Design of Self-Managing Dependable Systems with UML and Fault Tolerance Patterns*

Matthias Tichy, Daniela Schilling† and Holger Giese
Software Engineering Group, University of Paderborn, Warburger Str. 100, Paderborn, Germany
[mtt|das|hg]@uni-paderborn.de

ABSTRACT

The development of dependable software systems is a costly undertaking. Fault tolerance techniques as well as self-repair capabilities usually result in additional system complexity which can even spoil the intended improvement with respect to dependability. We therefore present a pattern-based approach for the design of service-based systems which enables self-managing capabilities by reusing proven fault tolerance techniques in form of *Fault Tolerance Patterns*. The pattern specification consists of a service-based *architectural design* and *deployment restrictions* in form of UML deployment diagrams for the different architectural services. The architectural design is reused when designing the system architecture. The deployment restrictions are employed to determine valid deployment scenarios for an application. During run-time the same restrictions are at first used to automatically map additional services on suitable nodes. If node crashes are detected, we secondly employ the restrictions to guide the self-repair of the system in such a way that only suitable repair decisions are made.

1. INTRODUCTION

Software plays a dominant role in today's high-integrity systems [2] like critical infrastructure (telephone system) or safety-critical systems (car). Dependability and its attributes availability, reliability, safety, and security [5] are key goals in developing these high-integrity systems. Fault tolerance techniques are widely employed to develop dependable systems as they allow, if correctly applied, to improve availability and reliability (cf. [1, 9]).

Today's high-integrity systems are distributed systems and as such are subject to additional problems like partial failures, network failures, etc. But distributed systems also provide means for enhancing the dependability as they allow for redundancy and healing by redeployment of software to working nodes. As a consequence, the software is not bound to a specific computation resource but may be executed on different nodes in the system over time. Service-based architectures [14] (as extended component-based systems) cope with the complexity and dynamics of these systems. They offer infrastructure services which support the lookup and use of services by clients, which do not know the exact location of services in the network and therefore are not affected by changing the location during redeployment. Therefore, service-oriented architectures are an ideal architecture for such distributed, high-integrity systems and as such targeted by our approach.

In [11, 12, 13], we developed a UML based approach to model service-based systems which also supports the later deployment of the services. Deployment rules and a run-time environment permit to realize systems with partially configurable reliability and availability attributes for redundant services. Deployment decisions are made online looking for appropriate nodes which satisfy the requirements stated in the deployment rules. Additionally, the automatic self-management detects crashed services and then employs the specified deployment rules to restart them on a node with appropriate characteristics. In those previous works, deployment restrictions concerning redundancy properties have not been addressed. In contrast to these scenarios, the deployment of systems, which employ fault-tolerant techniques, require a deployment mapping, which maps the different services taking into account the assumptions of a fault tolerance technique such as, for example, placement on distinct nodes with different hardware.

The presented approach addresses this problem by offering *reusable* fault tolerance patterns which include besides the architectural design (as presented in [10]) also deployment restrictions for all the architectural services of the patterns which make such assumptions explicit. These fault tolerance patterns are specified in a graphical way using UML component and deployment diagrams to improve the readability.

The architectural design is reused each time the pattern is employed during the architectural design. Any valid deployment scenario for the architecture has to respect the deployment restrictions of all employed patterns. The deployment restrictions are further employed for the automatic self-management at run-time. If a new service has to be added, the deployment restrictions are used to automatically find a suitable mapping of the new service to nodes. The restrictions also guide the self-repair of the system when node crashes are detected. Only suitable repair decisions are made which respect the constraints implied by the restrictions.

The outlined approach results in an automatic self-management capability for pattern-based designs. An error-prone textual and manual specification of deployment restrictions concerning redundancy for each service is rendered unnecessary, as all required information results from the embedding of the patterns into the UML model. The deployment restrictions are subsequently satisfied dur-

†Supported by the International Graduate School of Dynamic Intelligent Systems, University of Paderborn.
*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.

WOSS '04 Oct 31-Nov 1, 2004 Newport Beach, CA, USA

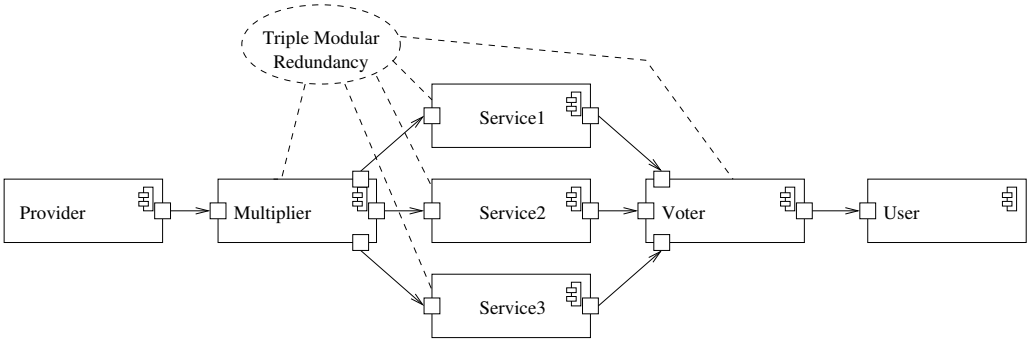


Figure 1: Structure of the triple modular redundancy template

ing runtime by an execution framework. Thus, the automatic self-management will satisfy all deployment constraints stemming from the roles the service plays in the deployment patterns.

We first describe in Section 2 the concepts for fault tolerance patterns. Then, we outline how the deployment restrictions can be used to compute suitable deployment mappings for a given static architecture in Section 3. In Section 4, these concepts are further extended to also cover the online addition of new services and the self-repair in case of node crashes. Finally, related work is discussed in Section 5 and we close the paper with a conclusion and outlook on future work.

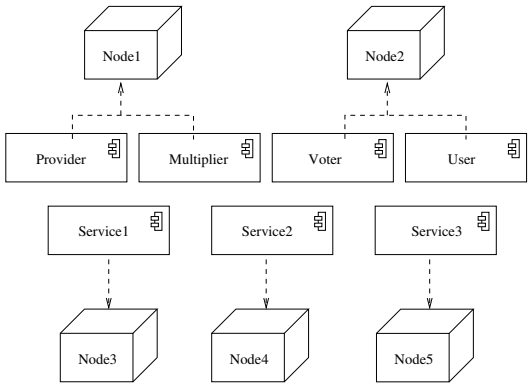
2. FAULT TOLERANCE PATTERNS

A fault tolerance pattern represents the specification of a certain fault tolerance technique. The specification consists of three parts. The first one represents the structure of the fault tolerance technique in form of a *fault tolerance template* [10] consisting of service roles and their connections. Based on this structural specification, in the second part, *deployment restrictions* concerning the different elements of the fault tolerance template are specified. The third part is the *behavior* specification of the fault tolerance technique. In this paper, we focus only on structure and deployment restrictions.

Figure 1 shows the *structure* of the triple modular redundancy (TMR) [9] fault tolerance pattern. A triple modular redundancy system uses three services Service1...3, which actually do the work. The Voter compares the different results and chooses the result which at least two of the services returned. The Multiplier triples the input values and propagates them to the three services. Thus, a triple modular redundancy system can tolerate one malfunctioning service. The Provider and User services are not part of the fault tolerance pattern but are added for the specification of additional deployment constraints.

Unfortunately common-mode failures spoil the fault tolerance enhancement of the TMR. For example, if two of these three services 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 services Service1...3 must be deployed to distinct nodes. The Multiplier and Voter as well as the Provider and User services are single points of failure in a simple application of TMR. Our observation is, that if a User service fails, the Voter service 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 services to the same node, i.e. both services do not crash fail independently of each other.

These observations must be appropriately specified in order that the actual deployment of the services satisfies these restrictions. A deployment diagram is used for the specification of these *deployment restrictions* for the fault tolerance pattern. A graphical specification as a diagram typically provides better readability than a textual representation.



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

Figure 2: Deployment pattern of the triple modular redundancy pattern

Figure 2 shows the deployment restrictions of the fault tolerance pattern of Figure 1. The following deployment restrictions are specified: (1) that each service is deployed to exactly one node (2) that each service Service1...3 is deployed on a different node, (3) that the services User and Voter as well as the services Provider and Multiplier both have to be deployed on the same node, (5) that Node3, Node4 and Node5 must have different CPU types.

The first restriction is visualized by the fact that each service is the source of exactly one deployment arrow. This ensures on the one hand that each service is mapped to a node and on the other hand that no service is mapped to more than one node. The second restriction is visualized by the deployment arrows of the services pointing to different node symbols. This constraint guarantees that Service1...3 are executed on different nodes and, thus, are not affected by common-mode failures. The third restriction is visualized by the deployment arrows of the services pointing to the same node symbol. This specifies that the Voter and the User services as well as the Multiplier and the Provider services do not crash independently. The fourth constraint is annotated in the graphic in a textual way. It assures that the software services do not suffer from failures that are specific to a certain CPU type.

After a short introduction to the system which is used to exemplify the approach, we show, in the next section, how the TMR fault tolerance pattern is applied to this system and how the deployment restrictions are used for an initial deployment of the services. In

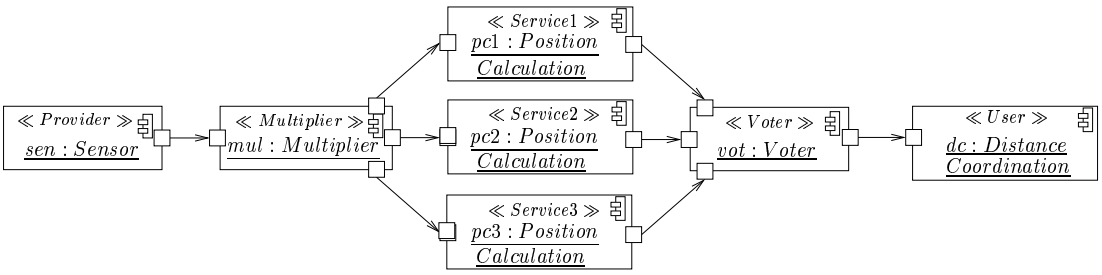


Figure 3: Service structure for shuttle position calculation

Section 4 the deployment reconfiguration based on the deployment restrictions in the case of node failures is presented.

3. INITIAL DEPLOYMENT

The approach at hand is developed within the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering. As main application example, autonomously driving shuttles[1] are developed. These shuttles drive on tracks like trains but they are powered by magnetic waves produced by stators which are assembled between the tracks. One particular problem is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and require a small distance between the different shuttles such that a high reduction of energy consumption is achieved. For the distance coordination of the shuttles it is indispensable to know their exact position. The position is calculated based on the stator waves that power the shuttles. As the position calculation is that important, we decided to implement it by using the TMR pattern. Thus, there are three services for the position calculation and four additional services. One for the sensor that measure the stator waves, one that multiplies the sensor data and delivers it to the calculation services, one voting service and one service that is responsible for the distance coordination. Figure 3 shows the structure of the implemented system after the application of the TMR fault tolerance template. Note, that the services are annotated by their respective role played in the fault tolerance template using a stereotype.

In order to find a suitable deployment, we propose to transform the graphically specified deployment restrictions to the constraint language of a standard constraint solver like ILOG's solver software. Thus, a standard solver is used to find a deployment that satisfies the deployment restrictions. For the constraint problem, we use the boolean variable $m_{i,j}$ to denote whether service i will be deployed to node j ($m_{i,j} = 1$) or not ($m_{i,j} = 0$).

As described in the previous section there are four restrictions that have to be met by a correct deployment. These restrictions are automatically transformed into constraints in the following way:

(a) each service has to be deployed to exactly one node. This corresponds to restriction (1) within the previous section and is formalized as:

$$\forall i : \sum_j m_{i,j} = 1 \tag{1}$$

(b) services with deployment arrows leading to different nodes must be executed on different nodes. In the above example this corresponds to restriction (2) and means no two position calculations may be executed on the same node. This can be formalized e.g. for the position calculation services pc1 and pc2 as:

$$\forall j : m_{pc1,j} + m_{pc2,j} \leq 1 \tag{2}$$

(c) services with deployment arrows leading to the same node must be executed on the same node. Within the example this means the Voter and DistanceCoordination services respectively the Sensor and the Multiplier services are deployed to the same host. This constraint is formalized e.g. for the services sen and mul as:

$$\begin{aligned} \forall j \quad : \quad & m_{mul,j} + m_{sen,j} = 0 \vee \\ & m_{mul,j} + m_{sen,j} = 2 \end{aligned} \tag{3}$$

(d) constraints that are annotated to the graphical constraint representation can be used by replacing the node names by the matching function. Here, the three nodes serving as hosts for the position calculation must be of three different CPU types.

$$\begin{aligned} & m_{pc1,j_1} \wedge m_{pc2,j_2} \wedge m_{pc3,j_3} \Rightarrow \\ & j_1.\text{CPU} \neq j_2.\text{CPU} \wedge j_1.\text{CPU} \neq j_3.\text{CPU} \wedge j_2.\text{CPU} \neq j_3.\text{CPU} \end{aligned} \tag{4}$$

As an example, we assume that there are six nodes that serve as hosts for the services. An initial mapping obtained from the constraint solver, which is correct w.r.t. the above given restrictions, is shown in Table 1 and graphically shown in Figure 4. ILOG's solver software returns an initial mapping for this rather small example almost immediately (0.004s) on an AMD Athlon 600 standard PC.

	sen	mul	pc1	pc2	pc3	vot	dc
avalon	1	1	0	0	0	0	0
gareth	0	0	0	0	0	0	0
taliesin	0	0	0	0	0	1	1
gorlois	0	0	1	0	0	0	0
uther	0	0	0	1	0	0	0
arthur	0	0	0	0	1	0	0

Table 1: Values of the $m_{i,j}$ variables obtained from the constrained solver

4. DEPLOYMENT RECONFIGURATION

In self-managed systems, on the one hand new services may be added during runtime. An additional service must be deployed to the system according to the deployment restrictions.[2] On the other hand, during runtime nodes and the executed services may fail. The application of a fault tolerance pattern provides the redundancy to

[1] http://nbp-www.upb.de/en

[2] Addition of new hardware instead does not result in redeployment with respect to fault tolerance.

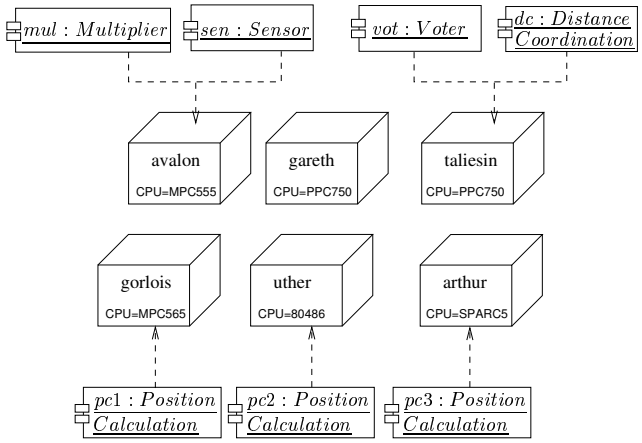


Figure 4: Example of a correct initial mapping

tolerate failures, but in order to tolerate additional failure restarting of the failed services on working nodes is necessary. This repair is subject to the deployment restrictions.

In order to reconfigure the deployment, we propose to use the mentioned deployment constraints. As this reconfiguration is performed during runtime, using the same constraint problem as in the initial deployment is naive w.r.t. the performance of the constraint solving. A better approach is to shrink the constraint problem by reusing the current values of the deployment variables $m_{i,j}$ for those services not affected by the failure. Only the deployment variables of the new or restarted services are still variable. If this very restricted constraint problem is not solvable, we need to relax some of the fixed variable values one after another in order to come up with a correct deployment solution. If a solution is found, which includes changed deployment variables, then service migration is necessary in order to fulfill the found deployment.

Additional Services

For example, during runtime a convoy service, which is responsible for the convoy management (i.e. it determines convoy speed and is responsible for communicating this convoy speed to all shuttles), may be added. In order to find a suitable deployment, the values of the deployment variables for the old services are fixed, e.g. the pc1 service should remain on gorlois:

$$m_{pc1,j} = \begin{cases} 1 & j = gorlois \\ 0 & \text{else} \end{cases} \quad (5)$$

For the new service convoy, new deployment variables $m_{con,j}$ are added to the constraint problem. After a run of the constraint solver, the variables hold the node, on which the new service will be deployed. In our example, the new service may be run on node gareth.

Repair

Consider the case that in our example gorlois fails and, thus, the service pc1 fails, too. The failed service must be restarted on a new node in order that the system keeps the required redundancy to tolerate an additional failure of the pc services. Selecting a node for this restarted service means in terms of the formalization, that the variables $m_{pc1,j}$ are not fixed, whereas the other variables are fixed to the values returned by the last invocation of the constraint solver. To accommodate this, we add constraints concerning the fixed values of the variables.

Since gorlois is not working, we need to fix the mapping variables concerning this node to 0. This leads to the following additional constraints:

$$\forall i : m_{i,gorlois} = 0 \quad (6)$$

	sen	mul	**pc1**	pc2	pc3	vot	dc
avalon	1	1	**?**	0	0	0	0
gareth	0	0	**?**	0	0	0	0
taliesin	0	0	**?**	0	0	1	1
gorlois	**0**	**0**	**0**	**0**	**0**	**0**	**0**
uther	0	0	**?**	1	0	0	0
arthur	0	0	**?**	0	1	0	0

Table 2: Values of the $m_{i,j}$ variables before repair

The values of the $m_{i,j}$ variables are shown in Table 2. The constraint solver will then return a new deployment mapping for the pc1 service. Concerning the initial mapping shown in Figure 4, the service pc1 can be restarted on gareth.

5. RELATED WORK

In [8] Sommer and Guidec present an approach for the specification of resource constraints for software deployment. The resource restrictions are used for the correct deployment of the software on resource-constrained systems. The to-be-deployed software specifies its required resources using program language constructs and the deployment software (the *resource broker*) determines whether the requirements can be met on the deployment node. The resource broker checks during the execution of the software, whether the software adheres to this requirements. This approach is similar to ours with respect to the specification of deployment restrictions. Contrary, we provide means for the deployment restriction specification on a more abstract level using UML deployment diagrams. Additionally, we specifically tackle the deployment for fault tolerant systems, which is not in the focus of [8].

Nentwich et al. present in [7] a consistency framework for UML models. The consistency checks are expressed in terms of first order logic over sets which are built by XPath expressions on XMI models. This approach could be used for checking the deployment constraints, too. The deployment constraint checking rule must be written in terms of XMI and XPath individually for each deployment constraint. We instead employ a graphical notation for the specific case of deployment constraints. This eases the process of specifying a deployment constraint, and improves the readability of the deployment constraints.

The Distribution Constraint Language DCL [3] provides flexible means for the specification of distribution constraints for components in form of a visual language. Distribution constraints are property based allocation expressions, e.g. never allocate a component with the property Personal Data = Yes to a hardware resource with the property WebData. Using DCL it is possible to generate deployment specifications for components based on the components properties, the network topology, and the distribution constraints. Though in the paper the generated deployment specification is erroneous with respect to network bandwidth and memory space. Fault tolerance is addressed by the addition of replication constraints, which guide the replication of components to several hardware resources. Using a visual language for specifying the distribution constraints is similar to our approach, whereas the actual graphical syntax differs. In its presented state, the generated deployment specification is only a helpful device for the developer

during the design phase, because the deployment at runtime is not supported and useless as long as the generated deployment specifications are erroneous. Generic templates for specifying reusable distribution constraints as supported by our approach are not possible.

Dearle et al.[4] also propose a declarative constraint-based deployment language. Their approach is based on an Autonomic Deployment and Management Engine (ADME). This engine is responsible for deploying software components to computational nodes in a way which satisfies the constraints specified for the component. In case of changes (crashes etc.), this engine redeploys the components according to the deployment constraints. During this redeployment, the engine tries to keep the number of redeployments low. The textual constraint language allows constraints for the deployment of components and the definition of connections between components. According to the authors, writing constraints is difficult. By our usage of a graphical language, which is an extension of UML deployment diagrams, we believe it is considerable easier to specify deployment constraints and understand them than using a textual language. Our idea of specifying deployment constraints for fault tolerance patterns makes it even easier to write deployment constraints respective to just reuse them. Reusing deployment constraints for common architectural patterns is already envisaged by Dearle et al.

The DeSi environment [6] 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. Several algorithms are described which try to compute a good deployment with respect to availability of the whole system. This approach also supports the specification of constraints for components to be deployed to same/different hosts. The resulting deployment is graphically shown in a proprietary diagram, while the constraints are only specified using drop-down-lists in a userinterface. We, in contrast to this approach, support the graphical specification of constraints using the standard modeling language UML. In addition, constraints are specified for a template and, thus, can be reused in all applications of this template.

6. CONCLUSIONS AND FUTURE WORK

In today's world, dependability of software is extremely important. Fault tolerance techniques are broadly applied in order to enhance the dependability of software. As these techniques increase the development complexity of software, an approach to add fault tolerance to software systems is needed. This approach has to ensure that the system is correctly deployed, new software can be added and in the case of a hardware crash the affected software is redeployed to another hardware component.

We presented fault tolerance patterns as means to build and deploy dependable systems. These fault tolerance patterns are a representation of known fault tolerance techniques like triple modular redundancy. Naive application of these fault tolerance techniques does not take deployment issues into account and, thus, renders the fault tolerance technique pointless. Therefore, we use deployment restrictions to avoid bad deployments. Based on these deployment restrictions an automatic initial deployment is created. During runtime the same deployment restrictions support the deployment reconfiguration of new services or failed services as a self-repair of the system. Tool support exists (1) for the specification of fault tolerance patterns and (2) the automatic transformation of the graphical deployment restrictions to constraints for the ILOG solver software.

In the future, we will look into adding objective functions like minimizing the communication distance between services or minimizing the individual host load. Therefore, the deployment will not only satisfy the deployment constraints, but it will also provide a better overall solution.

The shown example only uses restrictions concerning combined deployment of services on same/different nodes and a very general node type attribute inequality constraint. In [12] attributes are used in a more wider way to further specify the characteristics of nodes. A transformation of these attribute constraints to the input language of a constraint solver is planned.

Acknowledgements. The authors wish to thank Rainer Feldmann and Matthias Meyer for discussions and comments on earlier versions of this position paper.

7. REFERENCES

[1] T. Anderson and P. Lee. *Fault Tolerance - Principles and Practice*. Prentice Hall, 1981.

[2] J. Bowen and M. Hinchey, editors. *High-Integrity System Specification and Design*. Springer Verlag, 1999.

[3] F. Bübl and A. Leicher. Designing Distributed Component-Based Systems With DCL. In 7^{th} *IEEE Intern. Conference on Engineering of Complex Computer Systems ICECCS, Skövde, Sweden*, June 2001.

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

[5] 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.

[6] 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.

[7] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):28–63, 2003.

[8] 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.

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

[10] M. Tichy, B. Becker, and H. Giese. Component Templates for Dependable Real-Time Systems. In *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, September 2004.

[11] M. Tichy and H. Giese. An Architecture for Configurable Dependability of Application Services. In R. de Lemos, C. Gacek, and A. Romanowsky, editors, *Proc. of the Workshop on Software Architectures for Dependable Systems (WADS), Portland, USA (ICSE 2003 Workshop 7)*, May 2003.

[12] M. Tichy and H. Giese. Seamless UML Support for Service-based Software Architectures. In *Proc. of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003, Luxembourg*, volume 2952 of *Lecture Notes in Computer Science*, pages 128–138, November 2003.

[13] M. Tichy and H. Giese. A Self-Optimizing Run-Time Architecture for Configurable Dependability of Services. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems II*, number 3069 in Lecture Notes in Computer Science, pages 25–51. Springer Verlag, 2004. (to appear).

[14] J. Waldo, G. Wyant, A. Wollrath, and S. Kendal. A Note on Distributed Computing. techreport, Sun Microsystems Laboratories, November 1994. TR-94-29.