Representation and Optimization of Software Usage Models with Non–Markovian State Transitions

Karl Doerner* and Walter J. Gutjahr†

Abstract: We extend Markov usage models by admitting non–Markovian transitions between states. The suggested description syntax preserves the visualisation capacities of Markov models and, although it is kept simple, provides the computational power of a universal programming language. It is shown that a previously introduced optimization technique for computing optimal test transition probabilities can be generalized to the presented framework. A medium–size example from a real–world application illustrates the approach.

1 Introduction

Among the different approaches for modeling the usage profile of a software system, the Markov–chain modeling technique, as developed by Whittaker and Poore [10], Whittaker and Thomason [11], and Walton, Poore and Trammell [8], gets more and more popular. This is not surprising, since the technique combines several favorable features: formal precision, flexibility and versatility in the application, and the potential of giving easily understandable visual representations.

Starting from the basic idea of identifying *states* of a software system and visualizing them as nodes in a directed graph, Markov–chain usage models determine frequencies of transitions between these states and consider an idealized version of the underlying stochastic process satisfying the *Markov property*: Given that the system has reached a certain state, its transition probabilities to other states only depend on the current state, but not on previous states, i.e., not on the history of the process.

For a large class of applications, this may be a sufficient approximation to reality. There are many applications, however, which contain state transitions that are essentially non–Markovian. Consider, for example, a simple loop (see Fig. 1). Represented by a Markov model, the number of iterations of such a loop will always be geometrically distributed, the probability for exactly k iterations being $p^k(1-p)$ $(k = 0, 1, \ldots)$. The probability function of the geometric distribution is decreasing in any case, which makes it impossible to model a usage profile with, say, ten iterations as the most frequent value, or a profile where the number of iterations is a fixed integer, which may or may not depend on what has happened in the past. Fig. 2 shows a typical example of the last–mentioned kind.

Still more difficult to model are the frequently occurring *consumer / producer* situations in which some action asynchronously consumes items produced by another. A simple example is

*Dept. of Management Science, University of Vienna, Austria

†Dept. of Statistics and Decision Support Systems, University of Vienna, Austria

the following: A word processor allows the user to have an arbitrary number of documents open at a time. "Open document" and "Close document" actions may be mingled; each instance of "Open document", however, must later be synchronized with a "Close document" event.

Fig. 1. Markov model of a loop. The arcs are labeled with transition probabilities.

A straightforward technique to cope with this problem is to split the nodes, such that they get able to reflect events in the history of the process. In specific situations, this may be helpful, but simple examples show that the model can also explode beyond tractability if this procedure is applied. Three nested loops, e.g., each with exactly 100 iterations to be performed, would require a model with at least 10^6 nodes. From a theoretical point of view, the number of required nodes can even get infinite. Also in cases where the number of nodes can still be managed, node splitting will drastically impair the transparency and clearness of the model. So it seems highly desirable to enrich the Markov–chain usage model formalism by elements that can express dependencies of the transition probabilities on the history of the process.

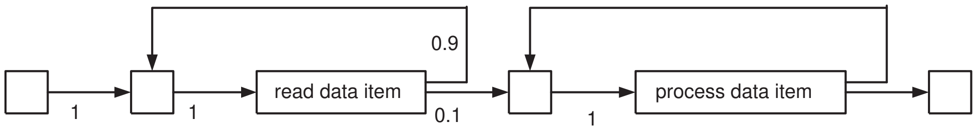

Fig. 2. Dependency between two loops: In the first (online) phase, an arbitrary number of data items is read and stored. In the second (offline) phase, each data item is processed by the system. The number of iterations in the second loop has to be the same as that in the first loop.

A general framework able to deal with the indicated problem has been developed by Woit [12]. The approach defines a rather powerful syntax and semantics for the description of usage profiles. Markov–chain models can be translated into this language, but it is not restricted to them. The profile specification consists of (a) a matrix with so–called *history classes* as rows and so–called *input classes* as columns, and (2) a C–language oriented function description. By means of the history classes, non–Markovian state transitions can be modeled.

The approach suggested in the present article has common features with Woit's technique, but, in our opinion, improves it in certain points:

1. Instead of replacing the Markov–chain formalism by a completely different description language, we design our syntax rather as an *extension* of Markov–chain models than as an alternative to them. This has the advantage that the "visualization power" of Markov–chain models can be fully exploited. In practise, a usage model has often only few states where non–Markovian transition descriptions are necessary. By applying our approach, only modifications in these places are required. The chance of a representation in graphical form is preserved.

2. The integration of the language elements needed for the description of non–Markovian dependencies into the visual representation is further supported by a very concise, but nevertheless easily understandable syntax.

3. Our syntax is "lean", i.e., restricted to the most simple and essential features, which favors an extension of analytic techniques, as they have been developed in the Markov usage model literature, to our framework.

4. Our additional language elements can be applied in a strictly "local" way, at least as the description is concerned: If, for example, a Markov transition graph with 1000 nodes and 5000 arcs needs modification in the immediate neighborhood of a single node, it is sufficient for the model designer to consider and to change only this small part of the transition graph. (This does not imply that the model statistics are affected only locally; after the change, the overall graph must be re–analyzed, but this can hopefully be done in an automated way.)

The suggested syntax and semantics, together with some introductory examples, are presented in Section 2. Furthermore, to underline the versatility of our formalism, we shall demonstrate there that it reaches the full computational power of a universal programming language. In other words: The whole range from a rough Markov model with only a few states to a complete, detailed statistical specification of the use of the software system, with all possible interdependencies, is available to the model designer; he/she can choose any granularity level between these poles.

It is probable that at least some of the analytic techniques described in recent publications on Markov usage models can be generalized to our framework. To illustrate this, we show that the task of computing optimal *test transition probabilities* based on a given usage model — a task that is already highly non–trivial in the Markovian case (see [3], [4], [5]) — allows such an extension. This topic is treated in Section 3. Let us mention that there exist also certain previous results on Markov usage models than cannot be extended to our framework. E.g., the computation of the expected time to occurrence of a state (see [7]) is impossible in our model from theoretical reasons. Thus, we are confronted with a tradeoff between generative power and analytic power.

In Section 4, we demonstrate the practical application of our approach by presenting a small fictitious and a medium–size real–world example.

Section 5 contains concluding remarks and a list of questions for future research.

2 Framework and Description Language

We start by describing a general formal framework for our extension of Markov usage models, and make it then more concrete by defining syntax and semantics of a specific description language fitting into this framework. Readers preferring to get a rough impression of the approach before considering it in abstract terms may immediately turn to the examples at the beginning of Subsection 2.2 and read Subsection 2.1 afterwards.

2.1 Extended Markov Usage Models

The underlying idea of our technique is to re–formulate a non–Markovian chain by extending its state information in such a way that the Markov property is gained. The added information will be called the (current) *frame* and denoted by the symbol y. Thus, the original state information, i.e., the reference to the current node, is replaced by the ordered pair

$$state = (node, frame).$$

While the sequence of visited nodes does not necessarily form a Markov process, our construction will guarantee that the sequence of (extended) states is a Markov process in any case. The frame y is an element of some finite or infinite set Y.

Let $G = (V, A)$ be the transition graph, where $V = \{1, \ldots, n\}$ is the set of nodes (the initial node and the terminal node being 1 and n, respectivly), and $A \subseteq V \times V$ is the set of arcs (cf. [3], [4]). For each arc $(i, j) \in A$ and each frame $y \in Y$, we define

$$p(i, j, y) \in [0, 1],$$

the transition probability from state i to state j, given that the current frame is y. Furthermore, for each node $i \in V$ and each frame $y \in Y$, we define

$$\varphi(i, y) \in Y,$$

the frame after the transition from state i to the next state, given that the current frame is y. Note that $\varphi(i, y)$ depends on the current state i, but not on what the next state is. It is always assumed that

$$\sum_{j \in Succ(i)} p(i, j, y) = 1 \quad \text{for each } i \in V \text{ and each } y \in Y,$$

where $Succ(i)$ denotes the set of successor nodes of node i in G. For the purposes of Section 3, we shall further assume that node n is *absorbing*, i.e., that $p(n, n, y) = 1$ for each $y \in Y$. In a general context, this assumption is not necessary; also recurrent chains can be modeled.

A specific frame $y_0 \in Y$ is marked as the *initial frame*, i.e., the current frame at the moment where the process starts in the initial node 1.

Let i be fixed. Then, an equivalence relation $\sim_i$ on Y can be defined by

$$y_1 \sim_i y_2 \quad \text{exactly if} \quad p(i, j, y_1) = p(i, j, y_2) \text{ for all } j.$$

The relation $\sim_i$ induces a partition of Y into equivalence classes $Y_r^{(i)}$ $(r = 1, 2, \ldots)$. An equivalence class $Y_r^{(i)}$ $(r \geq 1)$ consists of all frames y for which the same vector of transition probabilities is assigned to the node i.

In order to improve computational tractability, we assume that to each $i \in V$, there exist only a finite number $R(i)$ of equivalence classes $Y_r^{(i)}$, i.e., that $1 \leq r \leq R(i) < \infty$. The number $R(i)$ is allowed to depend on the node i. Under this assumption, we may define a finite number of probabilities $p_r(i, j)$ by

$$p_r(i, j) = p(i, j, y) \quad \text{for } y \in Y_r^{(i)} \quad (r = 1, \ldots, R(i)). \tag{1}$$

Note that the function $\varphi(i, y)$ needs not to be correlated in any way with the partitions $(Y_r^{(i)})$.

A fixed path through the graph G will be denoted by $x = (x_0, x_1, \ldots)$, where x_k is the node reached after the kth transition. A random path through G will be denoted by $X = (X_0, X_1, \ldots)$. In any case, $x_0 = X_0 = 1$. Similarly, we shall denote the sequence of *frames* by $(Y_0, Y_1, \ldots)$, whenever the randomness of the subsequent frames is to be emphasized. Again, the initial frame $y_0 = Y_0$ is deterministic. In total, the extended process can be written as

$$((X_0, Y_0), (X_1, Y_1), (X_2, Y_2), \ldots).$$

We have

$$\begin{aligned} Y_1 &= \varphi(X_0, y_0), \\ Y_2 &= \varphi(X_1, Y_1) = \varphi(X_1, \varphi(X_0, y_0)), \end{aligned}$$

etc. In particular, for fixed frame transition function φ, the frame Y_k only depends on the initial frame y_0 and on $X_0, \ldots, X_{k-1}$.

On the assumptions above, the extended process $((X_0, Y_0), (X_1, Y_1), \ldots)$ will be called an *Extended Markov Usage Model* (EMUM).

Proposition 2.1. An EMUM is a Markov chain, possibly with infinite state space.

Proof. The transition probability between the states (i, y) and (j, z) of the EMUM is given by

$$p((i, y), (j, z)) = \begin{cases} p(i, j, y), & \text{if } z = \varphi(i, y), \\ 0, & \text{otherwise.} \end{cases}$$

It depends only on the states (i, y) and (j, z), which proves the assertion. □

The reader should notice that if, for each i and r, the pairs (i, y) with $y \in Y_r^{(i)}$ are comprised to one single state (i, r), the resulting process is *not* Markovian in general: Although it is true that two different frames y, y' from the same equivalence class $Y_r^{(i)}$ entail the same transition probabilities in the *current* transition, it is possible that they entail different transition probabilities in some future transition. Therefore, an EMUM can have an infinite state space regardless of the fact that all partitions $(Y_r^{(i)})$ consist of finitely many subsets. For example, it is possible to define y as the sequence $(X_0, \ldots, X_{k-1})$ of all previously visited nodes, i.e., as the entire history (cf. Example 2.2 below).

As it is seen from Proposition 2.1, the alternative we propose for the solution of certain drawbacks of "classical" Markov models representing each state by a node, is a Markov model as well. So we do not claim to improve the general approach of modeling software usage by Markov chains, but rather to extend the representation technique.

2.2 A Description Language

Before introducing a concrete description language for EMUMs in formal terms, two nearly self–explaining examples might be helpful:

Example 2.1.

Intention: Two branches within a loop are to be traversed in an alternating sequence. After each single iteration, the loop is to be terminated with probability 0.05. However, after at most ten iterations, the loop is to be terminated in any case.

Realization: see Fig. 3.

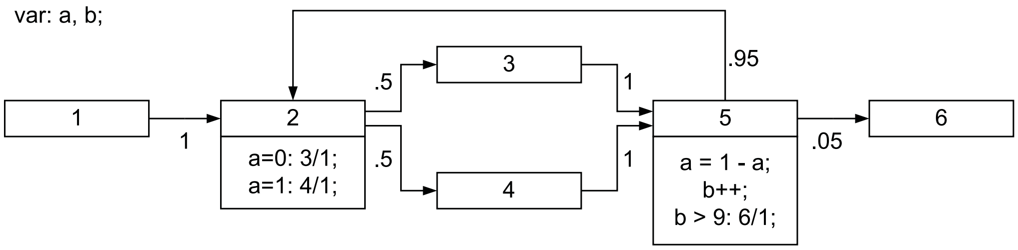

Fig. 3. EMUM of Example 2.1.

Comments:

- The `var` statement defines the names a and b as aliases for the first and the second frame variable, $y[0]$ and $y[1]$, respectively. The frame is the vector $y = (y[0], y[1])$. All frame variables are automatically initialized by zero. If the user wants to initialize some or all of them to something other than zero, he/she can assign the desired values in the initial node by the syntax described below.

- In node 2, if $a = 0$, the probability for the transition to node 3 is set to $p(2,3) = 1$ (consequently, $p(2,4) = 0$), and if $a = 1$, the probability for the transition to node 4 is set to $p(2,4) = 1$ (consequently, $p(2,3) = 0$). These explicitly assigned probabilities overrule the initial probabilities 0.5, 0.5.

- In node 5, the binary value of variable a is swapped, an the variable b, counting the number of iterations of the loop, is increased by one. After ten iterations, the probability for the transitions to node 6 is set to $p(5,6) = 1$ (consequently, $p(5,2) = 0$), which effects that the loop is terminated. However, the loop may have been terminated already before due to the original probability of 0.05 for transition $(5,6)$.

Example 2.2.

Intention: Loop as in Example 2.1 with two branches, each to be traversed with probability 0.5. If in the last iteration before the current iteration, the first of the two branches has been traversed, then the program is to be terminated immediately. Otherwise, the next iteration is to be performed.

Realization: see Fig. 4.

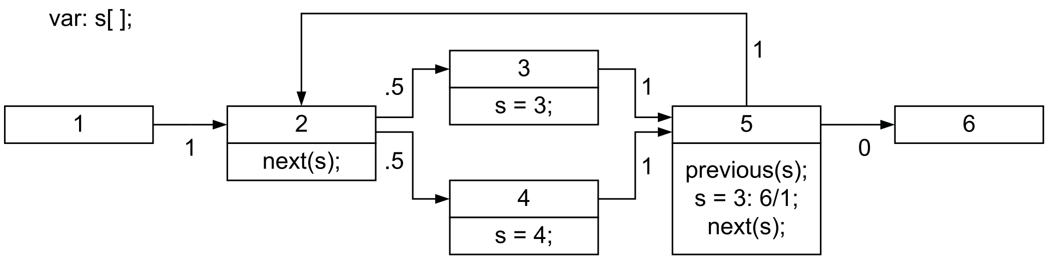

Fig. 4. EMUM of Example 2.2.

Comments:

- In this example, a (doubly linked) *list* of frame variables s is used. Initially, an implicit pointer refers to the first element of the list.
- In node 2, the pointer is switched to the next element of the list.
- In the nodes 3 and 4, the current node number (3 or 4) is written into the current element of the list.
- In node 5, a "look into the past" (the iteration before the current iteration) decides whether the loop is to be terminated.

The formulation above shows that the frame variables are able to record the whole path traversed up to now, which allows access to any event in the past.

Outline of the syntax of the description language:

We do not specify the syntax in every detail, leaving the exact specification of "boolean expression" and "arithmetical expression" (which are precisely defined in an abstract sense) as well as that of the variable names and literals open. The concrete form of the syntax of these elements can be filled in from an arbitrary programming language, say C++.

The symbols [...] and $\langle \ldots | \ldots \rangle$ are used as meta–symbols, denoting optional parts and alternatives, respectively.

EMUM:
[`var:` *var-name* [`[ ]`], *var-name* [`[ ]`], ... ;]
`node` [*node-index*]
  [*condition*:] $\langle$ *assignment;* | { *assignment; assignment;* ... } $\rangle$
  ...
  ...
`node` [*node-index*]
  ...
...

condition:
boolean expression, containing *simple-conditions*

simple-condition:
var-name $\langle = | \neq | \leq | \geq | < | > \rangle$ $\langle$ *int-literal* | *var-name* $\rangle$

assignment:
$\langle$ *var-assignment* | *prob-assignment* | `next` (*var-name*) | `previous` (*var-name*) $\rangle$

var-assignment:
var-name $\langle$ `++` | `--` | = arithmetical expression, containing *int-literals* and/or *var-names* $\rangle$

prob-assignment:
node-index / *float-literal*, *node-index* / *float-literal*, ...

Semantics of the description language:

1. The `var` statement defines aliases for the frame variables $y[0]$, $y[1]$, $y[2]$, …. By *var-name*, a single alias is defined. By *var-name*`[ ]`, a list with an implicit pointer is installed, and the pointer is initialized by directing it to the first element of the list. The aliases are assigned to the variables $y[0]$, $y[1]$, $y[2]$, … in a natural way. For example,

   ```
   var:  a, b[ ], c[ ];
   ```

   effects that $y[0] = a$, $y[1] = b[0]$, $y[2] = c[0]$, $y[3] = b[1]$, $y[4] = c[1]$, etc.

2. A node statement is activated as soon as the node with the indicated index is reached. If the statements are integrated into the graph representation (as in Example 2.1 and 2.2), the node statement is implicit.

3. *condition*, if present, formulates the condition for the execution of the subsequent assignment(s).

4. *var-name* refers to a certain frame variable, i.e., a component of the frame. If a list *var-name*`[ ]` has been defined, the current value of the pointer assigned to *var-name* decides which element of the list is selected.

5. *var-assignment* increments or decrements a frame variable, or it assigns the value computed from the r.h.s. to the variable.

6. *prob-assignment* updates the vector of transition probabilities from the current node to its successors. The arcs to the indicated successor nodes get the indicated probabilities, and all other arcs to successors get probability zero.

7. The statements `next` (*var-name*) and `previous` (*var-name*) switch the current pointer assigned to *var-name* to the next and to the previous element of the list, respectively.

The reader might wonder why we are not willing to support larger parts of a commonly used programming language as elements of our syntax. The reason is that a broader syntax would lead to uncontrollable effects preventing the generalization of analytical techniques in Markov usage modeling to our framework. E.g., if we would allow probability assignments to contain (expressions in) frame variables or other probabilities, the general assumption in Section 1 that there are only finitely many equivalence classes $Y_r^{(i)}$ would not necessarily be satisfied anymore, with the consequence that the technique described in Section 3 would not work.

Nevertheless, we have been very careful not to restrict the syntax in an essential way: The following result shows that there are no inherent limitations concerning the computational power of the presented language:

Proposition 2.2. The description language defined above has the computational power of a universal programming language.

The proof can be found in the Appendix. In the proof, the dynamically expandable size of the frame, i.e., the unbounded number of elements of Y, is essentially used. The assertion of Proposition 2.2 does not hold anymore for ordinary Markov chains with a finite number of states, not even if the "splitting technique" for nodes (cf. Section 1) is used.

It is easy to implement a precompiler translating an EMUM description, formulated in the presented language, into, say, C++ code for generating random paths $((X_0, Y_0),\ (X_1, Y_1), \ldots)$ according to the specified EMUM. Estimates for long–run probabilities, expected occurrences and some other parameters (see [7]) can be obtained from a large sample of such paths, i.e., by simulation. Also rare–event simulation techniques may be applied for improving the accuracy of such estimates. The *exact* determination of these parameters is more difficult and a topic of future research (cf. the Conclusion), or, for certain parameters, even infeasible, at least in the general case.

The task of optimizing *dependability tests* based on such a description is the topic of the next section. Before turning to this topic, let us mention a drawback of our approach: We cannot indicate a general procedure for deciding whether or not a given EMUM is *consistent* in the sense that the probability of an *infinite* random path is zero. Infinite paths may already occur in a "classical" Markov–chain model, e.g., if some part of the transition graph can be entered, but not left anymore. However, a reachability analysis would immediately reveal this situation. A procedure checking consistency of a given EMUM cannot exist: As long as we want to provide the power of a universal programming language, we necessarily also inherit the *halting problem.* So, after all, the designer of an EMUM is in the same situation as a programmer: No general algorithm exists to decide whether a given program will always terminate, but if the program *does* terminate correctly for a large number of test inputs selected according to the usage profile, then the probability of non–termination can be considered as negligible. Analogously, if a large number of trials to generate (finite) sample paths from an EMUM always succeed, we may conclude that the probability of an infinite path is either zero or of a negligible size.

For *practical* purposes, an EMUM can always be specified in a form where consistency is guaranteed: One way to achieve this is the following: Introduce an implicit small probability of ϵ to go from each node immediately to the terminal node (say, $\epsilon = 10^{-5}$) and reduce all other transition probabilities by multiplying them by $1 - \epsilon$. From the viewpoint of applications, this modification is even desirable, because it is *realistic:* it mimics a rare, but possible program abortion by some external event or by hardware problems.

A second pragmatic approach to guarantee consistency would be to introduce an additional (possibly implicit) frame variable counting the total number of transitions. If a pre–specified limit is reached, the path is completed by a transition to the terminal node.

3 Optimization of Test Transition Probabilities

In this section, the approach developed in [3], [4] and [5] for computing optimal *test* transition probabilities from a Markov usage model will be extended to non–Markovian state transitions. We are able to do this in the general formal framework of Subsection 2.1, i.e., without using the specific features of the description language presented in Subsection 2.2. Also other possible description languages are accessible to our approach, provided that they satisfy the conditions indicated in Subsection 2.1.

The idea of the approach is the following: One main purpose of Markov usage models (actually: usage models in general) is their application in *statistical testing* (see [7]), a testing technique that allows the determination of unbiased statistical estimates for dependability measures as *reliability*, *risk* or *safety.* In their simplest version, such tests use random inputs based on the usage profile itself, but it is also possible to change the test profile, with the advantage that the accuracy of the dependability estimate may be increased by exploiting information on especially error–prone or critical parts of the software system. In our approach, this information is represented by two list of numbers, a list $(f(i,j))$ and a list $(l(i,j))$ $((i,j) \in A)$, where $f(i,j)$ and $l(i,j)$ are the estimated failure probability and the estimated loss due to failure, respectively, for the processing step corresponding to arc (i,j). (For *reliability* estimation, only the list with the values $f(i,j)$ is required.) For more details to the definition and estimation of the numbers $f(i,j)$ and $l(i,j)$, as well as to the following concepts in the Markovian special case, the reader is referred to [5].

The set of possible paths $x = (x_0, x_1, \ldots)$ will be denoted by D. Obviously, for a fixed path $x \in D$,

$$P(x) = \prod_{k \geq 1} p(x_{k-1}, x_k, y_{k-1}) \tag{2}$$

gives the probability that x is chosen during operational usage. Therein, y_0, y_1,... are the current contents of the frame at the beginning of the first, the second, ... transition, respectively. In Subsection 2.1, we have seen that y_k only depends on y_0 and on $x_0, \ldots, x_{k-1}$. The probabilities $P(x)$ define a probability distribution on D.

In view of (1), the probabilities on the r.h.s. of (2) may also be written as

$$p(x_{k-1}, x_k, y_{k-1}) = p_r(x_{k-1}, x_k) = p_{r(y_0, x_1, \ldots, x_{k-1})}\,(x_{k-1}, x_k). \tag{3}$$

Note that r is determined by the subset $Y_r^{(x_{k-1})}$ into which y_{k-1} falls. Since y_{k-1} is a function of y_0 and $x_0, \ldots, x_{k-2}$, the integer r depends only on $y_0, x_0, \ldots, x_{k-1}$. The possible range for $r(y_0, x_1, \ldots, x_{k-1})$ is $\{1, 2, \ldots, R(x_{k-1})\}$.

Now, for the *test* of the software system, the probabilities $p_r(i,j)$ are changed to *test transition probabilities* $t_r(i,j)$. Because $1 \leq r \leq R(i)$, the number of variables $t_r(i,j)$ is $\sum_{(i,j) \in A} R(i)$ and hence finite. Under the changed transition probabilities, the probability of path $x \in D$ is

$$T(x) = \prod_{k \geq 1} t(x_{k-1}, x_k, y_{k-1}) = \prod_{k \geq 1} t_{r(y_0, x_1, \ldots, x_{k-1})}\,(x_{k-1}, x_k). \tag{4}$$

As P, also T is a probability distribution on D. By E_P and E_T, we denote the mathematical expectation w.r.t. P and T.

Let $c(x, \pi)$ denote a *cost function* representing the loss due to program failure if the software system π is executed at use x. The special form of this cost function depends on the particular dependability measure under consideration. Cost functions for the dependability measures reliability, risk and safety are presented in [5]. The (un)dependability measure itself is then given formally by $E_P(c(X, \pi))$, the expected loss under operational usage.

The uncertainty on the actual behavior of the software system is modeled by the assumption that π is subject to a distribution Q. (As to this assumption, we refer the reader to [2], where the last–mentioned distribution is denoted by P.) E_Q denotes the expected value w.r.t. the probability distribution Q. In [5], two models (a static and a dynamic model) are considered for representing the expected loss $E_Q(c(x, \pi))$ due to failure at use x by the values $f(i,j)$ and $l(i,j)$. For example, the static model for *risk* assumes

$$E_Q(c(x, \pi)) = \sum_{(i,j) \in A(x)} f(i,j)\, l(i,j),$$

where $A(x)$ denotes the set of transitions $(i,j) \in A$ occurring in path x.

As an estimator for the considered (un)dependability measure, we compute the average of the values

$$S = \frac{P(X)}{T(X)} \cdot c(X, \pi) = \prod_{k \geq 1} \frac{P(X_{k-1}, X_k, Y_{k-1})}{T(X_{k-1}, X_k, Y_{k-1})} \cdot c(X, \pi)$$

obtained after executing each single test case. Therein, X is selected according to the test distribution T, and $c(X, \pi)$ is the observed loss due to failure (zero if no failure has occurred) at the respective test case. When computing $P(X)$ and $T(X)$, eq. (3) and the corresponding formula for the values $t(x_{k-1}, x_k, y_{k-1})$ can be used. In sampling theory, the quotient $P(X)/T(X)$ is called the *likelihood ratio*; each test outcome has to be weighted by this quotient in order to compensate for the effect of the probability change.

Analogously as in [5], it is verified that

$$E_T(S) = E_P(c(X, \pi)),$$

i.e., that S is an unbiased estimator for the (un)dependability measure under consideration.

The accuracy of a statistical estimator is usually measured by its *variance*. Our aim is to choose the test transition probabilities $t_r(i, j)$ in such a way that the accuracy of the estimator is maximized, i.e., that its variance is minimized.

As a generalization of Lemma 3.1 in [5], we obtain:

Proposition 3.1. The expected variance $E_Q(var_T(S))$ is minimized with respect to T, if and only if

$$G(T) = E_P(L(X)\, H(X)) \tag{5}$$

is minimized, where $L(x) = P(x)/T(x)$ is the likelihood ratio, and

$$H(x) = E_Q([c(x, \pi)]^2).$$

The proof is quite analogous to that of Lemma 3.1 in [5] and is therefore omitted here.

For $H(x)$, specific formulas concerning the dependability measures risk, safety and reliability have been derived in [5]. Because $H(x)$ does not depend on $p_r(i, j)$ and $t_r(i, j)$, these formulas can be used without change in our context. For example, for *risk* under the static model,

$$H(x) \approx \sum_{(i,j) \in A} f(i, j)\, [l(i, j)]^2,$$

where the approximation is good whenever the failure probabilities $f(i, j)$ are small (this is always the case in practical applications of statistical testing).

Minimizing (5) under the constraints

$$\begin{array}{ll} 0 \leq t_r(i, j) \leq 1 & \text{for } (i, j) \in A,\ 1 \leq r \leq R(i), \\ \sum_{j \in Succ(i)} t_r(i, j) = 1 & \text{for } i \in V,\ 1 \leq r \leq R(i), \end{array}$$

is a *stochastic optimization problem* which can be solved by a combination of sampling and a gradient procedure, as in the algorithm below:

procedure `OptimizeTest` {
  set $T := P$;

draw a sample of paths according to P;
for $s := 1$ **to** $maxs$
  for $i := 1$ **to** n
    for $r := 1$ **to** $R(i)$ {
      for each j with $p_r(i, j) > 0$
        compute from the sample an estimate $D_r(i, j)$ of the
        partial derivative of $G(T)$ to the variable $t_r(i, j)$;
      determine the indices $j = j_{min}$ and $j = j_{max}$
        for which $D_r(i, j)$ is minimal and maximal, respectively;
      $t_r(i, j_{min}) := t_r(i, j_{min}) + const/s$;
      $t_r(i, j_{max}) := t_r(i, j_{max}) - const/s$; } }

Essentially, this is a generalization of the procedure `OptimizeTest` in [5]. There, only the *best* partial derivative of $G(T)$ is used for the computation of the next probability vector. The efficiency can be improved, if, as indicated above, also the *worst* partial derivative is used. The computation of the partial derivatives is outlined in [5] and can be adopted into our context by straightforward modifications.

Because it is based on sampling, `OptimizeTest` is a stochastic procedure the output of which depends on the influence of randomness and can therefore only be considered as an approximate solution. But even if we assume that the sample distribution reflects *exactly* the real distribution (this is the limiting case for a sample size tending to infinity), the algorithm `OptimizeTest` is only guaranteed to converge to the optimal solution if the function $G(T)$ is *convex* in the decision variables $t_r(i, j)$. A convexity proof for the Markovian special case has been given in [5], Theorem 5.1. Fortunately, it turns out that this result can be generalized to the extended framework considered here:

Proposition 3.2. For arbitrary $H(x)$, the function $G(T)$ in (5) is a convex function of the variables $t_r(i, j)$ $((i, j) \in A,\ 1 \leq r \leq R(i))$.

The proof is given in the Appendix.

Also modifications of the optimization technique concerning stepsize, bootstrap and initial test transitions, as they are described in [5], can be extended to our framework.

Examples for the application of the technique will be presented in the next section.

4 Examples

As an illustration for the application of our technique, we first present a relative small fictitious example of an Online Shop. Then, we investigate a second, medium-size real world example of an outbound call center which is used by the Austrian Red Cross.

4.1 Online Shopping Applet

The model in Fig. 5 reflects a small e–commerce application allowing the user, i.e., a potential customer, to order articles — say, books. The number of books that can be ordered in a single use is limited by three.

Comments:

- When a potential customer visits the virtual shop, the applet is initialized by the browser (node 1). After login (node 2), the virtual shop with the list of the available books is presented to the potential customer (node 3).
- Starting from this window, it is possible to call the online help (node 11), to view the already selected books in the shopping cart (node 4), or to select a (new) book (node 5).
- After the *first* visit of the online help (node 11), the probability to visit it again during a certain use is decreased (from 0.05 to 0.01). To express this, we introduce the frame flag h.
- If a particular book has been selected (node 5), the customer can decide whether to put it into the shopping cart (node 4) or to go back to the list of books (node 3).
- The flag f is used to indicate a selection of a book in node 5. We use this flag to mark whether the shopping cart window (node 4) was called to add a selected book (arc (5,4)) or to view the selected book(s) (arc (3,4)). In the first case, the book is added to the shopping cart, which means that the variable i counting the number of books in the cart has to be increased by one. Depending on the counter i, the probability of returning to the list of books is decreased more or less, and the probability to proceed the purchase by the confirmation of the selection (node 9) is increased. In particular, if the customer has already selected three books ($i = 3$), it is not possible anymore to go back to the list of books. Therefore, the probability for that action is set to zero.
- In our fictitious book-shop, we force the customer to confirm his/her selection before he/she can submit the order (node 7). Alternatively, he/she may cancel the order (node 6).
- For each selected book, a confirmation is required by the customer (node 10). To handle the number of confirmations, we use the loop frame variable a which is initially set equal to i and reduced by one each time a book is confirmed (node 10). In each iteration, the customer confirms the purchase of one selected book. After the last confirmation, the customer is forced to go back to the shopping cart window. Note that at this moment, the confirmation flag v has the value one.
- Only after confirmation of each single book, the customer can decide whether to submit or cancel the order. Before leaving the online shop, he/she may print his/her order (node 8). By means of the frame flags s, c, and p, we represent the fact that the functions submit, cancel, and print, respectively, can be called at most once, and that submitting and cancelling are mutually exclusive operations.

```
var: i, f, c, s, p, v, a, h;
```

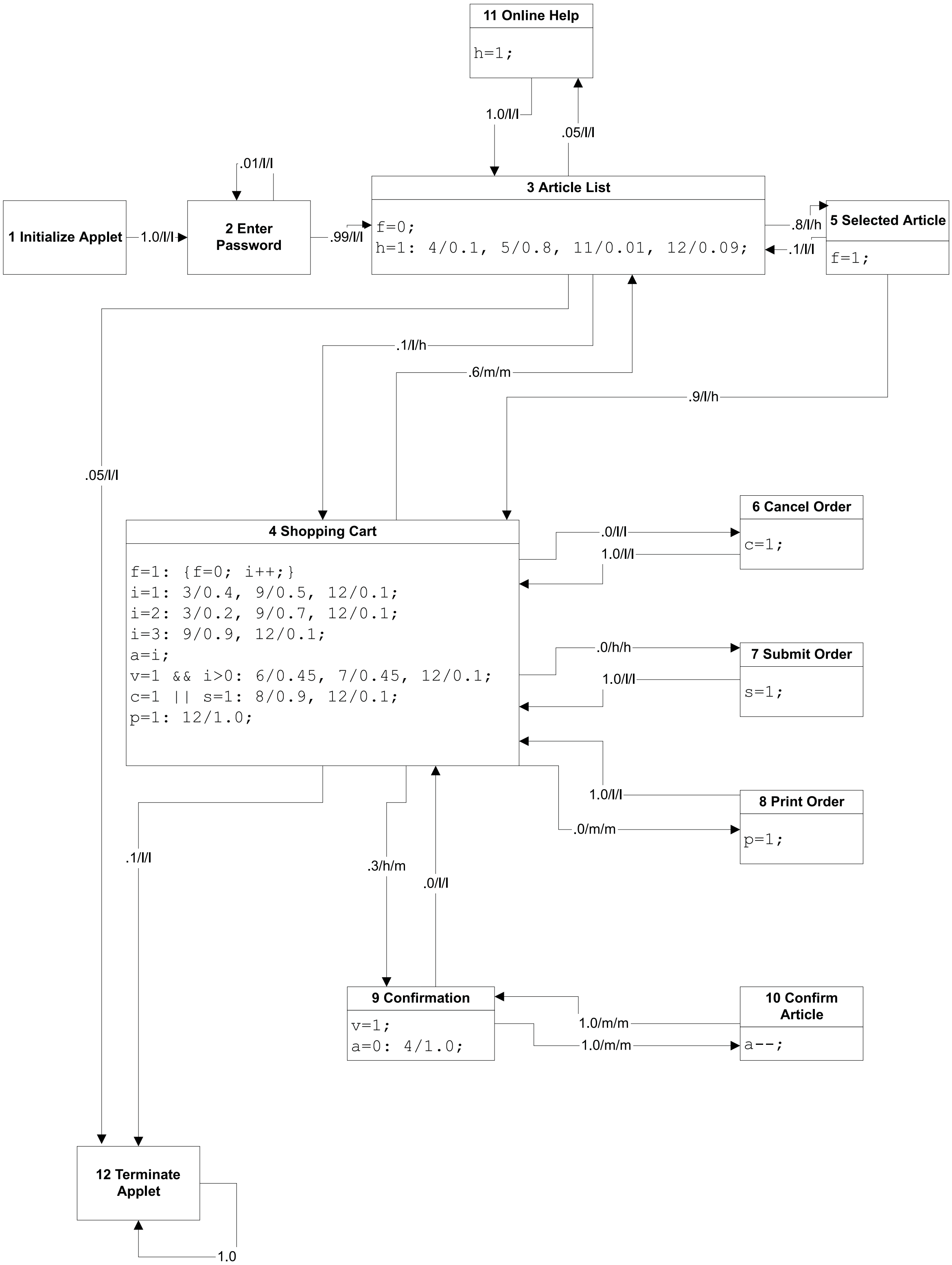

Fig. 5. A small EMUM of an Online Shop.

As the first type of output, our program can produce an arbitrary number of random traversals of the shopping system according to the EMUM description in Fig. 5. One traversal obtained in this way was the following: 1, 2, 3, 5, 4, 3, 5, 4, 9, 10, 9, 10, 9, 4, 7, 4, 8, 4, 12. In this use, two articles are added to the shopping cart. After the confirmation of the two articles by iterating the loop (9, 10) twice, the order is submitted and printed. Then the program is terminated.

Our description in Fig. 5 also includes an estimation of the failure rate $f(i,j)$ and of the loss in case of failure, $l(i,j)$, for each transition (i,j). We assume that the submission of an order has a high failure rate because of the necessary data transfer over the internet (arc (4,7)). Assuming that the confirmation function (arc (4,9)) has been newly implemented, we also expect a high failure rate for this function. In the case that the submission of the order fails, a high loss is expected with regard to the frustration of the customer. If the customer's attempt to call the shopping cart in arc (3,4) and arc (5,4) fails, a high loss is expected as well. In Fig. 5, the values $f(i,j)$ and $l(i,j)$ are integrated into the EMUM description. Each arc (i,j) is labeled with the following estimates: $p(i,j)$ / $f(i,j)$ / $l(i,j)$. Therein, the constants l (*low*), m (*medium*), and h (*high*) have the following values: For $f(i,j)$, $l = 0.001$, $m = 0.01$, and $h = 0.1$. For $l(i,j)$, $l = 1$, $m = 10$, and $h = 100$.

As a second output, optimal test transition probabilities can be computed. We have applied the procedure OptimizeTest of Section 3 with $N = 10000$, $maxs = 100$, and bootstrap as described in [4] with $K = 50$ iterations in the outer loop. The results are presented in Table 1. Therein, the line index $i(r)$ refers to node i and the rth equivalence class of the frame variables in node i (see Subsection 2.1).

	1	2	3	4	5	6	7	8	9	10	11	12
1(1)	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00	.00	.00
2(1)	.00	.05	.95	.00	.00	.00	.00	.00	.00	.00	.00	.00
3(1)	.00	.00	.00	.15	.70	.00	.00	.00	.00	.00	.13	.02
3(2)	.00	.00	.00	.09	.84	.00	.00	.00	.00	.00	.05	.02
4(1)	.00	.00	.65	.00	.00	.00	.00	.00	.32	.00	.00	.03
4(2)	.00	.00	.40	.00	.00	.00	.00	.00	.57	.00	.00	.03
4(3)	.00	.00	.20	.00	.00	.00	.00	.00	.77	.00	.00	.03
4(4)	.00	.00	.00	.00	.00	.00	.00	.00	.97	.00	.00	.03
4(5)	.00	.00	.00	.00	.00	.13	.84	.00	.00	.00	.00	.03
4(6)	.00	.00	.00	.00	.00	.00	.00	.89	.00	.00	.00	.11
4(7)	.00	.00	.00	.00	.00	.00	.00	.00	.00	.00	.00	1.0
5(1)	.00	.00	.09	.91	.00	.00	.00	.00	.00	.00	.00	.00
6(1)	.00	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00
7(1)	.00	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00
8(1)	.00	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00
9(1)	.00	.00	.00	.00	.00	.00	.00	.00	.00	1.0	.00	.00
10(1)	.00	.00	.00	.00	.00	.00	.00	.00	1.0	.00	.00	.00
11(1)	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00	.00
12(1)	.00	.00	.00	.00	.00	.00	.00	.00	.00	1.0	.00	1.0

Table 1. Optimal test transition probabilities in the Online Shop.

It can be observed that the intended effect of favoring the test of critical operations has been achieved. According to P, the critical transition (4,7) has only probability 0.45 of being tested, which is the same as for the uncritical transition (4,6). According to T, the probability of transition (4,7) is increased to 0.84 and the probability of transition (4,6) is decreased to 0.13. Also the test probability for the newly implemented function (4,9) is increased, as it can be seen in the rows 4(1), 4(2), 4(3), 4(4).

4.2 Outbound Call Center

In this example we model the marketing database of an outbound call center. To increase transparency, we have partitioned the entire EMUM description into four diagrams, the Main Section (Fig. 6) and three subsections (Fig. 7–9). Entry node and exit node of a subsection are represented in the Main Section and are repeated in the subsection. The statements assigned to a node that refer to the frame variables are only listed in the subsection copy of the node.

The main functionality is *customer processing* (node 6 and the subsequent part of the graph), allowing an operator to poll information about a certain customer (in the concrete case: a donor for the Red Cross). This information is used for a phone call with the customer. The results of the call can be used to update the database and another potential customer can be selected (node 6). To control the operators and to optimize the workflow, a supervisor can compute different *statistics* (node 20). The functionalities to distribute work and to maintain so–called campaign data form the *supervisor* section (node 24).

The description of the subsection *customer processing* is depicted in Fig. 7. By means of customer processing, the operator can select a potential new or an already registered customer, and the customer's data can be seen on the screen (node 7). There is also further information about the customer available: information on the last phone calls with the customer (node 8) or the complete chronology of interactions with him/her can be viewed (node 9). We assume that node 8 and node 9 are traversed at most once for each customer. We use the frame variables b and c to model this assumption.

As soon as the operator knows the important facts about the potential customer, he/she will, with a certain probability, proceed to perform a phone call (node 10). After the call, the operator can modify customer data, e.g., register an address change. If there are no changes, the operator can go directly to a function where he can feed the call results into the computer (node 15). We use frame variable a to count the number of customers already processed. After a certain number of processed customers, the probability to leave the current subsection is increased.

In the subsection *statistics* (see Fig. 8), several performance statistics can be computed and printed or exported to an accounting program. We use the frame flag f to model the probability increase for exporting or printing statistical data after their computation (node 20).

In the subsection *supervisor* (see Fig. 9), the three main functionalities *check campaign results* (node 31), *distribute campaigns* (node 36) and *maintain campaigns* (node 39) are implemented. One task of the supervisor is to control the work of the operators; for this purpose, he/she uses the functionality *check campaign results*. The number of results that can be checked consecutively is limited by ten to avoid inattention of the supervisor. To express this, we introduce the frame variable n. Another task of the supervisor is to distribute the work to the available operators (node 36). Certain customers are grouped to so–called *campaigns*, and each campaign is assigned to an operator (node 39). Here, frame variables are used to record the numbers of campaigns of certain types.

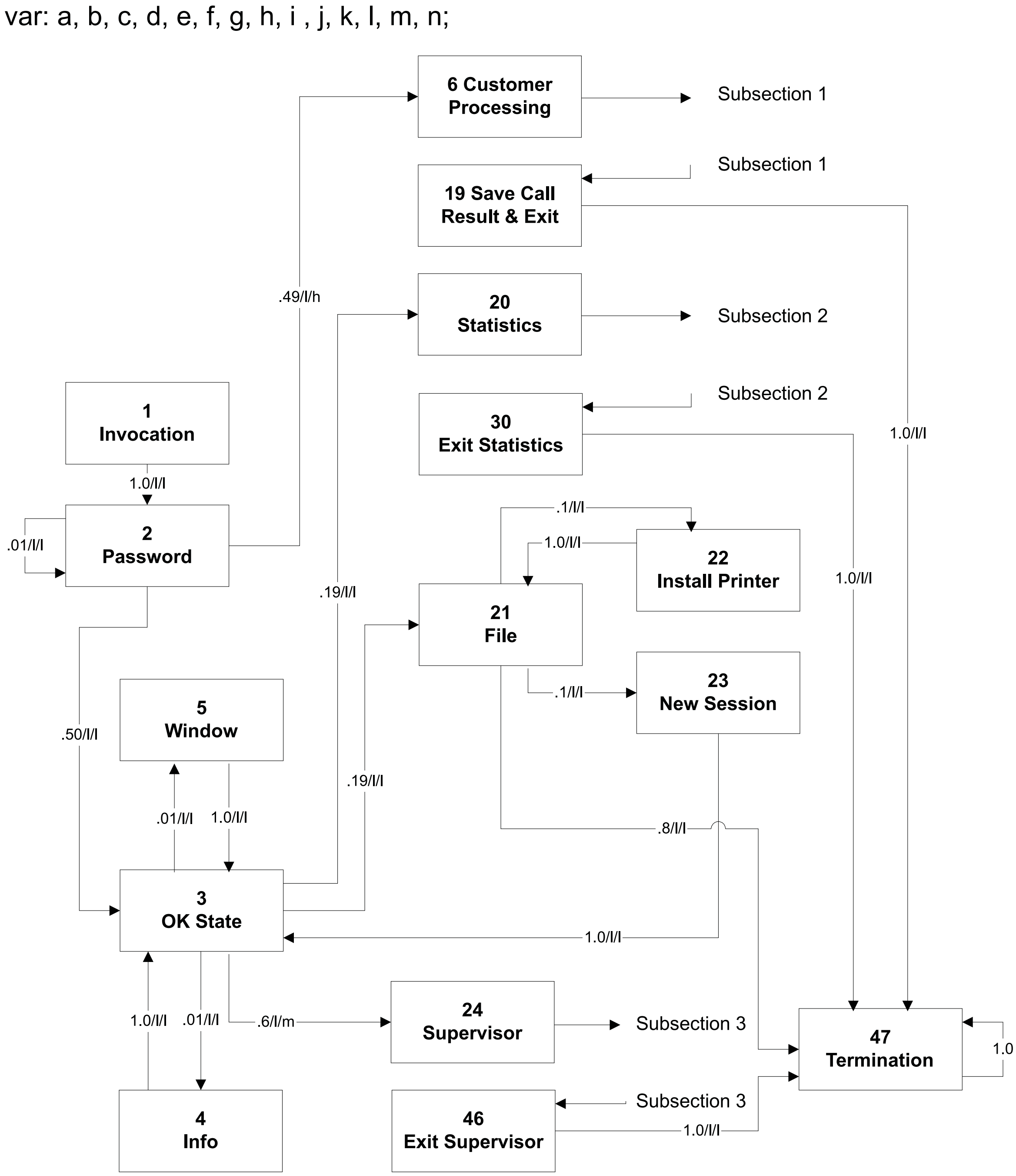

Fig. 6. An EMUM of an Outbound Call Center - Main Section.

As in the previous example, failure rates and losses in case of failure have been estimated. A high failure rate is expected for the functions (7,8), (7,9), (8,9) and (9,8). Also the loss in case of failure is high for these functions, since it is not possible to perform a phone call if the operator has wrong or missing information about the (potential) customer. We also expect a high failure rate for the function (36, 37) since it uses additional information from an error–prone legacy system, as well as a high loss in case of failure.

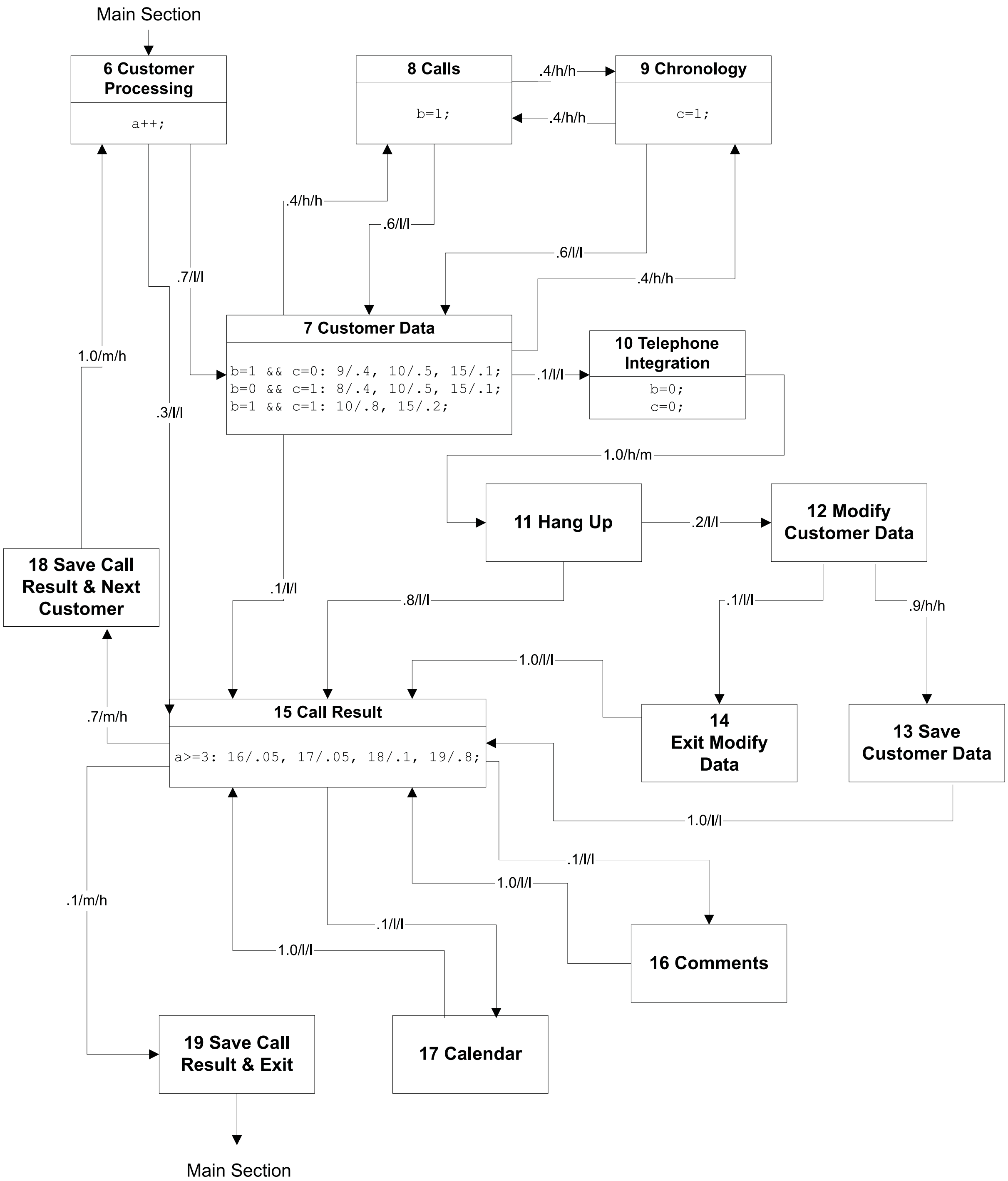

Fig. 7. An EMUM of an Outbound Call Center — Subsection 1.

The resulting optimal test transition probabilities for this EMUM have been computed by means of the procedure OptimizeTest of Section 3 with $N = 1000$, $maxs = 100$ and bootstrap as described in [4] with $K = 100$ iterations in the outer loop. For the sake of brevity, only part of the output is presented: Table 2 and Table 3 contain the test transition probabilities for node 7 and node 36, respectively. The line index $i(r)$ has the same meaning as in Subsection 4.1. It can immediately be observed that according to T, the probability for the critical functions $(7, 8)$, $(7, 9)$ and $(36, 37)$ has been increased.

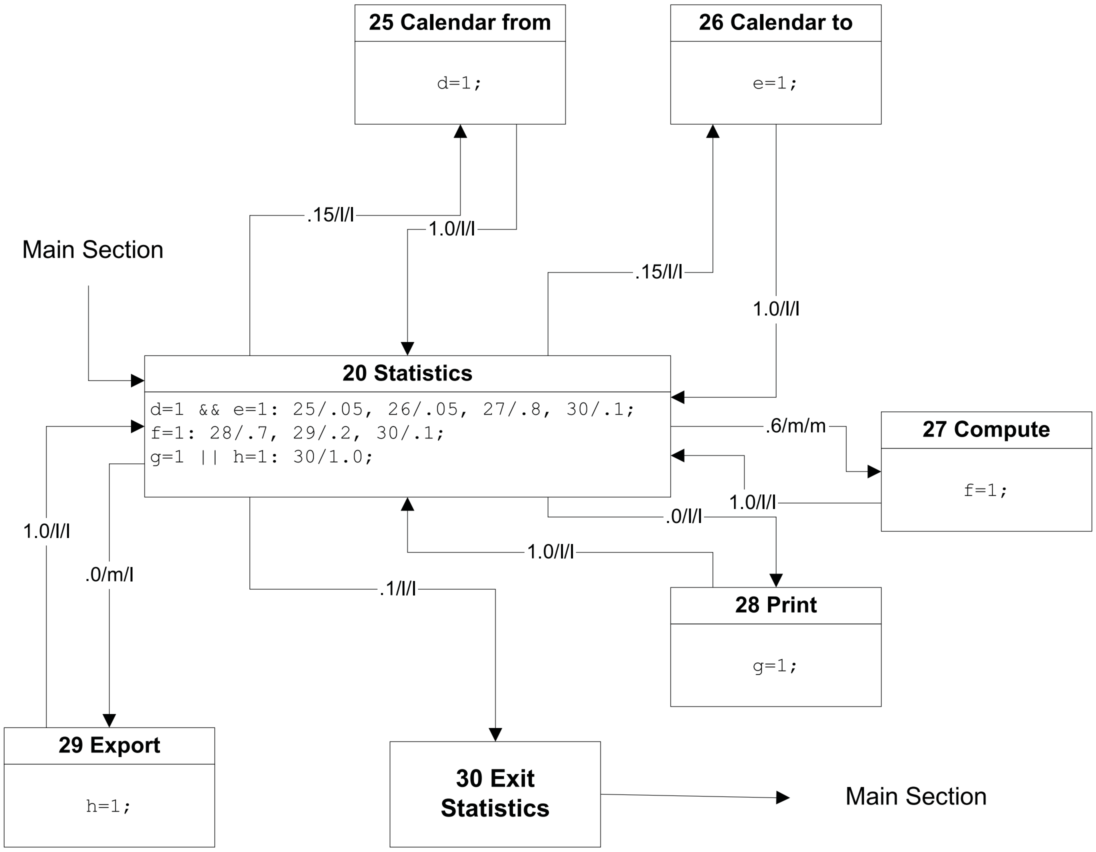

Fig. 8. An EMUM of an Outbound Call Center — Subsection 2.

	8	9	10	15
7(1)	.43	.43	.07	.07
7(2)	.00	.45	.47	.08
7(3)	.45	.00	.47	.08
7(4)	.00	.00	.81	.19

Table 2. Optimal test transition probabilities in the Call Center Example for node 7.

	37	38	46
36(1)	.82	.09	.09
36(2)	.13	.79	.08
36(3)	.13	.10	.77

Table 3. Optimal test transition probabilities in the Call Center Example for node 36.

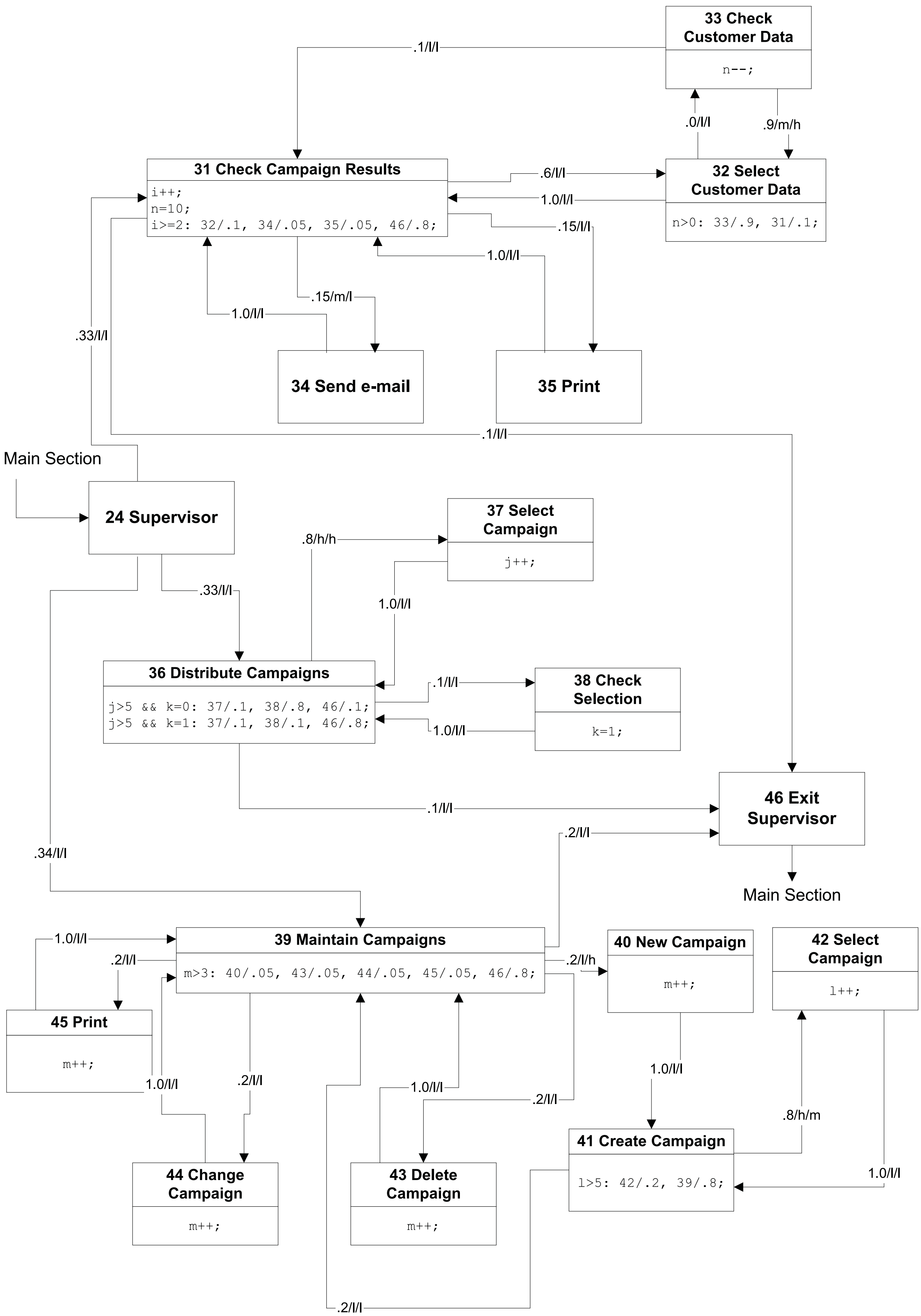

Fig. 9. An EMUM of an Outbound Call Center — Subsection 3.

5 Conclusions

We have shown that it is possible to extend Markov usage models to cases where also non–Markovian state transitions occur. This can be done by adding statements in a simple, easily understandable syntax to the nodes of the Markov model. Our formalism provides the full computational power of a universal programming language, which means that the specification of the software usage can be refined to any desired degree of accuracy. On the other hand, it contains ordinary Markov models as a special case, and it preserves their visualization capacities also in the case of non–Markovian transitions.

In our opinion, an extension of the suggested kind is not only valuable in view of a more adequate representation of the software usage, but, in some cases, even indispensable, at least if test cases are to be derived from the model: Often, the possible actions in a certain state essentially depend on what has happened before this state has been reached. A strictly Markovian model might therefore produce test scenarios that cannot be performed in reality. Using our approach, it is always feasible to obtain only paths through the state graph that can be realized by appropriate test inputs.

As we have demonstrated, one specific technique based on usage models, namely the computation of good transition probabilities between states for dependability tests with regard to failure–prone or critical operations, can be fully extended to our framework for the non–Markovian situation, and works also for medium–size models with real–world data. It would be desirable to generalize other established techniques as well, such as the exact or approximate computation of useful statistics (long run probabilities, probabilities of occurrence in a single sequence, expected number of occurrences, expected number of transitions until the first occurrence, expected sequence length), optimization with respect to minimal cost or maximal value of testing under given constraints, automated testing using scripts, recording test experience, application of experimental design techniques, partition testing with usage models, etc. (cf. [7]). Especially for very large usage models, also deterministic test case selection heuristics would be of interest, as they have been outlined in Avritzer and Weyuker [1] and Weyuker [9]. Given that the presented formalism for extended Markov models is considered as attractive and useful, there remains a wide area for future methodological and empirical research to exploit the inherent potential of the approach.

Acknowledgment. The authors are indebted to the anonymous reviewers for their helpful and constructive comments.

References

[1] Avritzer, A., Weyuker, E. J., "The automatic generation of load test suites and the assessment of the resulting software", *IEEE Trans. Software Eng.*, vol. SE-21, 705 – 716 (1995).

[2] Eckhardt, D. E., Lee, L. D., "A theoretical basis for the analysis of multiversion software subject to coincident errors", *IEEE Trans. Software Eng.*, vol. SE-11, 1511 – 1517 (1985).

[3] Gutjahr, W. J., "Failure risk estimation via Markov software usage models", *SAFECOMP 96, Proc. of the 15th International Conference on Computer Safety, Reliability and Security*, E. Schoitsch (ed.), 183 – 192, Springer (1997).

[4] Gutjahr, W. J., "Importance sampling of test cases in Markovian software usage models", *Probability in the Engineering and Informational Sciences*, vol. 11, 19 – 26 (1997).

[5] Gutjahr, W. J., "Software Dependability Evaluation Based on Markov Usage Models", to appear in: *Performance Evaluation* (2000).

[6] Hopcroft, J. E., Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley (1979).

[7] Poore, J. H., Trammell, C. J., "Application of statistical science to testing and evaluating software intensive systems", in: *Statistics, Testing, and Defense Acquisition*, Washington: National Academy Press (1998).

[8] Walton, G. H., Poore, J. H., Trammell, J., "Statistical testing of software based on a usage model", *Software – Practice and Experience*, vol. 25 (1), 97 – 108 (1995).

[9] Weyuker, E. J., "Using failure cost information for testing and reliability assessment", *ACM Trans. on Software Eng. and Methodology*, vol. 5, 87 – 98 (1996).

[10] Whittaker, J. A., Poore, J. H., "Markov analysis of software specifications", *ACM Trans. on Software Eng. and Method.*, vol. 2 (1), 93 – 106 (1993).

[11] Whittaker, J. A., Thomason, M. G., "A Markov chain model for statistical software testing", *IEEE Trans. Software Eng.*, vol. SE-20, 812–824 (1994).

[12] Woit, D. M., "Operational profile specification, test case generation, and reliability estimation for modules", Thesis, Queen's University, Kingston, Ontario, Canada (1994).

APPENDIX

Proof of Proposition 2.2. It suffices to show that a Turing machine (TM, see, e.g., [6]) can be simulated on an EMUM described by means of the presented language. Let Q denote the finite set of states of the given TM. Without loss of generality, $Q = \{1, \ldots, n-1\}$, where n is the number of elements of Q increased by one. We define node set and arc set of the EMUM by

$$V = \{1, \ldots, n\}, \quad A = \{(i,j) \mid 1 \leq i \leq n-1,\ 1 \leq j \leq n-1\} \cup \{(i,n) \mid 1 \leq i \leq n\}.$$

Let Γ denote the finite set of allowable tape symbols of the TM, and let Γ^* denote the set of words on Γ. Without loss of generality, $\Gamma = \{1, \ldots, m\}$. We set $Y = \Gamma^*$ and identify the current tape content of the TM with the current frame of the EMUM. The initial node of the EMUM is the initial state, say 1, of the TM. The initial frame of the EMUM is the input of the TM, i.e., its initial tape content. In other words, it is assumed that the frame variables $y[1]$, $y[2]$, … initially contain the input (which is easy to achieve by a `read` loop). The terminal node of the EMUM is the node n which is not a state of the TM.

Let $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ be the *next move function* of the TM. For given $i \in Q$ and $\gamma \in \Gamma$, $\delta(i, \gamma)$ consists of the components $\delta_1(i, \gamma) \in Q$, $\delta_2(i, \gamma) \in \Gamma$ and $\delta_3(i, \gamma) \in \{L, R\}$. Furthermore, let $Q_f \subseteq Q = \{1, \ldots, n-1\}$ denote the set of final states of the TM.

Now, we start the definition of the EMUM by the statements

```
var:  a[ ];
node [1]
  a = -1; next(a);
```

For each $i \in Q = \{1, \ldots, n-1\}$, we add a node statement of the following form to the definition of the EMUM:

```
node [i]
  a = 1:  {a = δ2(i,1); Δ12(i,1); Δ3(i,1);}
  a = 2:  {a = δ2(i,2); Δ12(i,2); Δ3(i,2);}
        ...
  a = m:  {a = δ2(i,m); Δ12(i,m); Δ3(i,m);}
```

Therein, $\Delta_3(i,\gamma)$ is to be substituted by `next(a)` if $\delta_3(i,\gamma) = L$, and by `previous(a)` if $\delta_3(i,\gamma) = R$, and $\Delta_{12}(i,\gamma)$ abbreviates the following: If $\delta_1(i,\gamma) \notin Q_f$ and $\delta_2(i,\gamma) \geq 0$, then $\Delta_{12}(i,\gamma)$ is to be substituted by $\delta_1(i,\gamma)/1$. This effects a (deterministic) transition from node i to node $\delta_1(i,\gamma)$. If, on the other hand, $\delta_1(i,\gamma) \in Q_f$ or $\delta_2(i,\gamma) < 0$ (cf. below), then $\Delta_{12}(i,\gamma)$ is to be substituted by $n/1$, which effects a (deterministic) change form node i to the terminal node n.

The assignment of $a = -1$ to the frame variable $y[0]$ ensures that if the head of the tape of the TM is shifted from position 1 to the left, i.e., to position 0, the processing terminates in the next step, since then $\delta_2(i,\gamma) < 0$.

It is easily verified that the EMUM defined as above behaves exactly in the same way as the given TM, reaching its terminal node when the TM halts, and producing the same output. □

Proof of Proposition 3.2. As shown in Section 2.1, y_k is a function of y_0 and of $x_0, \ldots, x_{k-1}$, say,

$$y_k = \bar{\varphi}(y_0, x_1, \ldots, x_{k-1}).$$

With $H(x)$ as in Proposition 3.1, we set $H_P(x) = P(x)\,H(x)$, such that

$$\begin{aligned} H_P(X) = P(X)\,H(X) &= \prod_{k\geq 1} p(X_{k-1}, X_k, Y_{k-1}) \cdot H(X) \\ &= \prod_{k\geq 1} p(X_{k-1}, X_k, \bar{\varphi}(y_0, X_1, \ldots, X_{k-1})) \cdot H(X). \end{aligned} \tag{6}$$

(Note that y_0 is not a random variable, since the initial frame is fixed.) The expression (6) does not depend on the decision variables $t_r(i,j)$. Using $H_P(X)$, the function $G(T)$ can be written as

$$G(T) = E_P(H_P(X) \cdot [T(X)]^{-1}), \tag{7}$$

where

$$[T(X)]^{-1} = \prod_{k\geq 1} \frac{1}{t(X_{k-1}, X_k, \bar{\varphi}(y_0, X_0, \ldots, X_{k-1}))} = \prod_{k\geq 1} \frac{1}{t_{r(y_0, X_0, \ldots, X_{k-1})}(X_{k-1}, X_k)}. \tag{8}$$

For fixed $x \in D$, let $M(i,j,r,x)$ denote the number of transitions k with $(x_{k-1}, x_k) = (i,j)$ and $r(y_0, x_0, \ldots, x_{k-1}) = r$. In other words: $M(i,j,r,x)$ is the number of those transitions of the path x that use the test probability $t_r(i,j)$ $(1 \leq r \leq R(i))$. Then we may rewrite (8) as

$$[T(X)]^{-1} = \prod [t_r(i,j)]^{-M(i,j,r,X)}, \tag{9}$$

where the product is over all triples (i,j,r) with $t_r(i,j) > 0$ and $1 \le r \le R(i)$. (For $t_r(i,j) = 0$, also $M(i,j,r,x) = 0$ with probability one, so transitions of this type need not to be considered.) Labeling the triples (i,j,r) satisfying the conditions above by $\sigma = 1, 2, \ldots$, the r.h.s. of (9) becomes of the form

$$z_1^{-m_1} \ldots z_s^{-m_s},$$

where $z_\sigma = t_r(i,j) \ge 0$ and $m_\sigma = M(i,j,r,X) \in \{0,1,\ldots\}$. As shown in [4], the function $z_1^{-m_1} \ldots z_s^{-m_s}$ is convex in $z_1, \ldots, z_s$. Hence, $[T(X)]^{-1}$ is convex in the decision variables $t_r(i,j)$. Because $H_P(X)$ does not depend on these variables, also $H_P(X)\,[T(X)]^{-1}$ is convex. The result follows from (7) and the fact that the linear operator E_P preserves convexity. $\square$