Rich Interfaces for Dependability: Compositional Methods for Dynamic Fault Trees and Arcade models

Hichem Boudali[1] Pepijn Crouzen[2] Boudewijn R. Haverkort[1] Matthias Kuntz[1]
Mariëlle Stoelinga[1]

[1] University of Twente, P.O. Box 217, 7500AE Enschede, The Netherlands

[2] Saarland University, Stuhlsatzenhausweg 45, 66123 Saarbrücken, Germany

Abstract

This paper discusses two behavioural interfaces for reliability analysis: dynamic fault trees, which model the system reliability in terms of the reliability of its components and Arcade*, which models the system reliability at an architectural level. For both formalisms, the reliability is analyzed by transforming the DFT or* Arcade *model to a set of input-output Markov Chains. By using compositional aggregation techniques based on weak bisimilarity, significant reductions in the state space can be obtained.*

1 Introduction

Dependability evaluation has become an important and integral part in the design of today's computer-based systems. To this end, a wide variety of modeling approaches has been developed for evaluating system dependability: General purpose models, such as CTMCs; stochastic Petri nets [**?**] (and their extensions); stochastic process algebras [**?**, **?**, **?**]; interactive Markov chains [**?**], Input/Output IMCs [**?**], and stochastic activity networks (as used in UltraSAN and Möbius [**?**]) serve the specification and validation of a wide variety of quantitative properties of computer and communication systems, including dependability properties. Dependability-specific formalisms, such as reliability block diagrams (RBDs), the System Availability Estimator (SAVE) language [**?**], dynamic RBDs (DRBDs) [**?**]; dynamic fault trees (DFTs) [**?**] and extended fault trees (eFTs) [**?**]; OpenSESAME [**?**], and TANGRAM [**?**] contains specific constructs for expressing dependability concerns. Some architectural (design) languages specific "error annexes" have been developed to allow for dependability analysis, most notably, the architectural description language AADL [**?**], and the UML dependability profile [**?**].

Compositionality is an essential feature that, in our opinion, any dependability formalism should possess, enabling one to manage today's involved dependability concerns by breaking down the complexity of large systems into small and manageable pieces. Compositionality comes in two aspects: *compositional modeling* means that a model can be created by composing smaller submodels and *compositional analysis* means that a model can be analyzed by combining the results of the analysis of the submodels.

Few of the approaches mentioned before are fully compositional: while dependability-specific and architectural approaches allow compositional modeling, they lack compositional analysis techniques: for the architectural approaches, few systematic analysis methods exist at all; the dependability-specific formalisms are usually analyzed by generating the global state space of underlying CTMC in a monolithic, non-compositional way. Several general-purpose formalisms are compositional; however, these lack specific constructs to model dependability concerns in a concise and convenient way.

In this paper, we illustrate how one can obtain compositional analysis techniques for higher-level formalisms by exploiting the compositionality properties of a lower-level, general-purpose formalism. More specifically, we consider two (distinct) formalisms for reliability analysis, viz. dynamic fault trees (DFTs) and Arcade. DFTs are a versatile, graphical formalism had has gained popularity among reliability engineers. Arcade [**?**] is an approach that we designed, by learning from the drawbacks from previous formalisms.

Our approach is based on a compositional translation of both formalisms into Input-Output Interactive Markov Chains (I/O-IMC), yielding an I/O-IMC representation for each modeling construct in these formalisms. Thus, the translation from a complete DFT or Arcade model is defined in terms of the translations of their components. These

translations pin down the semantics of both formalisms in a rigorous way. In this way, costly errors due to misunderstandings or misinterpretations are avoided. Then, by using the compositional analysis methods from the I/O-IMC formalism, we obtain compositional analysis techniques for DFTs and Arcade models. In particular, we generate the state space of the underlying CTMC in an efficient way by using compositional aggregation techniques, based on the repeated use of aggressive state space minimization techniques. We show the benefits of this approach on a number of examples, several of which feature a drastic reduction in the number of states generated.

The rest of this paper is organized as follows: Section **??** introduces the DFT and Arcade formalisms. In Section **??** we lay out the compositional analysis methods based on I/O-IMCs, while Section **??** reports on tools support and case studies. Finally, Section **??** presents the conclusions.

This paper summarizes and unifies results from [**?**, **?**, **?**, **?**].

2. The DFT and Arcade formalisms

2.1. Dynamic fault trees

A fault tree model describes the system failure in terms of the failure of its components. Standard FT are combinatorial models and are built using static gates (the AND, the OR, and the K/M gates) and basic events (BE). A combinatorial model only captures the combination of events and not the order of their occurrence. Combinatorial models become, therefore, inadequate to model today's complex dynamic systems. DFT introduce three novel modeling capabilities: (1) spare component management and allocation, (2) functional dependency, and (3) failure sequence dependency. These modeling capabilities are realized using three main dynamic gates: The spare gate, the functional dependency (FDEP) gate, and the priority AND (PAND) gate. Figure **??** depicts all DFT gates.

The PAND gate fails when all its inputs fail and fail from left to right (as depicted on the figure) order. The spare gate has one primary input and one or more alternate inputs (i.e. the spares). The primary input is initially powered on and when it fails, it is replaced by an alternate input. The spare gate fails when the primary and all the alternate inputs fail (or are unavailable). A spare could also be shared among multiple spare gates. In this configuration, when a spare is taken by a spare gate, it becomes unavailable (i.e. essentially seen as failed) to the rest of the spare gates. The FDEP gate is comprised of a trigger event and a set of dependent components. When the trigger event occurs, it causes the dependent components to become inaccessible or unusable (i.e. essentially failed). The FDEP gate's output is a 'dummy' output (i.e. it is not taken into account during the calculation of the system's failure probability). Along with static and dynamic gates, DFT also

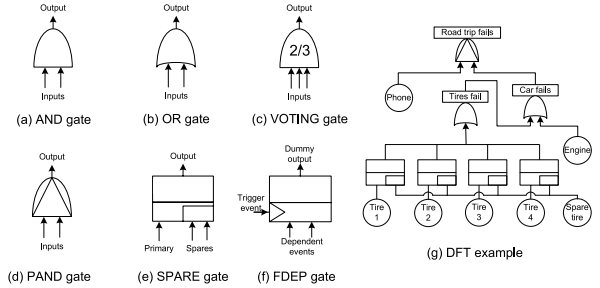

Figure 1. DFT gates and example.

possess basic events, which are leaves of the tree. A basic event usually represents a physical component having a certain failure probability distribution (e.g. exponential). A DFT element has a number of operational or failed states. In the case of a BE, operational states could be further classified as *dormant* or *active* states. A dormant state is a state where the BE failure rate is reduced by a factor called the dormancy factor α. An active state is a state where the BE failure rate λ is unchanged. Depending on the value of α, we classify BE as: cold BE ($\alpha = 0$), hot BE ($\alpha = 1$), and warm BE ($0 < \alpha < 1$). The dormant and active states of a BE correspond to dormant and active modes of the physical component. For instance, a spare tire of a car is initially in a dormant mode and switches to an active mode when it is fixed on the car for use.

Figure **??** shows a DFT modeling a road trip. Looking at the top PAND gate, we see that the road trip fails (i.e. we are stuck on the road) if the car fails after the mobile phone has failed; if the car fails first, then we can call the road services to tow the car and continue our journey. The car fails if either the engine fails or the tire subsystem fails, as modeled by the OR gate labeled 'car fails'. The car is equipped with a spare tire, which can be used to replace any of the primary tires. When a second tire fails, the tire subsystem fails, causing in turn a car failure. Thus, we model the tire subsystem by four spare gates, each having a primary tire and all sharing a spare tire The spare tire is a cold spare, i.e. it is initially in standby mode with failure rate 0.

Galileo DIFtree [**?**] was the first package to introduce, use, and analyze DFT. DIFtree allows a limited form of compositional modeling and analysis. On the modeling side, DFTs do allow bigger trees to be built from smaller subtrees, however there are some rather severe restrictions on the type of allowed inputs to certain gates (e.g. inputs to spare gates and dependent events of functional dependency gates have to be basic events), which greatly diminish the modeling flexibility and power of DFT.

Moreover, DFT lack modular analysis. That is, even though stochastically-independent sub-modules exist in a

certain DFT module (specifically those whose top-node is a dynamic gate), these sub-modules cannot be solved separately and still get an exact solution. Consequently, a DFT model, which is typically analyzed by first converting it into a Continuous Time Markov chain (CTMC), becomes vulnerable to the state space explosion problem, i.e. the number of states in the underlying CTMC is exponential in the number of basic events. The DIFtree methodology allows compositional analysis only if the top node is static, not for dynamic top nodes.

Using the I/O-IMC framework, we enable full compositionality, i.e. we lift several syntactic restrictions of DFTs and allow compositional analysis for any DFT.

2.2 Arcade

Arcade is a high-level architectural language for dependability analysis; it is a rich language as well as extensible which allows for even further expressivity. In Arcade, one models a system as a set of interacting components, where each component is characterized by a set of operational/failure modes, time-to-failure/repair distributions, and failure/repair dependencies, etc.

At this stage, there are, within Arcade, three types of components (i.e. building blocks) with which one can, in a modular fashion, construct a system model: (1) Basic Component (BC), (2) Repair Unit (RU), and (3) Spare Management Unit (SMU). A BC represents a physical/logical system component that has a distinct operational and failure behavior. A BC can have any number of operational modes (e.g., active vs. inactive, normal vs. degraded) and can fail either due to an inherent failure or due to its functional dependency upon another component. The RU component handles the repair of one or more BCs. Here, various repair policies (e.g., first-come-first-served, priority) and repair dependencies between BCs can be implemented. Finally, the SMU handles the activation and deactivation of BCs used as spare components.

The Arcade's building blocks can be readily modified or new building blocks can be added depending on the application domain. Arcade is also a formal language as the underlying semantics of each of these components is expressed in terms of I/O-IMC.

Figure **??** provides an example of a simplified nuclear reactor cooling system (NRCS) [4] modeled within Arcade. The figure shows the NRCS architectural design model. The NRCS consists of a reactor, two parallel pump lines, a heat exchanger and a bypass system for the heat exchanger. Each of the two pump lines consists of a single pump, a single filter and a number of control valves. The heat exchanging unit consists of the heat exchanger itself, a number of valves and one filter. The bypass system can be opened and closed by means of two motor driven valves.

All components, except the reactor itself (whose failure behavior is not considered here), are subject to failures and are repairable. The filters and the heat exchanger are either operational or failed. The valves can fail in two different modes, either stuck-open or stuck-closed. The pumps have two different operational modes and one failure mode. The pumps are either fully operational, or in a degraded operational mode, which is reached if one of the two pumps fails. In the degraded mode, the remaining pump fails with a higher failure rate. This is indeed a typical load sharing (load shared between the two pumps) situation.

Except for the two pumps, which share a single repair unit with a first-come-first-served (FCFS) repair strategy, each component has its own dedicated repair unit1. The system is down, if either none of the two pump lines is operational, or both the heat exchanger and the bypass system are not operational. A pump line is defective if one of its components is defective; where for the valves, only the stuck-closed case is considered to be a relevant failure. The heat exchanging unit is defective if the heat exchanger itself fails or one of its accompanying filters or valves fails. Finally, the bypass line fails if one of the motor driven valves is stuck-closed.

Arcade allows modeling the dependability characteristics described above by simply adding dependability annotations (rightmost in figure) to the various components. The dependability annotations are specified using a well-defined syntax. There exists an annotation for each of Arcades three building blocks. For example, in Figure **??**, we show the annotations for basic components named FP1, P1, and VIP1, and the repair unit named P.rep. Component FP1 has two fields time-to-failure and time-to-repair indicating the type of failure and repair distribution respectively. The annotation for the repair unit P.rep shows that it is in charge of the 2 pump components P1 and P2, and the repair policy is FCFS.

From these dependability annotations, Arcade automatically, and transparently to the user, derives a state model, analyzes it using standard numerical methods, and finally outputs the desired dependability measure (e.g. reliability or availability). Solving the NRCS with Arcade for a mission time of 1 year for instance, we get a system availability and reliability of 0.9999999990 and 0.9999986799 respectively.

3. Compositional Analysis based on I/O-IMCs

3.1. Input/Output Interactive Markov Chains

Input/Output interactive Markov chains (I/O-IMCs) [**?**] are a combination of Input/Output automata (I/O-automata) [**?**] and interactive Markov chains (IMCs) [**?**].

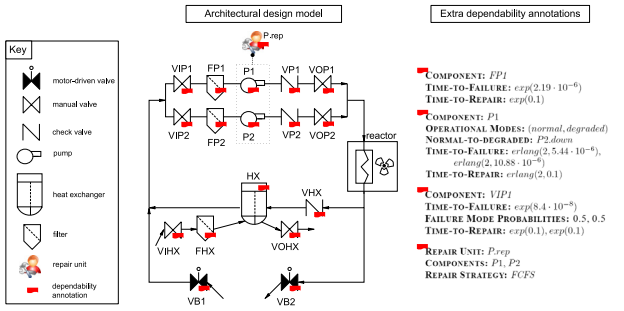

Figure 2. Reactor Cooling System

I/O-IMCs distinguish two types of transitions: (1) *Interactive transitions* labeled with actions; (2) *Markovian transitions* labeled with rates λ, indicating that the transition can only be taken after a delay that is governed by an exponential distribution with parameter λ. Inspired by I/O-automata, actions can be further partitioned into:

1. *Input actions* (denoted $a?$) are controlled by the environment. They can be *delayed*, meaning that a transition labeled with $a?$ can only be taken if another I/O-IMC performs an output action $a!$. A feature of I/O-IMCs is that they are *input-enabled*, i.e., in each state they are ready to respond to any of their inputs $a?$. Hence, each state has an outgoing transition labeled with $a?$.

2. *Output actions* (denoted $a!$) are controlled by the I/O-IMC itself. In contrast to input actions, output actions cannot be delayed, i.e., transitions labeled with output actions must be taken immediately.

3. *Internal actions* (denoted $a;$) are not visible to the environment. Like output actions, internal actions cannot be delayed.

States are depicted by circles, initial states by an incoming arrow, Markovian transitions by dashed lines, and interactive transitions by solid lines. Fig. **??** shows an I/O-IMC with two Markovian transitions: one from $S1$ to $S2$ with rate λ and another from $S3$ to $S4$ with rate μ. The I/O-IMC has one input action $a?$. To ensure input-enabling, we specify $a?$-self-loops in states $S3$, $S4$, and $S5$[1]. Note that state $S1$ exhibits a race between the input and the Markovian transition: in $S1$, the I/O-IMC delays for a time that is governed by an exponential distribution with parameter λ, and moves to state $S2$. If however, before that delay ends, an input $a?$ arrives, then the I/O-IMC transitions to $S3$. The only output action $b!$ leads from $S4$ to $S5$. We say that two I/O-IMCs *synchronize* if either (1) they are both ready to

[1] In the sequel we often omit these self-loops for the sake of clarity and simplicity of the I/O-IMC representation.

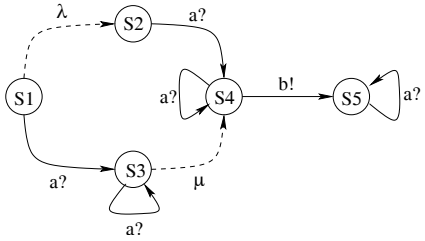

Figure 3. Example of an I/O-IMC

accept the same input action or (2) one is ready to output an action $a!$ and the other is ready to receive that same action (i.e., has input action $a?$). I/O-IMCs are also equipped with a parallel composition operator "||", to build larger I/O-IMCs out of smaller ones. The behavior of $P = Q||R$, i.e., the parallel composition of I/O-IMCs Q and R, is the joint behavior of its constituent I/O-IMCs and can be described as follows:

1. If an action does not require synchronization then Q and R can evolve independently, i.e., if Q (R) can make any transition (interactive or Markovian) and behaves afterwards as Q' (R'), the same behavior is possible in the parallel context, i.e., $Q||R$ can evolve to $Q'||R$ ($Q||R'$).

2. If an action of an interactive transition requires synchronization, then both I/O-IMCs Q and R must be able to perform that action at the same time, i.e., $Q||R$ evolves simultaneously into $Q'||R'$. Note that when an output and an input action synchronize the result is an output action.

Like in process algebras, the hiding operator hide A in P makes output actions in a set A internal, such that no further synchronization is possible over actions in A. Finally, aggressive minimization (also called lumping) techniques are available for I/O-IMCs that translate an I/O-IMC into one that is equivalent, but smaller. More details on the I/O-IMC formalism can be found in [**?**].

3.2. Compositional translation to I/O-IMCs

In this section we show three examples of how one obtains I/O-IMC models for the constructs in the DFT and Arcade formalisms. The full translation can be found in [**?**] and [**?**].

DFT basic event I/O-IMC model As pointed out in Section **??**, a basic event has a different failing behavior depending on its dormancy factor. For this reason we identify three types of basic events and correspondingly three types

of I/O-IMC. Figure **??** shows the I/O-IMC corresponding to a cold, warm, and hot basic events (all called A). The I/O-IMC clearly captures the behavior of the basic event described in Section **??**.

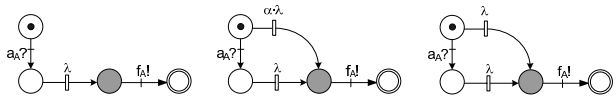

Figure 4. The I/O-IMC models of cold, warm, and hot basic events.

DFT PAND gate I/O-IMC model The PAND gate fires if all its inputs fail and fail from left to right order. If the inputs fire in the wrong order, the PAND gate moves to an operational absorbing state (denoted with an **X** on Figure **??**). Figure **??** shows the I/O-IMC of the PAND gate P with two inputs A and B (A being the leftmost input).

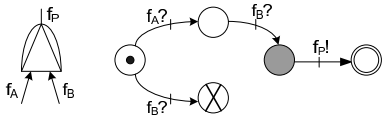

Figure 5. The I/O-IMC of the PAND gate.

Spare management unit I/O-IMC model in Arcade The spare management unit (SMU) handles the activation and deactivation of spare components. Figure **??** shows the I/O-IMC translation for an SMU that handles one primary and one spare, where the primary component is always in active mode, and thus always providing the service whenever it is operational. When the primary fails (input *failed_primary?*), the SMU activates (output *activate_spare!*) the spare component which takes over the primary. As soon as the primary is up again (input *up_primary?*), the spare is deactivated and the primary resumes operation.

3.3 Compositional aggregation approach

Our compositional semantics allows one to build the I/O-IMC associated to a DFT or Arcade model in a component-wise fashion, leading to a significant state-space reduction.

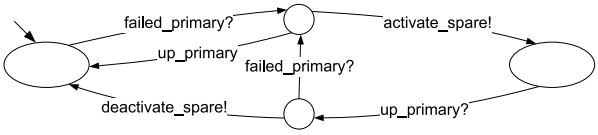

Figure 6. The SMU I/O-IMC model.

This kind of compositional aggregation approach has been previously successfully used, most notably in [**?**]. The compositional aggregation approach is to be contrasted with a more classical approach of model generation, such as the one used by DIFTree, where the CTMC model of a dynamic system is generated at once and as a whole and then possibly aggregated at the end. We propose the following conversion algorithm to transform a DFT or Arcade model into an I/O-IMC.

1. Translate each DFT or Arcade element to its corresponding (aggregated) I/O-IMC.
2. Pick two I/O-IMCs and parallel compose them.
3. Hide output signals that will not be subsequently used (i.e. synchronized on).
4. Aggregate, using weak bisimulation, the I/O-IMC obtained in step 3.
5. Go to step 2 if more than one I/O-IMC is left, otherwise stop.

The choice of I/O-IMCs we make in step 2 is important as this has an impact on the size of the generated state space during the intermediate steps. In the case studies (see Section **??**) we have used intuitive heuristics based on the level of interaction between models to decide the composition order.

Figure **??** illustrates the conversion algorithm on a simple DFT.

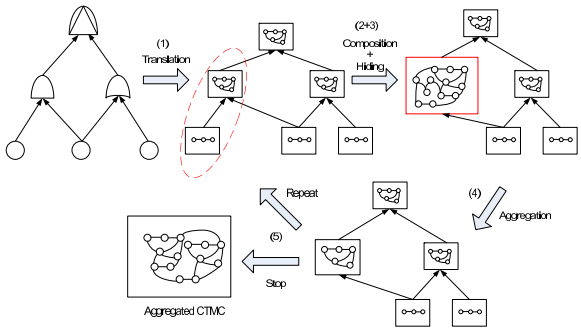

Figure 7. DFT to CTMC (or CTMDP) conversion algorithm.

4 Tool Support and Case studies

Both compositional-aggregation techniques have been implemented in a tool set based on CADP. Coral, the tool for DFT analysis is fully automated, whereas the Arcade tool is (still) partly manual. Using our tools, we have performed a number of case studies and compared our approaches with results from the literature. We summarize their results here.

Case study	Analysis method	Maximum number of states	Maximum number of transitions	Unreliability (Mission-time = 1)
CPS	DIFtree	4113	24608	0.00135668
CPS	Comp-Aggr	132	426	0.00135668
CAS	DIFtree	8	10	0.657900
CAS	Comp-Aggr	36	119	0.657900
MDCS	DIFtree	253	1383	$2.00025 \cdot 10^{-9}$
MDCS	Comp-Aggr	157	756	$2.00025 \cdot 10^{-9}$
FTPP	DIFtree	32757	426826	$2.56114 \cdot 10^{-11}$
FTPP	Comp-Aggr	1325	14153	$2.56114 \cdot 10^{-11}$

Figure 8. The results of the case studies.

4.1 DFT case studies

To compare the compositional aggregation (Comp-Aggr) approach with the traditional DIFtree method, we have conducted four case studies (none having non-determinism): the cascaded PAND system [**?**, **?**] (CPS), the cardiac assist system [**?**] (CAS), the multi-processor distributed computing system (MDCS) and the fault-tolerant parallel processor [**?**] (FTPP).

The results of the case studies are given in Figure **??**. The size of the largest model (with regard to the number of states) appearing during analysis is given for each experiment.

4.2 Arcade case studies

We carried out two case studies using the Arcade approach. First, we analyzed the nuclear reactor cooling system described before, which was modeled in [**?**] using the eFT approach. The CTMC for the pump subsystem has 10,404 states and 109,662 transitions; and the CTMC for the heat exchanger subsystem (including the bypass) has 240 states and 1,668 transitions. The largest model encountered during generation had 98,056 states and 411,688 transitions. The computed reliability of $52.9242 \cdot 10^{-10}$ (mission time 50 hours) coincides with [**?**].

Second, we analyzed a distributed database architecture (DDA), which was evaluated in [**?**] using SANs. It consists of a number of processors, disk controllers and hard disks, several of which are redundant. Using the methodology described in Section **??**, we generated the CTMC representing the behavior of the DDA. This CTMC has 2,100 states and 15,120 transitions. During the generation of this model, the largest I/O-IMC encountered had 6,522 states and 33,486 transitions. For comparison, the final model generated in [**?**] had 16,695 states.

5 Conclusions and Future work

In this paper, we have illustrated how one can obtain compositional modeling and analysis methods for high-level dependability formalisms via a element-wise translation to the I/O-IMC framework. For the two dependability formalisms (DFTs and Arcade) we showed the increase of the compositionality both at the analysis level and the model-building level.

Future work include completing the tool chain for Arcade models, considering more aggressive state space minimization techniques and generation diagnostics, explaining the most likely cause for a system failure, based on this framework.

Finally, we would like to stress that one could take a similar approach to formalisms for other dependability concerns, such as security or recovery, thus transferring the benefits of compositionality to other domains.

References

[1] Architecture Analysis and Design Language (AADL). SAE standards AS5506, Nov 2004.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. Wiley, 1995.

[3] M.-b. D. E. F. Arcade A Formal, Extensible. H. boudali and p.crouzen and and b.r.h.m. haverkort and m. kuntz and m.i.a. stoelinga.

[4] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theo. CS*, 202:1–54, 1998.

[5] H. Boudali, P. Crouzen, B. Haverkort, M. Kuntz, and M. Stoelinga. Arcade - a formal, extensible, model-based dependability framework. Technical report, University of Twente, to appear.

[6] H. Boudali, P. Crouzen, and M. Stoelinga. A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. *LNCS*, 4762:441–456, 2007.

[7] H. Boudali, P. Crouzen, and M. Stoelinga. Dynamic fault tree analysis using input/output interactive markov chains. In *Proc. of the 37th Annual IEEE/IFIP International Conference on DSN*, pages 708–717. IEEE, 2007.

[8] H. Boudali and J. B. Dugan. A discrete-time Bayesian network reliability modeling and analysis framework. *Reliability Engineering and System Safety*, 87(3):337–349, March 2005.

[9] H. Boudali and J. B. Dugan. A new Bayesian network approach to solve dynamic fault trees. In *Reliability and Maintainability Symposium*, Jan 2005.

[10] K. Buchacker. Modeling with extended fault trees. In *5th IEEE Int. Symposium on High Assurance Systems Engineering*, pages 238–246, Nov 2000.

[11] E. de Souza e Silva and R. M. M. Leao. The "TANGRAM-II" environment. In *Computer Performance Evaluation. Modelling Techniques and Tools: 11th Int. Conference, TOOLS 2000*, volume 1786, pages 366–369. LNCS, 2000.

[12] S. Distefano and L. Xing. A new approach to modeling the system reliability: dynamic reliability block diagrams. In *RAMS'06 proceedings*, pages 189–195, 2006.

[13] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Trans. on Reliability*, 41(3):363–377, September 1992.

[14] J. B. Dugan, B. Venkataraman, and R. Gulati. DIFTree: a software package for the analysis of dynamic fault tree models. In *Reliability and Maintainability Symposium*, pages 64–70, Jan 1997.

[15] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi. The system availability estimator. In *Proceedings of the 16th Int. Symp. on Fault-Tolerant Computing*, pages 84–89, July 1986.

[16] H. Hermanns. *Interactive Markov Chains*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *LNCS*, 1469:51–62, 1998.

[18] H. Hermanns and J. P. Katoen. Automated compositional Markov chain generation for a plain-old telephone system. *Science of Computer Programming*, 36(1):97–127, 2000.

[19] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[20] N. Lynch and M. Tuttle. An Introduction to Input/output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[21] OMG Group. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Technical report, june 2006.

[22] W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, 1992.

[23] M. Walter, M. Siegle, and A. Bode. OpenSESAME: the simple but extensive, structured availability modeling environment. *RESS*, In Press, corrected proof, April 2007.