Pattern-Based Synthesis of Fault-Tolerant Embedded Systems*

Matthias Tichy
Software Engineering Group
University of Paderborn, Germany
mtt@uni-paderborn.de

ABSTRACT

The general trend towards complex technical systems with embedded software results in an increasing demand for dependable embedded systems. In this position paper, we give an overview about a pattern-based approach for the development of fault-tolerant, component-based, embedded systems. Four aspects of this approach are addressed in this paper: (1) the approach is based on an appropriate modeling language, (2) an automatic deployment of software artifacts to hardware elements, (3) a compositional hazard analysis of the deployed system, which results in the identification of weak points for dependability, and (4) the subsequent application of fault-tolerance patterns to replace those weak points by more dependable architectures.
Advisor: Prof. Dr. Wilhelm Schäfer

1. INTRODUCTION

The general trend towards complex technical systems with embedded software results in an increasing demand for dependable embedded systems. Faults are the sources for low dependability. Random faults like hardware faults can be distinguished from systematic faults like software or design bugs. Faults can be removed by e.g. extensive testing or formal verification and tolerated by fault tolerance techniques [9]. We restrict ourselves to the problem of fault tolerance and refer to other approaches for fault elimination (e.g. [13, 1]).

The application of fault tolerance techniques to embedded systems is typically done manually by changing the system architecture. Architectural changes are for example addition of redundant parts and voters as well as changes to the behavior of software elements in order to cope with the additional redundancy.

*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.

SIGSOFT'06/FSE-14, November 5-11, 2006, Portland, Oregon, USA.

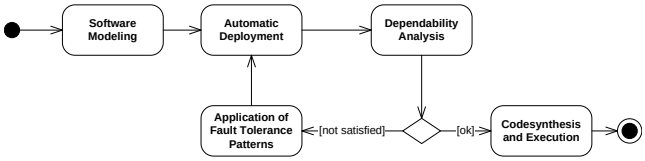

Figure 1: Approach overview

Unfortunately, the manual application of fault tolerance techniques can result in additional systematic faults due to the increased complexity. Consequently, we propose to automate the application of fault tolerance techniques into the embedded system. A formal model for the structure as well as the behavior of fault tolerance techniques is required for the automatic application.

We decided to add fault tolerance techniques into the software parts of the architecture rather than only in hardware to be more flexible w.r.t. which parts of the embedded software require more fault tolerance than others. As we consider hardware failures, we also need to consider the deployment of software to the hardware. As the automatic application of fault tolerance techniques influences the software's deployment, we also provide an automatic deployment.

Finally, the fault tolerance techniques should only be applied whenever and wherever they are needed. Thus, our process contains an analysis step of the dependability attributes[1] based on the combined software/hardware structure and known or assumed hardware failure rates. If the analysis is successful, the source code of the specified software model is automatically generated and subsequently executed on the target hardware. The above presented activities in our approach are shown in Figure 1.

In the next section, we present the language for the specification of the software as the foundation of our approach. We sketch the different dependability analyses of our approach in Section 3. Thereafter, we present the structural and behavioral specification of fault tolerance techniques by fault tolerance patterns. After reviewing relevant related work in Section 5, we conclude and present current and future work.

2. MODELING LANGUAGE

Embedded real-time systems are typically modeled in

[1]Availability, reliability, safety, and security are the attributes of dependability [9]. We do not consider security in our approach.

terms of a component based software and hardware system. The UML as an advanced technology provides in principle the essential concepts which are required to model the structure and behavior of software components. However, the current UML version does not directly apply to embedded real-time and fault tolerant systems. Therefore, an appropriate modeling language has to be used.

In [2], a graphical modeling language Mechatronic UML based on UML/RT [17] and ROOM [16] is proposed, which provides modeling constructs for the real-time properties of embedded systems. The Mechatronic UML contains means for the specification of structure using component diagrams and the specification of real-time behavior in form of Real-Time Statecharts. The Real-Time Statechart formalism is a variant of UML state machines which supports more powerful concepts for the specification of real-time behavior. They are semantically based on the timed automata formalism. Thus, Mechatronic UML provides the necessary means to specify structure and behavior of real-time software. Code generation is also available for Mechatronic UML models. Consequently, we base our approach on the Mechatronic UML.

3. ANALYSIS

Fault tolerance techniques should only be applied when they are required. Analysis techniques like Fault Tree Analysis are extensively used to check whether dependability requirements like reliability or safety have been properly addressed in the system.

As we explicitly consider component-based embedded systems, appropriate analysis techniques must be employed. We developed a component-based approach for the hazard analysis of the combined software/hardware models [5]. This approach supports the concurrent analysis of multiple system model variants. Each variant represents a different application of fault tolerance techniques. The analysis results in the optimal variant concerning system hazards.

Additionally, our approach includes a mapping of the combined software/hardware models to a stochastic petri net [3]. The stochastic petri net is then used to analyze reliability and availability. In stochastic petri nets, repair operations can be modelled in contrast to the above presented hazard analysis approach.

4. FAULT TOLERANCE PATTERNS

When the dependability analysis shows that dependability requirements are not satisfied, fault tolerance techniques are applied to the system. In the following, we present the different modeling aspects of *Fault Tolerance Patterns* which are required for the automatic application of fault tolerance techniques.

4.1 Structure

Fault tolerance techniques are often based on the principle of redundancy as redundant parts keep the system operational in case of faults. A first formalization of fault tolerance patterns was based on a declarative description using the component diagrams of the Mechatronic UML [21]. Figure 2 shows the specifaction of the triple modular redundancy pattern. The pattern consists of three component replicas and additional voter and multiplier components which connect the replicas to other software components.

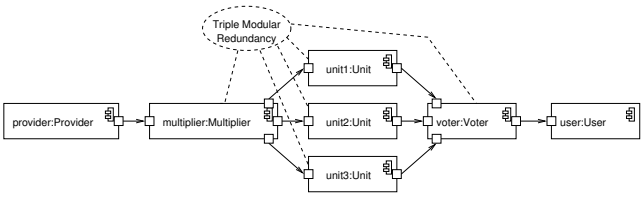

Figure 2: Triple Modular Redundancy Pattern

This declarative specification proved not flexible enough in terms of application to different types of component structure with varying number of ports. Consequently, we are in the process of formalizing the fault tolerance patterns as architectural refactorings using the graph transformation formalism [15, 4]. Early results have been presented for the specification of graceful degradation transformations [19] for fault-tolerant software systems.

4.2 Deployment Constraints

The redundant software components which are introduced by the application of a fault tolerance pattern must be deployed to some hardware element. The deployment of redundant software components should respect certain constraints. For example, redundant software components should not be deployed to the same execution hardware as a failure of this execution hardware would lead to a simultaneous failure of all redundant software components. The deployment of redundant software components to heterogeneous (e.g. processor type) hardware is another example of a deployment constraint.

We developed a graphical specification of those deployment constraints based on UML deployment diagrams and presented in [20] a formal mapping into a set of equations on integer and boolean variables. We also take resources like memory and bandwidth into account. The equations are then solved by a standard constraint solver. The assignment of the boolean variables then represents the deployment of software components to the hardware as well as the deployment of connectors between software components to the underlying communication network.

4.3 Behavior

For a complete specification, behavior of the components which are added to the component structure by the application of fault tolerance patterns is required. This behavior is the main source of the aforementioned increased complexity. Different techniques are used to tolerate different faults. Identical redundant parts (e.g. in triple modular redundancy) are used to tolerate random faults whereas functional diversity is used to tolerate systematic faults same function (e.g. n-version programming).

Fault tolerance techniques which employ identical redundant parts are especially suitable for automation since redundant copies as well as *glue components* like voters which interpret the results of the redundant parts can be automatically synthesized. Fault tolerance with functional diversity also employs synthesizable glue components like voters. However, the functional diverse parts cannot be automatically synthesized.

A synthesis algorithm for voter components needs to know the interaction protocol between the components. The interaction is described by the sent and received messages, possible message orders and timing constraints between sending and receiving of the messages. The Mechatronic UML employs Real-Time Statecharts for the specification of the interaction protocols and, consequently, the Mechatronic UML models provide all necessary information for the automatic synthesis of glue components.

4.3.1 Replica Determinism

If redundancy is used to achieve fault tolerant behavior, the redundant parts (replicas) must behave consistent with each other. This problem is known as *replica determinism* (cf. [14]). In non real-time systems, replica determinism of software components is for the most part determined by the reception order of messages. But even in non real-time systems, time outs are used for failure detection and, thus, time influences the replica determinism problem. This problem is even more important in real-time software as the behavior is even more dependent on clock values.

The message reception order on different replicas may lead to different behavior of the replicas. We propose to use a real-time group communication framework in order to ensure that all replicas receive the messages in identical order within a bounded delay.

Distributed systems suffer from the problem of different clocks. As replicas should be executed on different nodes, each replica uses a different clock for time related operations. In Real-Time Statecharts, timing constraints are only specified relative to a distinct point in time (e.g. a transition fires 500 time units after a message has been received). Thus, the absolute difference between clock values is no problem. We abstract from the problem of clock drift as the drift is negligible due to the typically small time difference between clock read and subsequent references to the clock value. Nevertheless, if lower and upper bounds of the clock drift are known, those bounds can be considered in the synthesis of the glue components.

Different types of scheduling algorithms are used in the context of real-time systems to schedule a number of software components on a hardware in a way that all real-time constraints (deadlines) are met. The replicas are executed on their respective hardware with different sets of other components. Thus, there will be a different scheduling for each replica. This does not pose a problem for replica determinism since we employ a *deadline first* scheduler which ensures all replicas meet their identical deadlines and thus behave consistently.

Different messages transmission delays of messages to different replicas are another problem which needs to be addressed. Considering that three replicas wait for a message, the message may be received before expiration of a time out by one replica. The other two replicas may receive the message after expiration of their time out and show different behavior from there on.

We are currently working on using model checking techniques which are already available for Real-Time Statecharts [1] in order to determine whether replicas may behave inconsistently due to different message transmission delays. The model checker is used to verify whether there exist a path in the state space where the components are in inconsistent states.

5. RELATED WORK

In the following, we review several related approaches. We present approaches concerning the different activities in the presented process (cf. Figure 1).

5.1 Analysis

Component-based hazard analysis is a hot topic in safety-critical systems research [10, 7, 6, 12]. The basic idea is to ease the hazard analysis by reusing already available information about failure behavior of the individual components rather than always start from scratch when performing a hazard analysis. The current approaches for component-based hazard analysis have in common that they describe the failure propagation of individual components (cf. failure propagation and transfer nets [3]). Outgoing failures are the result of the combination of internal errors and incoming failures from other components.

Different applications of fault tolerance patterns can be seen as variants. Our approach supports the analysis of all variants concurrently and computes the best variant w.r.t. the dependability attributes. All aforementioned approaches do not directly support the hazard analysis of product variants. Instead, variants must be manually created, analyzed, and compared individually.

5.2 Fault Tolerance

Grunske presents an approach [6] for the automatic improvement of safety properties for component-based software architectures. Transformational patterns are used as formalization of standard fault tolerance and safety techniques. The patterns are formalized as hyper-graph transformations. The structural part of an software architecture is transformed by the patterns. The approach does not consider the real-time behavior of the components neither the problem of replica determinism.

Kulkarni and Ebnenasir present an approach [8] to automatically transform fault-intolerant programs into fault-tolerant ones. In contrast to our approach, they do not consider component-based systems but base their approach on a formal model of processes, variables and assignments. This formal model, and thus their approach, is not directly applicable to component-based software architectures. Additionally, the real-time domain is not considered in this approach.

Both approaches do not consider deployment constraints for redundant parts. This complicates the automatic improvement of dependability attributes as the deployment must be known for the dependability analysis.

5.3 Deployment

In [18], Sommer and Guidec present an approach for the specification of resource constraints for software deployment. Resource restrictions are used for the correct deployment of the software on resource-constrained systems. Our approach allows the specification of fault tolerant deployment constraints. The DeSi environment [11] supports the assessment of deployment quality in terms of availability. It supports the automatic, constraint based deployment of components to hosts in a distributed system. This approach also supports the specification of constraints for components to be deployed to same/different hosts. We, in contrast to this approach, support the graphical specification of constraints using the standard modeling language UML. In contrast to

our approach, both approaches provide no means for specifying reusable distribution constraints using patterns.

6. CONCLUSIONS AND FUTURE WORK

We presented an approach for a pattern-based synthesis of fault tolerant real-time software. Currently, the parts concerning the modeling language, dependability analysis, and automatic deployment are conceptually finished. We currently work on improved concepts for structural transformations as well as concepts for the synthesis of generic components for systematic fault tolerance like voter components. In this area, we currently evaluate how we can employ model checking techniques to ensure replica determinism. Additionally, preliminary tool support for the concepts is available in form of plugins for the Fujaba Real-Time Tool Suite [1]. We are currently improving the tool support.

In this paper, we presented an approach for the synthesis of fault-tolerance at *design time*. We developed an approach [20] for self-healing a system in the case of failures *at runtime* which is well integrated into the presented approach.

The concepts as well as the tool support are evaluated in the railcab project[2]. This project aims at using a passive track system with intelligent shuttles that operate autonomously and make independent and decentralized operational decisions. The shuttles are safety-critical systems, which operate in real-time systems. Consequently, the railcab project is ideal to evaluate the presented approach. The real-time software of a shuttle will be (in part) developed using the presented approach in order to ensure that the approach can be used in non-trivial cases. Additionally, we plan to measure the benefit of the approach by a student case study. One group of students will use the approach whereas the other will not use it. We will compare the dependability of the students' software designs concerning availability, safety, etc. as well as the amount of time needed for the development of the software design.

7. REFERENCES

[1] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[2] S. Burmester, H. Giese, and M. Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In U. Assmann, A. Rensink, and M. Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science*, pages 47–61. Springer Verlag, August 2005.

[3] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius Framework and Its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[4] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language.

[5] H. Giese and M. Tichy. Component-Based Hazard Analysis: Optimal Designs, Product Lines, and Online-Reconfiguration. In *Proc. of the 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP), Gdansk, Poland*, Lecture Notes in Computer Science. Springer Verlag, September 2006. (to appear).

[6] L. Grunske. Transformational Patterns for the Improvement of Safety. In *Proc. of the The Second Nordic Conference on Pattern Languages of Programs (VikingPLoP 03)*. Microsoft Buisness Press, 2003.

[7] B. Kaiser, P. Liggesmeyer, and O. Maeckel. A New Component Concept for Fault Trees. In *Proceedings of the 8th National Workshop on Safety Critical Systems and Software (SCS 2003), Canberra, Australia. 9-10th October 2003*, volume 33 of *Research and Practice in Information Technology*, 2003.

[8] S. S. Kulkarni and A. Ebnenasir. Enhancing The Fault-Tolerance of Nonmasking Programs. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, pages 441–449. IEEE Computer Society, 2003.

[9] J. C. Laprie, editor. *Dependability : basic concepts and terminology in English, French, German, Italian and Japanese [IFIP WG 10.4, Dependable Computing and Fault Tolerance]*, volume 5 of *Dependable computing and fault tolerant systems*. Springer Verlag, Wien, 1992.

[10] J. A. McDermid. Trends in Systems Safety: A European View?

[11] 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.

[12] F. Ortmeier, A. Thums, G. Schellhorn, and W. Reif. Combining Formal Methods and Safety Analysis - The ForMoSa Approach.

[13] D. A. Peled. *Software reliability methods*. Texts in computer science. Springer Verlag, New York, 2001.

[14] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.

[15] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations*. World Scientific Pub Co, February 1997. Volume 1.

[16] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[17] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Techreport, ObjectTime Limited, 1998.

[18] N. L. Sommer and F. Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *Proc. of the Component Deployment : IFIP/ACM Working Conference, CD 2002, Berlin, Germany*, volume 2370 of *Lecture Notes in Computer Science*, pages 15–30, June 2002.

[19] M. Tichy and H. Giese. Extending Fault Tolerance Patterns by Visual Degradation Rules. In *Proc. of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, pages 67–74, September 2005.

[20] 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*, pages 1–6. ACM Press, May 2005.

[21] 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.

[2] `www.railcab.de/en/`