Extending Fault Tolerance Patterns by Visual Degradation Rules*

Matthias Tichy and Holger Giese
University of Paderborn, Software Engineering Group
Warburger Str. 100, 33095 Paderborn, Germany
[mtt,hg]@upb.de

Abstract

Embedded distributed systems play an important role in many advanced technical systems. In order to satisfy high availability and reliability requirements, fault tolerance techniques such as triple modular redundancy are employed. In addition, techniques for graceful degradation are required to handle situations when a system experiences too many faults to compensate them while still providing a reduced albeit sufficient functionality.

As a formal visual specification technique to describe known standard fault tolerance solutions we proposed fault tolerance patterns *[24] which capture the essential structure and relevant deployment restrictions of these solutions. Fault tolerance patterns are easily applied during the design of component-based systems to increase the reliability or availability of specific components or subsystems and permit to derive a correct initial deployment and guide the self-repair of the system.*

In this paper, we extend our fault tolerance pattern approach with additional visual degradation rules*. The rules can at first be employed to define reconfiguration steps for the system which reduce the provided level of fault tolerance while retaining the provision of functional properties. Secondly, steps which result in a graceful degradation and thus only a reduced functionality can be defined.*

1. Introduction

Today, distributed embedded systems are of paramount importance for the dependable operation of many advanced technical systems (cf. [22, 13]). In such systems reliability and availability are important non-functional attributes of embedded systems. As hardware failures can and will happen for some nodes of such complex distributed embedded systems, the systems must have appropriate failure handling capabilities. One possible direction are fault tolerance techniques [21] which employ redundant components in the space, time and/or information domain to improve the fault tolerance capabilities of a design. Techniques for graceful degradation can be employed to handle situations when a system experiences too many faults to compensate them while still providing a degraded level of functionality (cf. [27, 10]).

As the manual design and implementation of fault tolerance techniques is a complex and error prone task, we proposed to use *fault tolerance patterns* [24] as a formal visual specification technique to describe the structure and deployment restrictions of known standard fault tolerance techniques. The patterns support to easily apply well known strategies by simply reusing the patterns for components or structures of components where special attention for their reliability or availability is required. The deployment restrictions provided by the patterns can be further used to check that a given deployment respects the deployment restrictions of the pattern to exclude common-mode failures which could result if multiple redundant components are erroneously deployed on the same node. In addition, a correct initial deployment with respect to the deployment restrictions can be derived automatically.

While the fault tolerance technique, which are realized via the patterns, usually tolerate single hardware faults, additional hardware faults which may accumulate over time cannot be tolerated. Therefore, in practice a manual repair of the embedded system must be performed to restore the capability to tolerate faults. However, in many application domains manual repair is either impossible, only possible after rather long operation periods, or is more costly than spending more for the initial standby hardware. Thus, techniques which enable the self-repair or self-healing of the system without explicit maintenance would improve the situation considerably.

Several concepts [8, 7, 3, 4, 26] are proposed today which let the system reconfigure itself to repair detected

*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

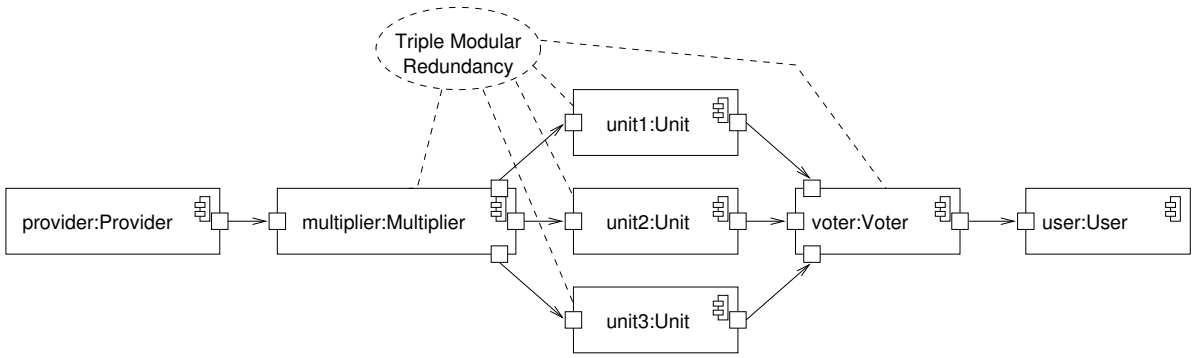


Abbildung 1. Triple Modular Redundancy Pattern

problems (cf. self-healing [16]).[1] A number of these approaches [4, 3, 26] suggest to derive a valid deployment mapping for self-repair actions in the presence of node crashes. In [26] a self-repair algorithm which is in line with these other approaches for fault tolerance pattern has been presented. If fault tolerant software components fail during runtime, the algorithm redeploys them in order to maintain the fault tolerance capabilities of the system. In case of crash failures at run-time, the deployment information given by the deployment restrictions of the fault tolerance patterns are used to determine an appropriate redeployment of the failed components. This approach has been further optimized in [25] looking for an optimal compromise between repair time and damage minimization.

In this paper, we extend the outlined fault tolerance pattern approach by means of visual *degradation rules*. These rules describe which elements of a given pattern can be removed to reduce the resource consumption of the configuration and which reduced non-functional and functional quality results. If only the reliability or availability of the subsystem is affected, we denote such a rule to be a *dependability degradation rule*. We name it a *graceful degradation rules* if also the functionality provided is restricted or has only a reduced quality.

The paper is organized as follows: The underlying concepts of the fault tolerance patterns are introduced in Section 2. Our extension in form of degradation rules follow in Section 3. Then, the extension of our approach for the self-repair which takes degradation rules into account is sketched in Section 4 and related work is discussed in Section 5. Finally, we conclude the paper and give an outlook on planned future work.

[1] While dynamic reconfiguration is not recommended by more traditional approaches (see IEC 61508), also prominent proposal exists which see reconfiguration as a key to dependable systems [23].

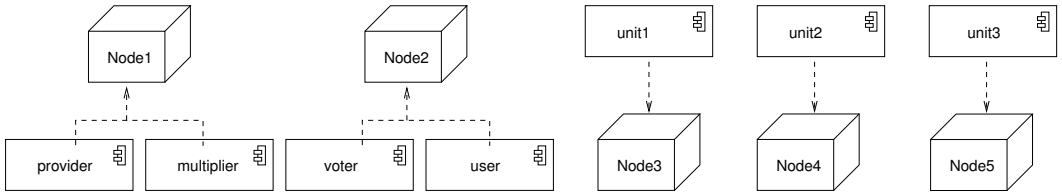
2. Fault Tolerance Patterns

Fault Tolerance Techniques have a long history in the development of dependable systems. Fault Tolerance Patterns [24] capture the structure and deployment restrictions of well known fault tolerance techniques in a formal and visual manner. We outline the underlying concepts in the following providing two examples: the triple modular redundancy pattern and the recovery block pattern. These examples are subsequently employed to present our extension with degradation rules.

2.1. Triple Modular Redundancy

Figure 1 shows the structural specification of the triple modular redundancy (TMR) fault tolerance pattern described by UML 2.0 [15] component diagrams. A triple modular redundancy system uses a multiplier component which triples the input received and forwards it to the three components unit1 . . . 3, which actually perform the computation. The voter compares the different results and chooses the result which at least two of the components returned. Thus, a triple modular redundancy system can tolerate one crashed or malfunctioning component. The user and provider components are not part of the fault tolerance pattern but are important for connecting the using and used components during application of the pattern. See [26] for an application of the triple modular redundancy pattern to an example from the railway domain.

In the case of improper deployment of the different components of the pattern, common-mode failures spoil the fault tolerance enhancement of a triple modular redundancy setup. For example, if two of these three redundant copies are executed on the same node, crash failure independence does not hold anymore for node failures and the usage of a TMR becomes pointless. Thus, the components unit1. . . 3 must be deployed to distinct nodes. The multiplier and voter as well as the provider and user components are single



$\{Node3.CPU \neq Node4.CPU \wedge Node3.CPU \neq Node5.CPU \wedge Node4.CPU \neq node5.CPU\}$

Abbildung 2. Deployment Restrictions of the TMR Pattern

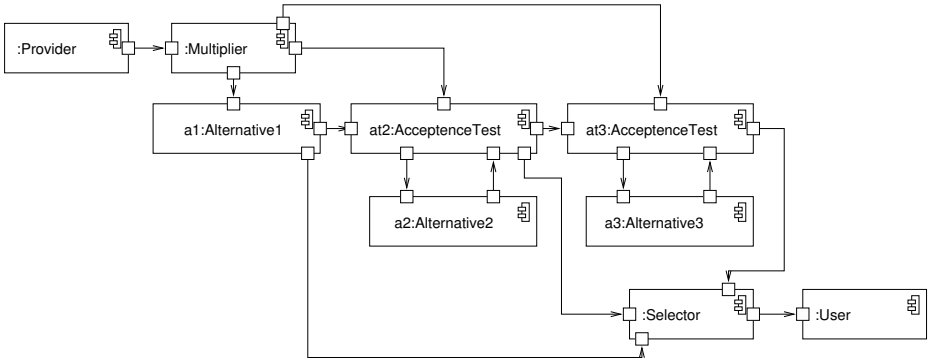


Abbildung 3. Recovery Block Pattern

points of failure in a simple application of TMR. Our observation is, that if a user component fails, the voter is not needed anymore. Thus, in order to enhance the fault tolerance of the TMR setup, we propose to deploy the provider and multiplier as well as the voter and user to the same node, i.e. both components do not crash fail independently of each other. In addition, the nodes executing the redundant copies should differ in order to tolerate hardware design faults. Figure 2 shows the deployment restrictions of the fault tolerance pattern of Figure 1 with a UML 2.0 deployment diagram. Such a graphical specification as a diagram typically provides better readability than a textual representation.

2.2. Recovery Blocks

As second example for a fault tolerance pattern we consider recovery blocks. Recovery Blocks [11, 1] aim at improving reliability due to different implementations for a specific function with different qualities and possibly different input data. An implementation of this technique consists of a number of alternative implementations and an acceptance test which is executed after each alternative. The acceptance test checks the values computed by the alternative. If the check does not execute successfully, the next alternative is executed.

In contrast to standard recovery blocks, our recovery block pattern targets distributed systems just as the approach presented in [12]. In order to tolerate crash failures, we replicate the different alternatives and the acceptance tests as shown in Figure 3. During deployment of the recovery block pattern elements, a copy of the acceptance test and one alternative are deployed as a pair to the same node whereas the different pairs are deployed to different nodes. Provider and multiplier as well as selector and user are deployed to the same node due to the same argument as given above in the description of the TMR deployment constraints.

2.3. Redeployment and Self-Repair

In [26] it is shown, how the deployment problem considering the above presented graphical deployment restrictions is translated into a standard integer problem which is solvable by standard constraint solvers like e.g. ILOG's Solver software.

Basically, for each component-node combination a boolean variable $x_{i,j}$ is used. The constraint solver then can either set the variable to 1 denoting that the component i should be deployed to node j or set it to 0 for the other case. Then, the following consistency constraint is required to ensure that each component is located on exactly one node (cf. [25]):

$$\forall j : \sum_{i \in C} x_{i,j} = 1 \tag{1}$$

The graphical deployment constraints are also transformed into constraints over these boolean variables. The constraint $\forall j : x_{unit1,j} + x_{unit2,j} \leq 1$ captures the deployment restriction that the two components $unit_1$ and $unit_2$ must not be deployed to the same node j. For r_i the resources required for each component i and u_j the resources available on a node j, we have to additionally ensure that $\forall i : \sum_{j \in N} x_{i,j} r_i \leq u_j$ holds.

If hardware resources fail, software components fail, too. As the hardware faults can either affect non-redundant components or accumulate over time, we support that the system can do a self-repair by redeployment of the failed software components. In [26, 25], we presented an approach to compute repair actions in a timely way, which are optimal w.r.t. damage of the component failures.

At first, the proposed algorithm tries to find a new deployment node for the failed software components. If a simple redeployment of just these components is not feasible due restrictions on the underlying hardware platform, other software components may be migrated in order to find suitable available hardware resources for the failed software components.

If the load of the hardware resources is high, the failure of just one hardware resource may lead to the situation that a redeployment of the failed software components may not be possible. If redundant copies of these failed software components are still working, the damage which is implied by this failure is rather small. But if non-redundant components are affected, the damage to the system is greater. Thus, we need to make resources available to redeploy those components. In the following, we therefore present degradation rules which aim at describing our options to repair a system by also taking the degrading of its non-functional or even functional properties into account.

3. Degradation Rules

In Section 2, we presented fault tolerance patterns which capture structure and deployment restrictions of existing fault tolerance techniques in a formal way. Thus, they are easily applicable to software models. Typically, fault tolerance techniques employ redundancy in order to tolerate failures. The redundant pieces of software either are identical copies (e.g. Triple Modular Redundancy) or provide different levels of functionality (e.g. Recovery Blocks). In times of need (e.g. excessive amount of failures), the number of redundant copies may be decreased in order to keep the system operational even at a decreased level. A specification is necessary which guides the system how to degrade the system without loosing too much functional quality. Thus, we propose *degradation rules* to complement the structural and deployment specifications of fault tolerance patterns.

For each fault tolerance pattern, we support the specification of a set of degradation rules. A degradation rule can decrease non-functional properties like reliability as well as decrease the functional quality of the software. Similar to structural and deployment specifications of fault tolerance templates, we use a visual specification language. These degradation rules specify the behavior which is executed while degrading the system's functional or non-functional properties. As those rules are important for the dependability of the systems, the rules must be formally defined using an appropriate formalism.

3.1. Story Patterns

Graph transformations [19] are a powerful formalism for the specification of structural changes to graph like structures and as such are an appropriate notion for the specification of degradation rules.

Graph transformation rules consist of structural specifications for a precondition and a postcondition. A graph transformation rule is applicable if each element of the precondition structure (excluding negative elements) can be mapped onto one element of the host graph. No two elements of precondition structure must be mapped onto the same element of the host graph (isomorphism check). Then the host graph is changed in such a way that each element of the postcondition can be mapped onto an element of the host graph while retaining the element mappings created during the precondition mapping and remove elements which are part of the pre- but not part of the postcondition structure.

As concrete formalism story patterns [28, 5] are employed. Story patterns combine pre- and postcondition structures into a single diagram based on UML [15] collaboration diagrams. The difference between pre- and postcondition structures is denoted by annotating elements by ≪create≫ and ≪destroy≫ stereotypes. An elemenent annotated by ≪create≫ will be created during application of the story pattern, whereas an element annotated by ≪destroy≫ will be destroyed.

Graph matching which is required in order to find a precondition in the host graph is a NP-complete problem [6]. As our degradation rules should be executed during runtime and failure recovery should be fairly fast, this is an unpleasant property. Fortunately, we can efficiently compute the relevant mappings of preconditions onto the host graph at deployment time due to knowledge about the occurrence of the fault tolerance patterns in the system.

In the following, we will present degradation rules for the TMR and the recovery block patterns. The TMR degradation rules degrade the reliability of the system (non-functional property) whereas the recovery block degradation rules degrade the functional property of the system.

3.2. Triple Modular Redundancy

Figure 4 shows a graceful degradation rule for the triple modular redundancy pattern. This rule specifies the degradation of a pattern application from a triple modular setup to a double setup, i.e. one redundant copy of Unit is removed from the deployment system. The removal of the redundant copy is specified using the ≪destroy≫ stereotypes. The connector between the voter and the redundant copy as well as the connector to the multiplier are also removed. The resulting double modular redundancy setup can tolerate crash failures, but cannot tolerate value failures.

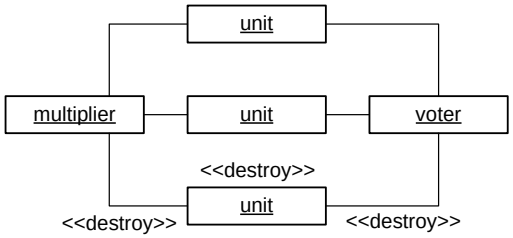


Abbildung 4. Triple → double redundancy

The second graceful degradation rule converts an already degraded double modular redundancy (former triple) setup into a non-redundant one. Figure 5 shows the story pattern which captures this behavior. This pattern removes the second redundant copy of the pattern application. As multiplier and voter are not necessary for a single computation unit, they are removed too. The final unit is directly connected to the user and provider components. As no redundancy is employed, the setup cannot tolerate a failure anymore. Note, that the rule requires that there is no other redundant copy available since then removing the multiplier and voter would not been possible. This is visualized by the crossed out unit component.

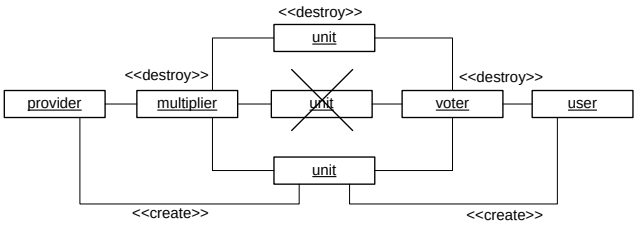


Abbildung 5. Double → no redundancy

3.3. Recovery Blocks

Graceful degradation for the recovery block pattern is performed by removal of alternatives and their acceptance test instance. As the number of alternatives can differ in applications of the fault tolerance pattern, we propose iterative degradation rules. Two rules complement each other. The first rule removes the first alternative and its accompanying acceptance test. The first one is removed, since its typical the most sophisticated one and therefore probably has the most resource requirements. Note, that the acceptance test, which should be removed, is connected to another via the directed next-edge. The negative acceptance test node in the lower part of the figure assures that there exists no previous acceptance test node, i.e. the acceptance test is the first one.

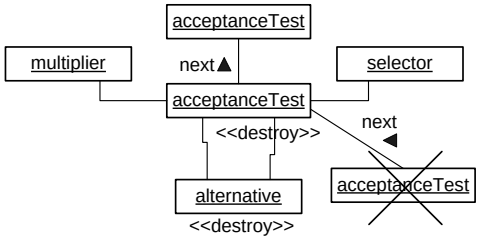


Abbildung 6. Remove first alternative

The first rule can only be executed as long as there is more than one alternative and acceptance test. If this is not the case, there is only one final alternative in the system. As in this case, the multiplier, selector and acceptance test components are unnecessary, the second degradation rule simply removes them from the model. Note, that both incoming and outgoing next-edges are negated, i.e. there are no other acceptance tests connected to this one and as such it is the final one.

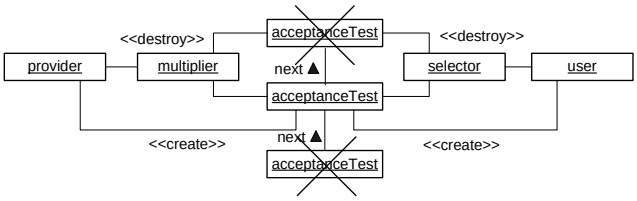


Abbildung 7. Garbage collection

4. Self-Repair with Degradation

Software components in distributed systems experience failures. In Section 2.3, we presented an approach to repair a partially failed system by redeployment of the failed software components. In high load systems, a redeployment of failed software components may unfortunately be not possible due to unsatisfiable resource constraints. In this situation, degradation rules as presented in Section 3 are used in order to free up resources by either decreasing non-functional

properties like fault tolerance and retain the required functionality or vice versa.

We have two options to process the degradation rules: (1) We may execute a set of rules[2] on the deployment constraint model and solve the constraint model using the algorithm presented in [25]. (2) We may map the application of all degradation rules into the objective function of the constraint model. Using an objective function, the constraint solver minimizes the degradation due to a value associated to the different configurations produced by applications of the degradation rules. The values denote the different qualities of the configurations, i.e. a triple modular redundancy setup has a higher value than a non-redundant setup.

The first option may result in non-optimal application of degradation rules as more rules are applied than necessary. But due to the incremental approach of [25], a fast recovery from the failure is provided. The second option offers the advantage that an optimal number of rules is selected for application but needs more time to find an minimal degradation due to the objective function.

We suggest to select each of these two options depending on the types of failures which are observed for the software components. If a timely reaction to at least on of these failures is required because one failed software component is an important non-redundant one, we suggest to use option one. After a successful redeployment of the degraded system, one can employ the second option to minimize the amount of degradation rules applications. If only redundant copies experience failures, the functionality of the system is not affected. Thus, it is reasonable to pursue option two directly. In the following, we sketch these two options.

4.1. Fast Degradation

The incremental algorithm [25] presented in Section 2.3 for the computation of self-repair actions starts with a small deployment model consisting of only a small number of components. This small model is repeatedly extended by other components until the deployment model is solvable.

The degradation rules are applied to the component structure of the deployment model during execution of the algorithm. In detail, the degradation rules are applied each time they can be applied to the model. Thus, whenever a reduced deployment model is not solvable, applicability of the degradation rules is tested. If the precondition of the rule can be matched[3] with the component structure, the rule is applied and the constraint solver is executed. If it does not succeed, the model is extended as in the original approach and the next iteration of the algorithm is performed.

Based on this extended algorithm, we preserve the fast responsiveness of the algorithm and additionally take into account the application of the degradation rules.

4.2. Minimal Degradation

In contrast to the above described approach for a fast recovery from a failure, we aim here at minimal degradation of the system. Thus, possible applications of degradation rules are added to the deployment constraint model using the objective function provided by the constraint solver.

The transformation of the deployment model to the input language of a constraint solver is shown in Section 2.3. There, boolean variables $x_{i,j} \in \{0,1\}$ are used denote whether a component $i \in C$ is deployed to host $n \in N$ or not.

It is not required that all components are deployed due to the possible application of the degradation rules. Thus, we get for $p \in P$ the set of pattern occurrences with related components C_p that for components $i \in C - \{C_p | p \in P\}$ still equation 1 must hold while otherwise the following inequality must hold:

$$\forall j \in N : \sum_{i \in \{C_p | p \in P\}} x_{i,j} \leq 1 \qquad (2)$$

In order to express the application of degradation rules, we introduce boolean variables $b_i \in \{0,1\}$, which denote whether a component i is deployed on any node ($b_i = 1$) or not ($b_i = 0$).

$$b_i = \begin{cases} 1 & : \sum_{i \in C} x_{i,j} = 1 \\ 0 & : \text{else} \end{cases}$$

The application of degradation rules for each pattern occurrence $p \in P$ can then be expressed in terms of boolean expressions over the b_i variables. This is formally represented using additional boolean variables $d_1^p, \ldots, d_k^p$ which denote whether different degraded configurations $1, \ldots, k$ hold. In this case, if $d_3^p = true$ holds, then the full triple modular redundancy setup is used. if $d_1^p = true$ holds, then only a single redundant copy is used. These configurations d_k^p are computed by the application of the degradation rules specified in Section 3. Finally, the constraint c^p which enforces that exactly one of the configurations is operational has to be added to the constraint system.

$$\begin{aligned} d_3^p &= b_{multiplier} \wedge b_{unit1} \wedge b_{unit2} \wedge b_{unit3} \wedge b_{voter} \\ d_2^p &= b_{multiplier} \wedge ((b_{unit1} \wedge b_{unit2}) \vee (b_{unit1} \wedge b_{unit3}) \\ &\quad \vee (b_{unit2} \wedge b_{unit3})) \wedge b_{voter} \wedge \neg d_3 \\ d_1^p &= (b_{unit1} \vee b_{unit2} \vee b_{unit3}) \wedge \neg d_2 \wedge \neg d_3 \\ c^p &: (d_1^p \vee d_2^p \vee d_3^p) \end{aligned}$$

[2] The actual number of graceful degradation rules is based on the amount of failed software components and can be specified by the administrator. A first rough estimation is to apply at least the same amount of rules as the number of failed components.

[3] This precondition check is fast, since we know which components are applicants of a fault tolerance pattern and as such we just have to check if all members of one pattern instance are contained in the deployment model.

The objective of the constraint solver is to maximize the deployed components as the execution of a degradation rules decreases the amount of deployed components. In terms of the deployment constraint model introduced in this section, the objective function is to maximize the value of the different fault tolerance pattern occurrences $p \in P$ using their set of degradation configurations $k \in D_p$ weighted value given by the constant factors α_k^p. Note that the components which are not related to any pattern are not subject of the optimization.

$$max \sum_{p \in P} \sum_{k \in D_p} \alpha_k^p d_k^p$$

5. Related Work

Many approaches (e.g. [17, 4, 14, 2]) address the problem of deployment and reconfiguration of fault tolerance enhanced systems. While [17, 14] do not specifically tackle the problem of minimal redeployment, [4] tries to compute optimal redeployment actions by a model extension approach similar to ours. In contrast to the others approaches, Arshad et al. present in [2] a planning based approach for failure recovery. Based on a domain model which specifies the components and its requirements on the system as well as reconfiguration actions, an AI planner is used to find a plan for failure recovery. The AI planner tries to find a sequence of actions which change the system state from the initial, failure state to a certain goal state (e.g. a number of redeployment steps). Degradation actions in combination with an appropriate goal may provide similar reconfiguration and degradation behavior as our approach. All above mentioned approaches do not use visual specifications.

In [23], Strunk and Knight present a formal approach for the specification of reconfiguration actions for real-time embedded systems. Those reconfiguration actions specifically allow the specification of real-time annotations as e.g. the action begins at the same time the system is no longer operating under a certain service configuration. Our approach uses a visual language for the specification in contrast to the textual notion of [23]. In addition, our degradation rules are part of fault tolerance patterns and thus can be easily reused.

Graceful degradation is used in a wide variety of approaches [10, 20, 27]. Gonzalez et al. present in [10] an approach to dynamically adapt the employed fault tolerance for each incoming computation request. Similarly to our approach, they can degrade non-functional properties like fault-tolerance. In contrast, to our approach, they use no visual language and adapt in case of variations to the rate of incoming computation requests. Shelton et al. present in [20] a framework for the analysis of graceful degradation rules. They divide a component-based software system into feature sets and compute what utility those different feature sets provide in case of failures. In contrast to our approach, they measure the system's degradation which is the implicit result of induced failures whereas our approach uses degradation rules to react to failures.

6. Conclusion and Future Work

Embedded software systems are an integral part of today's technical systems. Reliability and availability of those technical systems are important non-functional properties. As those properties are affected by failures, fault tolerance techniques are applied to provide protection from those failures. If the number of failures is too high, the system might degrade its functional or non-functional properties in order to maintain its operational status.

We presented a visual formalism for fault tolerance techniques in this paper. Fault Tolerance Patterns capture structure, deployment and degradation for existing fault tolerance techniques. We presented fault tolerance patterns for a triple modular redundancy setup and a distributed recovery block setup. Based on the visual specifications, we presented to self-repair the system in case of failures. Self-repair either restores the system by redeployment of failed software components or degrades the system's functional or non-functional properties by the execution of degradation rules.

We are currently integrating this approach into an execution framework and our case tool Fujaba[4] in order to further evaluate the benefits of the self-repair actions and the degradation rules. Proof of degradation rules correctness is an important prerequisite for the application of the rules in highly critical technical systems. Correctness includes detection of conflicts or unsafe states in the degradation rules. Thus, we are evaluating the formal verification approaches of [9] and [18]. In addition, we will address temporal requirements of degradation in future work. As a complement to degradation rules, improving rules will be considered which add components which were previously removed by a degradation rule in order to improve functional and non-functional properties during runtime.

Literatur

[1] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 447–457, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[2] N. Arshad, D. Heimbigner, and A. L. Wolf. A Planning Based Approach to Failure Recovery in Distributed Systems. In *Proceedings of the ACM SIGSOFT Internatio-*

[4] www.fujaba.de

nal Workshop on Self-Managed Systems (WOSS'04). ACM Press, Oct./Nov. 2004.

[3] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[4] A. Dearle, G. Kirby, and A. McCarthy. A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. Technical Report CS/04/1, University of St Andrews, 2004.

[5] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[7] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[8] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM Press.

[9] H. Giese and D. Schilling. Towards the Automatic Verification of Inductive Invariants for Invinite State UML Models. Technical Report tr-ri-04-252, University of Paderborn, Paderborn, Germany, December 2004.

[10] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 79, Washington, DC, USA, 1997. IEEE Computer Society.

[11] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, volume 16 of *Lecture Notes in Computer Science*, pages 171–187, London, UK, 1974. Springer Verlag.

[12] K. H. Kim and H. O. Welch. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Trans. Comput.*, 38(5):626–636, 1989.

[13] N. G. Leveson. *Safeware : system safety and computers*. Addison-Wesley, 1995.

[14] M. Mikic-Rakic, S. Malek, N. Beckman, and N. Medvidovic. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. In W. Emmerich and A. L. Wolf, editors, *Component Deployment, Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004, Proceedings*, volume 3083 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2004.

[15] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.

[16] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[17] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21164. IEEE Computer Society, 2004.

[18] A. Rensink, Á. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag Heidelberg, 2004.

[19] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations*. World Scientific Pub Co, February 1997.

[20] C. Shelton, P. Koopman, and W. Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Proc. of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003.

[21] A. K. Somani and N. H. Vaidya. Understanding Fault Tolerance and Reliability. *Computer*, 30(4):45–50, 1997.

[22] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[23] E. A. Strunk and J. C. Knight. Assured Reconfiguration of Embedded Real-Time Software. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 367–376, Washington, DC, USA, 2004. IEEE Computer Society.

[24] M. Tichy, B. Becker, and H. Giese. Component Templates for Dependable Real-Time Systems. In A. Schürr and A. Zündorf, editors, *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, volume tr-ri-04-253 of *Technical Report*, pages 27–30. University of Paderborn, September 2004.

[25] M. Tichy, H. Giese, D. Schilling, and W. Pauls. Computing Optimal Self-Repair Actions: Damage Minimization versus Repair Time. In R. de Lemos and A. Romanovsky, editors, *Proc. of the ICSE 2005 Workshop on Architecting Dependable Systems, St. Louis, Missouri, USA*. ACM Press, May 2005.

[26] M. Tichy, D. Schilling, and H. Giese. Design of Self-Managing Dependable Systems with UML and Fault Tolerance Patterns. In *Proc. of the Workshop on Self-Managed Systems (WOSS) 2004, FSE 2004 Workshop, Newport Beach, USA*, October 2004.

[27] D. Weber. Formal Specification of Fault Tolerance and its Relation to Computer Security. *ACM SIGSOFT Engineering Notes*, 14(3), 1989. (International Workshop on Software Specification and Design 1989).

[28] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.