# Extracting Test Sequences from a Markov Software Usage Model by ACO

Karl Doerner[1] and Walter J. Gutjahr[2]

[1] Department of Management Science, University of Vienna,
Bruenner Strasse 72, A-1210 Vienna, Austria
[2] Department of Statistics and Decision Support Systems, University of Vienna,
Universitaetsstrasse 5/3, A-1010 Vienna, Austria
{Karl.Doerner, Walter.Gutjahr}@univie.ac.at

**Abstract.** The aim of the paper is to automatically derive a suitable set of test paths for a software system. The design and the possible uses of the software system are modelled by a Markov Usage Model which reflects the operational distribution of the software system and is enriched by estimates of failure probabilities, losses in case of failure and testing costs. Exploiting this information, we consider the tradeoff between coverage and testing costs and try to find an optimal compromise between both. For that purpose, we use a heuristic optimization procedure inspired by nature, Ant Colony Optimization, which seems to fit very well to the problem structure under consideration. A real world software system is studied to demonstrate the applicability of our approach and to obtain first experimental results.

## 1 Introduction

Markov usage models, as introduced by Whittaker, Poore, Walton and other authors (see [30]–[32], [29], [23], [26], [11]) are versatile model descriptions for the use of a given software product in the application field. It is common knowledge that users do not exploit all functions described in the specification of a product with the same intensity; some features of a software application get "everyday routine", others are very rarely touched – or never. A software developer can schedule her/his resource allocations during different phases of the development cycle more cost-effectively, if she/he takes account of these differences. For example, it does not make sense to perform extensive design reviews, code inspections and tests for specific usage constellations that will probably never occur in practice, at least if a failure of the product on these constellations will not cause much harm.

A Markov usage model (MUM) is based on a *Markov chain* representation of the software usage. For putting it up, it is necessary to identify (1) *states* of the software, (2) possible *transitions* between these states (during these transitions, processing takes place), and (3) *probabilities* (or: relative frequencies) for the possible transitions. It is convenient to use a directed graph for the representation of the MUM; therein, the nodes are the states, the arcs are the transitions, and the transition probabilities are indicated as labels assigned to the arcs.

Related, but not quite identical to Markov usage models in the sense above is another class of probabilistic, graph-based processing models for software the first of which have already been developed three decades ago (see Littlewood [20], [21], [22], Cheung [5]); more recent articles concerning this other type of models are Siegrist [28] or Rajgopal and Mazumdar [24]: In these models, the nodes of the graph do not represent states of the processing, but components (modules etc.) of the software product, and the arcs do not represent processing steps, but control transfer between the components. In so far as they assume a Markovian (history-independent) probability structure, these models are often simply called *Markov models*; to distinguish them from Markov usage models in the sense defined above, we call them *component-based Markov models* in this paper. On a non-formal level, the main difference is that component-based Markov models already exploit "whitebox" knowledge of the software developer for putting up the model graph, whereas MUMs rather start with the "black-box" viewpoint of the software user: The developer knows the module structure of the given program and can estimate transfer probabilities between modules (knowledge on the actual use of the program is nevertheless necessary for that); the user, on the other hand, identifies states, e.g., certain input windows, menus, intermediate or final outputs etc., and can estimate probabilities of transitions between such states in her/his application environment. The present paper is based on the MUM modelling type, but we shall enrich the MUMs by some additional information accessible only to the developer.

The aim of the paper is to automatically derive a suitable set of *test paths* through the MUM graph, which may be elaborated in a second step to complete test cases for the software product under consideration. Similar problems have already been investigated in the literature, mainly in the context of protocol conformance testing or user interface testing (see. e.g., Belli and Grosspietsch [2], Aho et al. [1], Kumar and Venkataram [19], Shehady and Siewiorek [27], Kim et al. [18], Csöndes et al. [6]). While these approaches strive for a satisfaction of pre-defined coverage criteria (usually: arc coverage)[3], we do not force complete coverage (or coverage on a pre-defined level) in the present paper. Instead, we consider the tradeoff between coverage and testing costs and try to find an optimal compromise between both. For determining this solution, the uncertain knowledge the developer has on possible failures and losses in case of failure needs to be quantified, and the MUM must be enriched by this information.

We do not deal here with the question of how the selected test paths are extended to full test cases. In some cases of application, this may be a trivial problem, in other cases it can be highly nontrivial. Some general approaches to this topic have been outlined in the MUM literature, but we must treat this aspect as beyond the scope of the present paper.

[3] An exception is [6], where, among others, the problem of maximizing coverage subject to cost restrictions is treated. Contrary to our approach, however, information on program usage does not enter into the problem formulation.

A Markov usage model intends to represent the *operational distribution* of program inputs. For purposes of reliability estimation, it would be desirable to use this distribution also for testing, or to test with a distribution changed in a controlled way, as suggested in [12], [13], [7] and [8]. However, as argued by Rivers and Vouk in [25], resource limits often force the developer to give up this aim and to resort to non-operational testing. Our approach proposes a kind of compromise for such situations: while dropping the goal of making the derivation of reliability estimates from the test outcomes possible, we nevertheless try to exploit as much information about the operational use of the program when deriving test cases, including information on processing steps that are "critical" with respect to failure probability or to damage in case of failure.

## 2 The Model

As input we take a MUM represented by a strongly connected directed graph $G = (V,\ A)$ without multiple arcs, where a specific node (labeled by the number 1) is distinguished as the start node. A MUM with unique start node and unique end node, as considered in [12]–[13], can be reduced to this model by adding an arc from the end node to the start node, representing a new call of the program. The transition probabilities $p(e)$ assigned to the arcs $e \in A$ are assumed as strictly positive. Furthermore, we suppose that to each arc $e$, the following estimates are assigned: the failure probability $f(e)$, the loss $l(e)$ caused in case of failure, and the cost $c(e)$ for testing $e$. Note that since there are no multiple arcs, each arc $e$ can be identified with the ordered pair $(i, j)$ of its incident nodes. If $e = (i, j)$, we shall also write $p(i, j)$, $f(i, j)$, $l(i, j)$ and $c(i, j)$ instead of $p(e)$, $f(e)$, $l(e)$ and $c(e)$, respectively.

*Remark.* Let us emphasize that in our model, processing steps of the program are assigned to *arcs* and not to nodes of the graph. This convention is different from that used in component-based Markov models (see our remarks in the Introduction). For example, in our formalism, node $i = 2$ can refer to the state that a specific menu, from which the user is to select one of the program functions `fun1`, `fun2` or `fun3`, is presented to the user. According to the actual choice by the user, the program starts processing the selected function and gives an output to the user; node $i = 3$, $i = 4$ and $i = 5$ refer to the state that the output of function `fun1`, `fun2` and `fun3`, respectively, is presented to the user. In this way, the processing of function `fun1`, `fun2` and `fun3` corresponds to arc $(2, 3)$, $(2, 4)$ and $(2, 5)$, respectively. In particular, the chosen representation implies that estimates on failure probabilities and losses in case of failure have to be assigned, as indicated above, to the arcs $e \in A$ and not to the nodes $i \in V$. Since in our model, the states do not refer to processing activities, also sojourn times etc. are not essential in our case.

For modelling failures, we use the *static* model described in [13], i.e., we assume that a failing arc always fails, and that the resulting loss occurs only one time, no matter how often the arc is (or would be) traversed. Our technique can also be extended to the *dynamic* model of [13]. A discussion of the two models from the viewpoint of application can be found in [13].

We define a test sequence as a closed walk $w = (e_1, \ldots, e_m)$, $e_k \in A$ ($k = 1, \ldots, m$), containing the start node 1. Arcs $e$ may occur more than once in $w$. The cost for testing $w = (e_1, \ldots, e_m)$ is

$$C(w) = \sum_{k=1}^{m} c(e_k). \tag{1}$$

The assumption that $w$ is closed allows a decomposition of $w$ in a series of single test paths from the start node 1 to an "end" node, i.e., a predecessor node of 1 (a node where a new program call is necessary to continue the test).

After test, tested arcs are considered as correct (not failing). Of course, this is a simplification, since other program inputs than those actually chosen for the test of a given arc could have caused this arc to fail. The simplification can be justified by the consideration that not testing an arc at all leaves a failure probability of a higher order of magnitude than testing it with just one specific input combination. It should be mentioned that the well-known coverage measures for whitebox testing use the same simplification.

Whether an arc is failing or not is uncertain before test. For modelling this uncertainty, we use the probabilistic approach. Let us emphasize, however, that our technique can also be adapted without difficulties to other uncertainty measures, such as fuzzy values or plausibility or credibility values in the sense of the Dempster-Shafer theory.

As an indicator variable for the stochastic event that arc $e$ fails in case it is traversed, we introduce $I_f(e) = 1$ if $e$ is failing, given that $e$ is traversed, and $I_f(e) = 0$ otherwise. Evidently, the expected value of the random variable $I_f(e)$ is just $f(e)$.

In a similar way, we introduce an indicator variable for the traversal of arc $e$. When defining this variable as a random variable, it seems appropriate to refer to the *steady state* of the operational usage of the program, i.e., the distribution of traversals of arcs as it is reached after some time of use of the program, when "starting effects" have already become insignificant. Therefore, we set $I_r(e) = 1$ if a random transition in steady state passes through $e$, and $I_r(e) = 0$ otherwise.

The random variables $I_r(e)$ and $I_f(e)$ are independent by definition, since we have defined the event $I_f(e) = 1$ as the event that $e$ fails, conditional on the event that $e$ is traversed, so whether $e$ is actually traversed or not has no influence on whether $I_f(e) = 1$ or not.

Let $\bar{A} \subseteq A$ be the set of *not* tested arcs. According to our assumption, only the arcs in $\bar{A}$ can cause losses by failure. Denoting the mathematical expectation by $E$, the expected loss by failure for one steady-state transition can then be calculated as follows:

$$L(\bar{A}) = E\left(\sum_{e \in \bar{A}} I_r(e) \cdot I_f(e) \cdot l(e)\right) = \sum_{e \in \bar{A}} E(I_r(e) \cdot I_f(e)) \cdot l(e).$$

Using independence of $I_f(e)$ and $I_r(e)$ and the fact that the expectation of a product of independent random variables is the product of the expectations, we further obtain

$$L(\bar{A}) = \sum_{e \in \bar{A}} E(I_r(e)) \cdot f(e) \cdot l(e). \tag{2}$$

For abbreviation, let us set $q(e) = q(i,j) = E(I_r(e))$ for the probability that in steady state, arc $e = (i,j)$ is traversed. These probabilities can be computed as follows: Let $x_i$ denote the probability with which state (node) $i$ occurs in the steady-state distribution. Evidently, $q(i,j) = x_i \cdot p(i,j)$. The probabilities $x_i$ result by the solution of the system

$$(P^t - I)(x_1, \ldots, x_n)^t = (0, \ldots, 0)^t, \quad (1, \ldots, 1)(x_1, \ldots, x_n)^t = 1 \tag{3}$$

of linear equations, where $P = (p(i,j))$ is the matrix of transition probabilities, $I$ is the identity matrix, and $n$ is the number of states (nodes).

By using a sufficiently long test sequence $w$, it is of course possible to test *all* arcs, such that $\bar{A} = \emptyset$ and hence $L(\bar{A}) = 0$. However, there is a tradeoff between the achieved coverage and the testing costs. Let $\bar{A}(w)$ denote the set of arcs not covered by the walk $w$. Then we may determine the best point of this tradeoff by solving the following optimization problem:

**Problem 1.** Find a test sequence $w$, such that $\gamma L(\bar{A}(w)) + C(w) = \min$. Therein, $\gamma$ is a factor indicating how high the program developer values an average loss of one currency unit per transition in operational usage as his/her own loss. $\gamma$ depends on several factors, e.g., contracts, number of customers, market reputation effects etc. An alternate problem formulation where $\gamma$ needs not to be estimated is the following:

**Problem 2.** Find a test sequence $w$, such that $L(\bar{A}(w)) = \min$ under the constraint $C(w) \leq B$. Here, $B$ is a fixed given budget limit for testing.

**Proposition.** Both Problem 1 and 2 are NP-hard combinatorial optimization problems.

**Proof.** By choosing specific values for $f(e)$ or $l(e)$, the *Rural Postman Problem* (RPP), which is known to be NP-hard, can be reduced to Problem 1 or Problem 2. This can be seen as follows: The RPP consists in finding a shortest closed walk $w$ in a graph $G = (V, A)$ with the property that $w$ contains each arc $e \in A_0$, where $A_0$ is a given subset of $A$, at least once. Now define $f(e) = 1$ for all $e \in A_0$ and $f(e) = 0$ for all $e \notin A_0$, and choose $\gamma$ very high. Then a solution of Problem 1 solves the given RPP. In a similar way, the RPP can be reduced to Problem 2. Therefore, Problems 1 and 2 are NP-hard as well. □

The property stated in Proposition is an indicator that for larger problem instances, *heuristic* solution approaches will be required.

*Remark.* We want to emphasize that the problem of obtaining *complete* coverage, as investigated in [19] and in a series of related articles, is contained in our problem 1 as the special case where the constant $\gamma$ is very large and the values $f(e)$ and $l$ strictly positive, such that an optimal solution must be a solution where all arcs are covered.

## 3 Heuristic Solution by ACO

One can easily design a straightforward greedy heuristic for solving Problem 1 or 2. Nevertheless, already simple examples show that greedy-type heuristics can produce very poor solutions at our problems. So we apply instead a metaheuristic technique, *Ant Colony Optimization* (ACO), for which some types of convergence behavior towards the optimal solution have been shown ([14]–[15]), to our problems. The basic ideas of ACO have been developed by Dorigo, Maniezzo and Colorni [10] at the beginning of the 90ies; a formulation as a flexible metaheuristic applicable to the overall range of combinatorial optimization is given by Dorigo and Di Caro [9]. Compared with other possible metaheuristics like genetic algorithms, simulated annealing or tabu search, the application of ACO to our problems has the advantage that the graph structure of the problems fits very well to the graph-oriented "philosophy" of ACO (see [14]).

Roughly speaking, we can consider an ACO implementation as a learning system with a fixed number of cooperative agents, implemented on the same or on different processors. Each agent constructs solutions for the given problem, governed (1) by randomness, (2) by a greedy principle, and (3) by a blackboard mechanism representing the "collective memory" of the agent community. Learning takes place via this blackboard which contains *trail levels* used by all agents for the computation of the probabilities for performing possible partial construction steps. Agents that have found the best solution up to now, or the best solution generated within a certain construction period, or, in the so-called rank-based variant (see [4]), one of the $r$ best solutions generated within the period, are allowed to "write" on the blackboard, which means that the trail levels for the partial construction steps they applied in their solution are increased. For details of the general algorithm, we refer the reader to the literature cited above.

In our computational experiments, we applied three different variants; in each of them, trail levels are assigned to the arcs of $G$, and a construction step is an extension of the current partial walk $w^{(k)} = (e_1, \ldots, e_k)$ to the partial walk $w^{(k+1)} = (e_1, \ldots, e_k, e_{k+1})$, where $e_{k+1}$ is selected from the set of arcs starting at the end node of $e_k$. (Construction begins by choosing an arc $e_1$ starting at the start node 1.) In the first variant ("Standard"), the probability of an arc of being appended is chosen proportional to the trail level. In the second variant ("Variant A"), the probability that a particular arc $e$ is appended is chosen proportional to $\frac{\tau(e)}{n(e)+1}$, where $\tau(e)$ is the trail level of $e$, and $n(e)$ counts the total number how often $e$ has been traversed in all previous construction rounds in any walk constructed by any of the agents. Thus, arcs, that have already been used frequently get a lower probability of being considered in the future, which favors diversification of the walks.

In the third variant ("Variant B"), we apply the same selection mechanism as in the first (Standard) variant, but the trail level update mechanism is changed: Only an arc that lies on the best walk found so far with a minimum number of traversals in this best walk gets a trail level increment. Also this variant favors diversification.

## 4 Experimental Results

In this section we apply our approach to a medium-size real world system representing an outbound call center which is used by the Austrian Red Cross. We give a brief overview of the system and present numerical results of our analysis. The considered MUM consists of 47 nodes and 85 arcs and reflects three main menues and submenues including the functions for the *customer processing*, the *supervisor functions* and the functions for the *campaign management*. To increase transparency, we have partitioned the entire MUM description into a set of 4 diagrams–the *main section* and three subsections. Entry node and exit node of a subsection are represented in the *main section* and are repeated in the subsection.

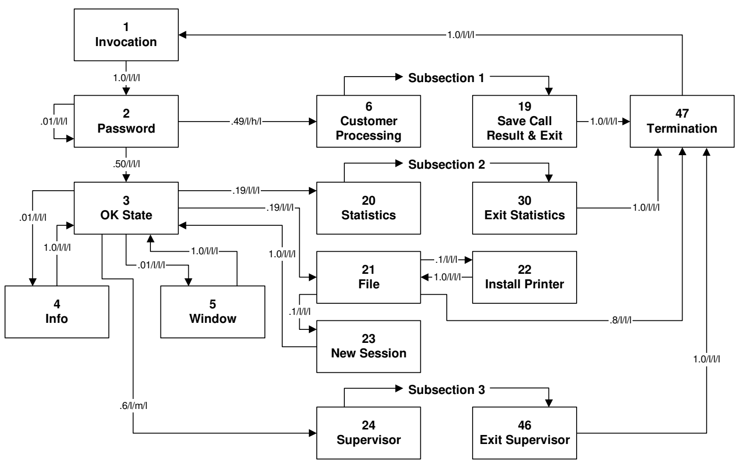


**Fig. 1.** Call Center Application – Main Section

The main functionality is *customer processing* (node 6 and the subsequent part of the graph given in Fig. 2). 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). The system provides also further information about the customer: information on the last phone calls with the customer (node 8) and the complete chronology of interactions with him/her (node 9). 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 a new e-mail address. If there are no changes the operator can go directly to a function where he can store the call results (node 15).

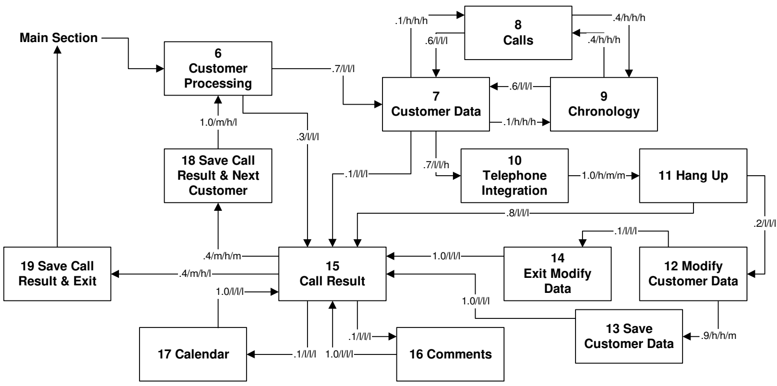


**Fig. 2.** Call Center Application – Customer Processing

In the subsection *statistics* , several performance statistics can be computed and printed or exported to an accounting program.

In the subsection *supervisor* (see Fig. 3), the three main functionalities *check campaign results* (node 31), *distribute campaigns* (node 36) and *maintain campaigns* (node 39) are implemented. One important task of the supervisor is to control the work of the operators. For this purpose he/she uses the functionality *check campaign results.* By using the functionality *send e-mail* the checked results can be distributed (node 34). 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.

Our description in Fig. 1–Fig. 4 also includes the operational probability $p(i,j)$, an estimation of the failure probability $f(i,j)$ and of the loss in case of failure $l(i,j)$, and also the cost $c(i,j)$ for testing the function $(i,j)$. These values are integrated into the MUM description. Each arc $(i,j)$ is labelled with the following estimates: $p(i,j) \,/\, f(i,j) \,/\, l(i,j) \,/\, c(i,j)$. Therein, the constants $l$ (low), $m$ (medium), and $h$ (high) have the following values: For $f(i,j)$: $l = 0.001, m = 0.01, h = 0.1$. For $l(i,j)$: $l = 1, m = 10, h = 100$. For $c(i,j)$: $l = 1, m = 10, h = 1000$. Failure probabilities, losses in case of failure and testing costs have been estimated by "educated guesses". A high failure probability 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. The costs for

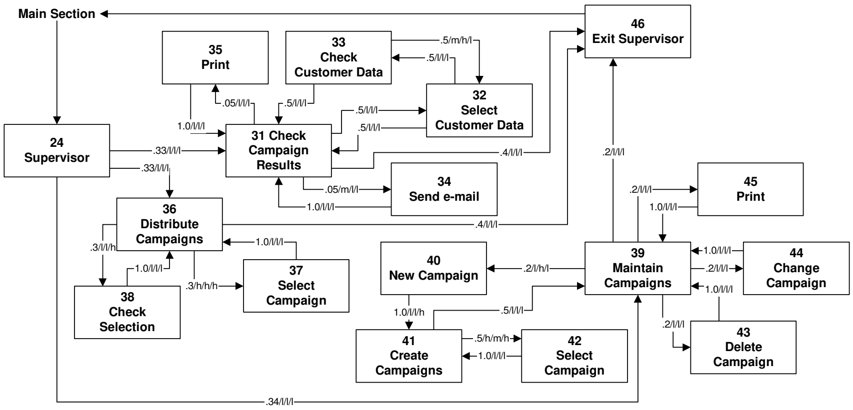


**Fig. 3.** Call Center Application – Supervisor Functions

testing these functions are high as well, because the data are extracted from a legacy system. To test this system an additional expert has to join the test team. Also the testing costs for testing function (7,10) are high: To verify the test results of function (7,10), a call center application expert is required.

The problem described above was used to test the three different variants of our ACO algorithm. To evaluate the performance of our novel approach, we compared the results with the results of a greedy heuristic. Our greedy heuristic works as follows: we chose the next not visited arc which has the maximum value according to the decision rule $(q(e) \cdot f(e) \cdot l(e))/(d(e)+1))$, where $d(e)$ denotes the distance from the current position to arc $e$. By using this heuristic we have to define the coverage rate of the arcs to be tested. For a fair comparison we use about the same coverage rate of good solutions found by the ACO algorithm. The solution quality of our greedy heuristic is 1045.45 cost units.

To show the performance of the ACO algorithm, we will also show the performance of the randomized heuristic information - to so called attractiveness - used in the ACO algorithm, which we will refer to as “Heuristic” in the figures. We performed six runs with each of the three algorithms. The plots presented in Fig. 4–5 show best solutions values found after a certain number of iterations, averaged over these six runs. In Fig. 4, the results of the greedy heuristic and our first (Standard) ACO variant are shown. It can be seen that the greedy heuristic is not able to find low-cost solutions: the best solution value stagnates at about 1000 cost units. Standard ACO finds good solutions, but only after about 35 iterations.

Fig. 5 shows the results of the ACO modifications, Variant A and B. We see that, compared to Standard ACO, relatively good solutions are obtained already after a smaller number of iterations. This makes Variants A and B particularly interesting for large applications, where it cannot be expected that computation time suffices for driving the optimization process to the point where optimal solutions are reached.

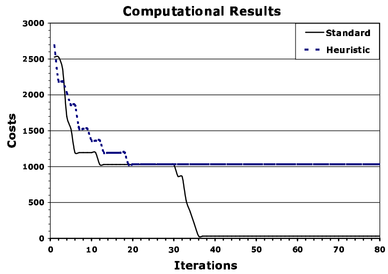


**Fig. 4.** Computational Results – Heuristic, Standard ACO

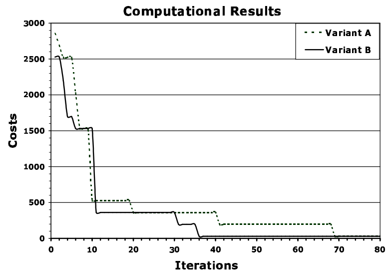


**Fig. 5.** Computational Results – Variant A, Variant B

All the three ACO algorithms find very reasonable solutions after 70 iterations. Of course, the trends of the presented results will have to be verified by more extensive experiments in the future.

## 5 Conclusion

We have proposed a technique for obtaining appropriate software test sequences from a Markov Usage Model description of the expected use of a software product. Our approach addresses the tradeoff between testing costs and failure risk caused by untested processing steps (state transitions). For the search for good solutions, we have applied a metaheuristic, Ant Colony Optimization. In this respect, our work is in the spirit of the currently emerging paradigm of "Search-Based Software Engineering", as it has been formulated by Harman and Jones [16] and propagated in a series of recent initiatives. As an additional input information completing the Markov Usage Model description, we use estimates of failure probabilities and of losses in case of failure. This gives the considered problem the character of a decision problem under uncertainty. Our technique has been applied to a medium-size real-world problem. The experimental results show indicate that our ACO approach outperforms a greedy-type solution heuristic. Two variants of the standard ACO technique have been specifically designed for the problem under consideration; they produce still more promising results than standard ACO. Nevertheless, much more experimental work will be required in the future as a prerequisite for the development of suitable toolkits supporting software testing along the lines of the described approach on an industrial level.

Practical application of our technique requires that the parameters values $p(e)$, $f(e)$, $l(e)$ and $c(e)$ are estimated. Sensitivity of the optimization results with respect to these estimates is an important topic that deserves further investigation. Our first impression is that the results are relatively robust with respect to all parameter types except the transition probabilities $p(e)$. On the other hand, just these probabilities can be estimated in a rather precise, objective way by using an instrumentation of the program code and measuring the transfer frequencies in a test with similar usage profile as that expected in the application field.

Our approach takes Markov usage models of any type as inputs and can therefore be used in a broad range of applications. Some recent articles indicate that the application of graph–based models for GUI (graphical user interface) testing is an area of growing importance, see, e.g., Kallepalli [17] or Belli [3]. Taking account of the specific requirements of GUI testing in the context of our problem formulation seems to be an interesting topic for future research. Another topic of research would be an extension to semi-Markov usage models for applications where the Markov assumption (claiming that state transition probabilities are history-independent) seems not adequate.

**Acknowledgement.** The authors are indebted to Fevzi Belli for drawing their attention to the optimization approaches in protocol conformance testing and for some helpful disussions.

## References


1. Aho, A. V., Dahbura, A. T., Lee, D., Uyar, M. Ü, "An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours", *IEEE Trans. on Communications* 39 (1991), pp. 1604–1615.
2. Belli, F., Grosspietsch, K.-E., "Specification of fault-tolerant system issues by predicate/transition nets and regular expressions – approach and case study", *IEEE Trans. Software Eng.* 17 (no. 6) (1991), pp. 513–526.
3. Belli, F., "Finite-state testing and analysis of graphical user interfaces", *Proc. 12th Int. Symp. on Software Reliability Engineering (ISSRE)*, IEEE CS Press (2001), pp. 34–43.
4. Bullnheimer, B., Hartl, R. F., Strauss, C., "A new rank-based version of the Ant System: A computational study", *Central European Journal for Operations Research* 7(1) (1999), pp. 25–38.
5. Cheung, R. C., "A user-oriented software reliability model", *IEEE Trans. Software Eng.*, vol. 6 (1980), pp. 118–125.
6. Csöndes, T., Kotnyek, B., Szabó, J. Z., "Application of heuristice methods for conformance test selection", *European J. of Operational Research*, vol. 142 (2002), pp. 203–218.
7. Doerner, K., Gutjahr, W. J., "Representation and Optimization of Software Usage Models with Non–Markovian State Transitions", *Information and Software Technology* vol. 47 (2000), pp. 873–884.
8. Doerner, K., Laure, E., "High performance computing in the optimization of test plans", *Optimization and Engineering*, vol. 3 (2002), pp. 67–87.
9. Dorigo, M., Di Caro, G., "The Ant Colony Optimization metaheuristic", in: *New Ideas in Optimization*, D. Corne, M. Dorigo, F. Glover (eds.), pp. 11–32, McGraw–Hill (1999).
10. Dorigo, M., Maniezzo, V., Colorni, A., "The Ant System: An autocatalytic optimization process", Technical Report 91–016, Dept. of Electronics, Politecnico di Milano, Italy (1991).

11. Dulz, W., Zhen, F., "How to use XML descriptions for generating Markoiv chain usage models from sequence diagrams", Tech. Rep. 06/02, Dept. of Comp. Sci. 7, University of Erlangen, Germany.
12. Gutjahr, W. J., "Importance sampling of test cases in Markovian software usage models", *Probability in the Engineering and Informational Sciences*, vol. 11, (1997), pp. 19–26.
13. Gutjahr, W. J., "Software Dependability Evaluation Based on Markov Usage Models", *Performance Evaluation*, vol. 40 (2000), pp. 199–222.
14. Gutjahr, W. J., "A graph–based Ant System and its convergence", *Future Generation Computing Systems* 16 (2000), pp. 873–888.
15. Gutjahr, W. J., "ACO Algorithms with Guaranteed Convergence to the Optimal Solution", accepted for publication in: *Information Processing Letters*.
16. Harman, M., Jones, B. F., "Search-based software engineering", *Information and Software Technology*, vol. 43 (2001), pp. 833–839.
17. Kallepalli, Ch., "Measuring and modeling usage and reliability for statistical Web testing", *IEEE Trans. Software Eng.* vol. 27 (no. 11) (2001), pp. 1023–1036.
18. Kim, Y. G., Hong, H. S., Bae, D. H., Cha, S. D., "Test cases generation from UML state diagrams", *IEE Proc.-Softw.* 146, no. 4 (1999), pp. 187–192.
19. Kumar, G. P., Venkataram, P., "Protocol test sequence generation using MUIOS based on TSP Problem", *Proc. IFIP TC6 Conf.* (1994), pp. 165–191.
20. Littlewood, B., "A reliability model for systems with Markov structure", *J. Royal Statistical Soc., Series C (Applied Statistics)*, vol. 24 (1975), pp. 172–177.
21. Littlewood, B., "A Semi-Markov model for software reliability fith failure costs". In: *MRI Symp. Computer Software Engineering*, Polytechnic Press, Polytechnic of New York, New York (1976), pp. 281–300.
22. Littlewood, B., "Software Reliability Models for modular program structure", *IEEE Trans. Reliab.*, vol. 28 (no. 3) (1979), pp. 241–246.
23. 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).
24. Rajgopal, J., Mazumdar, M., "Modular operational test plans for inferences on software reliability based on a Markov model", *IEEE Trans. Software Eng.*, vol. 28 (no. 4) (2002), pp. 358–363.
25. Rivers, A. T., Vouk, A. M., "Resource-constrained non-operational testing of software", *Proc. 9th Int. Symp. on Software Reliability Engineering (ISSRE)*, pp. 154–163 (1998).
26. Sayre, K., Poore, J. H., "Partition testing with usage models", *Information and software technology*, vol. 42 (2000), pp. 845–850.
27. Shehady, R. K., Siewiorek, D. P., "A method to automate user interface testing using variable finite state machines", *Proc. FTCS 27* (1997), pp. 80–88.
28. Siegrist, K., "Reliability of systems with Markov transfers of control", *IEEE Trans. Software Eng.*, vol. 14 (no. 7) (1988), pp. 1049–1053.
29. Walton, G. H., Poore, J. H., Trammell, J., "Statistical testing of software based on a usage model", *Software – Practice and Experience*, vol. 25 (1) (1995), pp. 97–108.
30. Whittaker, J. A., "Markov chain techniques for software testing and reliability analysis", Ph. D. dissertation, University of Tennessee (1992).
31. Whittaker, J. A., Poore, J. H., "Markov analysis of software specifications", *ACM Trans. on Software Eng. and Method.*, vol. 2 (1) (1993), pp. 93–106.
32. Whittaker, J. A., Thomason, M. G., "A Markov chain model for statistical software testing", *IEEE Trans. Software Eng.*, vol. SE-20 (1994), pp. 812–824.