This is the author's version of the work. The definitive work was published in *Proceedings of Design, Automation and Test in Europe (DATE 2009), pp. 320-323, 2009.* The work is supported in part by the German Science Foundation (DFG), SFB 694.

Incorporating Graceful Degradation into Embedded System Design

Michael Glaß, Martin Lukasiewycz, Christian Haubelt, and Jürgen Teich
University of Erlangen-Nuremberg, Germany
{glass,martin.lukasiewycz,haubelt,teich}@cs.fau.de

Abstract

In this work, the focus is put on the behavior of a system in case a fault occurs that disables the system from executing its applications. Instead of executing a random subset of the applications depending on the fault, an approach is presented that optimizes the systems structure and behavior with respect to a possible graceful degradation. *It includes a degradation-aware reliability analysis that guides the optimization of the resource allocation and function distribution, and provides data-structures for an efficient online degradation algorithm. Thus, the proposed methodology covers both, the design phase with a structural optimization and the online phase with a behavioral optimization of the system. A case study shows the effectiveness of the proposed approach.*

1. Introduction and Related Work

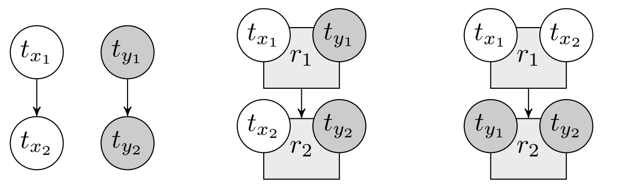
In a real-world system, failures that disable the system from executing its applications are inevitable in the presence of cost and performance constraints. In case a set of failures disables the system from carrying out all applications, a subset of less important applications can be dropped while the more important applications can be kept alive. This concept is known as *graceful degradation*, i.e., *functional degradation* [9].

Consider the example in Figure 1 of a simple system consisting of two applications x and y with two tasks per application. At design time, both shown implementations have the same overall system reliability, since a failure of one resource r_1 or r_2 will lead to a failure of at least one application and, thus, the overall system functionality is not executable anymore. Assuming application x being of higher priority than application y, a failure of, e.g., resource r_2 still allows to execute application x. Thus, implementation 2 should be preferred by the used optimization algorithm at design time.

In this work, a degradation-aware reliability analysis is proposed that permits to quantify the reliability of the system in the presence of functional graceful degradation. This allows the *design space exploration* to optimize the resource allocation and function distribution of the embedded system under design and, thus, saves costs while increasing system performance.

At run time, less important applications can be specifically deactivated to gain free resources for more important applications. For this purpose, a data-structure that can already be derived at design time is presented that guides a graceful degradation algorithm at run time. Based on binary decision diagrams (BDDs) [1], the structure enables a very efficient check of the current system state and provides rules when and which applications to shut down.

A lot of work has been done in the area of graceful degradation techniques. The approaches in [4, 11] are more focused on the problem of *performance degradation* where the performance



(a) application (b) implementation 1 (c) implementation 2

Figure 1. Two implementations with identical overall system reliability but differing degradation possibilities.

of an application is degraded instead of a complete shutdown of less important applications.

In [7], tasks are reconfigured in the system with respect to the available hardware. In the work presented in this paper, no complete reconfiguration of the system is allowed but predefined resource allocations and task bindings have to be selected as a reaction to failures in the system. An similar approach to the work at hand is proposed in [12], but is restricted to *series-parallel* systems where structural resource sharing is neglected. In the analysis proposed in this work, this problem is solved by using an extended BDD-based reliability analysis.

The remainder of the paper is outlined as follows: Section 2 introduces the system model. The degradation-aware reliability analysis is proposed in Section 3. The online algorithm is presented in Section 4. Section 5 shows the introduced techniques applied to a case study before the paper is concluded in Section 6.

2. System Model

The system *specification* is given in a graph-based manner. The specification consists of a so called *application* that models the applications and their tasks, the *architecture* that includes all resources that can be used, and a set of mapping constraints. An example of a system specification is shown in Figure 2.

The application is modeled by a task graph $g_\mathsf{t}(V_\mathsf{t}, E_\mathsf{t})$ that describes the behavior of the system. The vertices $t \in V_\mathsf{t}$ denote tasks, the directed edges E_t represent data dependencies between the tasks.

The architecture is modeled by a resource graph $g_\mathsf{a}(V_\mathsf{a}, E_\mathsf{a})$ and represents possible interconnected hardware resources. The vertices $r \in V_\mathsf{a}$ represent the individual resources, e.g., processors, buses, sensors, or actuators. The edges E_a correspond to available communication links.

The set of mapping constraints is given by a set of *mapping edges* E_m. Each mapping edge $m \in E_\mathsf{m}$ is a directed edge from a task to a resource that indicates whether a specific task can be executed on a hardware resource.

Given a specification, the goal of design space exploration is to find a set of high-quality *feasible implementations*. An implementation consists of two parts: The *allocation* $\alpha \subseteq V_\mathsf{a}$ represents the resources that will actually be used in the implementation and the *binding* $\beta \subseteq E_\mathsf{m}$ determines on which al-

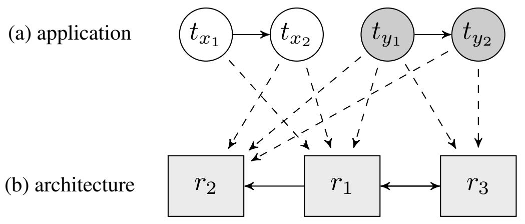


Figure 2. A system specification. The dashed edges correspond to a possible mapping of tasks to resources.

located resource each *task instance* can be activated and, thus, executed. With this knowledge, we define an implementation as a pair (α, β).

Definition 1. *An implementation is called* feasible *if it guarantees*

- *that each task is bound at least once and is bound to allocated resources only, and*
- *that data-dependent tasks can communicate correctly.*

Definition 2. *In the online phase, the implementation* works properly *if*

- *at least one instance of each task is activated,*
- *tasks instances are only activated on properly working, i.e., not failed resources,*
- *activated task instances of data-dependent tasks communicate correctly, and*
- *for each resource, the computational demand of activated task instances does not exceed the computational capacity of the resource.*

Although each application can be regarded as an individual functionality that can be influenced by failures, a more effective and applicable way is to cluster applications. Each *degradation mode* $d \in D$ is defined by a subset of tasks that have to work properly. The degradation modes are typically specified by the designer. In case failures occur that hinders the implementation to work properly in the current degradation mode, applications can be turned off by choosing a lower degradation mode using an online degradation algorithm. In this paper, for two degradation modes d_i and d_j with $i < j$, it holds that the tasks of the mode d_i are a subset of the tasks of the mode d_j such that a degradation from d_j to d_i only improves the reliability of the system.

3. Degradation-Aware Reliability Analysis

A so called *system function* $\psi(\boldsymbol{\alpha}, \boldsymbol{\beta})$ encodes one implementation as a Boolean function. This function returns 1 if the implementation is *properly working* for the state of resources encoded in $\boldsymbol{\alpha}$ as well as a task activation $\boldsymbol{\beta}$, and 0 otherwise. The binary vector $\boldsymbol{\alpha}$ contains a variable $\boldsymbol{r}$ for each resource $r \in V_\mathsf{a}$ that indicates whether the corresponding resource is defect (0) or not (1). Correspondingly, $\boldsymbol{\beta}$ contains a variable $\boldsymbol{m}$ for each task instance or mapping $m \in E_\mathsf{m}$ of the implementation, respectively, indicating whether the task instance is activated (1) or not (0). The determination of ψ for implementations without the aspect of degradation is presented in [3].

In this paper, the system function ψ is extended by incorporating the degradation modes as well as computational constraints that arise from the activation of task instances. For this purpose, the degradation modes are encoded by a binary vector $\boldsymbol{D} = (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|D|}})$ with $\boldsymbol{d_i} = \{0, 1\}$. Each variable $\boldsymbol{d_i}$ is 1 if the degradation mode d_i is active or 0 otherwise. The resulting system function ψ is the following Boolean function:

$$\psi(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{D}) = \bigwedge_{d \in D} [\neg \boldsymbol{d} \vee \psi_d(\boldsymbol{\alpha}, \boldsymbol{\beta})] \bigwedge_{r \in \alpha} C_r(\boldsymbol{\beta}) \qquad (1)$$

The system function $\psi_d(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is determined as in previous approaches (cf. [3]), but allows to ignore the tasks that are not part of the degradation mode $d \in D$. This $\psi_d(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is enabled if and only if the corresponding variable $\boldsymbol{d}$ is 1.

The computational constraints for each resource $r \in \alpha$ are given in the following linear form:

$$C_r(\boldsymbol{\beta}) = \sum_{m=(t,r) \in \beta} l_m \cdot \boldsymbol{m} \leq c_r \qquad (2)$$

Here, l_m denotes the computational load arising from activating task instance m which is bound to resource r while the maximum computational capacity of resource r is denoted as c_r. Incorporating the degradation modes directly into the system function $\psi(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{D})$ by using the binary vector $\boldsymbol{D}$, allows the analysis of the system in each mode without the need to store and calculate individual functions for each mode.

The functions are represented by binary decision diagrams (BDDs) [1]. Although, this data structure may grow exponentially in general, it is known to be a very compact representation for Boolean functions in many applications. A transformation scheme for the linear computational constraints into BDDs is presented in [2].

To model the system behavior under the influence of defects, the *structure function* $\varphi : \{0,1\}^{|\alpha|} \times \{0,1\}^{|D|} \to \{0,1\}$ is incorporated. This function φ returns 1 if and only if for the state of resources encoded in $\boldsymbol{\alpha}$ and the current degradation mode encoded in $\boldsymbol{D}$, there exists at least one task activation $\boldsymbol{\beta}$ that ensures a properly working implementation. For a given system function ψ, this function can be derived as follows:

$$\varphi(\boldsymbol{\alpha}, \boldsymbol{D}) = \exists \boldsymbol{\beta} : \psi(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{D}) \qquad (3)$$

The *structure function* φ_d for a specific degradation mode $d \in D$ is deduced by restricting this function. The function φ_d returns 1 if and only if for the state of resources encoded in $\boldsymbol{\alpha}$ and the specific degradation mode d, there exists at least one task activation β that ensures a properly working implementation. For each $d \in D$ this function is derived as follows:

$$\varphi_d(\boldsymbol{\alpha}) = \varphi(\boldsymbol{\alpha}, \boldsymbol{D}) \wedge \boldsymbol{d} \bigwedge_{\tilde{d} \in D \setminus d} \neg \tilde{\boldsymbol{d}} \qquad (4)$$

In the following, we assume that the reliability function $R(r, x)$ that delivers the reliability for each resource $r \in V_a$ at a time $x \in \mathbb{R}_{\geq 0}$ is known or can be approximated. The reliability function $R_d(x)$ defines the reliability for the tasks of the corresponding degradation mode $d \in D$ and is derived by *Shannon-decomposition* [10] that can be applied efficiently on the used BDD data structure:

$$R_d(x) = R(r, x) \cdot \varphi_d|_{\boldsymbol{r}=1} + (1 - R(r, x)) \cdot \varphi_d|_{\boldsymbol{r}=0} \qquad (5)$$

In our experimental results, the *Mean Time To Failure* (MTTF) $\int_0^\infty R_d(x)dx$ is used as the measure of reliability and is determined by a numerical integration of Equation (5). The MTTF

Algorithm 1 Online graceful degradation algorithm.

```
1: $\boldsymbol{D} := (0,\ldots,0,1)$ // initial state of degradation
2: $\boldsymbol{\alpha} := (1,\ldots,1)$ // initial resource state
3: $\boldsymbol{\beta} \in \{\boldsymbol{\beta}|\psi(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{D}) = 1\}$ // initial activation
4: while true do
5:     $\boldsymbol{\alpha}$ := observe() // observe
6:     if $!\exists\boldsymbol{\beta} : \psi(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{D})$ then
7:         $\boldsymbol{D} := \max_{\mathrm{D}}\{\boldsymbol{D}|\psi(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{D}) = 1\}$ // degrade
8:     end if
9:     $\boldsymbol{\beta} := \min_{\mathrm{dist}}\{\boldsymbol{\beta}|\psi(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{D}) = 1\}$ // new activation
10: end while
```

denotes the expected value for the failure-free time of the functionality of the corresponding degradation mode $d \in D$.

In former approaches [3, 13, 14], only the reliability with respect to the overall system functionality was used as an optimization criterion. In this work, the reliability of each degradation mode is respected and needs to be considered within the optimization. Here, a true multi-objective approach would treat the reliability of each degradation mode as an independent and competing objective. This results in $|D|$ competing objectives that tend to decrease the effectiveness of the used optimization approach. Thus, a new objective with the characteristics of a weighted reliability is proposed:

$$R_{MTTF} = \sum_{d \in D} (w_d \cdot \int_0^\infty R_d(x)dx), \text{ with} \sum_{d \in D} w_d = 1 \quad (6)$$

Weighting the MTTF of each degradation mode allows to shrink the number of reliability-specific objectives to one and, thus, represents an intuitive value to the designer. Moreover, the designer can weight the degradation modes with respect to the specification or usage of the system. For example, *full functionality* (d_3) as well as *safety-critical functionality* (d_1) can be weighted such that they strongly influence the objective while other degradation modes are optimized to a lesser extent. Other systems may need to focus in the safety-critical functionality only while full or *comfort functionality* (d_2) is of minor interest.

4. Online Graceful Degradation

In the online phase of a given system, graceful degradation takes place whenever the system resources are, due to defects, not able to carry out all the applications that are specified in the selected degradation mode. Here, an algorithm has to decide at run time if another degradation mode needs to be chosen. Moreover, due to the given constraints in computational load and communication, the instances of the executed tasks may need to be activated and deactivated at a certain state of degradation whenever a resource fails. In this section, an algorithm that is based on a data structure that can be derived from the system structure ψ is presented.

The proposed algorithm is based on an *observer* structure, i.e., there is one resource or a set of resources that is highly reliable and can detect defects of all resources in the system. In particular, this observer is able to keep track of the binary vector $\boldsymbol{\alpha}$ that encodes working and defect resources and controls the current state of degradation. Moreover, it decides which task instances $\boldsymbol{\beta}$ have to be activated or deactivated, respectively. The overall algorithm is outlined in Algorithm 1.

It is assumed that the system starts without any defects among the resources (line 2) and, thus, no degradation has taken part so far (line 1). For this initial state, an activation for the task instances can be derived from the system function ψ (line 3).

The failure detection and, thus, the update of the $\boldsymbol{\alpha}$ vector is performed by the function *observe()* (line 5). In case a failure occurs, it is tested whether there exists a feasible task activation with respect to the currently properly working resources and the current degradation mode (line 6). If the test fails, the function $\max_{\mathrm{D}}$ searches for the degradation mode of highest importance that can be carried out on the given set of working and defect resources $\boldsymbol{\alpha}$. Here, the actual degradation takes place (line 7). Finally, for a given $\boldsymbol{\alpha}$ and new $\boldsymbol{D}$, a new properly working implementation has to be achieved by a correct task activation $\boldsymbol{\beta}$ (line 9). This function $\min_{\mathrm{dist}}$ aims to find a task activation that is *similar* to the former task activation to avoid the unnecessary activation and deactivation of tasks and, thus, keeps performance reduction at a minimum. This is achieved by a special variable order of ψ that can already be done at design time:

$$\boldsymbol{d_1} < \ldots < \boldsymbol{d_{|D|}} < \boldsymbol{\alpha_0} < \ldots < \boldsymbol{\alpha_{|\alpha|}} < \boldsymbol{\beta} \quad (7)$$

Given this variable order, the degradation mode and the resource variables are setup first since they are fixed whenever the algorithm searches for a new task activation. The internal structure of a BDD is a tree with one root and a *true* and *false* terminal node. Each node of the tree represents a variable and has two outgoing edges that equal an assignment of *true* or *false*, respectively, to the corresponding variable. Thus, each traversal of the BDD from the root that reaches the *true* terminal node corresponds to a variable assignment that equals a properly working implementation. Here, the traversal for the $\boldsymbol{\beta}$ variables is guided by the previous task activation to reach the *most similar* task activation for the given BDD order. To prevent that an infeasible activation (*false* terminal node) is reached, for each node a look-ahead for both outgoing edges has to be done resulting in the overall traversal complexity of $O(2 \cdot |\boldsymbol{\alpha}| \cdot |\boldsymbol{\beta}| \cdot |\boldsymbol{D}|)$. This linear complexity also holds for the other operations of the algorithm (line 3,6,7) if ψ is given as a BDD as suggested.

5. Case Study

In this section, the proposed degradation-aware reliability analysis used in a design space exploration context as well as the presented online algorithm are applied to a system specification consisting of 11 ECUs connected to one bus that is inspired by typical automotive ECU networks. This specification consists of 6 applications with an overall number of 51 tasks. One application is *safety*-critical (d_1), two applications can be seen as *standard* functionality (d_2) while the remaining three applications are treated as *comfort* functionality (d_3). Each task of each application can be carried out by at least three ECUs, leading to a design space of $\approx 2^{80}$ implementations.

The proposed degradation-aware reliability analysis was included in JRELIABILITY [5] and used in the multi-objective design space exploration [6] tool OPT4J [8]. For this purpose, the results for the proposed weighted reliability were compared to a *common* reliability analysis that does only evaluate overall system functionality. The experiments were carried out on an Intel Pentium 4 3.0 GHz computer with 1.5 GB RAM. The results from the design space exploration are shown in Figure 3.

Shown is a two dimensional projection of a three dimensional objective space consisting of the objectives reliability, *area* consumption, and *power* consumption. The results show

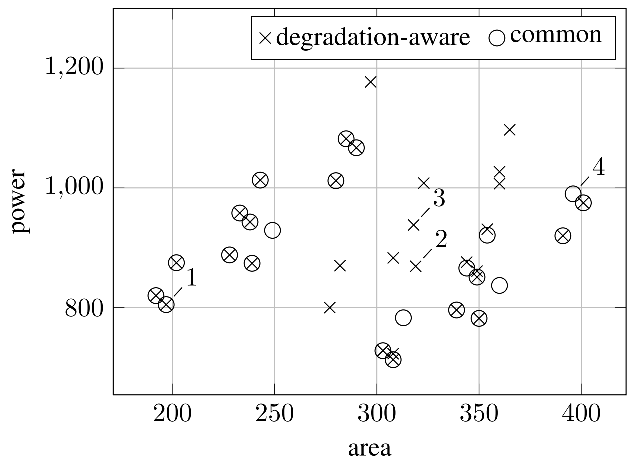


Figure 3. A two dimensional projection showing the best implementations found by a common and the novel degradation-aware optimization.

	area	power	MTTF [in years]		
			full	standard	safety
impl1	197	805	3.65	3.65	20.59
impl2	319	869	2.77	2.77	82.92
impl3	318	938	4.28	5.37	245.27
impl4	396	990	4.35	4.35	266.81

Table 1. Selected implementations and their properties. *impl2* **and** *impl3* **have only been found by incorporating graceful degradation.**

that the common approach is able to find relatively low-area implementations with low overall reliability and more expensive implementations with a higher overall reliability. Hence, it cannot find the implementations in between, where standard and safety-critical applications can achieve a higher reliability at lower costs compared to the implementations that have a high overall reliability. Thus, the incorporation of graceful degradation aspects in the analysis offers high-quality implementations that would not have been found otherwise. Table 1 shows some selected implementations and their objectives.

The introduced online algorithm was applied to the simplest and most complex feasible implementation regarding the induced structural redundancy of the given specification to enable the analysis of its scalability. Here, a Monte-Carlo simulation-based approach was used that generated $10,000$ failure scenarios, each starting with a completely working system and simulating the time until complete system failure is reached. The simulation was carried out on an embedded processor. The results are shown in Table 2.

The memory needed for the storage of the BDD data structure was $\approx 1.5kB$ to $\approx 175kB$. These additional memory requirements are acceptable for state-of-the-art ECUs. The computational time of the Algorithm 1 for reacting to failure ranges from $8.8\mu s$ to $21.1\mu s$ in average and is negligible small. Thus, the algorithm is also applicable for time-critical applications. Moreover, the execution time for Algorithm 1 in combination with the BDD data structure shows a very good scalability.

	BDD		time [μs]
	#nodes	size [kBytes]	
simple	60	1.44	8.8
complex	7198	172.75	21.1

Table 2. Proposed online algorithm evaluating the smallest and the most complex feasible implementation.

6. Conclusion

In this paper, a methodology to include graceful degradation into embedded system design, i.e., into the phase of design space exploration and an online algorithm for the online degradation and reactivation of tasks was presented. For the optimization, a new degradation-aware reliability analysis allows to take the systems ability to degrade into account and to quantify this ability in the reliability objective. In the online phase, an efficient degradation algorithm based on a BDD data structure that can be derived at design time allows an efficient task activation and performs a graceful degradation if needed. Both, static optimization and online runtimes show good scalability and perform well on the presented case study. In future work, different data structures than the memory intensive BDDs will be investigated for the online algorithm to trade-off memory requirements vs. runtime.

References

[1] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *Trans. on Computers*, 35(8):677–691, 1986.

[2] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

[3] M. Glaß et al. Symbolic Reliability Analysis and Optimization of ECU Networks. In *Proc. of DATE '08*, pages 158–163, 2008.

[4] O. Gonzalez et al. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proc. of RTSS '97*, pages 79–89, 1997.

[5] JReliability. The java-based reliability library. http://www.jreliability.org/, Version 1.2.

[6] M. Lukasiewycz et al. Efficient symbolic multiobjective design space exploration. In *Proc. of ASP-DAC '08*, pages 691–696, 2008.

[7] W. Nace and P. Koopman. A graceful degradation framework for distributed embedded systems. In *Workshop on Reliability in Embedded Systems*, 2001.

[8] Opt4J. The optimization framework for java. http://www.opt4j.org/, Version 1.3.

[9] B. Randell et al. Reliability Issues in Computing System Design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

[10] A. Rauzy. New Algorithms for Fault Tree Analysis. *Reliability Eng. and System Safety*, 40:202–211, 1993.

[11] K. Shin and C. Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *Proc. of ECRTS '99*, pages 29–36, 1999.

[12] H. Taboada et al. MOMS-GA: A multi-objective multi-state genetic algorithm for system reliability optimization design problems. *Trans. on. Rel.*, 57(1):182–191, 2008.

[13] S. Tosun et al. Reliability-Centric High-Level Synthesis. In *Proc. of DATE '05*, pages 1258–1263, 2005.

[14] Y. Xie et al. Reliability-Aware Cosynthesis for Embedded Systems. In *Proc. of ASAP '04*, pages 41–50, 2004.