Technical Report soft-08-04, Chair for Software Engineering, University of Konstanz

Debugging of Dependability Models Using Interactive Visualization of Counterexamples

Husain Aljazzar
Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
Email: Husain.Aljazzar@uni-konstanz.de

Stefan Leue
Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
Email: Stefan.Leue@uni-konstanz.de

***Abstract*—We present an approach to support the debugging of stochastic system models using interactive visualization. The goal of this work is to facilitate the identification of causal factors in the potentially very large sets of execution traces that form counterexamples in stochastic model checking. The visualization is interactive and allows the user to focus on the most meaningful aspects of a counterexample. We present the application of the visualization method as implemented in our prototype tool** DIPRO **to two significant case studies.**

I. INTRODUCTION

The success of model checking [1], [2] as an analysis technique in various areas of system design is founded in the automated nature of the state space exploration algorithms that this technique uses. Model checking tools are increasingly often used as debugging aids, not at last since model checkers for functional properties return diagnostic information describing a system execution from the initial state into a property violating state. Such an offending execution path is also referred to as a counterexample. Once a counterexample is available it is necessary to determine causal factors for the reachability of a property violating system state. This helps in identifying faults in the system design, and in debugging the model.

In the realm of stochastic model checking [3], [4] counterexamples are not as readily available as in functional model checking. Classical stochastic model checkers do not return counterexamples. Additionally, the notion of a counterexample in stochastic model checking is not that of a single execution path, but that of a set of paths from the initial system state into offending system states. For certain classes of dependability properties explicit state space search techniques have been used to compute such counterexamples in the stochastic setting. The fact that the counterexample consists of a potentially very large set of traces further complicates debugging. This problem is alleviated by the fact that the counterexample search methods construct the counterexample from those offending system traces that carry the most probability mass.

In this paper we present an approach to support the analysis and debugging of stochastic system models based on the interactive visualization of counterexamples for probabilistic reachability properties. The visualization is aiming at facilitating the determination of causal factors for property violations. We follow an interactive visualization approach in which the user can select portions of the state space that s/he considers important, while filtering out others. The visualization focuses on the execution traces belonging to the counterexample and brings out salient stochastic properties of the model, in particular the probability mass of system execution traces. We combine the presentation of variable valuations with the presentation of system traces so that different views on potential causal factors can be obtained. Finally, the visualization is dynamic and adds states and transitions as they are produced by the search algorithm. This aids in gaining a better understanding of the system, since the order in which transitions are added to the counterexample are indicative of their importance.

Precursory and Related Work: In precursory work we have devised heuristics guided [5] counterexample search methods for discrete-time and continuous-time Markov Chains (DTMCs/CTMCs) [6], [7]. An alternative approach to this problem based on k-shortest-paths search [8] has been proposed in [9], [10]. This approach provides more precise results compared to our approach presented in [6], [7]. The fact that this approach performs an exhaustive search on the complete state space dramatically constraints its practical applicability to models of realistic size. We proposed an on-the-fly algorithm called K* for solving the k-shortest paths in [11]. In this paper we briefly explain K* and its application to the counterexample generation for DTMCs and CTMCs. We then visualize the thus obtained counterexamples.

There is some work on analyzing counterexamples for the purpose of system debugging [12], [13] for functional model checking, relying on comparing property violating and property observing execution sequences. However, at the time of writing we are not aware of any other approach towards fault localization for counterexamples using interactive visualization techniques, or of any work suggesting counterexample analysis for stochastic system models.

The work in [14] presents an interactive 3D-visualization of the state space of a concurrent software model. The goal of this work is the visual discovery of structural properties of the state space, such as size, symmetries, connectedness of states or strongly connected components of states. It is not obvious how this approach can be applied to counterexample analysis. Our approach is based on a 2D visualization since we believe that 3D-visualizations are problematic due to the hiding of portions of the state space that is inevitable in 3D graph models.

The algorithm visualization work described in [15], [16] bears some similarity with our work in that they also reconcile algorithm animation and software visualization. While their objective is to enhance the understanding of algorithms, our objective is to better understand causal factors for property violations in stochastic system models.

Structure of the Paper: We introduce the foundations of stochastic models and its model checking in Section II. This section also sketches the K* algorithm and its application to counterexample search in stochastic model checking. The visualization techniques that we propose are explained in Section III. In Section IV we present our visualization tool DiPro and discuss the application of our approach to two case studies.

II. Stochastic Model Checking

A. Markov Chains

System dependability and performance models are often represented by variants of Markov chains. They describes the system behavior as a stochastic process in which system transitions are labeled with probability and time consumption values. A popular type of such dependability models are *discrete-time Markov chains* (DTMCs). DTMCs can be considered as probabilistic transition systems. In DTMCs, time is modeled in a discrete way by assuming that the system fires exactly one transition at every discrete time tick. Each transition is labeled with a numerical value that is interpreted as indicating the probability of firing this transition as the next step of the system if the system is in the origin state of the transition. Formally, a DTMC is defined as follows:

Definition 1: A labeled **discrete-time Markov chain (DTMC)** $\mathcal{D}$ is a quadruple $(\mathcal{S}, \hat{s}, \mathcal{P}, \mathcal{L})$, where $\mathcal{S}$ is a set of states, $\hat{s} \in \mathcal{S}$ is the initial state, $\mathcal{P} : \mathcal{S} \times \mathcal{S} \longrightarrow [0,1]$ is a transition probability matrix, satisfying that for each state s, $\sum_{s' \in S} \mathcal{P}(s, s') = 1$, and $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathsf{AP}}$ is a labeling function, which assigns to each state a subset of the set of atomic propositions AP. For any state s, we interpret $\mathcal{L}(s)$ as the set of valid propositions in that state.

For each pair of states s and s', $\mathcal{P}(s, s')$ gives the probability to move from s to s'. A move from s to s' is possible if and only if $\mathcal{P}(s, s') > 0$ holds. In this case we call (s, s') a *transition*.

Because of their conceptual simplicity, DTMCs are widely used in the modeling and analysis of stochastic systems, based on a discrete time abstraction. If a more realistic dense modeling of the time passing in states is required, then *continuous-time Markov chains (CTMCs)* can be used. While each transition of a DTMC corresponds to a discrete time step, in a CTMC transitions occur in dense real time. In a CTMC, each transition is labeled by a number that is interpreted to represent the probability of a transition from some state s to some state s' within t time units after being enabled. The duration t is determined by a random variable which follows a negative exponential distribution with the rate of the transition from s to s' as a parameter. A CTMC is defined as follows:

Definition 2: A labeled **continuous-time Markov chain (CTMC)** $\mathcal{C}$ is a quadruple $(\mathcal{S}, \hat{s}, \mathcal{R}, \mathcal{L})$, where $\mathcal{S}$ is a set of states, $\hat{s} \in \mathcal{S}$ is the initial state, $\mathcal{R} : \mathcal{S} \times \mathcal{S} \longrightarrow \mathbb{R}_{\geq 0}$ is a transition rate matrix and $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathsf{AP}}$ is a labeling function as in Definition 1.

The transition probability matrix $\mathcal{P}$, which we defined for DTMCs, is replaced by a transition rate matrix $\mathcal{R}$. For any pair of states s and s', $\mathcal{R}(s, s')$ is the time rate for moving from s to s'. A move from s to s' is possible if and only if $\mathcal{R}(s, s') > 0$. In this case we call (s, s') a *transition*. In this paper, we use the notion of a *Markov chain* as a generic term referring to either a DTMC or a CTMC.

A path in a Markov chain is intuitively a concrete execution of the system, i.e., it is encompassing a sequence of state transitions. Since systems considered in the domain of stochastic model checking are usually reactive systems, paths are assumed to be infinite. We often need to refer to finite path prefixes. We use the term *finite path*, or simply *path*, to refer to a path prefix and the term *infinite path* to denote a full path. The probability of paths in Markov chains is measurable using suitable probability measures for both DTMCs and CTMCs[17], [4]. We denote the measures for both DTMCs and CTMCs as Pr.

CTMCs and DTMCs can be modeled in the modeling language PRISM, supported by the identically named stochastic model checker [3]. The PRISM language is a state-based stochastic modeling language based on the Reactive Modules formalism presented in [18]. A PRISM model is composed of a number of modules which can interact with each other. A module contains a number of local variables. The values of all local variables as well as the points of control of some module define its current local state. The global state of the PRISM model is determined by the local states of all modules. The behaviour of each module is described by a set of command of the form

$$\begin{array}{lcl} [action]\ guard & -> & p_1 : update_1 \\ & + & ... \\ & + & p_n : update_n; \end{array}$$

in which "action" is an optional name of the command. The action name has a meaning in action synchronization, and we use action names in our visualization of counterexamples to denote observable events. The guard is a predicate over all the variables in the model. Each update describes a transition of the model that can be taken in case the guard evaluates to true. Each update is assigned a probability p_i in the case of a DTMCS, and a rate in the case of a CTMC. Depending on the model type the state space of a PRISM model is either a DTMC or a CTMC.

B. Probabilistic Reachability

In order to verify a dependability property using stochastic model checking the property has to be formulated as a formula of a stochastic temporal logic, such as PCTL [17] in the discrete-time case or CSL [19], [4] in the continuous-time case. For a given model and a given PCTL or CSL captured property a stochastic model checker can automatically verify whether the model satisfies the property or not. As we have shown in precursory work, for a limited class of properties called *probabilistic reachability* it is possible to efficiently compute counterexamples using explicit state space search on the respective Markov model. A probabilistic reachability property expresses that the probability of the system to reach a state where a given "undesired" condition φ holds does not exceed a given upper probability bound $\mathbf{p}$. One can add further constrains such as restricting the reachability to be satisfied within a given time bound $\mathbf{t}$. This kind of properties can be formulated in stochastic temporal logics by means of the *Until*-operator U. In PRISM such properties can be specified as

$$P \leq \mathbf{p}\ [true\ U\ \varphi] \quad \text{or} \quad P \leq \mathbf{p}\ [true\ U{\leq}\ \mathbf{t}\ \varphi],$$

or in a more general form as

$$P =?\ [\vartheta\ U\ \varphi] \quad \text{or} \quad P =?\ [\vartheta\ U{\leq}\ \mathbf{t}\ \varphi],$$

where ϑ denotes an arbitrary state formula. Note that in the second formulation "$\leq\ \mathbf{p}$" is replaced by "$=\ ?$". This causes PRISM to compute the total probability of the property and to deliver it as a result. We call a path which starts at the initial state $\hat{s}$ and satisfies the property $[\vartheta\ U\ \varphi]$ (or $[\vartheta\ U{\leq}\ \mathbf{t}\ \varphi]$) an *offending path*. The counterexample of a probabilistic reachability property is a set X of offending paths such that the accumulated probability of X violates the probability constraint "$\leq \mathbf{p}$". If the probability bound is not specified, i.e., we have "$=\ ?$" instead of "$\leq \mathbf{p}$", then we refer to any set of offending paths as a counterexample.

C. Generation of Counterexamples using K*

The algorithm K* finds k shortest paths in a given directed graph G for a start vertex and a set of target vertices. The following are the main design ideas for K*:

1) We apply the directed search algorithm A* [5] on G in order to determine a shortest path tree T on G.
2) All edges from G which A* explores are inserted into a special graph structure called *path graph* $\mathcal{P}(G)$. The details about the structure of $\mathcal{P}(G)$ are beyond the scope of this paper. For understanding our approach it is sufficient to know that $\mathcal{P}(G)$ is used to construct the k shortest paths in G. This is accomplished by a shortest path search on $\mathcal{P}(G)$, for example by using Dijkstra's algorithm.
3) Dijkstra's search on $\mathcal{P}(G)$ performs concurrently with A* on G. Consequently, Dijkstra will be able to deliver solution paths before G is completely searched by A*.

A* stores vertices on the search front in a priority queue open which is sorted by a *heuristic evaluation function* f. This function indicates the desirability of expanding a vertex. Vertices in the search queue open are called *open* vertices. open is sorted according to f. In each search iteration the head of open is removed from the queue and *expanded*. Vertices that have been expanded are called *closed* vertices and they are stored in the closed set, which is commonly implemented as a hash table. Notice that K* is correct in terms of delivering a shortest path tree only if the heuristic function used in A* is *monotone*.

In order to apply K* to the generation of counterexamples for a DTMC we use a probabilistic version of A* on the state transition graph of the DTMC. The result of K* is an enumeration of the most probable property violating paths. When the accumulated probability of the set X consisting of all found offending paths is sufficient to violate the probability bound, then X is provided as a counterexample. In order to analyze a CTMC we transform it into a DTMC using a uniformization step. Similarly to how it was suggested in [10], we ignore the self loops in the transformed DTMCs that are due to the uniformization step.

Each property violating path delivered in this way is a linear CTMC and its accurate probability can efficiently be computed using a stochastic model checker like PRISM. In our experiments monotone heuristics were not available for the problems we analyzed, hence we used K* with the trivial monotone heuristics $h(s) = 1$ for any state s. Note that in this case A* degenerates into Dijkstra's algorithm for determining shortest paths.

III. The Visualization Technique

Let $\mathcal{M}$ denote a Markov chain and let Φ denote a probabilistic reachability property. Our approach is to use interactive visualization techniques in order to facilitate debugging, which in this setting means analyzing a counterexample in order to determine the causality behind the violation of Φ. In principle we visualize the state space of $\mathcal{M}$ as a directed graph where states are represented by node icons and transition are represented by lines. The visualization has the following three objectives:

- First, we visualize the exploration of $\mathcal{M}$ using K*. The visualization component monitors the K* algorithm while it explores the state space of $\mathcal{M}$ on-the-fly. It displays changes caused by each search step immediately on the screen which helps in understanding how the search progresses through the state space The user can interact with the search algorithm thus enabling the type of simulation that we will explain in Section III-D.
- Second, our approach emphasizes the so far found portion of the counterexample using various highlighting mechanisms. This improves the readability of the counterexample and significantly increases the effectiveness of extracting information relevant for debugging.
- Third, we enable the use of interactive visual analytics functions that allow the user to selectively filter the displayed information, which helps to focus the visual analysis.

A. Drawing of States and Transitions

We use a circle to represent each state. We identify states that belong to the search front, i.e., states which are in the queue of the search algorithm, by depicting them as a rectangle. A hexagon represents the initial state. We also give visual indicators as to which states are going to be expanded next in order to assist the user in cases where the color differentiation between states in the search front is too subtle to be visually recognizable. We mark the first three states in open by drawing arrows of different width on the nodes.

Recall that K* employs A* to explore the state space of the Markov chain. A* uses an evaluation function f which estimates for each explored state s the probability of a potentially offending path which will be obtained by completing the current path leading to s. We use this fact in our visualization to bring out states having high f-values, i.e., states which very likely belong to offending paths having high probability. The highlighting is done by varying the colour intensity of nodes and edges and the line width of edges. This feature is intended to attract the user's attention to the critical behaviour of the system regarding the given safety property, while not discarding information belonging to the remainder of the behaviour.

The attribute variation factor used in emphasizing nodes and edges with high f-value is computed as follows. First, let f_{min} be the minimal f-value and f_{max} be the maximal f-value found so far during the exploration. Further, we extend the definition of f to cover transitions in the following way. For a transition $t = (s, s')$, we define $f(t)$ as the median of $f(s)$ and $f(s')$, i.e., $f(t) = \frac{f(s)+f(s')}{2}$. This extension relies on the fact that the relevance of a transition is derived from the relevance of its origin and destination states. Then we map the range $[f_{min}, f_{max}]$ to a given range of colours or line widths, respectively.

Colouring: We start with defining a colour scale of a particular number of colours C ranging from little emphasizing to strongly emphasizing colours. In DiPro, we created such colour scales using a tool developed in [20]. We map the range $[f_{min}, f_{max}]$ into the interval

$[0, 1]$ using the following monotonic function: $a : f \mapsto \frac{f - f_{min}}{f_{max} - f_{min}}$. Then, we determine the colour of the node representing a state s as follows: $colour(s) = a(f(s)) \cdot (C - 1)$. We use the same calculation to determine the colors of edges.

Edge and Line Width: In order to bring out the information which paths through the graph are more relevant than others we define the edge width to be proportional to the f-value of the corresponding transition. In other words, we render transitions with high f-values by thick lines, while transitions with small f-values are represented by thin lines. We map the range $[f_{min}, f_{max}]$ into the interval $]0, 1]$ using the following exponential grading function: $b : f \mapsto exp(a(f) - 1)$. Let W be the designated maximal edge width. Then, we determine the width of the edge representing a transition t by $width(t) = b(f(t)) \cdot W$. The use of the exponential function entails that the less important transitions are, the faster the corresponding edges become thinner. As a consequence the contrast between relevant and irrelevant transitions is extremely high so that irrelevant edges become almost imperceptible.

B. Layout Algorithm

In graph drawing, the arrangement of nodes and edges significantly influences the amount of clutter in the graph and hence determines its readability. Various graph layout algorithms have been proposed to optimize the graph layout in order to maximize graph readability, e.g. [21]. Since we are visualizing the on-the-fly state space exploration as it occurs we have to use an *incremental* layout algorithm. An incremental layout algorithm is an online algorithm that permits nodes and edges to be added at any time. It ensures a certain degree of stability in the layout of the already visible part of the graph. Without this stability the online visualization would be more confusing than helpful. The graph drawing library *yfiles* [22] used in our implementation provides four layout algorithms which are suitable for incremental visualization. In our tool we make all four algorithms available to the user.

C. Counterexample Highlighting

Once a counterexample is completely or partially found we surround each counterexample state with a red line and color the action name of each counterexample transition in red. This enables the user to quickely identify the offending behaviour of the system. We facilitate debugging by permitting a comparison of correct and offending behaviour. For instance, let n be a red node, i.e., a node representing a state which belongs to the counterexample. n must have a red successor n'. If n also has another successor n'' which is not red, i.e., n'' does not belong to the counterexample, then comparing both transitions (n, n') and $(n, n''$ can help determining causal factors for the property violation.

We also render the size of each node and action name proportional to the accummulated probability of all offending paths that have been found so far, including the corresponding state or transition. We increase the node size compared to the default size by

$$\alpha \cdot \frac{Pr(X')}{Pr(X)},$$

where X is the currently selected counterexample and $X' \subseteq X$ is the set of offending paths which include the considered state or transition. α is a factor which the user defines to control the amplification degree. Based on this scaling, states and transitions which contribute a large amount of probability mass to the property violation will appear very large on the screen. It then becomes much easier for the user to identify the actions which mostly contribute to the property violation.

D. Online Visualization

The on-the-fly nature of K* means that only a relatively small portion of the state space will be generated and be available for visualization. This greatly increases the practicability of our approach since it permits the visualization of much larger state spaces. The online visualization allows the user to interactively control the search algorithm. While watching the progress of the search, she or he can interactively halt the algorithm and modify the order of the search queue by selecting one or more states to be expanded next. It is also possible to execute the algorithm in step-wise exploration mode in which the user is free to manually determine the next state to be explored. These features makes our visualization technique very useful for interactive simulation purposes. The user can simulate the model in a stepwise fashion and alter its behaviour in order to detect model faults.

E. Visual Analytics

Our visualization approach is able to accommodate various visual analytics functions to analyse the information gathered during the search. This will help in detecting causal factors for the property violation. In our tool we provide a number of basic visualization functions:

- The user can hide nodes, edges or complete parts of the visible graph.
- The user can select nodes and edges.
- The user can query state information, such as the values of state variables or the effect of transition actions.
- The user can hide outgoing or incoming edges for selected states.
- The user can select paths through the state space which she/he is interested in and hide all of the model except for these paths. Optionally the model elements directly connected to the selected nodes can also be included in the selection.
- Selected paths can be analyzed and compared with each other by means of charts that can, for instance, display the development of the values of certain state variables along the selected paths.

The comparison of selected execution paths is a key tool in the analysis of counterexamples, as we shall illustrate in the following section.

IV. Experimental Validation

We have implemented a prototype tool for the visualization approach that we have described. The tool is called DiPro and is based on Java. DiPro calls K* in order to explore the state space of the PRISM models that we use as case studies. The state space is explored on-the-fly using the PRISM Simulation Engine [3]. For the drawing of the state space graph we mainly use the *yfiles* library [22]. We illustrate DiPro using two PRISM case studies.

A. Embedded Control System (ECS)

This case study models an embedded control system, closely based on the one presented in [23]. The system consists of a main processor (**M**), an input processor (**I**), an output processor (**O**), 3 sensors and two actuators. The input processor I reads data from the sensors and forwards it to M. Based on this data, M sends instructions to the output processor O which controls both actuators according to the received instructions. We analyze the failure behavior of the model. The possible failures are the following:

- Any of the three sensors can fail. This is modeled by the PRISM action *SensorFail*. The system is shut down if more than one sensor fails.
- The action *ActFail* models the failure of an actuator. The system is also shut down if both actuators fail.
- The I/O processors themselves can also fail, which is represented by the actions *IProcFail* and *OProcFail*. In either case, if I or O is unavailable, then the main processor M retries to read data from I or to send instructions to O. If the number of failure tries exceeds a limit MAX_COUNT, then the system is shut down. In our experiments we set MAX_COUNT=4.
- The action *MProcFail* indicates the failure of the main processor M, in which case the system is automatically shut down.

The model is translated by PRISM into a CTMC that consists of 4323 states and 18206 transitions.

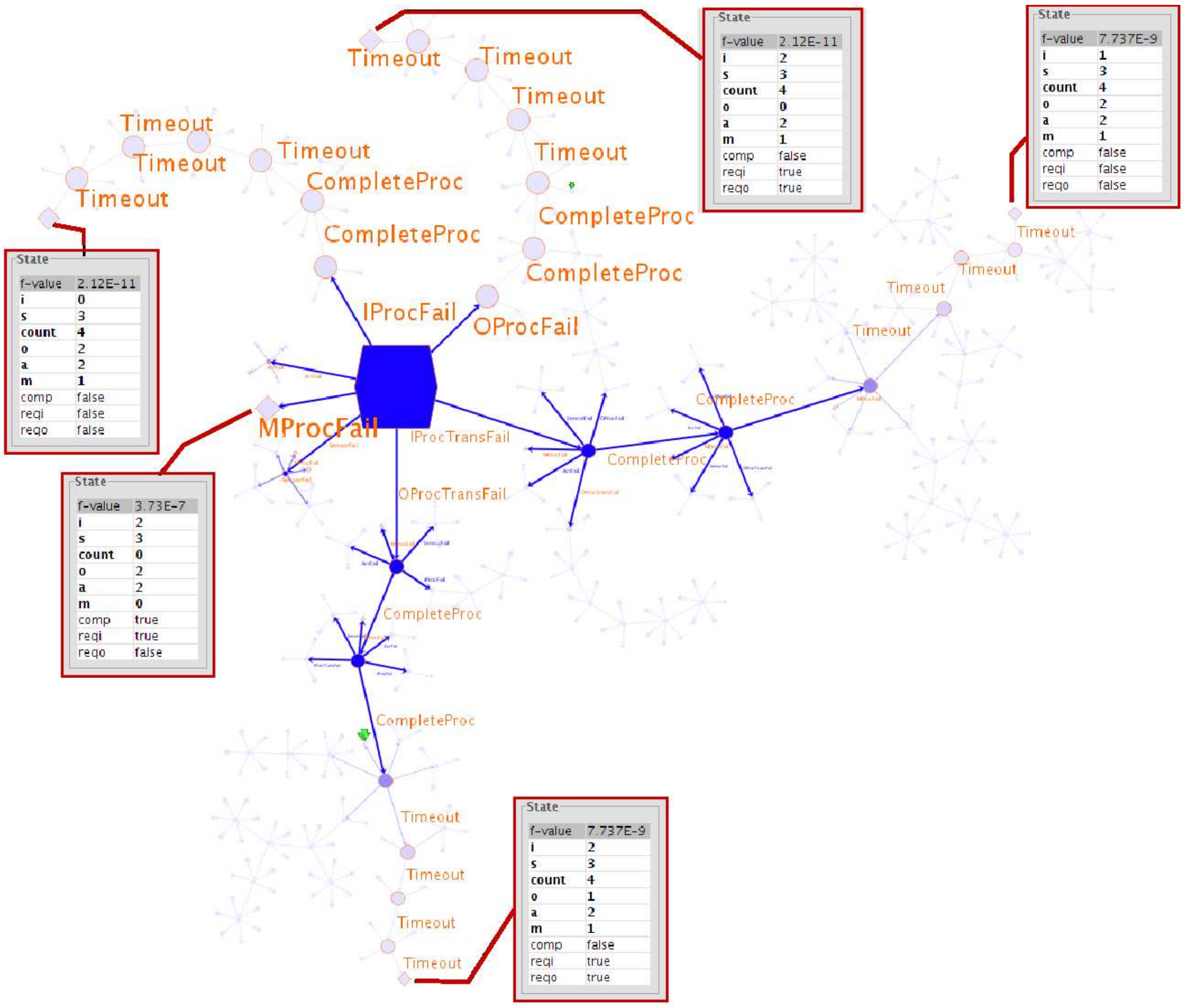
We are interested in the likelihood that the system is shut down within one hour, i.e., 3600 seconds. According to the description of the PRISM model, one time unit corresponds to one second. This induces the property

$$P =? \; [true \; U_{\leq 3600} \; down]. \tag{1}$$

For this property the Prism model checker computes a total probability of $4.363 \cdot 10^{-4}$.

In DiPro, the user is offered the possibility to perform a step-by-step exploration of the state space. The different intensities of the blue color in Figure 3 aid the user in identifying which states are more and which are less probable. The user can control which state to explore next or to let the algorithm automatically expand the next most probable state. This feature and many other interaction functions allow the user to navigate through the state space, thereby learning about modeling errors during the modeling phase or about faults when debugging the model.

Figure 3 represents the output that DiPro produces after a search progress of 208 iterations. The explored portion of the state space contains 417 states and 680 transitions. The generated counterexample is highlighted by surrounding states with red lines and writing the transition labels in red. The user can select a particular path and, for instance, analyze the state variable evaluations for selected nodes along this path. An important feature is the highlighting of the paths which are most responsible for the failure. The size of the symbol representing a state and the font size of a transition label are proportional to the probability of reaching an failure state through the respective state or transition. In Figure 3 we see three paths which are very much highlighted in this way compared to other paths. The first path (Path 0) consists of one transition with the action *MProcFail* which models the failure of the main processor. The



State

f-value	2.12E-11
i	2
s	3
count	4
o	0
a	2
m	1
comp	false
reqi	true
reqo	true

State

f-value	7.737E-9
i	1
s	3
count	4
o	2
a	2
m	1
comp	false
reqi	false
reqo	false

State

f-value	2.12E-11
i	0
s	3
count	4
o	2
a	2
m	1
comp	false
reqi	false
reqo	false

State

f-value	3.73E-7
i	2
s	3
count	0
o	2
a	2
m	0
comp	true
reqi	true
reqo	false

State

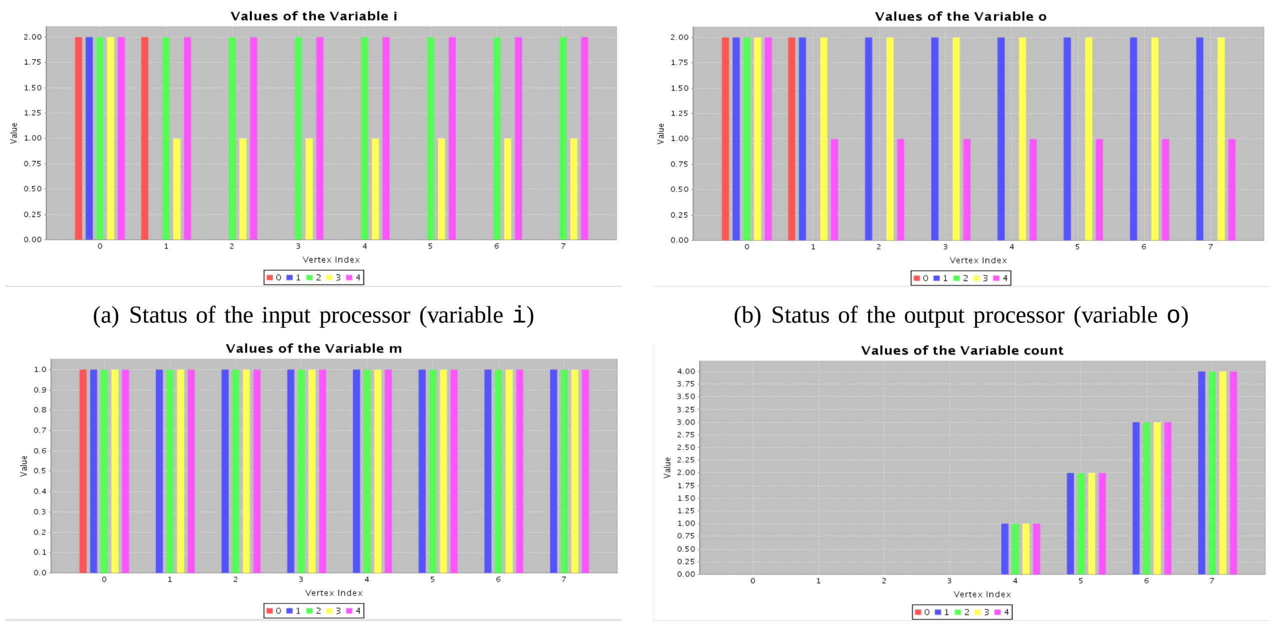
f-value	7.737E-9
i	2
s	3
count	4
o	1
a	2
m	1
comp	false
reqi	true
reqo	true

Fig. 1. Visualization of the Embedded Control System model.

second path (Path 1) starts with *IProcFail*, which indicates the failure of the input processor I, followed by *Timeout* actions representing the unavailability of input data. The third path (Path 2) is the path which starts with *OProcFail* representing the failure of the output processor O. This indicates that a major portion of the failure probability flows along these paths, indicating that a promising approach to make the system more reliable is to improve the reliability of the processors I, O and M. Conversely, it would not be effective to try to improve the reliability of the sensors, actuators or the communication bus.

We also notice another two failure paths which are interesting although they are not emphasized as much as the above mentioned three paths. The first of these is the path starting with the action *IProcTransFail* (Path 3), which models a transient failure of the input processor. The other is the one starting with *OProcTransFail* (Path 4), which models a transient failure of the input or output processor. This means that transient failures of the I/O processors represent further causal factors for the error. Such transient failures can be rectified automatically by the processor rebooting itself. However, the rate of reboot seems to be too low. Hence, speeding up the reboot in case of a transient fault is a further measure to increase the reliability of the system. Note that this change to the system has less impact than improving the reliability of the processors I, O and M.

A further visualization in DiPro that aids in debugging is the visualization of the development of variable valuations along execution paths of the model. This feature is aiming at identifying causal factors for property violations that are to be found in the development of variable valuations. In Figure 2 we depict bar charts



(a) Status of the input processor (variable `i`)

(b) Status of the output processor (variable `o`)

(c) Status of the main processor (variable `m`)

(d) Communication failure count (variable `count`)

Fig. 2. Visual analysis of some variable valuations along primary counterexample paths

to show the evolution of the valuation of some state variables along the five paths described above. From Figures 2(a) and 2(b) we can derive that the variable `i` in path 3 (yellow bars) and the variable `o` in path 4 (pink bars) drop to 1. In the PRISM model this phenomenon is caused by a transient failure of I and O. We can also observe that `i` along path 1 (blue bars) and `o` along path 3 (green bars) attain the value 0. This points at a failure of I and O. Each of these failures causes a failure in transferring the signals from the sensor or sending instructions to the actuators, as can be learned from the corresponding code in the PRISM models. This is confirmed by Figure 2(d) where the variable `count`, which stores the number of failing transfer tries, increases for all paths 1, 2, 3 and 4. Figure 2(c) shows that the variable `m` attains the value 0 for path 0 (red bars). This represents a failure of the main processor M, as can be seen by inspecting the PRISM code.

B. *Workstation Cluster (WsC)*

The second case study is given by a Prism model of a dependable cluster of workstations as first presented in [24]. The model is a CTMC which describes a system consisting of two sub-clusters connected via a backbone. Each sub-cluster consists of N workstations with a central switch that provides an interface to the backbone. We set $N = 10$ in the case study. Each of the components of the system (workstations, switches, and backbone) can break down. The failure of different components is indicated in the Prism model by the following actions:

- The actions *FailWSLeft* and *FailWSRight* indicate the failure of one workstation in the left or the right cluster, respectively.
- The action *FailLine* represents the failure of the backbone.
- The actions *FailToLeft* and *FailToRight* represent the failure of the switches of the left and the right cluster, respectively.

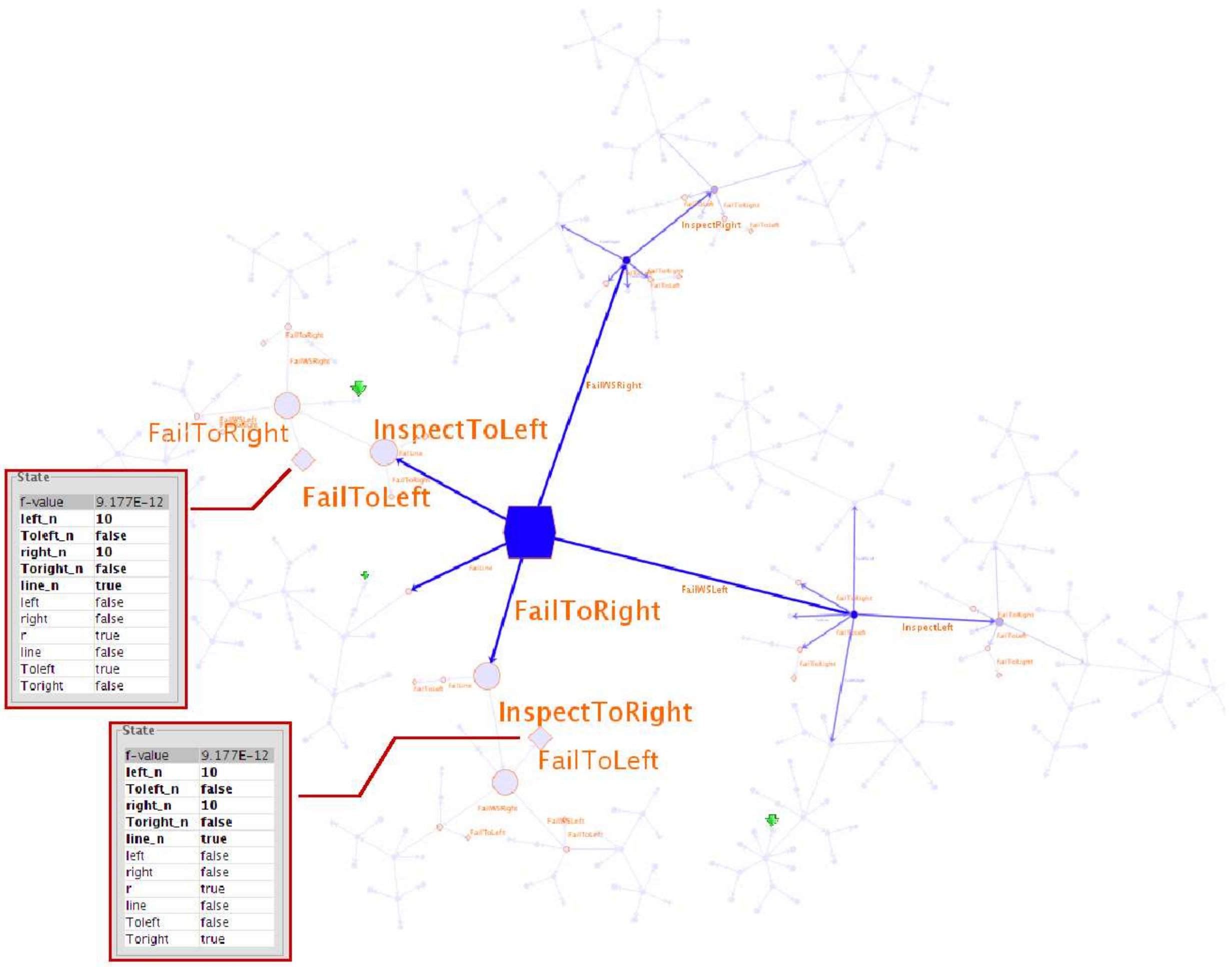
In order to provide minimum quality of service (QoS), at least 75% of the workstations have to be operational and connected to each other via operational switches and the backbone.

The CTMC derived from the PRISM model consists of 4180 states and 19552 transitions. We are interested in the likelihood that the quality of service drops below the minimum within 10 time units. In other words, we are interested in the property

$$P =?\ [\,true\ U\leq 10\ !\text{"}minimum\text{"}\,], \tag{2}$$

where "$minimum$" is a label which asserts the minimum QoS. Checking the property using Prism results in a probability for the full model of $3.252 \cdot 10^{-6}$.

Our visualization of this model as given in Figure 3 shows the counterexample. It emphasizes two paths, namely the path ⟨*FailToLeft*, *InspectToLeft*, *FailToRight*⟩



State

f-value	9.177E-12
left_n	**10**
Toleft_n	**false**
right_n	**10**
Toright_n	**false**
line_n	**true**
left	false
right	false
r	true
line	false
Toleft	true
Toright	false

State

f-value	9.177E-12
left_n	**10**
Toleft_n	**false**
right_n	**10**
Toright_n	**false**
line_n	**true**
left	false
right	false
r	true
line	false
Toleft	false
Toright	true

Fig. 3. Visualization of the Workstation Cluster System model.

and the path ⟨*FailToRight*, *InspectToRight*, *FailToLeft*⟩. This indicates that the most probable cause for the QoS to drop below the minimum required value are the failure of the left and the right switch, respectively. Other actions like *FailWSLeft* and *FailWSRight* are depicted using relatively small icons which means that they have only a secondary influence on the property violation. We can conclude that the appropriate approach for making the system more dependable is not to invest effort in increasing the reliability of the individual workstations but to increase the reliability of the switches.

V. Conclusion

We presented an approach to the debugging of system dependability models given as DTMCs and CTMCs using an interactive visualization of counterexamples to probabilistic reachability properties. The counterexamples are generated by an on-the-fly k-shortest-paths search using our K* algorithm. The objective of the interactive visualization that we use is to bring out information that is important to understand what execution traces contribute most probability mass to the violation of some property. The visualization also permits the interactive filtering out of portions of the state graph, and to follow the evolution of variable valuations along execution paths. All of these measures are designed to facilitate the discovery of causal factors in the property violation. We have implemented the approach in the DiPro tool and have shown its usefulness using two case studies.

In principle, the visualization approach can also be applied to the explicit-state based search for counterexamples in Markov Decision Processes (MDPs) that we presented in [25]. However, that approach searches for a scheduler that maximizes the total probability with every new offending path that is found. With each such iteration the counterexample changes, and this makes an incremental, step-wise visualization as the search progresses not very informative. However, a visualization of the final result of the search in a way that we have described in this paper will be very helpful and can be easily accomplished. We currently extend DiPro to handle MDPs in this way.

We currently work on extending the approach with a variety of data mining and visual analytics techniques in order to enhance the detection of property violation causalities. An example for that is our current work on visual comparison of correct and offending system executions, and to automatically elicit information from the sequences of variable valuations along counterexample paths. We also work on techniques to display the counterexamples on gigapixel displays such as the Powerwall at the University of Konstanz in order to be able to visualize large excerpts of complex state spaces.

Acknowledgements: We thank L. Meier for his contribution to the implementation of DiPro. The work greatly benefited from discussions with Daniel Keim and Ulrik Brandes.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999 (third printing 2001).

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[3] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, ser. LNCS, H. Hermanns and J. Palsberg, Eds., vol. 3920. Springer, 2006, pp. 441–444.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model-checking algorithms for continuous-time Markov chains," *IEEE Transions on Software Engineering*, vol. 29, no. 7, 2003.

[5] J. Pearl, *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addision–Wesley, 1986.

[6] H. Aljazzar, H. Hermanns, and S. Leue, "Counterexamples for timed probabilistic reachability." in *FORMATS*, ser. Lecture Notes in Computer Science, P. Pettersson and W. Yi, Eds., vol. 3829. Springer, 2005, pp. 177–195.

[7] H. Aljazzar and S. Leue, "Extended directed search for probabilistic timed reachability." in *FORMATS*, ser. Lecture Notes in Computer Science, E. Asarin and P. Bouyer, Eds., vol. 4202. Springer, 2006, pp. 33–51.

[8] V. M. Jiménez and A. Marzal, "Computing the k shortest paths: A new algorithm and an experimental comparison," in *Algorithm Engineering*, 1999, pp. 15–29.

[9] T. Han and J.-P. Katoen, "Counterexamples in probabilistic model checking," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 13th International Conference*, 2007.

[10] ——, "Providing evidence of likely being on time: Counterexample generation for ctmc model checking," in *ATVA*, ser. Lecture Notes in Computer Science, K. S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, Eds., vol. 4762. Springer, 2007, pp. 331–346.

[11] H. Aljazzar and S. Leue, "K*: A directed on-the-fly algorithm for finding the k shortest paths," Chair for Software Engineering, University of Konstanz, Gemany, Tech. Rep. soft-08-03, March 2008, submitted for publication. [Online]. Available: http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-08-03.pdf

[12] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Workshop on Software Model Checking (SPIN)*, ser. Lecture Notes in Computer Science 2648. Springer, 2003, pp. 121–135.

[13] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 3, 2006.

[14] J. F. Groote and F. van Ham, "Interactive visualization of large state spaces," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 1, pp. 77–91, 2006.

[15] S. Mukherjea and J. T. Stasko, "Applying algorithm animation techniques for program tracing, debugging, and understanding," in *ICSE '93: Proceedings of the 15th international conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 456–465.

[16] ——, "Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 3, pp. 215–244, 1994.

[17] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability." *Formal Asp. Comput.*, vol. 6, no. 5, pp. 512–535, 1994.

[18] R. Alur and T. A. Henzinger, "Reactive modules," *Formal Methods in System Design*, vol. 15, no. 1, pp. 7–48, 1999.

[19] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Model-checking continuous-time markov chains," *ACM Trans. Comput. Logic*, vol. 1, no. 1, pp. 162–170, 2000.

[20] A. Altintop, "Using color effectively in visualization," 2002. [Online]. Available: http://www.ub.uni-konstanz.de/

[21] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[22] yWorks GmbH, 2007. [Online]. Available: http://www.yworks.com/en/products_yfiles_about.htm

[23] J. Muppala, G. Ciardo, and K. Trivedi, "Stochastic reward nets for reliability prediction," *Communications in Reliability, Maintainability and Serviceability*, vol. 1, no. 2, pp. 9–20, July 1994.

[24] B. R. Haverkort, H. Hermanns, and J.-P. Katoen, "On the use of model checking techniques for dependability evaluation." in *SRDS*, 2000, pp. 228–237.

[25] H. Aljazzar and S. Leue, "Counterexamples for model checking of markov decision processes," Chair for Software Engineering, University of Konstanz, Gemany, Tech. Rep. soft-08-01, December 2007, submitted for publication. [Online]. Available: http://www.ub.uni-konstanz.de/kops/volltexte/2008/4530/