Automatic Success Tree-Based Reliability Analysis for the Consideration of Transient and Permanent Faults

Hananeh Aliee, Michael Glaß, Felix Reimann, and Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
{hananeh.aliee,glass,felix.reimann,teich}@cs.fau.de

Abstract—**Success tree analysis is a well-known method to quantify the dependability features of many systems. This paper presents a system-level methodology to automatically generate a success tree from a given embedded system implementation and subsequently analyzes its reliability based on a state-of-the-art Monte Carlo simulation. This enables the efficient analysis of transient as well as permanent faults while considering methods such as task and resource redundancy to compensate these. As a case study, the proposed technique is compared with two analysis techniques, successfully applied at system level: (1) a BDD-based reliability analysis technique and (2) a SAT-assisted approach, both suffering from exponential complexity in either space or time. Experimental results performed on an extensive test suite show that: (a) Opposed to the Success Tree (ST) and SAT-assisted approaches, the BDD-based approach is highly vulnerable to exhaust available memory during its construction for moderate and large test cases. (b) The proposed ST technique is competitive to the SAT-assisted analysis in analysis speed and accuracy, while being the only technique that is suitable to also handle large and complex system implementations in which permanent and transient faults may occur concurrently.**

I. Introduction

Constantly shrinking device structures allow to produce and design smaller, more efficient, and often also lower cost system components. However, this is bought by these device structures being susceptible to, e. g., environmental effects like cosmic rays, manufacturing tolerances, and aging effects. This leads to a growing inherent unreliability of the employed components [1]. In conclusion, future design methodologies need to automatically optimize and synthesize dependable embedded systems from unreliable components, not only for safety-critical, but for all types of application domains [2]. Dependability is typically not for free, but bought by certain kinds of redundancy in time, space, or energy to tolerate faults or by advanced hardening techniques during production and/or design. Thus, efficient automatic reliability analysis techniques are of utmost importance to investigate and carefully trade-off positive effects of reliability-increasing measures and their additional cost.

Existing work in the field of automatic reliability analysis can be coarsely divided into two categories: (a) Simple system models based on series-parallel structures and (b) more complex formal methods based on Binary Decision Diagrams (BDDs) [3] and related structures as well as SAT solvers [4]. While the former approaches are too simple to consider complex interdependencies of embedded systems, the latter inherently solve the *satisfiability problem* which comes at exponential complexity in either time (SAT solver) or space (BDD). To overcome this drawback, this work proposes an automatic methodology for the reliability analysis of embedded systems at system level based on *Success Trees* (STs), see [5], [6]. It concurrently considers permanent defects as a result of, e. g., aging effects that wear out components until they eventually fail permanently, and transient faults caused by, e. g. radiation inducing single event upsets that affect a single execution of a software task. The approach fully automatically (a) generates an ST for a given system implementation and (b) analyzes the success tree in the presence of transient and permanent faults by applying a state-of-the-art Monte Carlo-based simulation technique.

Fault tree analysis is a well-known method to quantify dependability parameters such as reliability and safety [7]. The top event in a fault tree is a system failure event, and the primary inputs of the tree are events that cause system failure. Success tree analysis uses the same representation of the system as fault trees, however the top event is a system success, and the primary inputs of the tree are those leading to system success. Employing ST-based analysis during design automation faces two serious challenges: (a) STs are typically employed to study the effects of interdependency and redundancy among hardware components and requires at least semi-manual construction by the designer [8]. In this work, we propose a novel construction technique that automatically extracts the ST of a system implementation, considering the dependency and redundancy of both software tasks as well as hardware resources while circumventing to implicitly solve the SAT problem as required by existing construction schemes. (b) While the actual analysis of the constructed ST is typically seamlessly applicable for hand-crafted use cases, the analysis may become a severe problem for complex and large systems or automatically derived STs [9]. In the work at hand, we employ a state-of-the-art ST analysis technique based on *stochastic logic* [10] to evaluate the generated STs. Here, each gate of the ST is translated into its corresponding stochastic logic template while a Monte Carlo simulation is used to calculate the reliability of the system.

To this end, this analysis technique provides a flexible environment to study the various effects of redundancy strategies at task level and resource level concurrently. Experimental results show that this approach can efficiently construct and analyze the success tree of a system, particularly where other existing analytical methods fail. Compared to another state-of-the-art simulation-based technique from [11], the proposed technique delivers more accurate results in a shorter or at least competitive time. The proposed automatic analysis approach enables a multi-objective design space exploration that is

Supported in part by the German Research Foundation (DFG) as associated project TE 163/16-1 of the priority program *Dependable Embedded Systems* (SPP 1500).

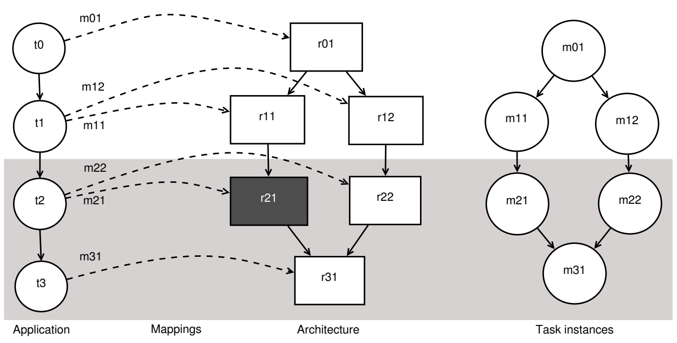

Fig. 1. A system specification consisting of an application with four tasks, six resources, and mapping edges (dashed). When allocating all resources and activating all mappings, an implementation is derived with a structure of communicating task instances depicted on the right. In this case, two redundant paths with respect to the tasks t_1 and t_2 exist.

capable of trading-off the achieved reliability for extra costs in other design objectives such as mounting space, energy consumption, or monetary costs.

The rest of the paper is outlined as follows: Section II gives an overview of the problem targeted in this paper and related work in this area. Section III introduces the proposed ST analysis approach and explains in details how the ST is constructed automatically and analyzed at system level. Finally, experimental results are discussed in Section IV while Section V concludes the paper.

II. Problem Statement and Related Work

This section first introduces the basic system model used in this work. Afterwards, the targeted problem and related approaches to solve it are discussed.

A system under design is given as a graph-based model:

- **Application:** The application of a system is modeled as a graph $g_{\mathsf{T}}(T, E_{\mathsf{T}})$, where the vertices $t_1, \ldots, t_{|T|} \in T$ denote *tasks* of the application, and the directed edges $e_1, \ldots, e_{|E_{\mathsf{T}}|} \in E_{\mathsf{T}} \in T \times T$ represent data dependencies between tasks.
- **Architecture:** A target architecture is represented as a graph $g_{\mathsf{R}}(R, E_{\mathsf{R}})$, where $r_1, \ldots, r_{|R|} \in R$ denote available interconnected hardware resources in the system, and the directed edges $e_1, \ldots, e_{|E_{\mathsf{R}}|} \in E_{\mathsf{T}} \in R \times R$ model available interconnection links between resources.
- **Mappings:** The relation between application and architecture is shown by a set of mappings. In particular, each mapping $m = (t, r) \in E_{\mathsf{M}}$ is a direct edge between a task *t* and a resource *r* and indicates that task *t* can be mapped (implemented) on resource *r*.

Thus, such a *specification* includes all the possible design alternatives, see Fig. 1 of an example system. From this, an *implementation* can be derived which represents the actual system to be implemented: An implementation consists of a subset of resources called *allocation* α that are actually used in the design and a subset of the mapping edges called *binding* β that guarantees that each task is mapped to at least one allocated resource in the system. Mapping tasks to multiple resources means creating multiple *instances* of the same task in the system, thus introducing redundancy at task level. An implementation is finally *feasible* if tasks are only mapped to allocated resources and all data-dependent tasks are executed on the same or adjacent resources.

Automatic reliability analysis approaches typically take a feasible implementation as input and evaluate the expected lifetime of this implementation with respect to a specified fault model. For this purpose, the *reliability function R* is calculated that gives the probability of the system having a life time τ_{LT} greater than time τ, i. e, $\mathcal{R}(\tau) = \mathcal{P}[\tau_{LT} > \tau]$. Given the reliability function, all well-known reliability measures, particularly the *Mean-Time-To-Failure* (MTTF) given as $\mathrm{MTTF} = \int_{\tau=0}^{\infty} \mathcal{R}(\tau)\,\mathrm{d}\tau$, can be derived seamlessly. To determine the reliability function, knowledge about the system behavior in the presence of faults is essential. Existing approaches typically determine a $characteristic\ function$ ψ first, which is a Boolean function that returns whether the system provides correct service (1) or failed (0). In this function, each allocated resource $r \in \alpha$ is modeled one-to-one by a binary variable $\boldsymbol{r}$ with $\boldsymbol{r} = 1$ indicating a properly working, defect-free resource and $\boldsymbol{r} = 0$ indicating a defective resource suffering from a permanent fault. The instances of each task are modeled by their respective mapping: $\boldsymbol{\beta} = (\boldsymbol{m_1}, \ldots, \boldsymbol{m}_{|\beta|})$ is a vector of binary variables where $\boldsymbol{m} = 1$ indicates that the task instance m is active and provides correct service and $\boldsymbol{m} = 0$, otherwise. From the simple system model employed here, and existing construction rules from literature, cf. [3], function ψ may be derived as follows:

$$\psi(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \bigwedge_{t \in T} \left[\bigvee_{m=(t,r) \in \beta} \boldsymbol{m} \right] \wedge \tag{1a}$$

$$\bigwedge_{m=(t,r) \in \beta} \boldsymbol{m} \rightarrow \boldsymbol{r} \wedge \tag{1b}$$

$$\bigwedge_{(\widetilde{t},t) \in E_{\mathsf{T}}} \bigwedge_{m=(t,r) \in \beta} \boldsymbol{m} \rightarrow \chi(m, \widetilde{t}) \tag{1c}$$

Eq. (1a) states that for each task $t \in T$, at least one active task mapping (or instance) $m = (t, r) \in \beta$ is necessary to provide correct service. Eq. (1b) states that an active task instance needs to be mapped to a non-defective resource. Finally, Eq. (1c) implies that if there are two data-dependent tasks, they must be able to communicate properly. The *communication function* χ is formulated according to the communication paradigm, i. e., single-hop or multi-hop communication, and the presence of majority voters. Based on the above definitions, the reliability analysis finally requires to solve the following *satisfiability problem* (SAT):

$$\varphi(\boldsymbol{\alpha}) = \exists_{m \in \beta} \boldsymbol{m} : \psi(\boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{2}$$

Here, the *structure function* φ evaluates to 1 if for a given allocation of resources $\boldsymbol{\alpha}$, there *exist* active mappings (task instances) m with $\boldsymbol{m} = 1$ for all tasks such that all previous constraints summarized in ψ are satisfied. State-of-the-art approaches use either BDDs [3] to represent function φ and apply an exact analytical approach or employ SAT solvers [11] to drive a simulation-based analysis. However, the BDD approach has the drawback of growing exponentially with the number of variables involved. The SAT solver on the other hand tackles the SAT problem over time, resulting in long simulation times. However, there exists no implicit SAT-solving ability for a success tree, because all primary inputs of the ST have fixed values. The work at hand avoids solving the SAT problem by proposing a novel construction scheme for success trees and an evaluation employing a recent simulation-based approach using stochastic logic. This way,

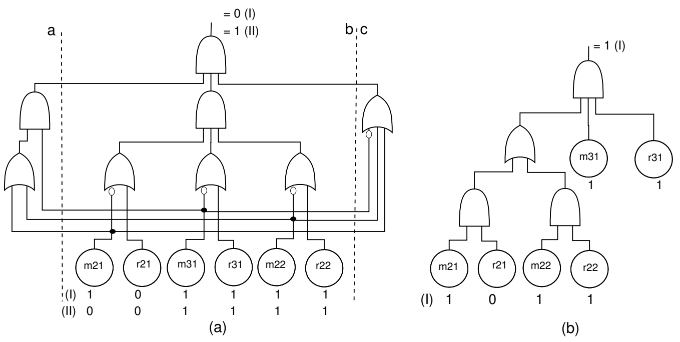

Fig. 2. The success tree generated for a part of the system shown shaded in Fig. 1: (a) Using Eq. (1), and (b) Considering a partial ordering scheme for evaluating the predecessors of a task instance first.

the problem of BDDs outgrowing available memory is avoided while being faster at higher precision compared to known SAT-based approaches.

III. Automatic Success Tree Analysis

In the following, the proposed automatic success tree analysis is introduced in three steps: (1) The construction scheme for STs, particularly discussing the need for a novel construction algorithm, (2) the evaluation of a constructed ST by a simulation-based approach, and (3) the integration of both permanent and transient faults within the ST analysis.

A. Success Tree Construction

Basically, success trees are a graphical representation of a Boolean formula by a logic circuit with arbitrary many primary inputs and one output. A success tree can be simply extracted from a given characteristic function as follows: Boolean operators are represented by logic gates, e. g., $\wedge$ by an AND gate and $\vee$ by an OR gate, and the binary variables are the primary inputs of the circuit[1].

The characteristic function introduced in Eq. (1) cannot be directly translated into a success tree as the following example shows: Consider again the system implementation depicted in Fig. 1. Suppose that resource r_{21} is defective (depicted in dark gray) and no transient fault occurred. Fig. 2 (a) illustrates the success tree of the Boolean formula related to the shaded area in Fig. 1 directly derived from Eq. (1). The success trees constructed from Eq. (1a), Eq. (1b), and Eq. (1c) are separated by a, b, and c parts respectively. Moreover, let the vector (I) hold the inputs for the corresponding allocation $\boldsymbol{\alpha}$ and binding $\boldsymbol{\beta}$. Evaluating the Boolean function (the corresponding success tree) with this vector results in a 0 although the system still provides correct service (via the redundant path over m_{12} and m_{22}). Opposed to ST, BDD and SAT-based approaches implicitly recognize that by disabling m_{21} ($\boldsymbol{m_{21}} = 0$), the function can still be fulfilled using vector (II).

To tackle this problem, Fig. 2 (b) proposes a success tree construction which evaluates all task instances in an ordered fashion, in which all predecessors of each task instance are processed first. Also, in this tree, the two possible redundant paths ending in m_{31} are considered independently, so a failure in one of them does not affect the other one. In the following, we propose a novel recursive construction scheme

Algorithm 1 Derive * Substitutions.

```
Require: $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$
Ensure: sortedTasks: topologically sorted list of tasks $T$
 1: substitutions // stores $m^* \rightarrow$ Boolean term
 2: while $t$ := sortedTasks.next() do
 3:     if $t$ has predecessor(s) $\widetilde{t}$ then
 4:         for all $(\widetilde{t}, t) \in E_T$ do
 5:             for all $m = (t, r) \in \beta$ do
 6:                 $m^* := \boldsymbol{m} \wedge \boldsymbol{r} \wedge \chi(m, \widetilde{t})$
 7:                 substitutions.put($m$, $m^*$)
 8:             end for
 9:         end for
10:     else // $t$ has no predecessors
11:         for all $m = (t, r) \in \beta$ do
12:             $m^* := \boldsymbol{m} \wedge \boldsymbol{r}$
13:             substitutions.put($m$, $m^*$)
14:         end for
15:     end if
16: end while
17: return substitutions
```

that includes partial ordering of the tasks and a substitution pattern to derive correct success trees.

To derive such a correct success tree, the dependency among tasks and resources must be properly reflected in the Boolean formula. So, whenever a permanent fault occurs in a resource r or a transient fault in a task instance m, it must only affect the impacted redundant paths, i. e., those paths that rely on the data from the faulty task or that include tasks executed on the defective resource. This may be achieved by a recursive substitution scheme of variables such that the output of gates considering preceding task instances serves as the direct input of gates considering succeeding tasks in the paths. Hence, we suppose to start the construction of the success tree by determining a map that stores the required substitutions, see Alg. 1. Suppose that application graph g_T is an acyclic graph[2]. In this algorithm, the tasks $t \in T$ are first ordered topologically with respect to their data dependencies, i. e., it holds that $\forall t, \widetilde{t} \in T, (t, \widetilde{t}) \in E_T : t < \widetilde{t}$. Hence, each task t appears in this list before any of its successors. For each task instance $m = (t, r) \in \beta$, it is ensured that the predecessor instances are evaluated first such that m is capable of detecting whether it received proper data ($\chi(m, \widetilde{t}) = 1$) and whether it itself provides correct service ($\boldsymbol{m} \wedge \boldsymbol{r} = 1$). Here, $\chi(m, \widetilde{t})$ particularly indicates the feasibility in the communication of task instance m with its predecessor $\widetilde{t}$ defined as follows:

$$\chi^{1/n}(m, \widetilde{t}) = \bigvee_{\substack{\widetilde{m}=(\widetilde{t},\widetilde{r}) \wedge \\ (\widetilde{m},m) \in E_\beta}} \widetilde{m}^* \tag{3a}$$

$$\chi^{k/n}(m, \widetilde{t}) = \bigvee_{\substack{\forall \widetilde{m_1}=(\widetilde{t},\widetilde{r_1}),\ldots,\widetilde{m_k}=(\widetilde{t},\widetilde{r_k}) \in \beta: \\ \widetilde{m_1} \neq \ldots \neq \widetilde{m_k} \wedge \\ (\widetilde{m_1},m),\ldots,(\widetilde{m_k},m) \in E_\beta}} V \wedge \widetilde{m_1}^* \wedge \ldots \wedge \widetilde{m_k}^* \tag{3b}$$

which hold for a *1-out-of-n* and a *k-out-of-n* system, respectively. Here, the edges $E_\beta \subseteq \beta \times \beta$ depict a feasible communication between task instances. Whenever a particular

[1]For explanatory purpose, we rather represent an ST also by a Boolean function instead of a graph in our methodology.

[2]The scheme for considering cyclic graphs is similar, but requires additional formulations also during analysis and shall not be discussed here.

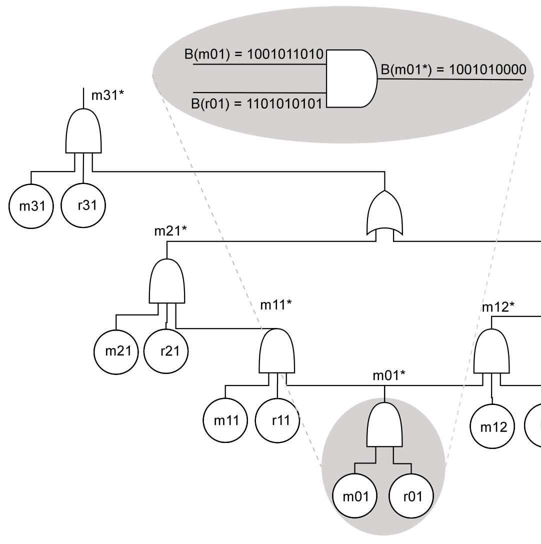

Fig. 3. The success tree of the system implementation in Fig. 1. *B(i)* refers to the bit stream representation of the success probability for the event *i*.

$\widetilde{m}^*$ appears in a formula, it is replaced by the Boolean term given by $\widetilde{m}^*$=substitutions.get($\widetilde{m}$). The result of Alg. 1 is a recursive substitution that, so far, stops for task instances that do not have predecessors. Finally, to construct the Boolean function (characteristic function) corresponding to the desired success tree, the recursion just has to be started from all task instances that do not have any successors (end tasks):

$$\psi^{1/n}(\boldsymbol{\alpha},\boldsymbol{\beta}) = \bigwedge_{t \text{ with } \nexists \widetilde{t}, (t,\widetilde{t}) \in E_T} \left[\bigvee_{m=(t,r)\in\beta} m^* \right] \tag{4a}$$

$$\psi^{k/n}(\boldsymbol{\alpha},\boldsymbol{\beta}) = \bigwedge_{t \text{ with } \nexists \widetilde{t}, (t,\widetilde{t}) \in E_T} \left[\bigvee_{\substack{\forall m_1=(t,r_1),\ldots,m_k=(t,r_k)\in\beta: \\ m_1 \neq \ldots \neq m_k}} m^* \right] \tag{4b}$$

This equation checks the possibility of correct execution for each end task in the case of *1-out-of-n* (Eq. (4a)) and *k-out-of-n* (Eq. (4b)) systems, respectively. Again note that other tasks than end tasks are not checked in this equation because they are successively inserted in the Boolean function by following the substitutions given by m^*. The resulting ST for the full example in Fig. 1 is depicted in Fig. 3.

B. Success Tree Analysis

Interpreting the success tree by providing either a 0 or 1 for each variable (primary input) in $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ determines whether the system provides correct service or fails for this particular scenario of defective resources and tasks being affected by a transient fault. Now, the key idea behind success tree analysis is to use a probability of success (or fault, respectively) instead of deterministic values for each primary input. This probability is, for each point in time τ, specified by the designer by providing a reliability function for each resource $\mathcal{R}_r(\tau)$ and task instance $\mathcal{R}_m(\tau)$. Analyzing the ST at time τ with the respective probabilities at the primary inputs delivers the probability of the whole system to work properly at time τ; implicitly delivering the desired $R(\tau)$ for the system implementation as outlined in the problem description.

In the following, the state-of-the-art stochastic fault tree analysis approach introduced in [10] is exploited. This approach models the probability of each primary input event i by a bit stream $B(i)$ of length l, where the number of 1s divided by l approximates the desired probability. In our case, the bit stream for an input i, i. e., a resource r or a task instance m, at time τ is given as:

$$B^{\tau}(i) = \{1,0\}^l \text{ with: } \frac{\#1}{l} \approx R_i(\tau) \tag{5}$$

To determine $R(\tau)$ of the overall system, a bit stream $B^{\tau}(i)$ is generated for each primary input. The ST is then interpreted in l iterations where in the j-th iteration, the j-th bit of each input bit stream is processed. Processing each iteration is determined as a separate Monte Carlo simulation run. Finally, the result is an output bit stream of the ST for which again the number of 1s divided by l approximates the probability of the complete system, hence, $\mathcal{R}(\tau)$. Fig. 3 presents an example where input streams $B(r_{01})$ and $B(m_{01})$ are processed via an AND gate (colored in gray). Suppose that $l = 10$, and $R_{r_{01}}$ and $R_{m_{01}}$ are 0.6 and 0.5, respectively. In the enlarged part of the figure, the input reliability values are presented in bit streams which are then passed through an AND gate to produce the output bit stream $B(m_{01}{}^*)$. The resulting bit stream represents $R_{m_{01}^*} = 0.3$ which is the multiplication of $R_{r_{01}}$ and $R_{m_{01}}$. Given the small l, other arrangements of 1's and 0's in the bit streams modeling $R_{r_{01}}$ and $R_{m_{01}}$ may lead to an inaccurate output. But the larger l, the more accurate the results are. As discussed in the experimental results, this technique can produce satisfiable results for $l > 2000$.

Given an ST that is only constructed once, this evaluation is invoked by simulation of $\mathcal{R}(\tau)$ over time $\tau \in [0, \tau_{max}]$, thus, enabling to calculate many reliability-related measures like the MTTF. Here, τ_{max} denotes the time that the success probability is near zero.

C. Modeling of Transient and Permanent Faults

Finally, we propose a combined success tree analysis which enables the analysis of transient as well as permanent faults. Previous works often analyze these effects separately and combine them afterwards, e. g., by adding the resulting failure rates. This, however, is not always feasible: Imagine a processor is permanently defective. The tasks executed on this processor are directly affected by being unavailable. Thus, they are affected by a permanent fault and become permanently defective themselves, although a separate analysis may assume they are only temporarily unavailable due to transient effects. Given the constructed ST as described in this section, such interdependencies may be considered correctly and hence, a concurrent analysis of transient and permanent effects is enabled. In this work, we distinguish the following effects:

Permanent Faults: In our analysis, we assume that hardware resources $r \in R$ may become permanently faulty. This may be the result of, e. g., aging effects that degrade hardware components such that they eventually violate specified characteristics. In that case, we assume that all task instances $m = (t, r) \in \beta$ mapped to such a resource r are affected by this malfunction and will not provide correct service anymore. Thus, for each such resource $r \in R$, the value of the primary input $\boldsymbol{r}$ is set to 0 or 1 according to the probability given by evaluating $\mathcal{R}_r(\tau)$.

Transient Faults: In addition to permanent faults, we also consider transient faults in hardware and we assume that these only affect individual task instances currently running on such a resource. For example, radiation effects have this kind of impact on hardware by flipping bits in either control or data path. As a result, the *transient failure rate* is directly influenced by the underlying hardware and the time duration each task instance is executed on the hardware. Assuming a given constant transient fault probability $\mathcal{P}_r$ for each resource $r \in R$ (e. g., due to radiation), for periodic tasks, the transient failure probability may coarsely be estimated for each task instance $m = (t, r)$ executed on this resource as:

$$\mathcal{P}_m = \mathcal{P}_r \cdot \frac{\tau_{\text{ET}}^m}{\tau_{\text{P}}^m} \tag{6}$$

with τ_{ET}^m being the duration of a single task instance's execution and τ_{P}^m being its period. Given this is a constant value over time, the reliability function of a task instance m is given as:

$$\mathcal{R}_m(\tau_{\text{ET}}^m, \tau_{\text{P}}^m) = 1 - \left(\mathcal{P}_r \cdot \frac{\tau_{\text{ET}}^m}{\tau_{\text{P}}^m}\right) \tag{7}$$

such that for each binary variable $\mathbf{m}$ and for each specific point in time τ, a bit stream encoding the same probability given by $\mathcal{R}_m(\tau_{\text{ET}}^m, \tau_{\text{P}}^m)$ is generated. Note that now, permanent resource defects still directly affect the task instances but there is also a probability that a task instance does not deliver correct results although the underlying resource is providing correct service; resulting in a correct modeling of the interdependency between transient and permanent faults.

IV. Experimental Results

This section evaluates the scalability, efficiency, and accuracy of the proposed automatic success-tree-based reliability analysis methodology. To this end, ST is compared with two state-of-the-art reliability analysis approaches: (1) the BDD-based approach proposed in [3], and (2) the SAT-assisted simulation presented in [11]. It should be noted that the comparison is based on the related work considering permanent faults only. The SAT-assisted simulation cannot be seamlessly adapted to consider the transient faults since it determines only one time-to-failure for each component and assumes it is permanently defective afterwards. The BDD-based approach can be adapted, but, as we will show, it heavily suffers from the increased number of variables to consider. Thus, this section will show the competitiveness of ST with existing techniques for permanent faults only. Afterwards, we will show that ST is superior when concurrently considering permanent and transient faults.

The proposed technique uses the Java-based reliability library (JReliability) [12] to analyze reliability-related measures, e. g. MTTF. Moreover, in this paper, the open-source optimization tool Opt4j [13] has been employed which provides a system-level optimization framework being also used by previous methods: SAT-assisted and BDD-based approach. The test suite employed here contains eight system-level design specifications including several real-world as well as multiple synthetic test cases ranging from simple, moderate, up to hard examples. The specification's size ranges from about 50 tasks with 25 resources up to 250 tasks with 1000 resources. For each of the specifications, 10 different implementations with a varying number of available resources are generated. To gather examples of different analysis complexity, not the sheer size is the key factor, but the degree of redundancy. Thus, each of the 80 implementations is evaluated with three different configurations, enabling hardly any redundancy (simple), a moderate degree of redundancy (moderate), and a very high degree of redundancy (hard). For the resulting test suite of 240 implementations with different complexities, each is analyzed with respect to its reliability in terms of MTTF. The experiments are carried out a standard desktop PC.

Accuracy w.r.t. Analysis Time: The BDD-based approach is precise in the sense that it gives an exact value for each required $\mathcal{R}(\tau)$ and only negligible errors occur, e. g., during numerical integration to determine MTTF. Both ST and the SAT-assisted simulation can only approximate $\mathcal{R}(\tau)$, such that an investigation of their accuracy becomes necessary. Table I investigates the achievable accuracy and speed-up of the proposed technique using stochastic logic and the SAT-based approach and compares them to exact results derived by the BDD-based approach. The table shows that with an increased number of simulation runs (equal to the length l of the bit streams in ST), the accuracy increases for both ST and SAT. For simple and moderate test cases, SAT approach is more accurate than ST, but the runtime of ST is still a factor of 3-5 lower. As will also be discussed in the next paragraph on scalability, BDD is the best option for simple and several moderate examples, being exact at a significantly shorter runtime. The important conclusion here is that for hard test cases, ST provides a higher accuracy at lower runtime compared to SAT.

Scalability: A serious issue when using the BDD-based approach is that its SAT solving capability is bought by an exponential complexity in terms of memory in the worst case. Thus, we investigate the number of examples where BDD fails to highlight the superiority of ST in terms of scalability. Consider again Table I: For simple and moderate implementations, BDD-based technique does never (simple) or hardly (moderate) fail due to outsized memory. It is noteworthy that already for the consideration of permanent defects only, BDD fails to analyze $50\,\%$ (40) of the hard implementations. In contrast, both SAT-assisted and ST-based approaches can seamlessly analyze these test cases with individual runtimes being affordable, cf. also Fig. 4. Extending the BDD-based approach to consider transient effects as well, an even larger amount of memory is required. This significantly influences its scalability with $0\,\%$ fail for simple, $20\,\%$ fail already for moderate, and $100\,\%$ fail for the hard examples. ST on the contrary can seamlessly analyze these test cases at almost the same speed as when not considering transient faults. The reason is that for permanent faults only, the $\mathbf{m}$ variables are set to all 1 but are none the less already considered.

Recommendation: (1) Considering permanent faults only, BDD is the technique of choice for examples of small and moderate complexity. For hard examples, BDD fails too frequently and ST is superior to SAT in terms of accuracy and speed. (2) Considering permanent and transient faults concurrently, BDD fails for a huge amount of examples, particularly when analysis complexity rises. ST on the contrary seamlessly evaluates these examples and is, through the unavailability of an extension for the SAT-assisted simulation for transient faults, the technique of choice for the concurrent consideration of permanent and transient faults.

TABLE I

COMPARISON OF THE (A) AVERAGE ERROR WITH RESPECT TO MTTF OF ST AND SAT, (B) THE SPEED-UP FACTOR (FOR EXAMPLE, ST OVER SAT 5.38 MEANS ST IS 5.38 TIMES FASTER THAN SAT), AND (C) THE PERCENTAGE OF EXPERIMENTS WHERE BDD AS THE EXACT TECHNIQUE FAILS.

	# of Runs	MTTF Error (%)		Speed-up Factor			Failed Experiments for BDD (%)	
		ST	SAT	ST over SAT	BDD over ST	BDD over SAT	Perm.	Perm. & Trans.
simple	500	4.45	3.85	5.38	3.96	18.48	0	0
	1000	3.29	2.54	5.85	6.80	34.4	0	0
	2000	2.72	1.89	5.99	11.91	62.02	0	0
	4000	2.30	1.39	5.95	23.50	118.80	0	0
moderate	500	4.53	3.84	3.08	5.78	15.84	2.5	20
	1000	4.08	2.77	3.09	9.42	26.19	2.5	20
	2000	3.63	1.98	3.01	19.66	52.84	7.5	20
	4000	3.02	1.35	2.91	31.44	82.97	4.0	20
hard	500	3.27	4.59	2.68	1.22	4.21	50	100
	1000	2.22	2.45	2.53	2.24	7.60	50	100
	2000	1.59	1.67	2.37	4.29	14.14	50	100
	4000	1.04	1.18	2.31	6.71	21.70	50	100

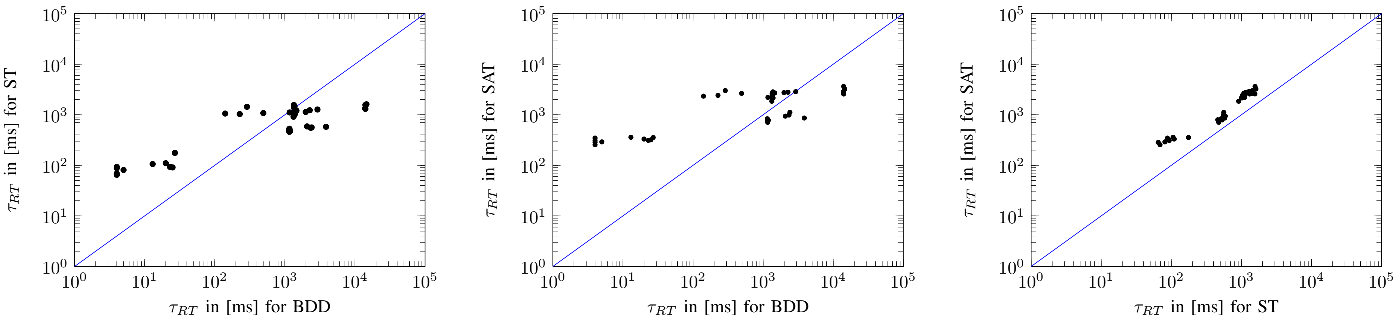

Fig. 4. Comparison of the runtime τ_{RT} of the compared techniques Success Tree (ST) analysis, BDD-based analysis and SAT-assisted simulation where BDD did not fail. As can be seen, although BDD is a lot faster on average, ST and SAT may also be faster in some cases and ST is typically faster than SAT (being also more accurate for the interesting hard examples). Noteworthy is that (given the times in milliseconds) all times are reasonable for a design space exploration.

V. CONCLUSION

This paper presents a fully automatic system-level reliability analysis methodology based on generating a success tree from a given system implementation and subsequently analyzing it based on a state-of-the-art simulation technique. Due to a novel construction scheme, it avoids the exponential growth related approaches in either memory (BDD) or time (SAT solver) by circumventing to implicitly solve the Boolean satisfiability problem. Moreover, it is capable of concurrently investigating the effects of permanent and transient faults that is currently not seamlessly implementable by the SAT-assisted simulation and tends to max-out BDD-based approaches. Compared to the existing techniques by considering only permanent faults, ST is superior in the case of complex implementations with a large degree of redundancy. In case of a concurrent consideration of permanent and transient faults, it clearly outperforms the BDD approach in terms of scalability. In summary, while BDD-based approaches are still the technique of choice for simple and moderately-sized examples, the proposed ST approach should be the preferred technique to analyze complex implementations. Given both kinds may have to be investigated during an automatic design space exploration, the novel ST technique provides an efficient and highly scalable solution to the reliability analysis problem.

REFERENCES

[1] N. Miskov-Zivanov and D. Marculescu, "Multiple transinet faults in combinational and sequential circuits: A systematic approac," *IEEE Transactions on Computer-Aided Dsign of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1614–1627, 2010.

[2] T. Streichert, M. Glaß, C. Haubelt, and J. Teich, "Design space exploration of reliable networked embedded systems," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 53, no. 10, pp. 751–763, 2007.

[3] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, and J. Teich, "Reliability-aware system synthesis," in *Design, Automation and Test in Europe (DATE)*. France: IEEE Computer Society, 2007, pp. 409–414.

[4] M. Glaß, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich, "Symbolic system level reliability analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. USA: IEEE Computer Society, ACM, 2010, pp. 185–189.

[5] A. Reibman and M. Veeraraghavan, "Reliability modeling: An overview for system designers," *Computer*, vol. 24, no. 4, pp. 49–57, 1991.

[6] N. Rasmussen, "The application of probabilistic risk assessment techniques to energy technologies," *Annual Review of Energy*, vol. 6, no. 1, pp. 123–138, 1981.

[7] J. Dugan, K. Sullivan, and D. Coppit, "Developing a low cost high-quality software tool for dynamic fault-tree analysis," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 49–59, 2000.

[8] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick, and J. Railsback, *Fault Tree Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance, 2002, pp. 1–218.

[9] G. Merle, J. Roussel, J. Lesage, and A. Bobbio, "Probabilistic algebraic analysis of fault trees with priority dynamic gates and repeated events," *IEEE Transactions on Reliability*, vol. 59, no. 1, pp. 250–261, 2010.

[10] H. Aliee and H. Zarandi, "Fault tree analysis using stochastic logic: A reliable and high speed computing," in *Annual Reliability and Maintainability Symposium (RAMS)*. USA: IEEE Computer Society, 2011, pp. 1–6.

[11] M. Glaß, M. Lukasiewycz, C. Haubelt, and J. Teich, "Towards scalable system-level reliability analysis," in *Proceedings of the 2010 ACM/EDAC/IEEE Design Automation Conference (DAC '10)*, Anaheim, USA, Jun. 2010, pp. 234–239.

[12] M. Glaß, M. Lukasiewycz, and F. Reimann, *Java-based Reliability Library*, 2008 (accessed February, 2012). [Online]. Available: http://jreliability.sourceforge.net/

[13] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4j - a modular framework for meta-heuristic optimization," in *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, Ireland, 2011, pp. 1723–1730.