{F} : out(i \ \ \mathbf{after}\ \sigma) \subseteq out(s \ \ \mathbf{after}\ \sigma)$
where $out(p\ \mathbf{after}\ \sigma) =_{\text{def}} init(p\ \mathbf{after}\ \sigma) \cap (L_U \cup \{null\})$.

The correctness of an implementation with respect to an implementation 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 $\theta$ which means the observation of a refusal (detection of the absence of actions). A test suite is a set of test cases. 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

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 implementations. For deriving tests the following specification of an algorithm is presented in [Tret96]:

**The specification of the test derivation algorithm**
Let $S$ be the suspension automaton of a specification and let $F \subseteq traces(S)$; then a test case $t \in \mathcal{TEST}\ (L_I, L_U)$ 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 $a \in L_I$ such that $F_a \neq \emptyset$

  t = $a; t_a$

  where $F_a = \{\sigma \mid a\sigma \in F\}$ , $S \xrightarrow{a}_{null} S_a$ and $t_a$ is obtained by recursively applying the algorithm for $S_a$ and $F_a$;

- 3.(*check the next output of the implementation*)

  t = $\sum\{$x;Fail $\mid x \in L_U \cup \{\theta\}, \overline{x} \notin$ out(S)$, \epsilon \in F\}$

  $+ \sum\{$x;Pass $\mid x \in L_U \cup \{\theta\}, \overline{x} \notin$ out(S)$, \epsilon \notin F\}$

  $+ \sum\{x; t_x \mid, x \in L_U \cup \theta, \overline{x} \in$ out(S)$\}$

  where $\overline{x}$ is a notation for $x$ in which the *null* action is replaced by $\theta$ action and vice versa, $F_x = \{\sigma \mid \overline{x}\sigma \in F\}$, $S \xrightarrow{\overline{x}}_{null} S_x$ and $t_x$ is obtained by recursively applying the algorithm for $S_x$ and $F_x$. The summation symbol $\sum$ denotes the generalized choice as usual in process algebra.

In the implementation of the algorithm initially $F$ equals $traces(S)$.

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, because this does not mean that the algorithm detected an error, it finishes with a Pass verdict. 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 become also 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 satiesfies the following properties (for a proof see [Tret96]):

**Theorem 2.6** *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 take the following example: the suspension automaton for a simple candy machine (Figure 1). 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 the suspension automaton 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 $liq_u$ 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 are given, pushing the button will give only a void response (*null* output).

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

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

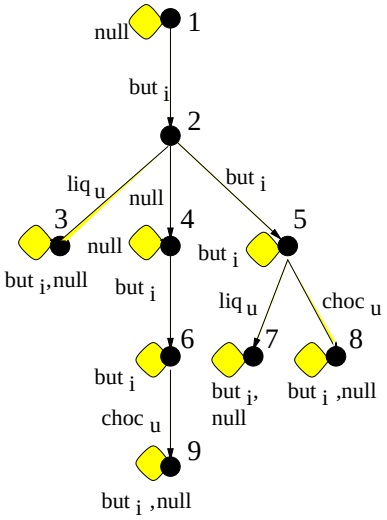


Figure 1: The suspension automaton for a candy machine

- 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 the $\theta$ action is produced, it chooses *Choice 1* ($S = S_8$, $F = traces(S_8)$):
  $t_{31} =$ Pass.

The resulting test is shown in Figure 2. Because in the *ioco* theory the test contains the output $\theta$ for the observation of a refusal, the output $\theta$ will replace the *null* output.

## 3 Adding probabilities

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:

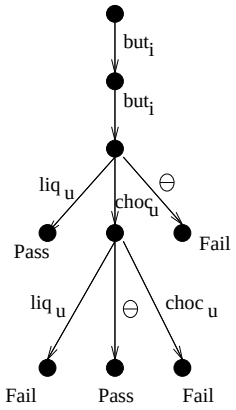


Figure 2: The test generated for the candy machine

- 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 2.6): 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 still remains is: what value we should give to these probabilities?

The answer to this question is related to 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? The answer will be formulated by the Lemma and the Theorem which will follow.

Lemma 3.1 will provide a formula for the probability that the algorithm will arrive at the end of one given trace. Theorem 3.2 will compute a configuration for $p_1$, $p_2$ and $p_3$ which maximizes the probability to arrive at the end of one given trace.

Now, for a good understanding, we will define a special tree which will be used in the subsequent proofs. We call this tree *the behaviour tree* and it is formed by the union of all the traces derived from a specification $S$. An example of this tree is given in Figure 3.

This *behaviour tree* is composed of the following kinds of nodes:

- *Final:* in this node the trace of the test stops with a verdict (Pass, Fail);
- *Intermediate:* this node contains the name of the input or of the output.

In the behaviour tree in Figure 3 the *Pass Final* state appears twice in one level because one *Pass* verdict can be generated from *Choice 1* and one from *Choice 3*. When we refer to a *given trace* we refer to a trace of this tree which starts from the *Root* state and stops somewhere in the tree (the traces can be infinite). The signals from the trace are mapped

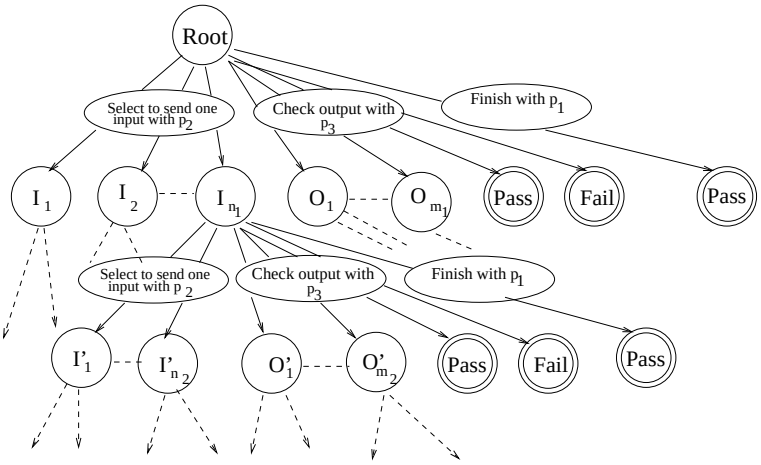


Figure 3: A probabilistic walk through the behaviour tree

on the internal nodes of the *behaviour tree*. In the *behaviour tree* all the tests generated for the algorithm are included.

**Lemma 3.1** *Consider an arbitrary but fixed finite trace which does not end in a final verdict. Let $n$ be the number of inputs on the trace and $p$ the length of the trace. Let $n_l$, $l = 1, 2, 3..n$ be the number of inputs which can be selected when the $l$-th input on the trace is selected. Let $P_k$, $k = 1, 2, 3..p - n$ be the probability of the $k$-th output in the trace to be produced by the implementation (IUT). Then the probability to generate this trace with the TorX algorithm is computed in the following way:*

$$P = \prod_{l=1}^{n} (\frac{1}{n_l} \times p_2) \times \prod_{k=n+1}^{p} (P_k \times p_3)$$

The full proof of the theorem can be found in Appendix A. Here an example will be given which is a good illustration for the way of computing the probability to generate a given trace (see Figure 4).

**Example** In the considered trace there are two inputs $I_a$, $I_d$ and three outputs $O_b$, $O_c$ and $O_e$. The number of all inputs which can be selected when $I_a$ is selected is five and for $I_d$ it is three. Then the probability that the input $I_a$ or the input $I_d$ is chosen from the set of inputs is $\frac{1}{5}$ respectively $\frac{1}{3}$ (independent events). The probability that the implementation (IUT) sends the output $O_b$, $O_c$ or $O_e$ is $\frac{1}{3}$, $\frac{1}{4}$ respectively $\frac{1}{5}$. The probability of arriving in the *Root* state is always one. With this the computation of arriving at the end of this trace is:
$P(S_1) = P(\text{Root}) = 1$
$P(S_2) = P(S_1) \times P(\text{Choice 2}) \times P(\text{Select input } I_a) = 1 \times (\frac{1}{5} \times p_2)$
$P(S_3) = P(S_2) \times P(\text{Choice 3}) \times P(O_b) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3)$
$P(S_4) = P(S_3) \times P(\text{Choice 3}) \times P(O_c) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3) \times (\frac{1}{4} \times p_3)$
$P(S_5) = P(S_4) \times P(\text{Choice 2}) \times P(\text{Select input } I_d) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3) \times (\frac{1}{4} \times p_3) \times (\frac{1}{5} \times p_2)$
$P(S_6) = P(S_5) \times P(\text{Choice 3}) \times P(O_e) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3) \times (\frac{1}{4} \times p_3) \times (\frac{1}{5} \times p_2) \times (\frac{1}{2} \times p_3) = \prod_{l=1}^{2} (\frac{1}{n_l} \times p_2) \times \prod_{k=3}^{5} (P_k \times p_3)$
with $n_1 = 5, n_2 = 3, P_3 = \frac{1}{3}, P_4 = \frac{1}{4}, P_5 = \frac{1}{2}$

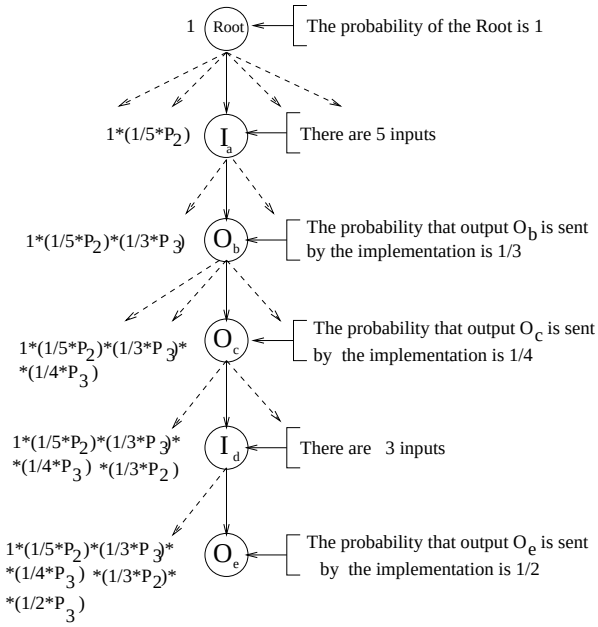


Figure 4: An example for computing the probability to generate a given trace

Once we have the formula for the probability of generating a trace we can look for the optimal configuration of the global probabilities in function of a given ratio between inputs and outputs. The following theorem solves this optimality problem.

**Theorem 3.2** *Consider an arbitrary trace which does not end in a final verdict. Let $n$ be the number of inputs on the trace and $m$ the number of outputs on the trace .*

*a) The probability to generate this trace reaches a maximum for $p_1 \to 0$, $p_2 = \frac{n}{n+m} \times (1-p_1)$ and $p_3 = \frac{m}{n+m} \times (1-p_1)$.*

*b) For every trace with ratio $r = \frac{n}{m}$ the probability to generate this trace reaches a maximum for $p_1 \to 0$, $p_2 = \frac{r}{r+1} \times (1-p_1)$ and $p_3 = \frac{1}{r+1} \times (1-p_1)$.*

*Proof* : a) The probability to generate a given trace is a function of two variables $p_2$ and $p_3$ (Lemma 3.1). For obtaining the extremal values for this probability the differential of the probability must be 0. Before doing this we will change the probability $P(p_2, p_3)$ to depend on $P(p_1, p_2)$.

$$p_1 + p_2 + p_3 = 1 \Rightarrow p_3 = 1 - p_1 - p_2$$

Conform Lemma 3.1

$$P(p_2, p_3) = \prod_{l=1}^{n}(\frac{1}{n_l} \times p_2) \times \prod_{k=n+1}^{n+m}(P_k \times p_3) = p_2^n \times p_3^m \times \prod_{l=1}^{n}\frac{1}{n_l} \times \prod_{k=n+1}^{n+m} P_k \Rightarrow$$

$$P(p_1, p_2) = p_2^n \times (1-p_1-p_2)^m \times \prod_{l=1}^{n}\frac{1}{n_l} \times \prod_{k=n+1}^{p} P_k$$

Now we observe that that $\prod_{l=1}^{n}\frac{1}{n_l}$ and $\prod_{k=n+1}^{n+m} P_k$ are constants which will be called $C_1$ and $C_2$.

$$dP(p_1,p_2) = \frac{\partial P}{\partial p_1}(p_1,p_2)dp_1 + \frac{\partial P}{\partial p_2}(p_1,p_2)dp_2$$

$$dP(p_1,p_2) = 0 \Rightarrow \begin{cases} \frac{\partial P}{\partial p_1}(p_1,p_2) = 0 \\ \frac{\partial P}{\partial p_2}(p_1,p_2) = 0 \end{cases}$$

$$\frac{\partial P}{\partial p_1}(p_1,p_2) = C_1 \times C_2 \times m \times (-1) \times p_2^n \times (1-p_1-p_2)^{m-1}$$

$$\frac{\partial P}{\partial p_2}(p_1,p_2) = C_1 \times C_2 \times (n \times p_2^{n-1} \times (1-p_1-p_2)^m + m \times (-1) \times p_2^n \times (1-p_1-p_2)^{m-1})$$

$$\frac{\partial P}{\partial p_1}(p_1,p_2) = 0 \Rightarrow$$

$$C_1 \times C_2 \times m \times (-1) \times p_2^n \times (1-p_1-p_2)^{m-1} = 0 \Rightarrow \begin{cases} p_2 = 0 & \text{or} \\ p_2 = 1-p_1 \end{cases}$$

The points are $(p_1,0),(p_1,1-p_1)$.

$$\frac{\partial P}{\partial p_2}(p_1,p_2) = 0 \Rightarrow$$

$$C_1 \times C_2 \times p_2^{n-1} \times (1-p_1-p_2)^{m-1} \times (n \times (1-p_1-p_2) - m \times p_2) = 0 \Rightarrow$$

$$\begin{cases} p_2 = 0 & \text{or} \\ p_2 = 1-p_1 & \text{or} \\ n - n \times p_1 - n \times p_2 - m \times p_2 = 0 \Rightarrow p_2 = \frac{n}{n+m} \times (1-p_1) \end{cases}$$

The points are $(p_1,0),(p_1,1-p_1),(p_1,\frac{n}{n+m} \times (1-p_1))$.
The graph of the function is sketched in Figure 5.

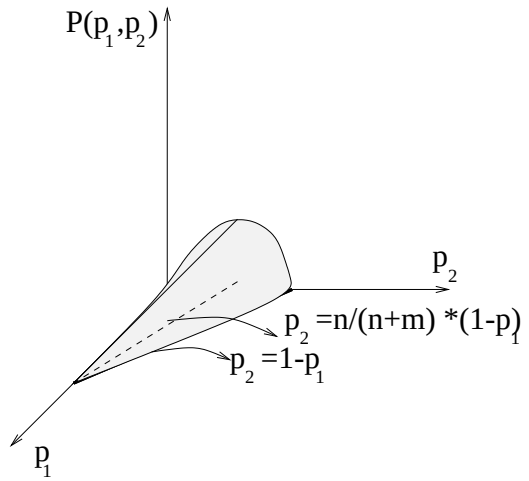


Figure 5: The graph of the probability function

Now

$P(p_1, 0) = 0$ point of minimum because the probability is $0 \leq P \leq 1$ ;
$P(p_1, 1 - p_1) = 0$ point of minimum;
$P(p_1, \frac{n}{n+m} \times (1-p_1)) = C_1 \times C_2 \times (\frac{n}{n+m})^n \times (1 - \frac{n}{n+m})^m \times (1-p_1)^{n+m}$ point of maximum.
We have the following:

- $P$ maximal $\Rightarrow (1 - p_1)^{n+m}$ maximal $\Rightarrow p_1 \to 0$;
- $p_3 = 1 - p_1 - p_2 = \frac{m}{n+m} \times (1 - p_1)$.

b) Let us consider a finite trace with ratio $\frac{n}{m}$ where $n$ is the number of inputs and $m$ the number of outputs on the trace. Then to maximize the probability to generate this trace we have (point a))
$p_2 = \frac{n}{n+m} \times (1 - p_1) \Rightarrow p_2 = \frac{\frac{n}{m}}{\frac{n}{m}+1} \times (1 - p_1)$ and
$p_3 = \frac{m}{n+m} \times (1 - p_1) \Rightarrow p_3 = \frac{1}{\frac{n}{m}+1} \times (1 - p_1)$

**Example**

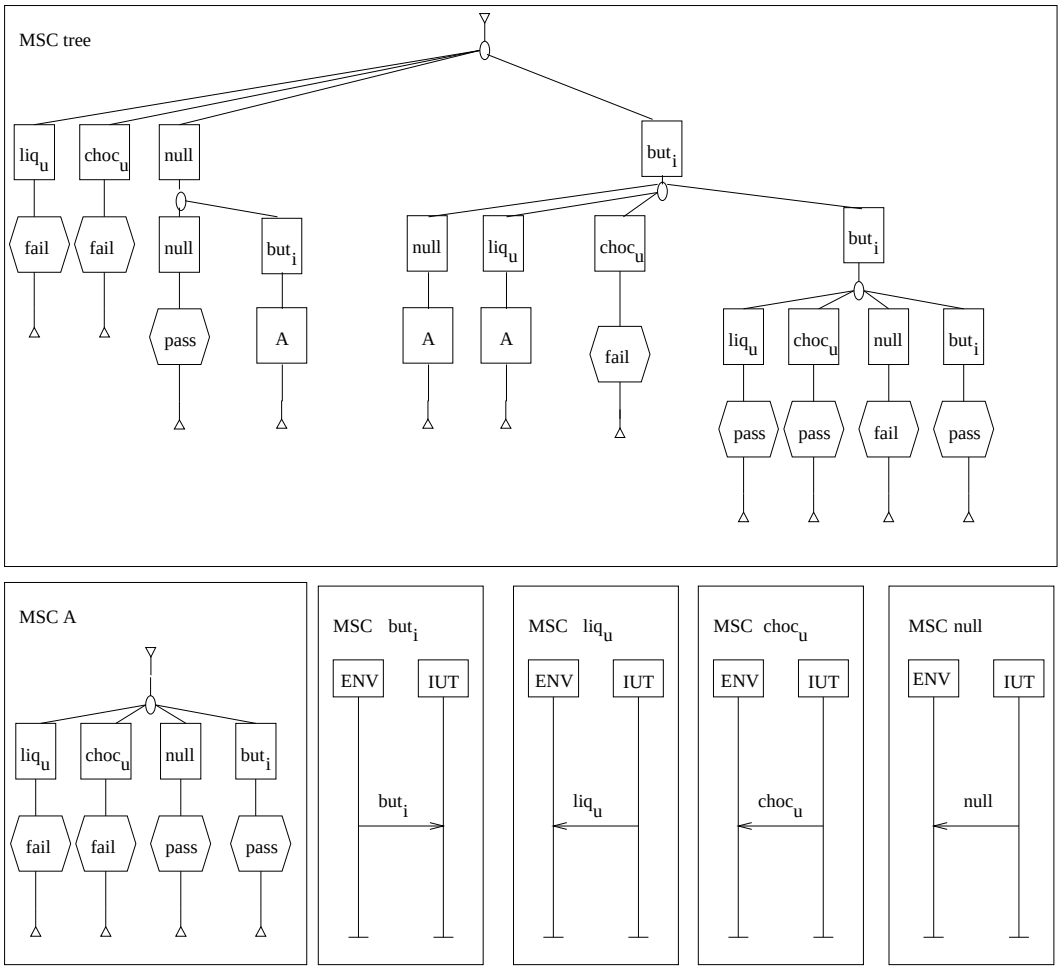


Figure 6: Tests derived from candy machine represented in an HMSC

Let us consider all execution traces of the tests generated from the candy machine with a length less then or equal to three. These traces are represented in the HMSC (see

[MR97]) from Figure 6. 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 7. 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.

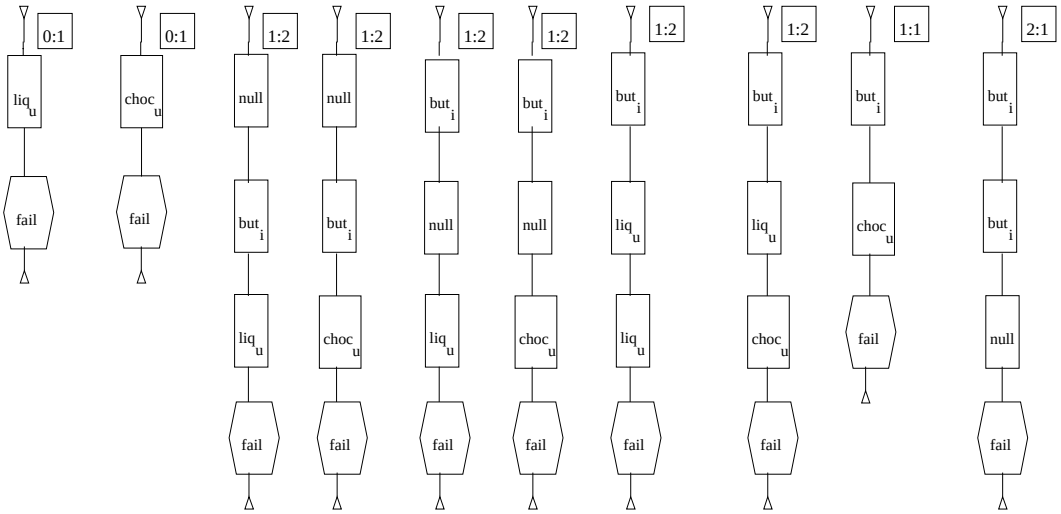


Figure 7: *Fail* traces represented in HMSC

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 longest 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 (Theorem 3.2) 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. Because the theorem applies to the traces which do not end in a *Fail* verdict, in the computation of $p_2$ and $p_3$, $p_1$ will be zero. So by applying the Theorem 3.2 b) we obtain:

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

and

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

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. The probability that the TorX algorithm generates and executes a trace is given by Lemma 3.1. A graphical

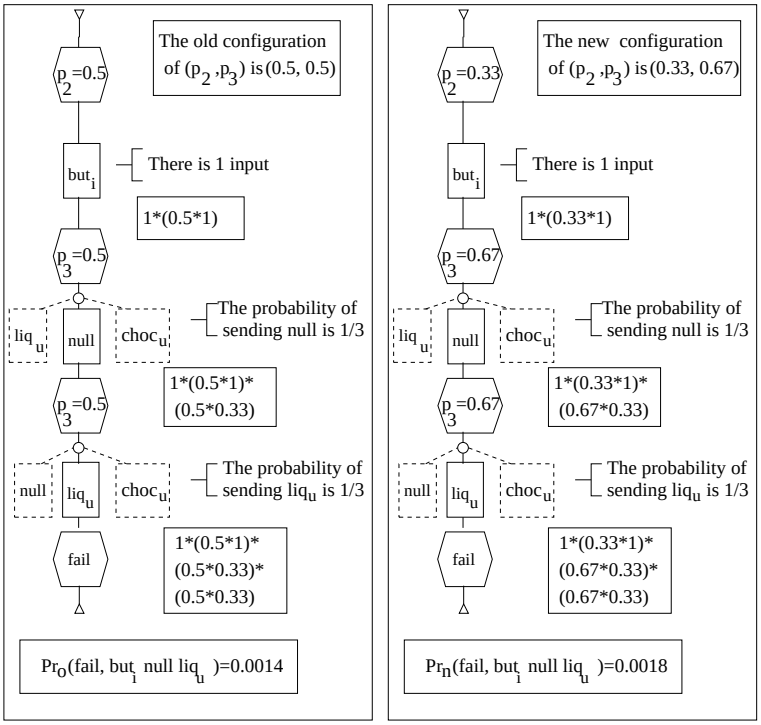


Figure 8: The probability of generating and executing the trace $but_i$ *null* $liq_u$

representation for the computation of the probability of the trace ($but_i$ *null* $liq_u$) is given in Figure 8 for the old and the new configuration of $(p_2, p_3)$.

After performing the trace $but_i$, the IUT can send three outputs *null*, $liq_u$, and $choc_u$, so the probability of sending one of it, such as *null*, is 0.33. In the same way the probability of sending $liq_u$ is also 0.33. By applying the lemma it results that 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 9 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).

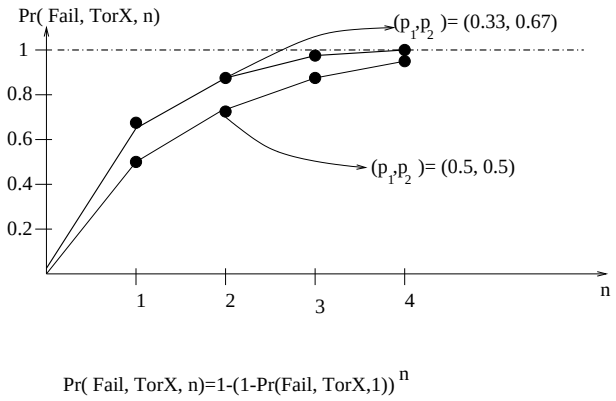


Figure 9: The probability of getting a *Fail* in function of the number of test runs

# 4 Conclusions

In this paper 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 example of the candy machine also showed that an appropriate choice of the probabilities improves the ability 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. It is an option for futher research to study the impact of this work on the ongoing series of case studies performed in the CdR project.

An important option for follow up of the current research is the extension of the testing theory from [HT96] in more ways with probabilities. In particular the study of the probabilistic coverage seems promising.

# References

[Tret96] 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.

[BFVT99] 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.

[MR97] 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.

[KGHS98] 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.

[Tret99] 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.

[FGKM96] 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.

[HT96] L. Heerink, J. Tretmans. Formal methods in conformance testing: a probabilistic refinement. In Bernd Baumgarten, Heinz-Jürgen Burkhardt and Alfred Giessler, editors, Ninth International Workshop on Testing of Communicating Systems, pages 261–276, Chapman & Hall, 1996.

# A Proof of Lemma 3.1

*Proof* : (By induction):
1)Basic step:
Node= Node after *Root*

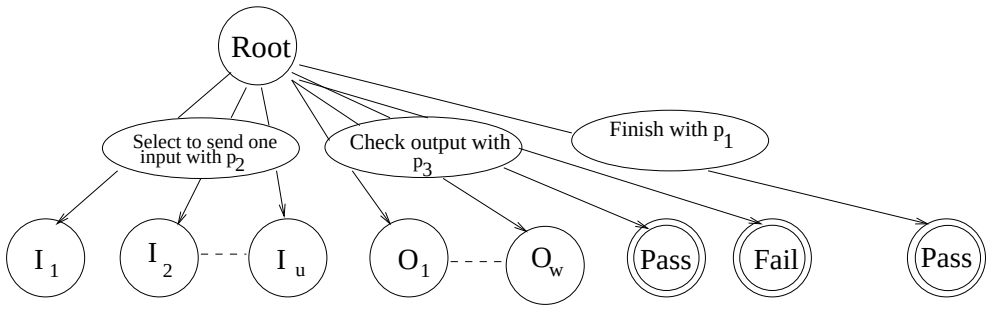


Figure 10: Root situation.

The *Choices: 1, 2 and 3* are independent events.
The *Root* state has probability 1 (sure event).
The probability to select an input $I_i, i = 1, 2, 3..u$ from the set of the inputs is $\frac{1}{u}$ (independent events).
Now:
a)Node= Contains an input:

$$P_N = \mathrm{P}(\textit{Choice 2}) \times \mathrm{P}(\textit{Select an input from the set of inputs}) = p_2 \times \frac{1}{u} = \prod_{l=1}^{1} (\frac{1}{n_1} \times p_2) \times \prod_{k=2}^{1} (P_k \times p_3)$$

with $n_1 = u$;
a)Node= Contains an output $O_i$

$$P_N = \mathrm{P}(\textit{Choice 3}) \times \mathrm{P}(\textit{output } O_i \textit{ is sent }) = p_3 \times P_{O_i} = \prod_{l=1}^{0}(\frac{1}{n_l} \times p_2) \times \prod_{k=1}^{1}(P_k \times p_3)$$

with $P_k = P_{O_i}$;
2)Induction step:
Let us assume that the probability for the current node is computed like

$$P_N = \prod_{l=1}^{n_N}(\frac{1}{n_l} \times p_2) \times \prod_{k=n_N+1}^{p_N}(P_k \times p_3)$$

then the probability to arrive in one of the next nodes (except *Final* state) is computed like:

$$P_{N+1} = \prod_{l=1}^{n_{N+1}}(\frac{1}{n_l} \times p_2) \times \prod_{k=n_{N+1}+1}^{p_{N+1}}(P_k \times p_3)$$

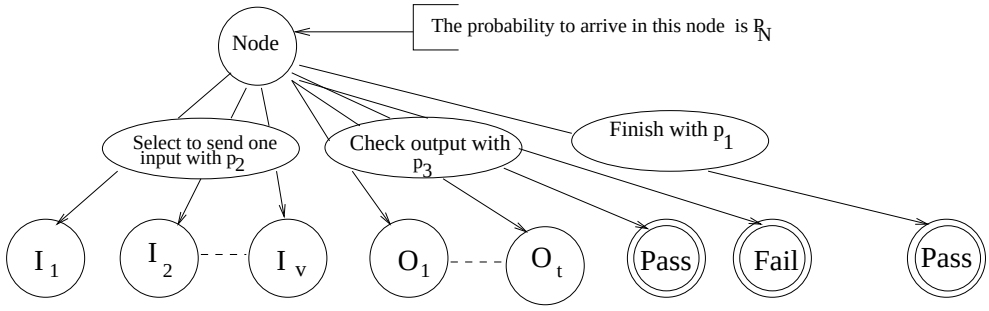


Figure 11: Intermediary situation.

The probability to select an input from the set of inputs in this case is $\frac{1}{v}$ (independent events):
Now:
a)Next node= Contains an input:

$$\begin{aligned} P_{N+1} &= \mathrm{P}(\textit{Current node}) \times \mathrm{P}(\textit{Choice 2}) \times \mathrm{P}(\textit{Select an input from the set of inputs}) = \\ &= (\textstyle\prod_{l=1}^{n_N}(\frac{1}{n_l} \times p_2) \times \prod_{k=n_N+1}^{p_N}(P_k \times p_3)) \times p_2 \times \frac{1}{v} = \\ &= \textstyle\prod_{l=1}^{n_{N+1}}(\frac{1}{n_l} \times p_2) \times \prod_{k=n_{N+1}+1}^{p_{N+1}}(P_k \times p_3) \end{aligned}$$

with $n_{N+1} = n_N + 1$, $p_{N+1} = p_N$, $P_k = P_{k-1}$, $n_{n_N} = v$;
b)Next node= Contains an output $O_i$:

$$\begin{aligned} P_{N+1} &= \mathrm{P}(\textit{Current node}) \times \mathrm{P}(\textit{Choice 3}) \times \mathrm{P}(O_i) = \\ &= (\textstyle\prod_{l=1}^{n_N}(\frac{1}{n_l} \times p_2) \times \prod_{k=n_N+1}^{p_N}(P_k \times p_3)) \times p_2 \times P_{O_i} = \\ &= \textstyle\prod_{l=1}^{n_{N+1}}(\frac{1}{n_l} \times p_2) \times \prod_{k=n_{N+1}+1}^{p_{N+1}}(P_k \times p_3) \end{aligned}$$

with $n_{N+1} = n_N$, $p_{N+1} = p_N + 1$ $O_{p_{N+1}} = O_i$.