Workflow Fault Tree Generation Through Model Checking

Herbert, Luke Thomas; Sharp, Robin

Published in:
Safety, Reliability and Risk Analysis: Beyond the Horizon

Publication date:
2014

Workflow Fault Tree Generation Through Model Checking

Luke Herbert (lthhe@dtu.dk) & Robin Sharp (robs@dtu.dk)
DTU Compute - Technical University of Denmark
DK-2800 Lyngby - Denmark

ABSTRACT: We present a framework for the automated generation of fault trees from models of real-world process workflows, expressed in a formalised subset of the popular *Business Process Modelling and Notation* (BPMN) language. To capture uncertainty and unreliability in workflows, we extend this formalism with probabilistic non-deterministic branching. We present an algorithm that allows for exhaustive generation of possible error states that could arise in execution of the model, where the generated error states allow for both fail-stop behaviour and continued system execution. We employ stochastic model checking to calculate the probabilities of reaching each non-error system state. Each generated error state is assigned a variable indicating its individual probability of occurrence. Our method can determine the probability of combined faults occurring, while accounting for the basic probabilistic structure of the system being modelled. From these calculations, a comprehensive fault tree is generated. Further, we show that annotating the model with rewards (data) allows the expected mean values of reward structures to be calculated at points of failure.

KEYWORDS: BPMN, Stochastic BPMN, Stochastic Model Checking, Quantitative Model Checking, Formal Risk Analysis, Fault Tree Analysis, Fault Tree Generation

1 INTRODUCTION

Creating fault tolerant and efficient process workflows poses a significant challenge. Individual faults, defined as an abnormal conditions or defects in a component, equipment, or sub-process (Crockford 1986), must be handled so that the system may continue to operate, and are typically addressed by implementing various domain specific safeguards. In complex systems, individual faults may combine to give rise to system failure, defined as a state or condition of not meeting a desirable or intended objective (Crockford 1986). The safety analysis of such systems is labour-intensive and requires a key creative step where safety engineers imagine what undesirable events can occur under which conditions.

Fault Tree Analysis (FTA) attempts to analyse the failure of systems by composing logic diagrams of separate individual faults to determine the probability of larger compound faults occurring. FTA is a commonly used method (Ericson 2005) (Stephans 2005) to derive and analyse potential failures and their impact on overall system reliability and safety. Originally developed in 1962 at by H.A. Watson, FTA has seen extensive refinement and widespread adoption and is today considered a proven and accepted reliability engineering technique (Ericson 1999), often required for regulatory approval of systems. However, fault trees are typically manually constructed (Ericson 2005) and determining the probabilities of faults occurring in systems which exhibit stochastic behaviour in the course of their correct execution is difficult, time-consuming and error prone.

Ideally, safety engineering starts during the early design of a system. Even at this stage in the design process, systems may exhibit a high degree of complexity (Roy et al. 2013), and formal modelling of the system typically captures the ideal, fault-free, conception of the system. Subsequently, the resultant FTA is based on an informal description of the underlying system (Ericson 2005), or requires modelling the system in an separate FTA specific modelling language (Liggesmeyer & Rothfelder 1998, Banach & Bozzano 2011). This makes it difficult to check the consistency of the analysis, because it is possible that causes are noted in the tree which do not lead to the failure (incorrectness) or that some causes of failure are overlooked (incompleteness).

FTA analysis is therefore often avoided early in a system design due to the significant effort involved in performing the analysis and ensuring its consistency with the system of interest, balanced against the likelihood that changes will be made to the design. However, correcting problems late in the development pro-

cess can be costly, cause significant delays, and even require complete system redesign. Being able to perform automated FTA directly from the initial conceptual models of a system is thus of vital importance.

1.1 *Contribution*

Liggesmeyer & Rothfelder 1998 coined the term *formal risk analysis* and developed an approach for automatically generating a fault tree from finite state machine-based descriptions of a system where the generated fault tree is complete with respect to all failures assumed possible. The ideas presented here can been seen as an extension of their developments combined with our previous work (Herbert & Sharp 2012, Herbert & Sharp 2013b) where we presented a theoretical framework, based upon the *Business Process Modelling and Notation 2.0* (BPMN) (Object Management Group 2011) modelling language, which allows the modelling and analysis, via model checking, of a wide range of real-world workflows.

In this paper we present an approach where workflows are modelled in the industry standard BPMN workflow language, which is a common choice for traditional, non FTA orientated, systems modelling (Chinosi & Trombetta 2012). We formalise and extend the BPMN formalism so that systems which exhibit stochastic behaviour can be modelled, and sketch how such models can be automatically converted into a format amenable to model checking. We then present *Probabilistic Computation Tree Logic* PCTL formulae, and an algorithm that exploits these to generate fault trees which reflect the base stochastic behaviour of the system from the generated statespace of the BPMN model as shown in fig. 1.

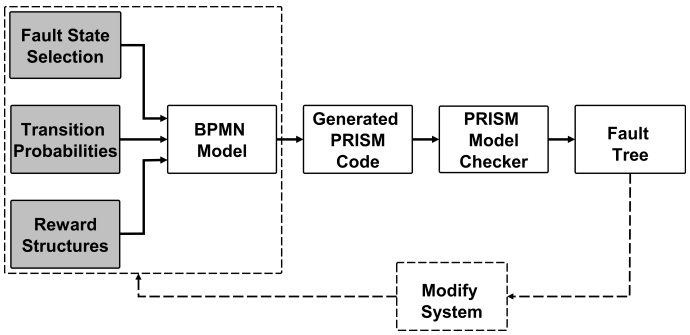


Figure 1: BPMN Fault tree generation overview

In fig. 1, the grey boxes are the additional material that must be supplied by a user in addition to an BPMN model. Note that these additions are simply annotations to an existing BPMN model and require no structural changes to the model. The *reward structures* are data annotations which are not strictly needed to simply perform a qualitative FTA. However, if data is associated with a system model this can also be mapped to the resulting fault tree, which allows the expected values of properties of interest, such as time, power usage or financial cost, to be included in the fault tree. This motivates our choice of quantitative probabilistic model checking, a formal verification method for the analysis of systems which exhibit stochastic behaviour, as opposed to the more limited methods of symbolic model checking empoyed by Liggesmeyer & Rothfelder 1998.

1.2 *Related Work*

Work by Banach and Bozzano 2011 allows for the automated generation of fault trees. However, their work focuses on digital circuits, which must be modelled in a custom modelling language for automated generation to be possible. In order not to add a further source of error, our models are automatically translated into fault trees with no remodelling required.

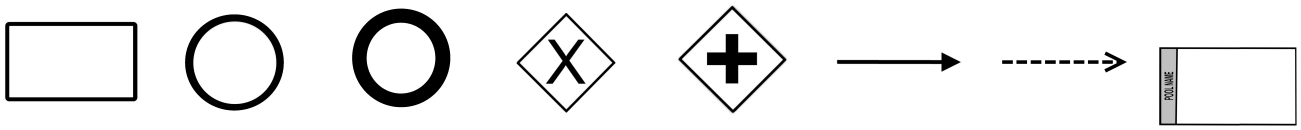
Work by Thums and Schellhorn 2003 Xiaocheng et al. 2010 employs model checking to verify the correctness of an FTA and to perform model checking of fault trees. In both cases, however, the source model for the FTA requires custom modelling of the system of interest and does not incorporate the use of rewards into the analysis.

An interesting approach employing Duration Calculus (DC) model checking is suggested by Schäfer 2003, where fault trees are generated from DC models. While similar to our approach, although lacking reward structures, it does not have a practical implementation, as it has since been shown by Fränzle and Hansen 2008 that model-checking of DC is a 4-fold exponential complexity model checking problem.

2 BUSINESS PROCESS MODEL AND NOTATION

The *Business Process Model and Notation* (BPMN) language (Object Management Group 2011) is a graphical notation for specifying workflows. Unfortunately, the semantics and pragmatics of BPMN are only informally defined in the relevant standards (Object Management Group 2011), thus leaving a number of questions open to interpretation.

The current version of BPMN (2.0) allows models to consist of nearly 100 graphical elements, covering the description of many categories of tasks, events, errors, areas of responsibility, and general annotations. However, there are essentially only two fundamental types of object in BPMN, *nodes* and *flows*. Nodes represent basic elements of a workflow such as activities performed and decisions made. Flows are links between nodes which express the relationship of one node to another; predominantly expressing the simple ordering in which elements of a workflow are performed, but may also capture message passing or errors. Finally, BPMN diagrams make use of various structural elements which help organise nodes within BPMN models.



(a) State (b) Start (c) End (d) Decision (e) Parallel Gateway (f) Sequence Flow (g) Message Flow (h) Pool

Figure 2: Core BPMN elements

2.1 *Core BPMN*

In this work, only a small subset of BPMN, often known as the *core* subset of BPMN, will be used. It consists of the eight elements found to be the most commonly used in a large survey of real-world BPMN usage by Muehlen and Recker 2008. The graphical elements of core BPMN are shown in fig. 2 and described below.

In core BPMN modelling, a workflow involves composing a number of core BPMN elements into a *business process diagram* (BPD).

Definition 1 (Core BPD)**.** *A core BPD is a tuple* $BPD = (\mathbf{N}, \boldsymbol{\mathcal{F}}, \mathbf{P}, \mathsf{pool}, \mathbf{L}, \mathsf{lab})$ *where* $\mathbf{N} \subseteq \mathbf{T} \cup \mathbf{E} \cup \mathbf{G}$*, is a set of nodes composed of the following disjoint sets:*

- *Tasks* $\mathbf{T}$*, are the basic actions done as part of a given workflow.*
- *Events* $\mathbf{E} \subseteq \mathbf{E^S} \cup \mathbf{E^E}$*, where the disjoint sets* $\mathbf{E^S}$ *and* $\mathbf{E^E}$ *respectively represent start and end events.*
- *Gateways* $\mathbf{G} \subseteq \mathbf{G^D} \cup \mathbf{G^F} \cup \mathbf{G^M}$*, where the disjoint sets* $\mathbf{G^D}$*,* $\mathbf{G^F}$ *and* $\mathbf{G^M}$ *respectively represent exclusive decision gateways, parallel fork gateways and parallel merge gateways.*

$\boldsymbol{\mathcal{F}} \subseteq \mathcal{S} \cup \mathcal{M}$ *is a set of flow relations, where sequence flows* $\mathcal{S} \subseteq \mathbf{N} \times \mathbf{N}$ *relate nodes to each other and* $\mathcal{M} \subseteq \mathbf{T} \times \mathbf{G^M}$ *is a relation between tasks and parallel merge gateways.* $\mathbf{P} \subset \wp(\mathbf{N})$ *is a set of disjoint pools and* $\mathsf{pool} : \mathbf{N} \to \mathbf{P}$ *maps nodes to a pool* $p \in \mathbf{P}$*.* $\mathbf{L}$ *is a set of unique labels and* $\mathsf{lab} : \boldsymbol{\mathcal{F}} \to \mathbf{L}$ *is a labelling function which assigns labels to flows.*

The definition of a BPD given in definition 1 models workflows by using elements of $\boldsymbol{\mathcal{F}}$ to define a directed graph with nodes which are elements of $\mathbf{N}$. However, definition 1 allows for graphs which are unconnected, do not have start or end elements, and are free-form or have various other properties which place them outside what is implied to be permitted in standard BPMN models. To ensure that a BPD describes a meaningful workflow we have developed a set of well-formedness rules, discussed at length in previous work (Herbert & Sharp 2012), which enforce restrictions on connecting elements, pool boundaries, and message passing. These are chosen such that they impose the minimum semantic interpretation necessary to determine the control flow of a model. In the case of a BPD, they add no more semantic interpretation than implied by the standard (Object Management Group 2011).

It should be noted that by combining several Core BPMN elements any element of the complete BPMN language can be simulated. Even inclusive gateways, which pose a challenge as their implied semantics includes a non-trivial and non-local backwards search of the flow graph of the BPD, can be addressed through the work of Christiansen et al. 2011, who present a method to translate this construct into other BPMN processes with minor restructuring of the overall BPD.

2.2 *Stochastic Core BPMN with Rewards*

BPMN makes use of external conditions on decision gateways to select the outgoing flow from a decision point. These decisions are modelled by the set $\mathbf{L}$ and assigned to specific flows by the function lab introduced in definition 1. In practice, decision points in a workflow will have outcomes which depend on some inherent property of the task or on outside factors. The idea is that at a decision point an active choice is made, and then that choice results in a number of different possible probabilistic outcomes. This behaviour, which is similar to a Markov decision process (White 1993), preserves the intention of actors in a process while enabling probabilistic behaviour, and can be effectively captured by annotating the possible outcomes of specific decisions with pairs of labels and probabilities (l, p). We employ the following function to ensure meaningful assignment of these intention-preserving probabilistic annotations.

Definition 2 (Gateway Flow Probability Function)**.** *A decision gateway probability function is a partial function* $\mathcal{P}_g : \mathcal{S} \times \mathbf{L} \to [0, 1]$ *which for a node* $g \in \mathbf{G^D}$ *and label* $l \in \mathbf{L}$ *assigns probabilities to all outgoing sequence flows* $((g, x), l)$*, such that for a given* l*:*

$$\sum_{\forall x \in \mathsf{out}(g)} \mathcal{P}_s((g, x), l)) = 1$$

Definition 2 ensures that all decision gateways have an associated probability and that the sum of all probabilities for a given label l is 1. Figure 3 illustrates the application of $\mathcal{P}$ to a decision gateway g.

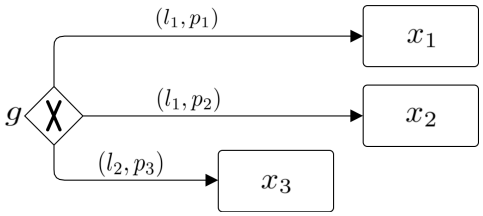


Figure 3: Assignment of label probability pairs to a decision gateway. Here application of $\mathcal{P}$ requires $p_1 + p_2 = 1$ and $p_3 = 1$

To enable quantitative analysis of a workflow we add numerical data to our models by using the following function which associates positive real numbers with tasks in a BPD.

Definition 3 (BPD Task Reward Function)**.** *For a BPD a reward function for a task $t \in \mathbf{T}$ is a partial function $\mathcal{R} : \mathbf{T} \to \mathbb{R}_{\geq 0}$.*

This function captures the notion that certain nodes have some reward or cost associated with the task. There is no practical distinction between costs and rewards, and we can use these annotated values to keep track of whichever quantities may be of interest in a process. We may associate as many reward structures as we wish with a given BPD, so that a single task may have multiple different numerical properties which are incremented when the task is performed. Further details of these structures and model checking of these properties can be found in Herbert & Sharp 2012.

3 MODELLING FAULTS

To allow the automated generation of fault trees, we extend the definition of tasks, and the relations between them, in a BPD to include two types of faults. The approach taken can be seen as employing special gateway flow probability functions that direct workflow execution to special faulty task states.

Note that faulty decision gateways in a workflow can be modelled simply by redefining gateway flow probability function $\mathcal{P}$ associated with a gateway.

3.1 *Fault State Generation*

We will allow for the addition of faults by means of the following definitions:

Definition 4 (Fail-Stop Task Fault Injection Function)**.** *For a task $t_n \in \mathbf{T}$ in a BPD, the partial function $\overset{\twoheadrightarrow}{\mathcal{E}} : \mathbf{T} \times [0,1] \to (\mathbf{T} \times \mathbf{T}) \times [0,1]$ adds a fail-stop execution sequence to a BPD as follows:*

$$\overset{\twoheadrightarrow}{\mathcal{E}}(t_n, p) = \begin{cases} t_{n-1}\mathcal{S}\overset{\twoheadrightarrow}{t_n} & \textit{with probability } p \\ t_{n-1}\mathcal{S}t_n & \textit{with probability } 1-p \end{cases}$$

Application of definition 4 to add *fail-stop* behaviour to a task is illustrated in fig. 4. Here, after task t_{n-1} has been performed, then with a probability of $P_{\overset{\twoheadrightarrow}{t_n}}$ the task t_n is not performed and instead a transition $t_{n-1}\mathcal{S}\overset{\twoheadrightarrow}{t_n}$, to a state $\overset{\twoheadrightarrow}{t_n}$ representing the process halting (deadlocking) during execution of t_n is made. We denote the set of all fail-stop tasks as $\overset{\twoheadrightarrow}{\mathbf{T}}$.

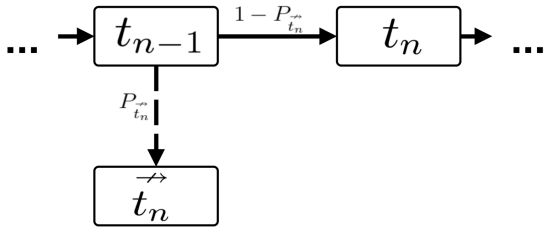


Figure 4: Fail-Stop behaviour

Definition 5 (Fail-Continue Task Fault Injection Function)**.** *For a task $t_n \in \mathbf{T}$ in a BPD, the partial function $\overset{\curvearrowright}{\mathcal{E}} : \mathbf{T} \times [0,1] \to (\mathbf{T} \times \mathbf{T}) \times (\mathbf{T} \times \mathbf{T}) \times [0,1]$ adds a fail-continue execution sequence to a BPD as follows:*

$$\overset{\curvearrowright}{\mathcal{E}}(t_n, p) = \begin{cases} t_{n-1}\mathcal{S}\overset{\curvearrowright}{t_n}\mathcal{S}t_{n+1} & \textit{with probability } p \\ t_{n-1}\mathcal{S}t_n\mathcal{S}t_{n+1} & \textit{with probability } 1-p \end{cases}$$

Application of definition 5 to add *Fail-continue* behaviour to a task is shown in fig. 5. In this case, after task t_{n-1} has been performed, then with a probability of $P_{\overset{\curvearrowright}{t_n}}$ the task t_n is not performed and instead a sequence of transitions $t_{n-1}\mathcal{S}\overset{\curvearrowright}{t_n}\mathcal{S}t_{n+1}$ passing through state $\overset{\curvearrowright}{t_n}$ representing the task being performed in some faulty, but not deadlocking, fashion. The set of all fail-continue tasks will be denoted as $\overset{\curvearrowright}{\mathbf{T}}$.

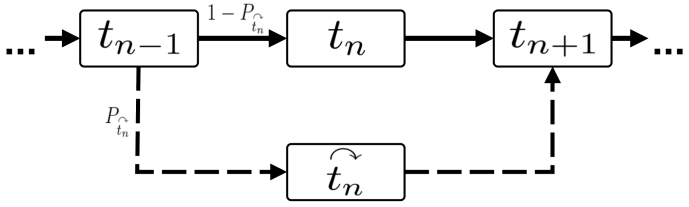


Figure 5: Fail-Continue behaviour

Note that the total application of $\overset{\twoheadrightarrow}{\mathcal{E}}$ and $\overset{\curvearrowright}{\mathcal{E}}$ produces the maximal set of possible fault states, where all tasks can exhibit both fail-stop and fail-continue behaviour. In practice it can be beneficial to restrict the set of states believed to be prone to failure to eliminate *false-positive* failures that are impossible in the system. However, total application of $\overset{\twoheadrightarrow}{\mathcal{E}}$ and $\overset{\curvearrowright}{\mathcal{E}}$ is certain to find all failure states.

4 GENERATING FAULT TREES

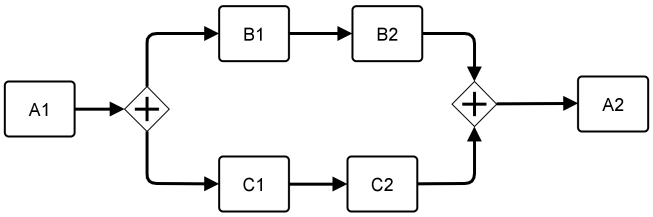
Producing a fault tree involves systematically building all possible chains of one of more fail-continue

faults that can occur in the system. It is therefore necessary to examine all specific systems configurations which are possible during execution. These state-spaces can become extremely large and model checking encompasses the current state of the art methods for efficiently searching this space. Specifically, we employ the model checker *PRISM* (Kwiatkowska et al. 2011) to perform this search.

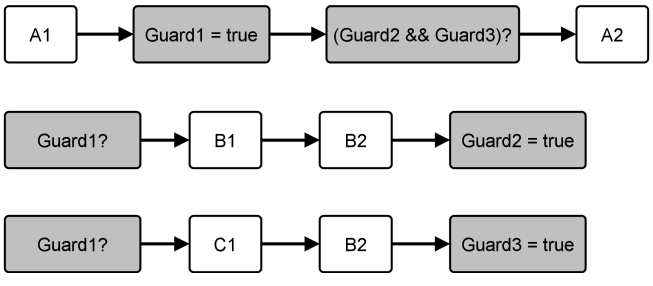
4.1 *Generating PRISM Code*

The theoretical details of conversion of BPMN models to PRISM model code are comprehensively covered in previous work (Herbert & Sharp 2012). The central idea is to identify sub-processes of the source Core BPMN model, and then map these to modules of PRISM code so that the encoding would be compositional, and will not impose further semantic interpretation on the source BPMN model.

We employ a two pass algorithm to build lists of pairs of linked nodes at each level of depth of nested parallel fork gateways in the source model. We record which pairs of nodes are present at a given parallel fork gateway nesting depth of the BPDs pool. With this list defined, the second pass translates a BPD into PRISM code by constructing PRISM modules where the corresponding fork and merge gateways at a given depth are retained in both the level above and below the parallel fork depth. Such that in the outer containing level a fork gateway is linked directly to a merge gateway. The concurrent synchronisation between modules is achieved by generating appropriately named PRISM *actions* that enforce synchronisation between modules, effectively simulating concurrency by generating the set of all possible interleavings of all parallel tasks. An illustration of this process for a simple pair of parallel fork and merge gateways is shown in fig. 6(a) and the resulting three guarded PRISM modules are shown in fig. 6(b).



(a) BPMN parallel process



(b) PRISM Modules

Figure 6: Illustration of the BPMN to PRISM translation process for parallel sequences.

When all pools are converted into appropriate sets of modules, messages between pools are identified in the form of dependent pairs of states, and appropriate synchronising PRISM actions between them are generated in a similar fashion. In addition, reward structures are extracted from the BPD as they are encountered and added to corresponding PRISM reward elements which associate rewards with specific configurations of the model.

4.2 *Fault tree construction*

Fault trees provide a convenient symbolic representation of the combination of faults causing a system failure, and they are represented as a parallel or sequential combination of logical *AND* and *OR* gates. Algorithm 1 employs the PRISM model checker to construct a fault tree, by means of executing queries in Probabilistic Computation Tree Logic (PCTL) (Aziz et al. 1995), based on classical continuous stochastic logic (Hansson and Jonsson 1994) extended to probabilistic quantification of described properties. PCTL is a rich temporal logic with a wide range of operators used to reason about properties of paths through a state-space. However, to construct fault trees we will rely only on the following operators (Kwiatkowska et al. 2011):

- The $\mathsf{P}_{=?}$ operator refers to the probability of an event occurring, more precisely, the probability that the observed execution of the model satisfies a given specification.
- The $\mathsf{R}^{l}_{=?}$ operator is used to express properties that relate to rewards, more precisely, the expected value of a random variable, associated with particular reward structure. To distinguish different reward structures we employ a label l.
- The binary *until* operator $a \mathsf{U} b$ specifies that, for a given path, in some state of the path the property b is true and in all preceding states the property a is true.
- The *eventually* operator $\mathsf{F}a$ specifies that, for a given path, a eventually becomes true at some point along the path.

In algorithm 1 a fault tree is constructed by performing queries of the probabilities of, and the values of *rewards* of interest, in each possible system configuration where one or more faults have occurred. In essence these calculations are made for every possible subsequence of elements of $\overset{\curvearrowright}{\mathbf{T}}$ appended with every element from $\overset{\rightharpoonup}{\mathbf{T}} \cup \emptyset$. As the probabilities of every possible meaningful combination of faults are generated, nodes of the faults tree are constructed and annotated with the calculated rewards values.

Algorithm 1 is guaranteed to terminate as it operates over a finite set. It has an upper complexity bound

Algorithm 1: Fault Tree Generation

Input: BPD annotated via $\mathcal{P}$, $\mathcal{R}$, $\vec{\mathcal{E}}$ and $\overset{\curvearrowright}{\mathcal{E}}$ functions.
Output: An exhuastive fault tree FT for the input BPD

Let $\mathbf{S}$ be the set of U separated sub-strings of $\overset{\curvearrowright}{\mathbf{T}}$
for $s \in \mathbf{S}$ **do**
  $\mathbf{Q} \leftarrow \mathbf{Q} \cup s$
  for $t \in \vec{\mathbf{T}}$ **do**
    $\mathbf{Q} \leftarrow \mathbf{Q} \cup (s \text{ append } \mathsf{U}t)$
for $q \in \mathbf{Q}$ **do**
  for $l \in \mathbf{Labels}$ **do**
    $p \leftarrow \mathsf{P}_{=?}[\mathsf{F}q]$
    $r \leftarrow \mathsf{R}^{l}_{=?}[\mathsf{F}q]$
  if *q is length 1* **then**
    Add OR node to Fault Tree FT with probability p and rewards r
  else
    Add AND node to Fault Tree FT with probability p and rewards r
Return FT

of $O(|\overset{\curvearrowright}{\mathbf{T}}|! \cdot |\vec{\mathbf{T}}|)$, which while large is still feasible as all queries are performed on the same state space generated by the PRISM model checker, and the construction of the statespace by PRISM still dominates the computational burden.

5 EXAMPLE

A straightforward example of a workflow that can be described using BPMN is shown in figure 7.

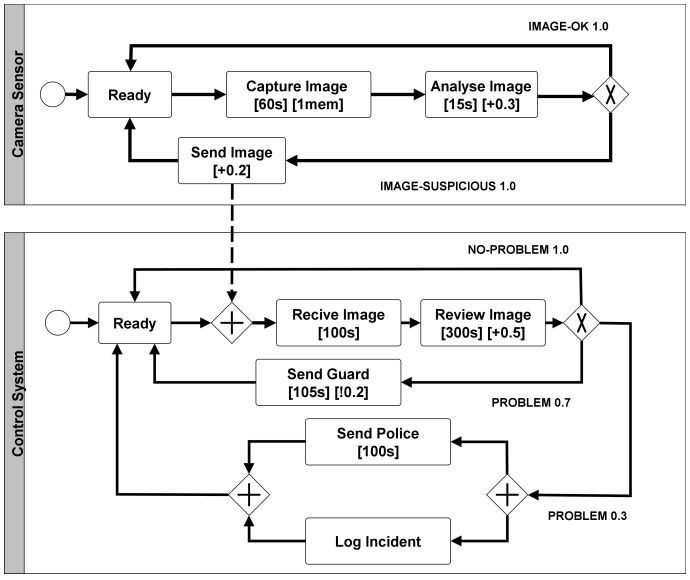


Figure 7: BPMN model of an unreliable "Smart" security camera system

In this example we have two BPMN pools modelling a security camera and control centre which responds to images sent from the camera. The camera process begins by entering a ready state and then proceeds to perform a capture image task, which is annotated with reward structures to indicate the time taken (60 seconds) and memory consumed (1 unit of memory). After that, the camera analyses the image to determine if there is suspicious activity. This process is similiarly annotated with a time reward, but also marked with a 0.3 probability of experiencing a fail-continue behaviour (indicated by the $+$ symbol), e.g. the image buffer could be corrupted. The camera process proceeds to make a choice if the image is suspicious or not. If the image is not suspicious then, with a probability of 1, the camera process loops back to the ready state. If the image is suspicious the message is sent to the control system, a task which may also be faulty.

The control system, after entering its ready state, will only progress to the receive image state when an image has been received. The image is then reviewed by a human operator who decides whether there is an actual problem. In the case where there is a problem, a non-deterministic choice is made between calling a guard or calling the police, with probabilities of respectively 0.7 and 0.3. In the case where a guard is sent there is a possibility of fail-stop behaviour, indicated by ! symbol and an associated probability of 0.2 that the guard does not respond to the call.

For this system, determining the fault tree of the possible combined faults that could occur allows us to obtain knowledge about how this system can fail to maintain security. The fault tree shown in fig. 8 determines probabilities, the minimum time taken and the expected amount of memory used for every possible failure, and combination of failures, encoded in the system model. The combination of failure probabilities combined with quantitative system performance data determined by this analysis allows systems designers to adjust the system reliability level. They can see the effect these changes will have on performance directly from a industry standard system model with just a few annotations.

6 CONCLUDING REMARKS

We have presented the central theoretical ideas of a method for the direct generation of fault trees from systems models in BPMN. Enhancing the original formal risk analysis concept of Liggesmeyer and Rothfelder 1998 with direct translation from a industry standard modelling language and allowing for the addition of data values at points of failure in a fault tree. For the cost of a small amount of additional data added to a workflow model we are able to automatically derive rich fault trees with no user interaction needed.

The method is computationally feasible and we have built a prototype implementation of these ideas, where some algorithm 1 is implemented with some

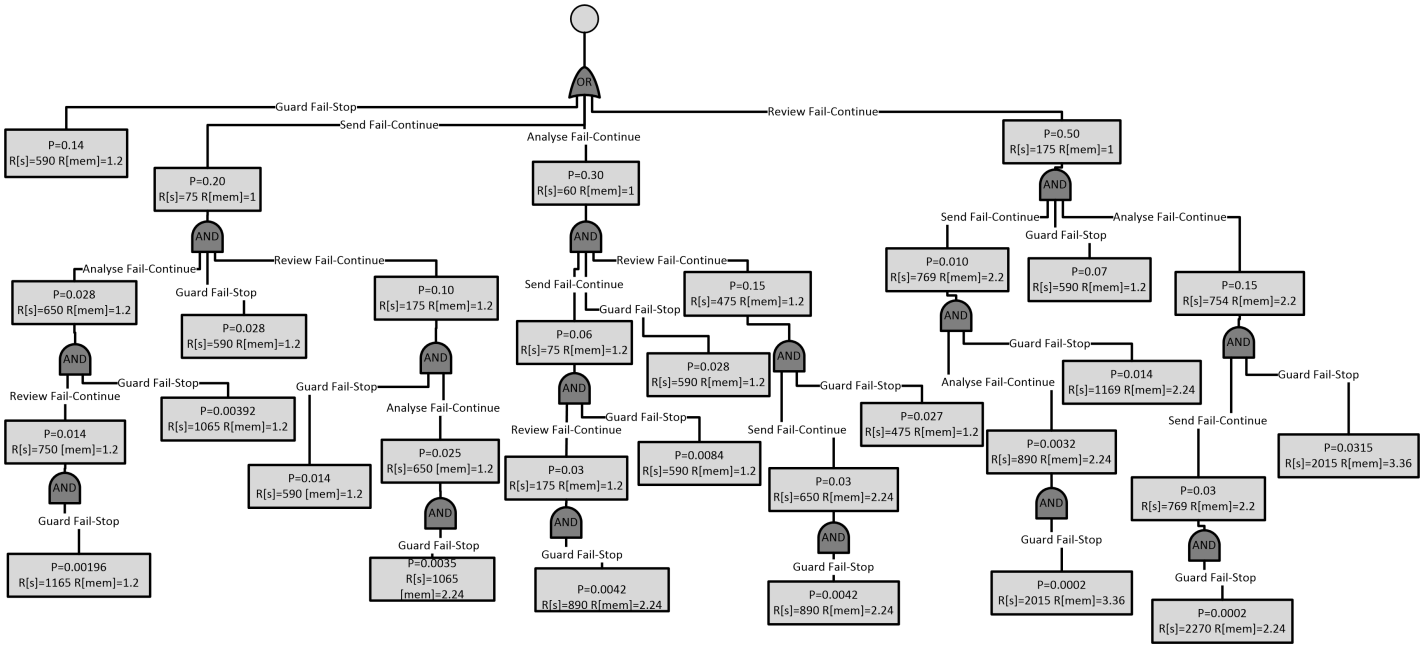


Figure 8: Generated fault tree for the example shown in fig. 7

practical tuning. This system, which is based upon the PRISM model checker, is able to generate fault trees for BPMN models with a total number of states, including fault states, of up to 10^{10} (Kwiatkowska et al. 2011). The full details of this tool will be reported elsewhere (Herbert & Sharp 2013c).

While this paper focused on BPMN, we have developed a similar formalisation of UML statecharts (Herbert & Sharp 2013a) which could equally well have been used to illustrate the automatic derivation of fault trees. Indeed, most workflow modelling languages which are fundamentally based on a graph structure should be amenable to extension with stochastic behaviour, rewards and faults, and thus possible to analyse with the approach described.

6.1 *Future Work*

A novel approach by Mukherjee et. al. (Mukherjee & Chakraborty 2007) to generating fault trees from maintenance logs can be greatly enhanced by the work presented here. Specifically, our methods allow systems models with a considerably greater degree of freedom in their behaviour to be built from logs prior to fault tree generation creating fault trees which better reflect the system and which include data.

The real-valued rewards introduced here can be expanded to include rewards which functions which are dependent on the variables of the overall system state. These rewards can be analysed with PRISM with only a modest complexity increase. Allowing for fault trees where nodes are annotated with functions which describe the evolution of the reward at different times when the failure can occur, e.g. in the example given we would have not just the minimum time until a fault happens but all the times of the cyclically occurring moments when failure is possible.

Handling faults in message passing between processes poses a similar set of problems to those explored in this paper. In this case lost messages and also situations where messages are delayed and may arrive out of sequence must be accounted for. In future work we will present an approach to these faults by constructing queues of lost messages which have associated probabilities of being sent each time a transition is made in the system. Further messages can be received by different recipients than intended. Combined with advanced rewards, this allows for modelling and deriving fault trees for considerably more complex systems.

REFERENCES

Aziz, A., V. Singhal, F. Balarin, R. K. Brayton, & A. L. Sangiovanni-Vincentelli (1995). It usually works: The temporal logic of stochastic systems. In P. Wolper (Ed.), *Proceedings of the 7th International Conference on Computer Aided Verification*, Volume 939 of *Lecture Notes in Computer Science*, pp. 155–165. Berlin, Heidelberg: Springer-Verlag.

Banach, R. & M. Bozzano (2011). The mechanical generation of fault trees for reactive systems via retrenchment ii: clocked and feedback circuits. *Formal Aspects of Computing Oct*, 1–49.

Chinosi, M. & A. Trombetta (2012, January). BPMN: An introduction to the standard. *Computer Standards & Interfaces 34*(1), 124–134.

Christiansen, D. R., M. Carbone, & T. Hildebrandt (2011). Formal semantics and implementation of BPMN 2.0 inclusive gateways. In *Proc. of the 7th international conf. on Web services and formal methods*, Web Services and Formal Methods 2010, Berlin, Heidelberg, pp. 146–160. Springer-Verlag.

Crockford, N. (1986). *An Introduction to Risk Management* (2nd Edition ed.). Cambridge England: Woodhead-Faulkner.

Ericson, C. A. (1999). Fault tree analysis - a history. In *Proceedings of the 17th International System Safety Conference*, ISSC'99, Unionville, Virginia, USA, pp. 1–9. System Safety Society.

Ericson, C. A. (2005). Fault tree analysis. In C. A. Ericson (Ed.),

Hazard Analysis Techniques for System Safety, pp. 183–221. New Jersey, USA: John Wiley & Sons, Inc.

Fränzle, M. & M. R. Hansen (2008, November). Efficient model checking for duration calculus based on branching-time approximations. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '08, Washington, DC, USA, pp. 63–72. IEEE Computer Society.

Hansson, H. & B. Jonsson (1994). A logic for reasoning about time and reliability. *Formal Aspects of Computing 6*(5), 512–535.

Herbert, L. & R. Sharp (2012, October). Using stochastic model checking to provision complex business services. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pp. 98–105.

Herbert, L. & R. Sharp (2013a, July). Optimised safe execution strategies for UML statecharts. In *Proceedings of the International SPIN Symposium on Model Checking of Software (SPIN) 2013*. (Forthcoming 2013 SPIN Symposium).

Herbert, L. & R. Sharp (2013b, March). Precise quantitative analysis of probabilistic BPMN workflows. *Journal of Computing and Information Science in Engineering 13*, 2–11.

Herbert, L. & R. Sharp (2013c). SBAT a stochastic BPMN analysis tool. In *Proceedings of the 18th International Workshop on Formal Methods for Industrial Critical Systems*. (Forthcoming 2013 FMICS Workshop).

Kwiatkowska, M. Z., G. Norman, & D. Parker (2011). PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer (Eds.), *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, Volume 6806 of *Lecture Notes in Computer Science*, London, UK, pp. 585–591. Springer-Verlag.

Liggesmeyer, P. & M. Rothfelder (1998, June). Improving system reliability with automatic fault tree generation. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pp. 90–99.

Muehlen, M. Z. & J. Recker (2008). How much language is enough? theoretical and practical use of the business process modeling notation. In *Proc. of the 20th international conf. on Advanced Information Systems Engineering*, Conference on Advanced Information Systems Engineering 2008, Berlin, Heidelberg, pp. 465–479. Springer-Verlag.

Mukherjee, S. & A. Chakraborty (2007, January). Automated fault tree generation: Bridging reliability with text mining. In *Reliability and Maintainability Symposium, 2007. RAMS '07. Annual*, pp. 83–88.

Object Management Group (2011, January). Business process model and notation (BPMN) 2.0. Standards Document formal/2011-01-03, Object Management Group, Needham MA, USA.

Roy, S., A. Sajeev, S. Bihary, & A. Ranjan (2013). An empirical study of error patterns in industrial business process models. *IEEE Transactions on Services Computing 99*(PrePrint).

Schäfer, A. (2003). Combining real-time model-checking and fault tree analysis. In K. Araki, S. Gnesi, and D. Mandrioli (Eds.), *Proceedings of the 2003 International Symposium of Formal Methods Europe, FME 2003*, Volume 2805 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, pp. 522–541. Springer-Verlag.

Stephans, R. A. (2005). Fault tree analysis. In R. A. Stephans (Ed.), *System Safety for the 21st Century*, pp. 169–188. New Jersey, USA: John Wiley & Sons, Inc.

Thums, A. & G. Schellhorn (2003). Model checking fta. In K. Araki, S. Gnesi, and D. Mandrioli (Eds.), *Proceedings of the 2003 International Symposium of Formal Methods Europe, FME 2003*, Volume 2805 of *Lecture Notes in Computer Science*, pp. 739–757. Berlin, Heidelberg: Springer-Verlag.

White, D. J. (1993). *Markov decision processes*. John Wiley & Sons.

Xiaocheng, G., R. Paige, & J. McDermid (2010, June). Analysing system failure behaviours with PRISM. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pp. 130–136.