Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool CASPA

Michael Günther
Universität der Bundeswehr München
micha1.guenther@web.de

Johann Schuster
Universität der Bundeswehr München
johann.schuster@unibw.de

Markus Siegle
Universität der Bundeswehr München
markus.siegle@unibw.de

ABSTRACT

CASPA is a stochastic process algebra tool for performance and dependability modelling, analysis and verification. It is based entirely on the symbolic data structure MTBDD (multi-terminal binary decision diagram) which enables the tool to handle models with very large state space. This paper describes an extension of CASPA's solving engine for path-based analysis. We present a symbolic variant of the k-shortest-path algorithm of Azevedo, which works in conjunction with a symbolic variant of Dijkstra's shortest path algorithm. A non-trivial case study illustrates the use of this kind of path-based analysis.

Keywords

stochastic process algebra, k-shortest paths, MTBDD

1. INTRODUCTION

CASPA is a tool for performance and dependability modelling, based on a stochastic process algebra. Its development began in 2003 [10], and it has since been the environment in which its developers have experimented extensively with symbolic (i.e. MTBDD-based) techniques that enable the tool to generate and analyse Markov chains with very large state spaces in a highly efficient manner [6].

Recently, CASPA's modelling language has been extended to support immediate actions (beside the usual Markovian actions) [2]. With this extension, CASPA has been proven to be very suitable for dependability evaluation purposes, in particular for analysing the dynamic behaviour of fault tolerant systems. For example, it is also used as a back-end solver of the OpenSESAME modelling tool [9]. In the context of dependability evaluation, the most probable action sequences that lead to an error state are often of interest. In order to calculate those sequences, we propose to use k-shortest-path algorithms (in this paper, we use the term "k-shortest-path" to denote the k-most probable path). This paper presents symbolical variants of Dijkstra's algorithm and the Azevedo k-shortest-path algorithm. These new algorithms are implemented and used in CASPA.

The paper is organised as follows: Section 2 is dedicated to our symbolic variant of Dijkstra's algorithm for calculating the shortest (i.e. most probable) path in a given transition system. Section 3 introduces a symbolical derivative of Schmid's variant of the Azevedo algorithm for calculating the k-shortest-path. Section 4 presents a non-trivial application case study that illustrates the efficiency of the implemented symbolic data structures and algorithms, and Section 5 concludes the paper. The appendix A is devoted to the MTBDD operations used for the symbolic algorithms presented in the paper.

DYADEM-FTS 2010, April 27, Valencia, Spain

2. SPANNING TREE ALGORITHM

This Section introduces a variant of Dijkstra's algorithm which is used for calculating the most probable path. Our variant is a set-theoretic approach, in order to exploit the possibilities of symbolic data structure. It was developed within Guenther's Master thesis [5]. This variant is called *flooding Dijkstra algorithm* in the sequel. In order to present a more readable version of the algorithm we use a slight modification of Guenther's original presentation, i.e. we work on a *maximal projection* of the transition system of interest. Suppose we are given a set of states S, a set of labels L and a labelled transition system $Trans$ defined as follows:

$$Trans \subseteq S \times L \times [0,1] \times S,$$

where we further assume that "parallel" transitions carry diffent labels, i.e. if $(x,a,p,y) \in Trans$ and $(x,a',p',y) \in Trans$, then $a \neq a'$ must hold. The real number in the interval $[0,1]$ is the probability of taking the corresponding transition, i.e.

$$\forall x \in S : \sum_{(x,a,p,y) \in Trans} p = 1.$$

We will also write $x \overset{a,p}{\rightarrow} y$ for the tuple (x,a,p,y). The *maximal projection* $Trans_{max}$ of $Trans$ is defined as

$$\begin{aligned} Trans_{max} \quad := \quad & \{(x,p,y) \in S \times [0,1] \times S | \\ & (\exists a \in L : (x,a,p,y) \in Trans) \wedge \\ & (\forall a \in L, p' \in [0,1] : \\ & ((x,a,p',y) \in Trans) \Rightarrow (p' \leq p))\}. \end{aligned}$$

The existence condition ensures that a lifting to $Trans$ exists and the second condition ensures that p is maximal. So these are the maximum transition probabilities one can get by choosing an arbitrary action from a source to a target

state. Again, we will use $x \xrightarrow{p} y$ as a synonym for the tuple (x, p, y).

In contrast to Dijkstra's algorithm we do not only carry out optimal, final updates but there may also be some updates of probabilities that are re-updated later on. The resulting graph is not a real spanning tree, in the sense that we include paths that are equally probable. We will resolve this issue when we read out the most probable path by making choices then. Nevertheless, in the sequel we will call such a quasi-spanning tree a spanning tree. (Alternatively one would have to ensure that in lines 26-28 of Algorithm 1 only one transition for each target y is added, in order to get a real spanning tree.)

The code of Algorithm 1 is explained as follows. Lines 1-6 perform the necessary initialisations, i.e. all state probabilities are set to zero except the probability of state 0 which is set to 1. The current BorderSet from where the updates start is set to state 0. As the algorithm just starts, the SpanningTree of course is the empty set. Lines 7-31 constitute the main loop that is carried out until there is no state in the BorderSet any more. First, in lines 8-10 the new probabilities of all states are set to zero. The loop from line 11 to 14 then calculates the maximum probability for every state $y \in S$ to be reached from the BorderSet. The UpdateSet is defined in line 16 to be those states that gained a higher probability in the current round. The update of the state probabilities to the higher probabilities gained by the BorderSet is done in lines 17 to 19. The remaining part is to update the spanning tree for those states that got a higher probability. Therefore the old transitions that reached the UpdateSet are deleted from the SpanningTree in lines 20-24 and the new transitions are added in lines 25-29. Note by looking at the condition in line 26 that by only looking at Prob(y) the update in line 27 might not be unique and therefore we do not get a spanning tree but a spanning tree that contains redundant, i.e. equally probable, paths. We will come to this in Sec. 3. Finally in line 30 the BorderSet is set to the UpdateSet and the loop can start again. Of course, the flooding Dijkstra algorithm shown in Algorithm 1 also performs a reachability analysis, since only reachable states will be in the spanning tree.

Algorithm 1 Flooding Dijkstra algorithm

```
 1: for all x ∈ S : x ≠ 0 do
 2:    Prob(x) = 0
 3: end for
 4: Prob(0) = 1
 5: BorderSet = {0} ⊆ S
 6: SpanningTree = ∅ ⊆ Trans_max
 7: while BorderSet ≠ ∅ do
 8:    for all y ∈ S do
 9:       newProb(y) = 0
10:    end for
11:    for all (x, y) ∈ BorderSet × S do
12:       if x -p-> y ∈ Trans_max then
13:          newProb(y) = max(Prob(x) · p, newProb(y))
14:       end if
15:    end for
16:    UpdateSet = {x ∈ S | newProb(x) > Prob(x)}
17:    for all x ∈ UpdateSet do
18:       Prob(x) = newProb(x)
19:    end for
20:    for all (x, y) ∈ S × UpdateSet do
21:       if x -p-> y ∈ SpanningTree then
22:          SpanningTree = SpanningTree \ {x -p-> y}
23:       end if
24:    end for
25:    for all (x, y) ∈ S × UpdateSet do
26:       if (x -p-> y ∈ Trans_max) ∧ (Prob(x) · p = Prob(y)) then
27:          SpanningTree = SpanningTree ∪ {x -p-> y}
28:       end if
29:    end for
30:    BorderSet = UpdateSet
31: end while
32: return SpanningTree
```

3. CALCULATION OF $K-$SHORTEST PATHS

This Section shows how to calculate k-shortest-paths, thereby employing the flooding Dijkstra algorithm presented in Sec. 2. First we show the basic procedure of finding the correct action labels for a path within the spanning tree calculated in the previous Section. Then we show how the transition system has to be transformed such that the shortest path of the transformed transition system is the second shortest path of the original transition system. With this algorithms at hand, one can calculate the $k-$shortest paths up to a certain fixed k.

3.1 Reading the action labels from a path

Before proceeding with the algorithm, we first have to define a *maximal lifting*. The maximal lifting of a subset $T \subseteq Trans_{max}$ to $Trans$ is given by

$$T^* \quad := \quad \{(x, a, p, y) \in Trans | \exists (x, p, y) \in T\}.$$

You may observe that this lifting is not unique: There can be more than one action fulfilling the maximality condition while p as the maximum is unique for a certain pair $(x, y) \in S^2$. Note that the algorithm for reading the most probable path works on the transition system $Trans$ and uses the maximal lifting of the path found by flooding Dijkstra in $Trans_{max}$. Let $Dest$ be the state which is the target of the path analysis and let $Init$ be the starting state of the analysis. Further we use the function $PickOne(X)$ that chooses one arbitrary element of a set X. Algorithm 2

Algorithm 2 Reading action labels

```
 1: y = Dest
 2: while y ≠ Init do
 3:    print ←
 4:    Predecessors =
         {x ∈ S | ∃(p) ∈ [0, 1] : (x, p, y) ∈ SpanningTree}
 5:    x = PickOne(Predecessors)
 6:    for all (a, p) ∈ L × [0, 1] do
 7:       if x -a,p-> y ∈ SpanningTree* then
 8:          print <a>
 9:          Path = Path ∪ {x -a,p-> y}
10:       end if
11:    end for
12:    y = x
13: end while
14: return Path
```

prints the shortest path from the $Dest$ to $Init$ by a traversal of SpanningTree choosing unique predecessor states. The explanation is straightforward. Line 1 sets the current state to $Dest$. The main loop that jumps back to the current state's predecessor goes from line 2 to 13 as long as we did

not reach $Init$. In line 3 prints the arrow indicating the next step. Line 4 computes all predecessor states of y in $SpanningTree^*$ and line 5 arbitrarily chooses one of them. The loop from line 6 to 11 picks out the corresponding transitions from state x to state y in $SpanningTree^*$. In line 8 the corresponding action is printed and line 9 adds the corresponding transition to $Path$. Line 12 sets the current state to the predecessor x.

Note that there can still be concurring actions in the Path generated. Of course due to the $SpanningTree^*$-property it holds that

$$\begin{aligned}\forall(x,y,a,a') \quad &\in S^2 \times L^2 : \\ &((x,a,p,y) \in SpanningTree^* \wedge \\ &(x,a',p',y) \in SpanningTree^*) \Rightarrow p = p'.\end{aligned}$$

So the probabilities of concurring actions in $Path$ are equal, no need to distinguish them for further processing.

3.2 Second shortest path

This Subsection will show how to generate a transition system $Trans'$ out of a given transition system $Trans$ such that the shortest path of $Trans'$ corresponds to the second shortest path of $Trans$. Inductively follows that by this concept the k shortest paths can be calculated for an arbitrary but fixed k. The algorithm is due to Azevedo [1] using a refinement of Schmid [7].

Starting from a set of states S and a set of transitions $Trans$ as before we use an additional set of states S' with $S \cap S' = \emptyset$ and $|S| = |S'|$. We use a fixed bijection

$$\begin{aligned}' : \quad & S \to S' \\ & x \mapsto x'\end{aligned}$$

to identify elements x of S with their corresponding copy x' in S' and vice versa. Further we define the set of states

$$S_{new} := S \cup S'$$

for the new transition system $Trans' \subseteq S_{new} \times L \times [0,1] \times S_{new}$. Let $Path$ be the path calculated in Sec. 3.1.

Algorithm 3 Second shortest path

```
1: Trans' = Trans
2: PathCopy := {(x', a, p, y') | ∃(x, a, p, y) ∈ Path ∧ y ≠ Dest}
3: Trans' = Trans' ∪ PathCopy
4: SourceStates := {x ∈ S | ∃(x, a, p, y) ∈ Path}
5: for all (x, a, p, y) ∈ Trans do
6:    if x ∈ SourceStates then
7:       if x -a,p-> y ∉ Path then
8:          Trans' = Trans' ∪ {x' -a,p-> y}
9:       end if
10:   end if
11: end for
12: return (Trans', Init = Init')
```

Algorithm 3 works as follows: The seed of the new transition system $Trans'$ are the old transitions $Trans$ (of course the initial state has to be changed in order not to find the same path as before) which are set in line 1. The shortest path (including its parallel actions) is copied to S' without the last transition to $Dest$ in line 2. Line 3 adds this path to $Trans'$. Line 4 calculates the source states of the transitions in $Path$. The loop from line 5 to 11 picks all the transitions that emanate from a source state in $Path$ but are not on the shortest path and adds them as cross connections from S' to S to the transitions of $Trans'$. Finally the initial state is set to the copy of the initial state in S' in line 12.

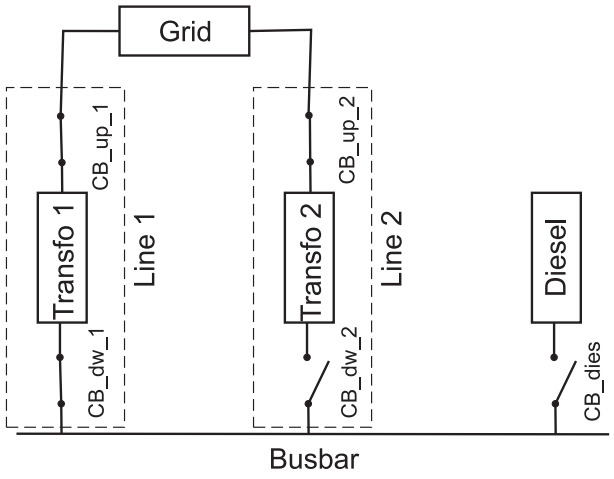

Figure 1: Sketch of the busbar model

4. CASE STUDY

To show the applicability of our algorithm we modelled the busbar given in [3] and noticed that the most probable paths published did not fit the textual description there (e.g. an initial failure of trafo 2 would not be possible if it was a cold spare). This is why we give an alternative description of the model that to our best knowledge produces the desired paths.

4.1 Description of the model

The model is shown in Fig. 1. The aim of the model is to provide the busbar with electrical energy. Each of the main lines consists of upper and lower circuit breakers and a transformer. They route electrical energy from the grid to the busbar. If the lines fail or the grid does, the diesel generator has to be used. The initial configuration is as seen in Fig. 1 where only CB_dw_2 and CB_dies are in the open position, the other switches are closed. The following constraints for the operation and dynamic behaviour are given:

- States of the components can be WORKING, STANDBY or FAILED
- Either line 1, line 2 or the diesel engine is used. Mode switches can only be line 1 $\leftrightarrow$ line 2 $\leftrightarrow$ diesel, no direct switches from line 1 to diesel and vice versa are allowed.
- The trafos and the grid are hot spares and always fail with the same rate, no matter if they are active or not.
- The circuit breakers CB_up and CB_dw are cold spares (i.e. they do not fail as long as no current runs over them) and they can produce on-demand-failures.
- CB_dies does only fails on-demand, it does not fail internally
- An on-demand-failure of a circuit breaker does not change its internal state. Reconfiguration can change the internal state of a component.
- Switching from trafo 1 to trafo 2 means trying to open CB_up_1 and to close CB_dw_2. Note that when trafo 1 fails and CB_up_1 fails to open, one must try to close CB_dw_2 even if it's clear that the diesel engine has to be

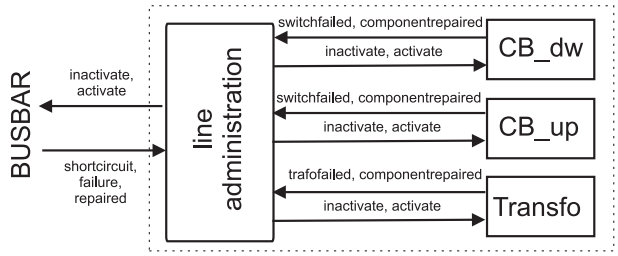

Figure 2: Sketch of the model of a trafo line

used, as the grid is short-circuited by line 1 in this case.

- Switching on the diesel engine means closing CB_dies and trying to start the engine. Both operations have on-demand failures
- Switching back after a repair always works without on-demand failures
- When a trafo fails, its upper circuit breaker has to be opened, otherwise a shortcircuit will make the grid unavailable for the other trafo.

We have introduced a total ordering of all switching actions as follows: (CB_up_1 > CB_dw_1 > CB_up_2 > CB_dw_2 > CB_dies > S_dies). Whenever there are more switching operations to be done in our model, the greater operation will always be switched first. Following the philosophy of shortest paths to error states we stress that whenever a greater switching action fails, we do not explore the path any further (except for the short circuit situation given above). All intended switchings succeed with probability $\frac{999}{1000}$, all failure rates are equal to 10^{-4} per hour and all repair rates equal to 10^{-1} per hour.

4.2 CASPA implementation of the model

The model has been built in CASPA using a compositional modelling approach with synchronisations. Each line is the parallel composition of two switches, a transformer and a line administration. Its synchronised actions are given in Fig. 2. The line administration process has in addition to its internal state a counter variable that keeps track of how many components of the line are currently in the FAILED state. From this it can be determined, when the entire line has been repaired. A top-level process synchronises with two line subprocesses and takes care of the grid and the entire diesel line. As the immediate actions used only for synchronising are of no interest for the resulting paths, they are eliminated. The elimination is done by four semisymbolic elimination rounds [2] and the resulting model has 774 reachable states.

4.3 Experimental results

All experiments have been carried out on an Intel Xeon 3.06 GHz machine with 2 GB of main memory running SUSE Linux version 9.1. The elimination of the superfluous synchronisations takes 0.285 seconds (including reachability analysis). For the calculation of the first 100 paths, the mean time needed to calculate one path is 0.61 seconds with a minimum of 0.15 seconds for the 4th path and a maximum of 1.14 seconds for the 24th path.

Table 1 shows the 13 most probable paths calculated by our algorithm. The first column is the sequence of actions, the second one the numerical result provided by our algorithm. The third column shows the exact result calculated by hand. For the by-hand calculation one only has to take care, which failure event(s) and repair event(s) can occur for a certain state. For example, the most probable path calculates as follows: In the initial configuration no component has to be repaired and Transfo1, CB_up_1, CB_dw_1, Transfo2 or Grid can fail. Therefore

$$P\left(\begin{array}{l}\texttt{Failure\_OF\_GRID} \rightarrow \\ \texttt{Occurrence\_OF\_RC\_CB\_dies}\end{array}\right) = \frac{10^{-4}}{5 \cdot 10^{-4}} \cdot \frac{1}{1000} = \frac{1}{5} \cdot \frac{1}{1000}.$$

Due to the fact that CASPA uses the CUDD library [4] with a default Cudd_Epsilon of $1.0 \cdot 10^{-12}$, this is the maximum accuracy one can expect from the results. For example the tool calculates

$$P\left(\begin{array}{l}\texttt{Failure\_OF\_Transfo2} \rightarrow \\ \texttt{OK\_OF\_RO\_CB\_up\_2} \rightarrow \\ \texttt{Failure\_OF\_GRID} \rightarrow \\ \texttt{Occurrence\_OF\_RC\_CB\_dies}\end{array}\right) = P\left(\begin{array}{l}\texttt{Failure\_OF\_Transfo2} \rightarrow \\ \texttt{Occurrence\_OF\_RO\_CB\_up\_2} \rightarrow \\ \texttt{OK\_OF\_RC\_CB\_dies} \rightarrow \\ \texttt{OK\_OF\_RS\_dies} \rightarrow \\ \texttt{Failure\_OF\_dies\_generator}\end{array}\right),$$

but the exact values differ:

$$\frac{1}{5} \cdot \frac{999}{1000} \cdot \frac{10^{-4}}{10^{-1}+4 \cdot 10^{-4}} \cdot \frac{1}{1000} \neq \frac{1}{5} \cdot \frac{1}{1000} \left(\frac{999}{1000}\right)^2 \cdot \frac{10^{-4}}{10^{-1}+3 \cdot 10^{-4}}$$

As the difference is below $1.0 \cdot 10^{-12}$, the values are taken as equal. Reducing Cudd_Epsilon would improve accuracy but also slow down the calculations.

Path	numerical result	theoretical result
Failure_OF_GRID Occurrence_OF_RC_CB_dies	$2.0e-04$	$\frac{1}{5}\frac{1}{1000}$
Failure_OF_GRID OK_OF_RC_CB_dies Occurrence_OF_RS_dies	1.998e-04	$\frac{1}{5}\frac{999}{1000}\frac{1}{1000}$
Failure_OF_GRID OK_OF_RC_CB_dies OK_OF_RS_dies Failure_OF_dies_generator	1.990e-04	$\frac{1}{5}\left(\frac{999}{1000}\right)^2\frac{10^{-4}}{10^{-1}+3\cdot10^{-4}}$
Failure_OF_Transfo2 Occurrence_OF_RO_CB_up_2 Occurrence_OF_RC_CB_dies	2.000e-07	$\frac{1}{5}\left(\frac{1}{1000}\right)^2$
Failure_OF_CB_dw_1 Occurrence_OF_RC_CB_dw_2 Occurrence_OF_RC_CB_dies		
Failure_OF_CB_up_1 Occurrence_OF_RC_CB_dw_2 Occurrence_OF_RC_CB_dies		
Failure_OF_Transfo2 Occurrence_OF_RO_CB_up_2 OK_OF_RC_CB_dies Occurrence_OF_RS_dies	1.998e-07	$\frac{1}{5}\frac{999}{1000}\left(\frac{1}{1000}\right)^2$
Failure_OF_CB_dw_1 Occurrence_OF_RC_CB_dw_2 OK_OF_RC_CB_dies Occurrence_OF_RS_dies		
Failure_OF_Transfo1 OK_OF_RO_CB_up_1 Occurrence_OF_RC_CB_dw_2 Occurrence_OF_RC_CB_dies		
Failure_OF_Transfo1 Occurrence_OF_RO_CB_up_1 OK_OF_RC_CB_dw_2 Occurrence_OF_RC_CB_dies		
Failure_OF_CB_up_1 Occurrence_OF_RC_CB_dw_2 OK_OF_RC_CB_dies Occurrence_OF_RS_dies		
Failure_OF_Transfo1 OK_OF_RO_CB_up_1 Occurrence_OF_RC_CB_dw_2 OK_OF_RC_CB_dies Occurrence_OF_RS_dies	1.996e-07	$\frac{1}{5}\left(\frac{999}{1000}\right)^2\left(\frac{1}{1000}\right)^2$
Failure_OF_Transfo1 Occurrence_OF_RO_CB_up_1 OK_OF_RC_CB_dw_2 OK_OF_RC_CB_dies Occurrence_OF_RS_dies		

Table 1: Start of the list of most probable paths

5. CONCLUSION

In this paper we have presented a set of algorithms to calculate the k-most probable paths for CASPA models. Well-known algorithms like Dijkstra and Azevedo have been adapted to fit the symbolic data structure used in the CASPA tool. A non-trivial case study shows the applicability of the approach. The results have been verified to fit the list given in [3] and for the first 13 paths additionally exact formulas are given. The exactness of the algorithms has been shown to be dependent on the exactness of the calculations in the underlying MTBDD package CUDD. With this path-based analysis the CASPA tool provides an alternative analysis method to the numerical analysis. The path-based analysis can further nicely be used for debugging models. Moreover, with the algorithmic framework for the symbolic calculation of the k-most probable paths at hand it is a small step to calculate the MTTF (Mean Time To Failure) and MTTR (Mean Time To Repair) in an approximate, path-based way. A prototypical implementation of the MTTR/MTTF calculations exists and will be presented in a subsequent paper.

Acknowledgements: Thanks to Alexander Gouberman for pointing us to the work of Azevedo and Schmid. Further we would like to thank Deutsche Forschungsgemeinschaft (DFG) who supported this work under grants SI 710/2 and SI 710/3.

APPENDIX

A. SPANNING TREE MTBDD CODE

In this Section we give the MTBDD versions of the algorithms presented in the paper. In the description of the algorithms the following name conventions for MTBDD variables are used: $\vec{a}$ (action labels), $\vec{s}$ (source states) and $\vec{t}$ (target states). The variable ordering in the MTBDD is $a_1 \prec \ldots \prec a_n \prec s_1 \prec t_1 \ldots \prec s_m \prec t_m$ according to commonly accepted heuristics. The following MTBDD operations are used to describe the algorithm (see e.g. [8] for a more detailed explanation):

- $\langle$MTBDD1$\rangle\langle$OP$\rangle\langle$MTBDD2$\rangle$, the general apply operator
- $ITE(\langle$MTBDD01$\rangle, \langle$MTBDDT$\rangle, \langle$MTBDDE$\rangle)$, the general if-then-else operator: MTBDD01 is a 0-1 MTBDD and whenever it is equal to 1, the value of MTBDDT is returned, the value of MTBDDE otherwise
- $\langle$MTBDD$\rangle_{\langle VAR\rangle=\langle VAL\rangle}$, returns $\langle$MTBDD$\rangle$ with $\langle$VAR$\rangle$ set to $\langle$VAL$\rangle$ (Restriction)
- $ABSTRACT(\langle$MTBDD$\rangle, \langle$VAR$\rangle, \langle$OP$\rangle)$:= $\langle$MTBDD$\rangle_{\langle VAR\rangle=0}\langle$OP$\rangle\langle$MTBDD$\rangle_{\langle VAR\rangle=1}$

Abstraction and restriction of more than one variable is defined in the canonical recursive way. For the general APPLY operation, the following operators are used:

- $\langle$ MTBDD1$\rangle$ == $\langle$ MTBDD2$\rangle$ returns 1 whenever MTBDD1 and MTBDD2 coincide, 0 otherwise
- $\langle$ MTBDD1$\rangle$ > $\langle$ MTBDD2$\rangle$ returns 1 whenever MTBDD1>MTBDD2, 0 otherwise
- $\langle$ MTBDD1$\rangle$ + $\langle$ MTBDD2$\rangle$ returns MTBDD1+MTBDD2

A.1 Spanning tree algorithm

For the algorithm let us assume we are already given a transition system $Trans$ where for every state s with at least one emanating transition the probabilities of all emanating transitions sum up to one, i.e. they define a discrete probability distribution. Whenever needed, subscripts show to which variable set a MTBDD belongs to (e.g. $Prob_s$ means that MTBDD $Prob$ is defined by s-variables. Let the initial state be stored in the variable $Init$ (by s-variables).

Algorithm 4 Flooding Dijkstra algorithm

```
1: Prob_s = Init
2: Border_s = Init
3: SpanningTree = 0
4: Trans_max = ABSTRACT(Trans, a, max)
5: while 1 do
6:    Prob_s^Border = Border_s · Prob_s
7:    Prob_st^Trans = Prob_s^Border · Trans_max
8:    Prob_t^new = ABSTRACT(Prob_st^Trans, s, max)
9:    UpdateSet_t = (Prob_t^new > (Prob_s)_{s→t})
10:   if UpdateSet_t == 0 then
11:      break
12:   end if
13:   Prob_s = (ITE(UpdateSet_t, Prob_t^new, (Prob_s)_{s→t}))_{t→s}
14:   Prob_t^Updated = UpdateSet_t · Prob_t^new
15:   NewTransitions =
      (Prob_st^Trans == Prob_t^Updated) · UpdateSet_t · Trans_max
16:   SpanningTree =
      ITE(UpdateSet_t, NewTransitions, SpanningTree)
17:   Border_s = (UpdateSet_t)_{t→s}
18: end while
```

Algorithm 4 works as follows. In line 1-3 some initialisation assignments are given. Line 4 calculates the maximal projection of $Trans$. The loop from line 5 to 18 calculates the updates. In line 6, the probabilities for the states in the current border set are calculated and in line 7 they are multiplied by the maximal transition probabilities. Line 8 calculates the new probabilities for successor states of the border set by abstraction of $Prob_{st}^{Trans}$. From this the $UpdateSet_t$ can be calculated that encodes the states that are reached with a higher probability by transitions from the border states. In lines 10-12 the exit condition is checked: Whenever there are no more states to be updated, the algorithm has finished. With the knowledge of the states to be updated, line 13 calculates the correct new probabilities. The new probabilities that have been updated are calculated in line 14 and are used in line 15 to get the transitions that provide the maximum probability for those states (not necessarily unique). Looking at the updated states the spanning tree can be updated in line 16: All transitions that lead to a state in $UpdateSet_t$ are taken from $NewTransitions$, the others remain the same. Finally $Border_s$ is updated in line 17.

A.2 Reading the action labels from a path

This code fragment reads one shortest path from the spanning tree calculated in A.1. Note that the path read may include parallel actions, but the predecessor states have to be unique. Suppose that $Init$ encodes the initial state, $Dest$ the destination state of the analysis (by s-variables). It is notable that two versions of the SpanningTree are used: The maximal projection and the maximal lifting. All paths are printed from $Dest$ to $Init$. The code in algorithm 5 reads as follows: Line 1 calculates the probability of $Dest$ by abstracting over all s-variables. In lines 2-4 some initialisations are done, most notably the maximal lifting of SpanningTree in line 3. The main loop starts at line 5 and terminates when $Init$ is reached. All incoming edges of $CurrentState$

Algorithm 5 Reading action labels

1: $p = ABSTRACT(Prob_s \cdot Dest, s, max)$
2: $ShortestPath = 0$
3: $SpanningTree^* = (SpanningTree == Trans) \cdot Trans$
4: $CurrentState_s = Dest$
5: **while** $CurrentState_s! = Init$ **do**
6: $\quad Edges_{st} = (CurrentState_s)_{s \to t} \cdot SpanningTree$
7: $\quad Edge_{st} = PickOne(Edges_{st})$
8: $\quad Edges_{ast} = (Edge_{st} == SpanningTree^*) \cdot SpanningTree^*$
9: $\quad ShortestPath = ShortestPath + Edges_{ast}$
10: $\quad$ **while** $Edges_{ast}! = 0$ **do**
11: $\quad\quad Action = PickOne(Edges_{ast})$
12: $\quad\quad$ **print** $getName(Action)$
13: $\quad\quad Edges_{ast} = Edges_{ast} - Action$
14: $\quad$ **end while**
15: $\quad$ **print** $\leftarrow$
16: $\quad CurrentState_s = ABSTRACT(Edge_{st}, t, +)$
17: **end while**

are read from $SpanningTree$ in line 6. Out of them a single edge is picked in line 7. Line 8 calculates the maximal lifting of $Edge_{st}$ in $SpanningTree^*$. The current $Edge_{ast}$ (with possible parallel actions) is added to $ShortestPath$ in line 9. The loop from line 10-14 prints all parallel actions belonging to $Edge_{ast}$. Line 16 switches to the next predecessor of CurrentState given by $Edge_{st}$.

A.3 Changing the transition system

In this Subsection the symbolic operations for changing the transition system are shown. The transition system is altered in such a way that the shortest path of the new transition system is the second shortest path of the old transition system. Suppose $Prob_s$ contains the probabilities of states and $ShortestPath$ contains a shortest path with possible parallel actions as calculated in A.2. The Expand function used in the code is just a short-hand notation, given as follows:

$$\begin{aligned} Expand(Trans, s, t) &:= NewVariable_s(s) \cdot \\ &\quad\; NewVariable_t(t) \cdot Trans \\ Expand(State, s) &:= NewVariable_s(s) \cdot State \end{aligned}$$

Two new variables are introduced (here between action and state variables) in order to encode original and copied states in source and target variables. $NewVariable_s$ encodes whether the source state lies in the original or the copied set of states. If it is set to 0 it means that the source state lies in the original set of states, if it is set to 1 the state is a copied one. A similar statement applies to the target states. Algorithm 6 works as follows. First the reachability analysis done by the spanning tree calculation is used to minimise the $Trans$ by leaving out unreached states (i.e. states that have a maximal probability of 0). This is done in line 1 and 2. Next, the shortest path without the last transition to $Dest$ is calculated in line 3. Line 4 calculates all allowed deviations, i.e. those transitions that emanate from states on the shortest path but do not lie on the shortest path. In lines 5-7 some expansions are done. From the given parameters one sees that $PathWithoutDest$ lies in the copyied states, while $Deviations$ lead from copies to original states and $Trans$ still lies in the original states. The new transition system is built in line 8 as the union of the three precalculations and finally in line 9 and 10 $Init$ and $Dest$ are copied.

Algorithm 6 Second shortest path

1: $NonZeroProb_s = (Prob_s > 0)$
2: $Trans = NonZeroProb_s \cdot Trans$
3: $PathWithoutDest = ITE((Dest)_{s \to t}, 0, ShortestPath)$
4: $Deviations = ITE(ShortestPath, 0, Trans) \cdot (ABSTRACT(ShortestPath, t, +) > 0)$
5: $PathWithoutDest = Expand(PathWithoutDest, 1, 1)$
6: $Deviations = Expand(Deviations, 1, 0)$
7: $Trans = Expand(Trans, 0, 0)$
8: $Trans' = Trans + Deviations + PathWithoutDest$
9: $Init' = Expand(Init, 1)$
10: $Dest' = Expand(Dest, 0)$

B. REFERENCES

[1] J. Azevedo, J. Madeira, E. Martins, and F. Pires. A Shortest Paths Ranking Algorithm. In *Proc. of the Annual Conference AIRO'90 Operational Research Society of Italy*, pages 1001–1011, IEEE, 1990.

[2] J. Bachmann, M. Riedl, J. Schuster, and M. Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra tool CASPA. In *SOFSEM 2009: Theory and Practice of Computer Science*, pages 485–496, Špindlerův Mlýn, Czech Republic, 2009. Springer, LNCS 5404.

[3] M. Bouissou and J.-L. Bon. A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. *Reliability Engineering and System Safety*, 82:149–163, 2003.

[4] CUDD website. `http://vlsi.colorado.edu/~fabio/CUDD/`, (last checked March 2010).

[5] M. Günther. Pfadbasierte Algorithmen zur Zuverlässigkeitsanalyse. Master's thesis, Univ. der Bundeswehr München, Fakultät für Informatik (in German), 2009.

[6] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Europ. Perf. Engineering Workshop*, pages 293–307. LNCS 3236, 2004.

[7] W. Schmid. *Berechnung kürzester Wege in Straßennetzen mit Wegeverboten*. PhD thesis, Universität Stuttgart, Fakultät für Bauingenieur- und Vermessungswesen, 2000.

[8] M. Siegle. *Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties*. Shaker Verlag, Aachen, 2002.

[9] M. Walter, M. Siegle, and A. Bode. OpenSESAME: The Simple but Extensive, Structured Availability Modeling Environment. *Reliability Engineering and System Safety*, 93(6):857–873, 2007.

[10] E. Werner. Leistungsbewertung mit Multi-terminalen Binären Entscheidungsdiagrammen. Master's thesis, Univ. Erlangen, Computer Science 7 (in German), 2003.