A Compositional Semantics for Repairable BDMPs

Shahid Khan[1](✉), Joost-Pieter Katoen[1], and Marc Bouissou[2]

[1] Software Modeling and Verification, RWTH Aachen University, Aachen, Germany
{shahid.khan,katoen}@cs.rwth-aachen.de

[2] EDF-R&D, Electricité de France, Palaiseau, France
marc.bouissou@edf.fr

Abstract. Boolean-logic Driven Markov Processes (BDMPs) is a graphical language for reliability analysis of dynamic repairable systems. Simulation and trace-based analysis tools for BDMPs exist and have been used to analyze reliability, safety and security aspects of industrially relevant case studies. To enable a model-based analysis of BDMPs, such as probabilistic model checking, formal semantics is indispensable. This paper presents a rigorous semantics to repairable BDMPs using Markov automata (MA), a variant of continuous-time Markov chains (CTMCs) with action transitions. The semantics is modular: an MA is associated with each BDMP element and these are combined to obtain an automaton for the entire BDMP. By ignoring the actions that are used to "glue" the MA of BDMP elements, a CTMC is obtained that is amenable to analysis by e.g., model checking. We report on a prototypical implementation and experimentally show that our semantics corresponds to the BDMP interpretation by the tool Yet Another Monte Carlo Simulation.

Keywords: Reliability · Dependability · Formal methods · Probabilistic model checking · Monte-Carlo simulation · Compositional analysis

1 Introduction

Static Fault Trees. Fault trees [18] are used for safety and reliability engineering in many application areas. *Static Fault Trees* (SFTs) are the simplest; their leaves, called *basic events* (BEs), model individual component failures or human errors. The failure times are governed by continuous probability distributions. Internal nodes, called *gates*, model how component failures lead to system failures. Gates are like logic circuit elements such as AND and OR, both instances of the voting (VOT) gate. Fault tree analysis amounts to determine the failure probability of the root of the fault tree, called the top event. SFTs are simple; the ordering of failures is irrelevant and repairs are excluded.

S. Khan—Supported by a HEC-DAAD Scholarship.

Extensions. SFTs have been extended in numerous ways, e.g., with priority-AND PAND gates [20], by dynamic fault trees (DFTs) [9] possibly extended with repairs [6,11], state-event fault trees (SEFTs) [14], component fault trees (CFTs) [15], Boolean logic Driven Markov Processes (BDMPs) [5] and a generalisation thereof [17]. These extensions are driven by the need to model e.g., (a) the replacement of failed components by spare ones, (b) hot and cold redundancies, (c) complex failure orderings, (d) repairs and maintenance strategies, or (e) the failure of a component by going through several degraded modes.

The Need for Semantics. The added expressive power leads to more modeling flexibility but comes at a price: the interpretation of these graphical fault tree languages becomes involved. (Also the analysis is more complex; e.g., maximal cut set computations—the main technique for SFTs—no longer suffice.) The interpretation of fault trees is not just of theoretical interest. Slightly different interpretations may lead to significantly divergent reliability measures and give rise to distinct underlying stochastic (decision) processes. This issue is discussed in detail for DFTs in [13]. Moreover, a rigorous semantics is a prerequisite to enable the analysis of fault trees using multiple tools. Our overall aim is to analyze Bouissou's BDMPs [5] by means of probabilistic model checking [1]. It has been shown in the last years that such an approach is quite successful for analyzing DFTs [11,19]. This paper, therefore, focuses on *providing a rigorous model-based semantics to BDMPs.*

BDMPs in a Nutshell. BDMPs are used in the probabilistic safety assessment of nuclear power plants. Two major mechanisms in BDMPs are *triggers* and *triggered Markov processes*. The triggers model *activation*: if the source of a trigger fails, then the target of the trigger is activated, i.e., woken from a stand-by mode, provided at least one of its parents is activated. Triggered Markov processes are pairs of Markov chains associated with BEs: one describes the behavior when being activated while the other considers the case when de-activated. BE can thus be in four states: working or failed in standby, or working or failed in the activated mode. BDMPs facilitate repairs by transiting from a failed to a working state. BDMPs allow modeling state-dependent failures, an aspect that is not supported by SFTs. BDMPs can be analyzed by the discrete-event simulator YAMS and the trace-based analysis tool FIGSEQ [4].

Contributions of This Paper. The main contribution is a *formal semantics* of BMDPs. The semantics is operational: we map each BDMP onto a *Markov automaton* [10]. This semantics assumes all continuous failure distributions to be negative exponentials. It covers two versions of triggers and takes triggered Markov processes as basic events. It thus includes repairs. This complicates matters a bit, as—in contrast to SFTs and classical (non-repairable) DFTs—the property is lost that once a sub-tree has failed it remains to do so. Many existing semantics of extensions of SFTs do not contain repairs; a notable exception is recent work on rare event simulation of DFTs with repairs [6]. Our semantics is *modular*, following the compositional approach for DFTs in [3]. This entails that each BDMP construct, i.e., basic event, gate (such as VOT and priority AND), triggers, etc. are mapped onto a single Markov automaton (MA). The MA for the entire BMDP is obtained by composing the constituting MA together. In order to enable this composition, the MAs for the BDMP elements are equipped with extra signals (a.k.a. *actions*). Once the entire MA of a BDMP is obtained, the actions are not of any further use and are abstracted away.

A prototypical implementation of the semantics has been made. It constructs the MA in a modular way as defined in this paper using the tool `moconv`[1] and uses the probabilistic model checker Storm [8] for reliability and availability analysis. Our semantics is validated by comparing the analysis results of our prototype with the simulation results of the BDMP analysis tool Yams [4]. (In the absence of any other formal semantics of BDMPs, this is the best we can hope for.) We stress that the focus of this work has not been to define a memory-efficient semantics—the peak memory consumption of a compositional approach can be substantial [19]. The modular approach, however, is conceptually simple, easily reveals the intricacies of some of the individual BMDP elements and can easily be extended with new types of gates. We, therefore, believe that our semantics increases the understanding of BDMPs and can provide the basis for more efficient state-space generation techniques for BDMPs, e.g., our semantics is amenable to partial-order reduction techniques for MA. The main contributions of this paper are:

1. A compositional, operational semantics of repairable BDMPs.
2. A prototypical implementation of this semantics.
3. An experimental validation of the semantics by comparing analysis results.

Organization of the Paper. Section 2 discusses the formal model, Markov Automata (MA). The components of BDMPs and their semantics are detailed in Sect. 3. The proposed prototypical implementation along with experimental results are discussed in Sect. 4. Section 5 concludes the paper and discusses some future work.

2 Markov Automata

Markov automata are a mathematical model that support discrete probability distributions, exponential delays, non-determinism among choices, and parallel composition. MA subsume DTMCs, CTMCs, CTMDPs and PAs as detailed by [12]. Here we outline the theory of MA necessary to understand the present work. We use $\mathbb{R}$ for the set of real numbers, μ for a distribution over the set S $\mu\colon S \rightarrow [0,1]$ such that $\sum_{s\in S}\mu(s) = 1$, $\mathit{Dist}(S)$ for the set of discrete probability distributions over the set S, $\mathit{supp}(\mathit{Dist})$ for the support of distribution function Dist, and $\perp$ ($\top$) for FALSE (TRUE). Syntactically, a Markov automaton $\mathcal{M}$ is a tuple $(S, \mathit{Act}, \rightarrow, \dashrightarrow, s_0)$ where S is a finite set of states, Act is a finite set of actions, $\rightarrow \subseteq S \times \mathit{Act} \times \mathit{Dist}(S)$ is a set of action transitions, $\dashrightarrow \subseteq S \times \mathbb{R}_{>0} \times S$

[1] The Modest Toolset: http://www.modestchecker.net/.

is a set of Markovian transitions, and $s_0 \in S$ is an initial state. Semantically speaking, if an action a can be performed from a state s such that $(s, a, \mu) \in \rightarrow$ (we write $s \xrightarrow{a} \mu$), then the probability of moving to state $s' \in S$ from state s is $\mu(s')$. Moreover, in case of a Markovian transition $s \overset{\lambda}{\dashrightarrow} s'$, this transition can be performed within time t with a probability which is exponentially distributed with rate λ, i.e., $(1 - e^{-\lambda \cdot t})$. The states of MA are called *Markovian* (iff having only Markovian outgoing transitions), *interactive* (iff having only probabilistic outgoing transitions), *deadlock* (iff having no outgoing transition), or *hybrid* (otherwise). The *exit* rate of a *Markovian* state s is computed as $E(s) = \sum_{s' \in S} R(s, s')$, where $R(s, s') = \sum\{\lambda | s \overset{\lambda}{\dashrightarrow} s'\}$ is the *rate* between state s and s'. The probability of leaving s is $1 - e^{-E(s) \cdot t}$. If s has more than one successor state, then there exists a race between such states after leaving state s; the probability of s' winning the race equals $P(s, s') = \frac{R(s,s')}{E(s)}$.

The parallel composition (denoted $\|$) of two MA $\mathcal{M}_i = (S_i, Act_i, \rightarrow_i, \dashrightarrow_i, s_{0,i})$, where $i \in \{1, 2\}$ w.r.t. $A = Act_1 \cap Act_2$ can be formally defined as the MA $\mathcal{M}_1 \parallel \mathcal{M}_2 = (S_1 \times S_2, Act_1 \uplus Act_2, \rightarrow, \dashrightarrow, (s_{0,1}, s_{0,2}))$, where $\rightarrow$ and $\dashrightarrow$ are defined as the smallest relations satisfying the following six rules (R_1 through R_6):

$$\frac{s_1 \xrightarrow{\alpha}_1 \mu_1 \quad \alpha \notin A}{(s_1, s_2) \xrightarrow{\alpha} \mu_1 \cdot \{s_2 \mapsto 1\}} R_1 \quad \frac{s_2 \xrightarrow{\alpha}_2 \mu_2 \quad \alpha \notin A}{(s_1, s_2) \xrightarrow{\alpha} \{s_1 \mapsto 1\} \cdot \mu_2} R_2 \quad \frac{s_1 \xrightarrow{\alpha}_1 \mu_1 \quad s_2 \xrightarrow{\alpha}_2 \mu_2 \quad \alpha \in A}{(s_1, s_2) \xrightarrow{\alpha} \mu_1 \cdot \mu_2} R_3$$

$$\frac{s_1 \overset{\lambda}{\dashrightarrow}_1 s_1' \quad s_1 \neq s_1'}{(s_1, s_2) \overset{\lambda}{\dashrightarrow} (s_1', s_2)} R_4 \quad \frac{s_2 \overset{\lambda}{\dashrightarrow}_2 s_2' \quad s_2 \neq s_2'}{(s_1, s_2) \overset{\lambda}{\dashrightarrow} (s_1, s_2')} R_5 \quad \frac{s_1 \overset{\lambda_1}{\dashrightarrow}_1 s_1 \quad s_2 \overset{\lambda_2}{\dashrightarrow}_2 s_2}{(s_1, s_2) \overset{\lambda_1+\lambda_2}{\dashrightarrow} (s_1, s_2)} R_6$$

In natural language, rules R_1 and R_2 state that an MA can independently take any action not in the common action set. Rule R_3 states that both MA must progress synchronously on the common action α. Rule R_4 and R_5 define that no synchronization takes place on Markovian transitions. The last rule R_6 defines that the rates of self-loops are added in parallel states. The composition operator ($\|$) for MA is commutative and associative. Thus, the order of composition among n MA does not matter. MA is called *open* if it can be composed with another MA. Once all composition is done, the MA is called *closed*. This paper takes a state-based view of Markov automata for model checking and actions are only required for parallel composition. This implies that all actions in *closed* MA are turned into *invisible* actions (τ) and *maximal progress* assumption can be applied thereafter. The *maximal progress* property states that if Markovian and action-based transitions are enabled simultaneously in a state, then the MA will always follow the latter and the former can be removed from the MA.

We do not have input and output actions (as adopted by [3]) in our framework. The composition is done over common *alphabets* as per rule R_3 using synchronization vectors of the form $\langle a, a \rangle \mapsto a$ (both MA must synchronize on action a and this action will behave as a in resultant composed MA) as detailed in [12]. This rule can be extended to an arbitrary number of MA and intuitively speaking, all MA having a shared action must synchronize to perform this action.

In order to enable model checking on Markov automata, we equip states with *atomic propositions* and introduce *variables* in the MA. The *variable expressions*, defined on top of these *variables*, represent visible aspects of the system and can be considered as *labeling* functions returning *atomic propositions* of each state. The action transitions can be equipped with *guards* (predicates over the *variables*) and *updates* (*variable* assignments).

Example 1. These concepts are summarized in Fig. 1 by parallel composition of two MA; (a) and (b). Where (a) has one transition guarded by when(⊥), this guard makes the transition impossible (we use gray color to highlight impossible transitions). This guard also makes the a_1 labeled action transition of MA (b) impossible which was supposed to synchronize with this impossible transition of (a). We associate one *atomic proposition* L with state p_1 of (a) (double circle is used to distinguish this state from the other states). The treatment of L through different stages of parallel composition can be followed. As a convention, we drop the distribution part of the transition whenever there is only one reachable state after an action, i.e., $supp(Dist(S)) = 1$. Consequently, we have dropped the probabilistic part for both actions a_0 and a_2 in $(a) \parallel (b)$, see Fig. 1(c). Conversion to invisible labels (τ) is shown in Fig. 1(c) and *closed* MA after removing all action-labeled transitions is shown in Fig. 1(d). The selection of Markovian transition rate λ was deliberate to highlight the application of rules R_4 and R_5.

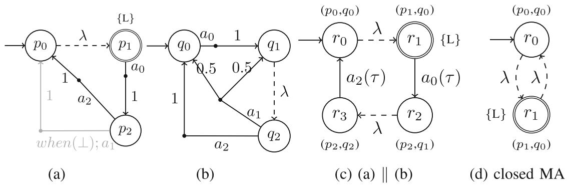


Fig. 1. MA example

3 BDMPs and Their Semantics

BDMPs [5] syntactically extend SFTs by one new element called *trigger* (Trig). The syntax of this formalism is very flexible. The triggers can have any type of node as their source and target. There are three syntactic restrictions: (1) trigger source and origin cannot be the same, i.e., trigger loops are not allowed; (2) Top cannot be the target of a trigger; (3) two triggers cannot target the same node. However, the semantics of this language is quite involved. For instance, there are four variants of *triggers* available in BDMPs. The name “logic-driven” stems from the fact

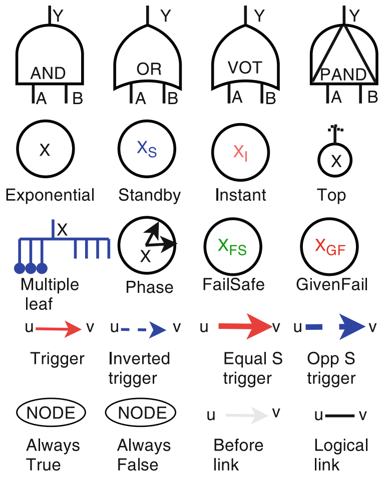
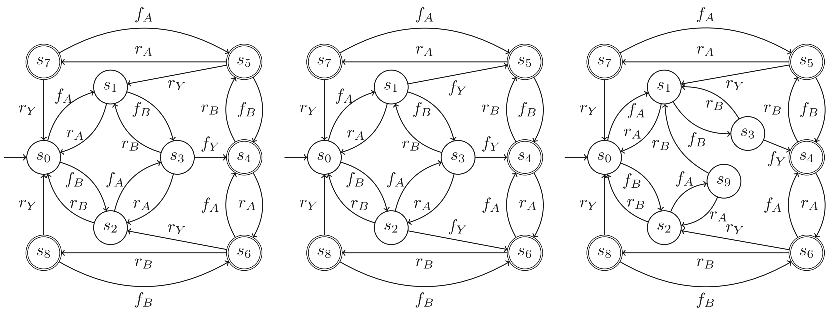


Fig. 2. BDMP elements



(a) MA of AND gate (b) MA of OR gate (c) MA of PAND gate

Fig. 3. MA of the BDMP gates

that predicates are used to switch between the BE modes, e.g., from *standby* to *active* mode. The structure-function SF and activation-function AF are used to control the failure and activation mechanisms in a BDMP. In the original paper [5], a predicate (*trimming*) was used to do computational optimization and to incorporate don't-care propagation assumption. The don't-care propagation assumption is based on the view that the components of an already failed subsystem can not fail while the system is being repaired. This assumption drastically reduces the state space explosion and is close to reality. For the sake of understandability, in this paper, we only focus on the activation behavior of BDMPs and do not consider this trimming and any other optimizations for BDMPs [5].

Compositional Semantics. In our compositional approach, we use actions to model the failure and mode switching mechanism of BDMPs. We introduce four actions namely active (a), de-active (d), fail (f) and repaired (r) to inform the rest of the system when a component is *active*, *standby*, *failed* or *repaired*, respectively. These four actions are enough to model BDMP elements. We define two MA for each BDMP element corresponding to *activation* and *failure*. We have defined different templates for each BDMP element and depending upon the position and the configuration of an element in a BDMP we invoke the corresponding template. These templates correspond to the semantics of each element. The approach is modular therefore it is easy to add more templates if, for instance, other elements are considered in the future. An exhaustive list of BDMP elements is presented in Fig. 2. We segregate these elements into four categories; the first row of Fig. 2 defines *gates*, the second and third row define *leaves*, the fourth row defines *triggers*, and the fifth row defines *others*. We discuss the proposed semantics under each category:

Gates: BDMPs have four types of gates; AND, OR, VOTing and PAND as shown in the first row of Fig. 2. The AND, OR and VOT are inherited from SFTs. PAND is a *dynamic* gate, i.e., its behavior depends on the order of input failures.

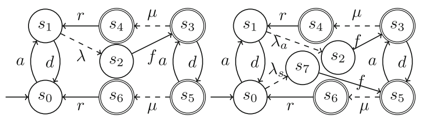
All gates shown in Fig. 2 have two inputs (A, B) and one output (Y). We use subscripts to denote actions of a particular BDMP element, e.g., the failure of input A is denoted by f_A. Any n input gate can be represented by a cascade of $n-1$ two-input gates (a.k.a: binary gates). Therefore it suffices to provide semantics of binary AND, OR and PAND gates. The structure function of AND is true when SF of both inputs A and B is true, i.e., $SF(Y) = SF(A) \wedge SF(B)$.

The failure of an AND is captured by the $s_3 \rightarrow s_4$ transition in the MA of Fig. 3(a). The gate is considered repaired if either one or both of its inputs get repaired. Therefore, we have two repair action transitions ($s_5 \rightarrow s_1$ and $s_6 \rightarrow s_2$) for single input repairs and two repair transitions ($s_7 \rightarrow s_0$ and $s_8 \rightarrow s_0$) for the repair of both inputs. The states representing the failure of the gate output are double circled. Note that the gate can only leave the double circled states by means of r_Y-labeled transitions.

The structure function of OR is true when SF of any of its inputs is true, i.e., $SF(Y) = SF(A) \vee SF(B)$. This behavior is captured in the MA of Fig. 3(b) by introducing two fail transitions ($s_1 \rightarrow s_5$ and $s_2 \rightarrow s_6$).

The PAND in BDMPs follows an *exclusive-PAND* semantics, i.e., simultaneous input failures do not cause output failure. The structure-function of PAND is true only when f_A occurs strictly before f_B. The precise semantics are depicted in Fig. 3(c) where intricacies of repair orders can be followed. It is remarked that *exclusive-PAND*, in-general, cannot capture the behavior where both inputs are INST (introduced in the next paragraphs) and both fail simultaneously.

Strictly speaking, the failure behavior of a *gate* does not depend on its activation status. Therefore we did not have any activation transition in the MA of *gates*. This is different for *leaves* as we discuss next. Although the behavior of a gate is independent of the activation, the gates are involved in propagating the activation behavior towards the leaves. Therefore we also define the activation MA for *gates*. (In-fact activation MA templates, in general, only depend on the type of trigger pointing to a node and the number of parents inheriting this nodes.)



(a) MA of EXP (b) MA of STDBY

Fig. 4. MA of EXP and STDBY

Leaves: There are eight types of leaves in BDMPs as shown in the second and third row of Fig. 2. The EXP represents component failures which follow an exponential probability distribution with rate $\lambda \in \mathbb{R}_{>0}$. The EXP can fail upon activation as depicted in Fig. 4(a). The *repair rate* for this BE is μ and it is unaffected by the activation status of EXP. The STDBY is used to model components which can fail in *standby* mode. Hence, two failure rates are relevant: the standby failure rate λ_s and the active failure rate λ_a. If we remove the standby failure behavior of a STDBY, then it behaves as EXP. This can also be observed in Fig. 4(b), i.e., if we remove the Markovian transition ($s_0 \dashrightarrow s_7$) and the then unreachable state s_7, the resulting automaton is identical to Fig. 4(a).

The MULTI element represents a batch of independent and identical EXP components. MULTI with parameters n and m can be modeled as n EXPs inherited by a VOT gate where $K = m$ and $N = n$. The structure function of MULTI becomes true after $N-K$ of these identical units have failed. The semantics of MULTI with $K = 2$ and $N = 3$ are presented in Fig. 5. Notice that the automaton structure of Fig. 5 can be easily extended to other values of K and N. Since the functionality of a VOT is hidden in MULTI, the MA of VOT is also hidden in MA of MULTI. If we remove all a and d labeled action-transitions from the MA of MULTI, we get the MA of a VOT gate with three EXPs having identical failure rates (λ) and repair rates (μ).

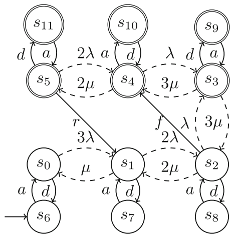


Fig. 5. MA of MULTI

BDMPs model on-demand failures using INST BEs. The SF of this element (upon activation) becomes one with probability γ or remains zero with probability $1 - \gamma$ as depicted in state s_1 of Fig. 6. The failing mechanism of this BE is quite straightforward but its repair mechanism is not obvious. INST is receptive to activation and deactivation actions even when it is failed. The INST will keep track of the activation requests and if it has a valid activation request at the time of repair, then INST will be checked again.

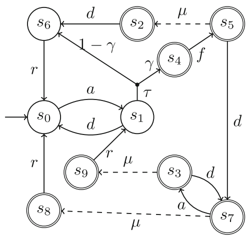


Fig. 6. MA of INST

The structure function of PHASE becomes immediately true upon an activation request and once true it switches to false upon the occurrence of a Markovian transition governed by rate μ, see Fig. 7(a). Notice that action names like *start-phase* and *end-phase* are not used because they are just aliases of failure and repairing actions, respectively. The SF of G.Fail (F.Safe) becomes true (false) at the start of BDMP analysis and remains true (false) thereafter. This behavior is achieved by adding a $when(\bot)$ guard in the repair (fail) action transition of Fig. 7(b) (Fig. 7(c)). BDMPs achieve the G.Fail (F.Safe) feature by associating a flag $FailF\_Frozen$ to BEs. Setting this flag to true forces the BE to keep its failure state set by the user during modeling. The user can set the failure state through another flag called $FailF$. BDMPs (like other fault tree formalisms) analyze how BEs contribute towards the occurrence of an undesired top-level event (called TOP in BDMPs). It can inherit only one child v and follows failure f_v and repair r_v transitions of this child, see Fig. 7(d).

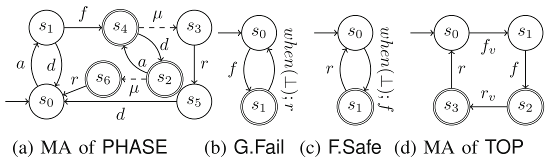


Fig. 7. MA of other BDMP leaves

BDMP Modularization: Before presenting the activation semantics we discuss modularization; an important concept for our approach. A *module* is a subset of BDMP elements disjoint (from activation point of view) from other elements and having the same activation behavior. The modularization creates a partition of BDMP elements. In order to modularize, we traverse the BDMP and identify the nodes that are target of triggers. We call these nodes the *module representatives*. The *module-representative* is the element which interacts with other parts of the BDMP and decides the activation status of the module it is representing. The activation/deactivation mechanism of all *module-members* are tethered to activation and deactivation of *module-representative*. It becomes apparent that BDMP can be segregated in modules and cardinality of these modules can range from one (imagine module consist of single BE) to the size of entire BDMP (when BDMP is a simple SFT).

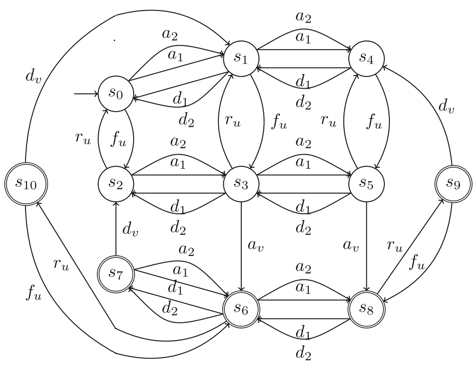
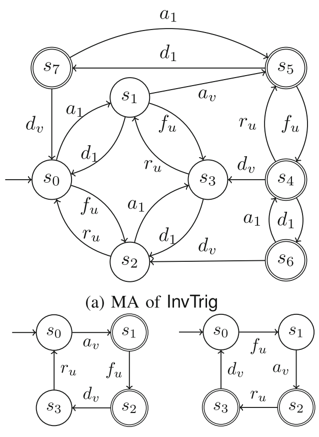


Fig. 8. MA of Trig and two parents

Triggers: There are four types of triggers in BDMPs: *trigger* Trig, *inverted trigger* InvTrig, *equal separating trigger* EqSTrig, and *opposite separating trigger* OppSTrig. The Trig link connecting nodes u and v means that, $AF(v)$ is true when $SF(u)$ is true and v is input of some gates $g_1, g_2, \dots, g_k$ and AF of at least one of these k gates is true. The semantics of Trig for the case where v is inherited by two gates (a.k.a.: nodes) are shown in Fig. 8, where we perform activation action a_v after reception of two actions; failure of u (f_u) and activation of gate 1 or 2 (a_1 or a_2 respectively). Here, we only presented MA for the case where Trig target v is input of two gates but the pattern in this automaton can be extended by adding states and transitions capturing activation, deactivation of more gates (if added). We double circle the states where v is active to distinguish them from the other states.

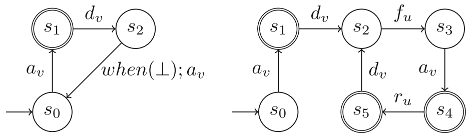
The InvTrig provides complementary behavior to Trig. Here $AF(v)$ is false when either $SF(u)$ is true, or $SF(u)$ is false and v is the input of some gates $g_1, g_2, \ldots, g_k$ and AF of all of these gates is false. The automata-based view for target v of InvTrig inherited by one gate is shown in Fig. 9(a). Notice that the deactivation action is performed as soon as parent is deactivated or trigger origin u in repaired. The OppSTrig isolates the activation behavior of the target node v from its parents. The semantics of this link is presented in Fig. 9(b). Informally speaking, v is activated at the start of BDMP analysis and it is deactivated as soon as OppSTrig origin u performs a fail action. The EqSTrig achieves complementary behavior of the OppSTrig. The target v of EqSTrig is activated upon failure of the trigger origin, i.e., u. The deactivation of v is performed upon repair of u, see Fig. 9(c).



(a) MA of InvTrig (b) MA of OppSTrig (c) MA of EqSTrig

Fig. 9. MA of other *triggers*

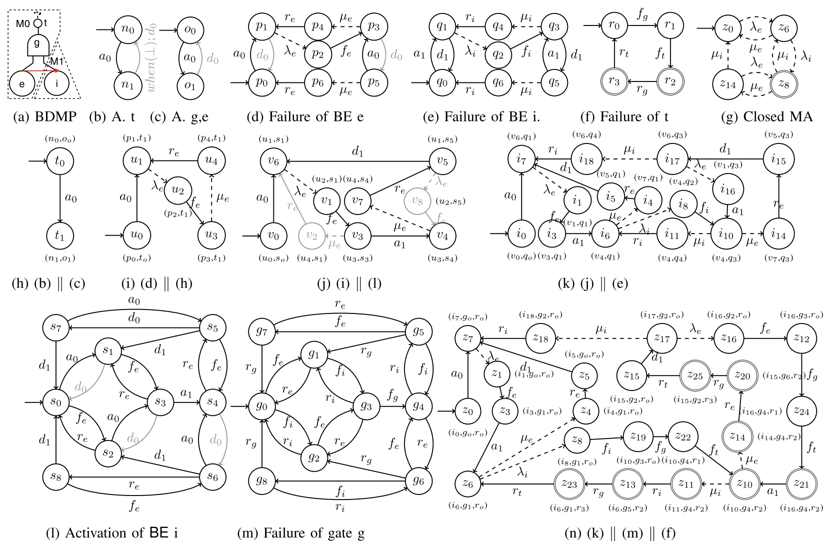
The aforementioned activation semantics captures the behavior of all BDMP elements except PHASE, which is treated differently. In BDMPs, a flag *In_progress* is associated to PHASE. If this flag is set to true, then the activation action has to be performed at the start of the BDMP analysis. The *In_progress* flag of only one phase element should be set to true so that we can clearly identify the first phase of the phased-mission profile. This flag is effective at the beginning of the BDMP analysis. Subsequent activation is conditioned to the fact whether PHASE is the target of a trigger or not. If it is not the target of a trigger, then PHASE can not be activated afterward, see Fig. 10(a).



(a) *In_prog.*=$\top$,no trigger (b) *In_prog.*=$\top$, one trigger

Fig. 10. MA of PHASE activation

On the other hand, if PHASE is the target of the trigger then it will get activated on the failure of trigger origin u. This activation mechanism is similar to EqSTrig MA of Fig. 9(c) and we only need to add an activation action at the start provided *In_progress* flag is true, see Fig. 10(b). We remark that BDMPs are *multi-top* trees and each top module is activated at the start of the analysis. We assign a different number to each top module because the top element can be the target of the trigger.

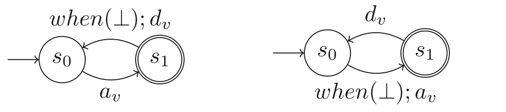

(a) BDMP (b) A. t (c) A. g,e (d) Failure of BE e (e) Failure of BE i. (f) Failure of t (g) Closed MA

(h) (b) ∥ (c) (i) (d) ∥ (h) (j) (i) ∥ (l) (k) (j) ∥ (e)

(l) Activation of BE i (m) Failure of gate g (n) (k) ∥ (m) ∥ (f)

Fig. 11. Composition example

Others: This includes node activation flags (AlwaysFalse, AlwaysTrue), BeforeLink, and LogicLink. The AlwaysTrue (AlwaysFalse) flags can be associated to any node of the BDMP. If this flag is true, then that node and associated module will remain active (de-active) throughout the analysis of the BDMP as indicated by the MA of Fig. 12(a) (Fig. 12(b)). The BeforeLink connecting two INST type nodes u and v enforces an order in checking the Bernoulli distribution associated to the INST. That is, v is checked only after checking u. The BeforeLink were proposed later in BDMPs as an optimization. They are mentioned here for the sake of completeness and we did neither outline its precise semantics nor discuss it further because we do not consider optimizations. The last element we mention is the LogicLink. The semantics of LogicLink are quite subtle. When we syntactically say node B is child of node A, we are semantically implying that the actions of the MA corresponding to B and A are visible to each other.



(a) MA of AlwaysTrue (b) MA of AlwaysFalse

Fig. 12. MA of AlwaysTure and AlwaysFalse

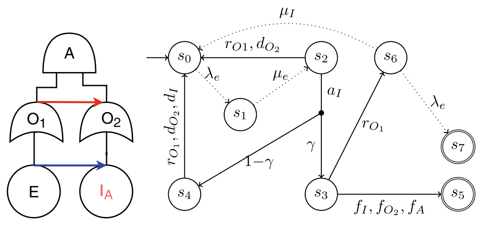
Composition Example. Let us explain our compositional semantics on a BDMP example. The example BDMP, depicted in Fig. 11(a), has two modules (0 and 1) as there is only one Trig. The BDMP consists of four elements; therefore we need eight automata to construct the complete state-space of the BDMP. The automata and their relevant compositions are:

$$\underbrace{MA_{F_{bdmp}}}_{Fig.(n)} = \underbrace{MA_{F_t}}_{Fig.(f)} \parallel \underbrace{MA_{A_t}}_{Fig.(b)} \parallel \underbrace{MA_{F_g}}_{Fig.(m)} \parallel \underbrace{MA_{A_g}}_{Fig.(c)} \parallel \underbrace{MA_{F_e}}_{Fig.(d)} \parallel \underbrace{MA_{A_e}}_{Fig.(c)} \parallel \underbrace{MA_{F_i}}_{Fig.(e)} \parallel \underbrace{MA_{A_i}}_{Fig.(l)}$$

The final MA (see Fig. 11(g)) of the BDMP is obtained by turning all actions into internal actions and applying maximal progress (as described in Sect. 2).

The activation mechanism of t (TOP) is shown in Fig. 11(b) with an impossible deactivation transition. Since all other components of the *primary module* will follow that behavior, the deactivation transition of their MA is colored gray to indicate this impossibility, see Fig. 11(c). The same approach is followed in Fig. 11(d) to indicate that e cannot be deactivated once activated. All states corresponding to the failure of TOP are double circled. Notice in Fig. 11(n) that some states after the Markovian transition from z_{10} are still tagged as fail states but no time is spent in these states because they all have only outgoing action transitions. Another important observation on some paths, e.g., $z_{11} \rightarrow z_{13} \rightarrow z_{23} \rightarrow z_6$ is that there is an interleaving behavior but since all interleavings of action transitions lead to the same end result, we do not draw all possible paths. A similar phenomenon occurs in Fig. 11(j). The transitions to v_2 and v_8 are highlighted as impossible transitions because they are paralleled by immediate (action-based) transitions a_1 and d_1, respectively. Each state is annotated with the state identifiers from the composing automata for the sake of clarity. State identifiers are neither required for composition nor for model checking.

Non-determinism. Due to the compositional nature of semantics, non-determinism can occur. That is to say, the *closed* MA of a BDMP may contain states that have more than one outgoing action-transition. These scenarios occur if there are several possible ways in which failures, activation, and de-activation are propagated through the BDMP. As our semantics is modular, these propagations are initiated locally, though their effect is global. Let us illustrate this by an example, see the BDMP in Fig. 13(a).



(a) BDMP (b) Automaton for reachability property

Fig. 13. Non-determinism example

Our semantics obtains a choice between the activation and deactivation actions of INST I_A. This can be seen as follows. Consider the *closed* MA in Fig. 13(b) of the BDMP obtained after applying maximal progress. The transitions are labeled with actions (instead of the invisible action τ) for the sake of clarity. Note that transition $s_4 \rightarrow s_0$ has three labels indicating that these transitions can be arbitrarily permuted. Their order, however, does not matter and we will finally reach state s_0, regardless of which order is taken. Consider the execution trace of the BDMP in Fig. 13(a), initiated by a repair of E: $r_E \rightarrow r_{O_1} \rightarrow d_{O_2}$. This trace does not activate I_A. However, this propagation of E's repair resulting in O_2's de-activation is not atomic. Therefore, the activation action of I_A can interleave with this trace. This is reflected by state s_2 in the MA which also has an outgoing a_I-action transition. Another non-deterministic choice exists at state s_3. This is due to the two possible execution traces initiated by E's repair: $r_E \rightarrow a_I \rightarrow f_I$ and $r_E \rightarrow a_I \rightarrow r_{O_1} \rightarrow f_I$. The first trace will directly lead to failure. But the second trace repairs O_1 (r_{O_1}) before performing f_I; failure state S_7 will be reached only after E's next failure. In the meantime, I could be repaired leading to a return to the initial state. This behavior is captured through the Markovian transitions from state s_6. These subtle non-deterministic scenarios are mainly due to the instantaneous behavior of the INST element.

4 Prototypical Implementation and Experimentation

Our compositional BDMP semantics has been implemented in a software prototype. The overall setup of our implementation is presented in Fig. 14. We describe each block:

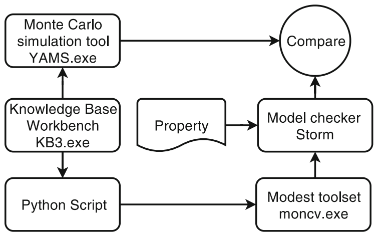


Fig. 14. Prototypical tool

KB3.exe: The *Knowledge Base Workbench* [4] is a GUI based tool used to create, simulate, and export the BDMP as Figaro definition. Figaro is a domain-specific object-oriented probabilistic modeling language defined to carry out operation safety studies [4]. It generalizes various reliability models, e.g., reliability block diagrams and can cast generic models in *knowledge bases* KB. In fact, BDMPs are originally defined as a *knowledge base* in Figaro.

Python Script: This constitutes the core of our implementation. The script takes a BDMP description as Figaro definition and generates a process-algebraic description of its MA using the Modest language [2]. In order to do so, the following five steps are performed;

1. ReadFI: we read the English or French version of BDMP and populate it into a dictionary-based data structure.
2. ModularizeBDMP: the structure of the parsed BDMP is analyzed and segregated into *modules*. We use a depth-first search on the underlying graph to identify *module-representatives*. Whenever we encounter a node having different activation behavior, i.e., the node is either target of Trig, InvTrig, OppSTrig, EqSTrig, tagged as AlwaysTrue or AlwaysFalse, or inherited by more than one *module*, then we consider this node as a *module-representative*. As stated earlier, *modules* create a partition of a BDMP and from an activation point of view we only need to create interaction between partitions. We remark that a module having TOP as *module-representative* is called *primary module* and INST in *primary modules* of BDMPs as originally conceived in [5] have no semantics, i.e., they are never tested.

3. RemoveVOT: converts a VOT gate into a cascade of AND and OR gates.
4. BinarizeGates: This step turns any gate into a cascade of binary gates.
5. WriteModestDefinitions: This step generates a process-algebraic description for each component present in the BDMP under consideration.

monovonv.exe: This component of the Modest tool-set converts a Modest file to a JANI file. JANI is an intermediate language originally designed to exchange models between different formal analysis tools [7]. The model checker Storm has direct support for the JANI format.

Property: We use probabilistic temporal logic to encode the properties of interest, i.e., the *unreliability* and the *unavailability*. In a repairable BDMP, there is no "permanent failure". What we are interested in is an entry in a tangible fail-labeled state. Note that *unreliability* is a simple reachability property and mostly a partial state-space suffices for this property. However, we need complete state-space to compute the *unavailability*.

Storm: We feed the JANI file to Storm along with the property of interest and Storm computes and returns the numerical value for the desired metric. Storm [8] is a state-of-the-art probabilistic model checker. It is an open-source and freely available tool. Recently, Storm participated in the QComp 2019[2] competition and outperformed all competitors on most of the benchmarks. It uses numerical and symbolic methods.

YAMS.exe: Yams is a freely available Monte-Carlo based simulation tool for BDMPs [4]. Yams uses a standard event-driven Monte-Carlo simulation method. It can report the mean value of an indicator function along with its standard deviation, i.e., the range of uncertainty against a given confidence level. Yams can be configured to compute different reliability metrics, e.g., *mean-time-to-failure* (MTTF), *unreliability* and *unavailability*, etc. The simulation time increases with increasing precision requirement, e.g., approximately $\mathcal{O}(10^k)$ simulations are required to obtain a (low) probability of 10^{-k} with a 10% confidence interval.

Results of Test-Cases. The test cases considered along with detailed documentation are available online[3]. These test cases range from simple interactions, e.g., mutual exclusion to literature benchmarks. For each test case, we compare Storm generated results with those of Yams. Yams reports results for different confidence bounds and we benchmark, as tolerance, 99% *confidence level* for our comparison. Storm was run for the precision of 10^{-8}. Yams is a Windows-based tool, whereas Storm was run on a Linux-based machine having 5x: 2 Intel® Xeon® Platinum 8160, 48 threads 2.1 GHz, 384 GB RAM. The symbolic engine of Storm, i.e., *sylvan* was restricted to 8 threads with maximum allocated memory of 40 GB. We build complete state-spaces symbolically and MAs reported here are *sparse* models build from the symbolic models

[2] HTTP://qcomp.org/competition/2019/.
[3] HTTP://sourceforge.net/projects/visualFigaro/files/Doc_and_examples/Francais/.

Table 1. Indicative statistics for test-cases

	#elements			Yams				Storm						Comparison	
Test			Mission					Complete MA		Reduced MA		CTMC		Absolute Error	
case	stat	dyn	Time	Unrel.	Tol.	Unavl.	Tol.	#state	#trans.	#state	#trans.	#state	#trans.	Δ_{Unr}	Δ_{Unavl}
1	10	2	10	0.7872	0.0011	0.5157	0.0013	987	1889	313	1299	172	984	0.0000	0.0002
4	19	2	10	0.9101	0.0074	0.6864	0.0120	34336	99002	20027	86719	3105	26436	0.0001	0.0005
17	24	4	10	0.8562	0.0009	0.5978	0.0126	1793667	4630531	511752	3593876	207744	2845056	0.0001	0.0018
23	23	3	20	0.8690	0.0275	0.4950	0.0407	16110	40220	5677	29787	2048	22528	0.0017	0.0096
39	34	6	10	0.7712	0.0034	0.5620	0.0040	144483	336048	50436	242001	12288	157120	0.0008	0.0003
49	13	3	10	0.8811	0.0026	0.6148	0.0040	934	2109	574	1739	117	548	0.0011	0.0010
53	14	5	10	0.6770	0.0038	0.4042	0.0040	101761	220941	36712	166196	10840	104420	0.0015	0.0017

after application of the *maximal progress*. We only look into the accuracy of results and disregard computation requirements in terms of memory and verification time because the implementation is only a sanity check of semantics. The results for six test-cases are reproduced in Table 1. Detailed results along with the python script can be accessed at[4]. The last two columns of Table 1 report the absolute errors of the unreliability (Δ_{Unr}) and unavailability (Δ_{Unavl}). Where $\Delta_{Unr} = |Unrel_{\text{YAMS}} - Unrel_{\text{STORM}}|$ and $\Delta_{Unavl} = |Unavl_{\text{YAMS}} - Unavl_{\text{STORM}}|$. We do not reproduce Storm calculated values as they can be easily reproduced from the absolute errors. It is clearly visible (in the last two columns of Table 1) that the *unreliability* and the *unavailability* values computed by Storm are always within Yams-computed range. The size of the complete MA, the reduced MA (after application of maximal progress), and CTMC (after removal of spurious non-determinism) are also provided.

Non-determinism. As described earlier, the compositional semantics may lead to non-determinism. This occurred in eight test cases. The non-determinism is primarily due to the instantaneous character of INST elements. After delaying the INST activation actions by a very high rate Markovian transition, i.e., 10^5, and applying maximal progress, the minimal and maximal values obtained by Storm coincide. That is, all remaining non-determinism is spurious. Our analysis with Storm yields the same values as the simulation tool Yams. The usage of high-rate Markovian transitions increases the *stiffness* of the underlying Markov chain and it results in an increased analysis time as convergence is slower.

5 Conclusion

We presented a formal, compositional semantics for repairable BDMPs. Its modularity provides insight into the subtleties of BDMPs and yields a comprehensible semantics that is amenable to model-based analysis such as probabilistic model checking. Experimental evaluations using a prototypical implementation reveal that our semantics coincides with the BDMP interpretation of the simulation tool YAMS. Future work includes reducing the peak memory consumption by leveraging partial-order reduction, bi-simulation, and symmetry reduction as shown to be successful for dynamic fault trees [19]. The challenge is to adapt these techniques to repairs. It would also be interesting to extend our semantics

[4] HTTP://github.com/moves-rwth/dft-bdmp/.

to generalized BDMPs [17], and to exploit priorities in GSPNs [16] to obtain a fully deterministic compositional semantics as in [13] for DFTs.

References

1. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
2. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MODEST: a compositional modeling formalism for hard and softly timed systems. IEEE TSE **32**(10), 812–830 (2006)
3. Boudali, H., Crouzen, P., Stoelinga, M.: A rigorous, compositional, and extensible framework for dynamic fault tree analysis. IEEE TDSC **7**(2), 128–143 (2009)
4. Bouissou, M.: Automated dependability analysis of complex systems with the KB3 workbench: the experience of EDF R&D. In: ICEE. CIEM (2005)
5. Bouissou, M., Bon, J.L.: A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. Rel. Eng. Sys. Safety **82**(2), 149–163 (2003)
6. Budde, C.E., Biagi, M., Monti, R.E., D'Argenio, P.R., Stoelinga, M.: Rare event simulation for non-Markovian repairable Fault Trees. TACAS 2020. LNCS, vol. 12078, pp. 463–482. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_26
7. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
8. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part II. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
9. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans. Reliab. **41**(3), 363–377 (1992)
10. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351. IEEE Computer Society (2010)
11. Guck, D., Spel, J., Stoelinga, M.: DFTCalc: reliability centered maintenance via fault tree analysis (tool paper). In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 304–311. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_19
12. Hartmanns, A., Hermanns, H.: A modest Markov automata tutorial. In: Krötzsch, M., Stepanova, D. (eds.) Reasoning Web. Explainable Artificial Intelligence. LNCS, vol. 11810, pp. 250–276. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31423-1_8
13. Junges, S., Katoen, J.-P., Stoelinga, M., Volk, M.: One net fits all: a unifying semantics of Dynamic Fault Trees using GSPNs. In: Khomenko, V., Roux, O.H. (eds.) PETRI NETS 2018. LNCS, vol. 10877, pp. 272–293. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91268-4_14
14. Kaiser, B., Gramlich, C., Förster, M.: State/event fault trees - a safety analysis model for software-controlled systems. Rel. Eng. Sys. Safety **92**, 1521–1537 (2007)

15. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A new component concept for fault trees. In: SCS. CRPIT, vol. 33, pp. 37–46. Australian Computer Society (2003)
16. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets, vol. 292. Wiley, New York (1995)
17. Piriou, P.Y., Faure, J.M., Lesage, J.J.: Generalized Boolean logic Driven Markov Processes: a powerful modeling framework for model-based safety analysis of dynamic repairable and reconfigurable systems. Rel. Eng. Sys. Safety **163**, 57–68 (2017)
18. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15**, 29–62 (2015)
19. Volk, M., Junges, S., Katoen, J.P.: Fast dynamic fault tree analysis by model checking techniques. IEEE Trans. Ind. Inform. **14**(1), 370–379 (2018)
20. Walker, M., Papadopoulos, Y.: Synthesis and analysis of temporal fault trees with PANDORA: the time of priority AND gates. Nonlinear Anal. Hybri. Syst. **2**(2), 368–382 (2008)