Computing Optimal Self-Repair Actions: Damage Minimization versus Repair Time

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

ABSTRACT
The dependability of a software system can be improved by online redeployment of failed software components using appropriate system self-repair actions. The effect of different self-repair actions can vary to a great extent w.r.t. the resulting temporary service unavailability and reduced redundancy of services. We therefore developed an approach to efficiently compute self-repair actions which realize requested repair steps in a nearly optimal manner. We show that our approach achieves a suitable compromise between the usually infeasible optimal deployment modification w.r.t. damage minimization and repair time minimization by presenting a number of simulation results.

Categories and Subject Descriptors
D.2.11 [Software]: Software Architectures; D.4.5 [Software]: Reliability—Fault-tolerance; D.4.7 [Software]: Organization and Design—Distributed systems

General Terms
Reliability, Algorithms, Design

Keywords
Deployment, Self-Healing, Distributed Systems, Dependability

1. INTRODUCTION
Software plays an important part in today’s dependable systems like critical infrastructures or transportation systems. High availability and reliability are key dependability properties which have to be satisfied by the software and hardware of such systems. In most embedded dependable systems in addition rather severe resource and timing restrictions must be respected by the software.

While traditional approaches to dependability avoid dynamic reconfiguration altogether (see IEC 61508), several concepts [7, 6, 3, 4, 15] exist that instead propose to let the system reconfigure itself to repair detected problems. If high availability is required, the capability of the system for online reconfiguration enables that required maintenance activities can be done while the system is online. If the system is even able to autonomously re-deploy software components affected by a hardware crash failure, it can ensure that despite transient hardware faults the availability of the system does not decay step by step. Thus, such systems can be classified as self-healing (cf. [9]). Accordingly, a self-repair action consists of a set of components scheduled for re-deployment and an appropriate new deployment for each of these components. In this paper, we only address the issue of hardware crash failures. The term crash failure of a software component is used as a shorthand for a component which is affected by a crash failure of the executing hardware node.

A number of approaches [4, 3, 15] permit to derive a valid deployment mapping for self-repair actions. Considering the limited processing power and rather restricted resources available with embedded devices, we require that the deployment modifications in such systems are planned in such a manner that the resource constraints are met and the negative effects on the availability or reliability are minimized. This naturally means that the self-repair actions should be computed and realized as fast as possible to minimize the repair time. In addition, any migration of software components which either results in temporarily not providing a service or weakening redundancy should be avoided in order to minimize the resulting temporary damage. We abstract from the migration technique provided by an underlying operating and communication system and pessimistically assume that the migrated components are stopped on the old node and restarted on the new node.

In [15], we have presented an approach on how to synthesize deployment rules for the fault tolerant deployment of software components. In this paper, we describe how to compute nearly optimal self-repair actions for a given set of deployment rules. In addition, we provide an evaluation showing that the improved algorithm can be employed in the outlined environment and in most cases offers a nearly optimal compromise between repair time and damage minimization. While computing the optimal solution w.r.t. damage minimization is usually not feasible, the presented algorithm results in most cases in a nearly minimal damage but much shorter repair time due to the significant faster computation.

We first outline the underlying foundations of our approach for the rule based description of component deployment and automatic
deployment in Section 2. Then, the extensions for the efficient computation of deployment mappings for self-repair actions are presented in Section 3. An experimental evaluation of the approach follows in Section 4. We close the paper with a review of relevant related work in Section 5 and some final conclusions as well as an outlook on future work in Section 6.

2. FOUNDATIONS

In [14, 13] we presented an approach that permits to model a service-based system and its deployment at runtime using a UML-based graphical notation. The automatic self-management of the system detects if one of these services crashed at run-time. In this case, it restarts the crashed service taking into account the given deployment restrictions.

The approach mentioned above has been extended in [12, 15] in such a way that it also regards fault-tolerance techniques. To model fault-tolerance in a reusable fashion, we introduced fault-tolerance patterns. Each pattern represents the specification of a certain fault-tolerance technique, like e.g. triple modular redundancy or hot standby. A pattern consists of an architectural part modeled as UML component diagram and deployment restrictions given as UML deployment diagrams. We are currently working on synthesis of the behavioral specification of generic components participating in a fault tolerance pattern as e.g. voting components.

An example for such a pattern is the Triple Modular Redundancy (TMR) pattern. A triple modular redundancy system implements a certain service by three redundant components. The input for these components Component1,..,3 is tripled by a Multiplier. The three results of the components are reduced to one by a Voter. The Voter compares the three results and chooses the result which is calculated by at least two of the components. Thus, a TMR system can tolerate one malfunctioning component. Unfortunately, common cause failures spoil the fault tolerance enhancement of the TMR. E.g. if two of the three redundant components run on the same node, crash failure independence does not hold anymore for node failures and thus the usage of TMR becomes pointless.

In Figure 1, the deployment restriction of this pattern is depicted. The Provider component delivers the value to be tripled. As the Multiplier cannot work without these values, we restrict the deployment in a way that both components do not crash fail independently of each other, i.e. both components must be deployed to the same node, which is depicted by the deployment edges from both components to the same node. The same holds for the Voter and the User which uses the results. On the other hand the three components Component1,..,3 have to run on three different nodes which is depicted by the deployment edges which lead from the components to three different nodes.

To find a suitable initial deployment, i.e. a deployment that meets all deployment restrictions given in the patterns, the graphically specified deployment restrictions are mapped to inequalities over boolean and integer variables. 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. The graphical deployment constraints are then transformed to constraints over these boolean variables. The constraint

$$\forall j: x_{\text{Component1},j} + x_{\text{Component2},j} \leq 1$$

captures the deployment restriction that the two components Component1 and Component2 must not be deployed to the same node. This transformation is further described in [15].

In self-managed systems, it is sometimes necessary to reconfigure the system. This is the case if a further component has to be added or if an existing service failed. In both cases, the initial deployment results can be reused. As reconfiguration is performed at runtime it is essential to minimize the time needed to compute the reconfiguration. Therefore, we suggested in [15] to shrink the constraint problem by leaving the values of the running services unchanged. Thus, only the deployment variables of the failed services have to be calculated. If this constraint problem is not solvable some of the fixed variables have to be relaxed. In this case, some running components are migrated, e.g. by shutting them down and restarting them on another node, to make room for redeployment of failed components. The selection which running components should be migrated was previously not addressed but is of crucial importance.

3. OPTIMIZED SELF-REPAIR

In order to improve this reconfiguration management, we aim at improving two different aspects. First, the amount of time for computing self-repair actions is reduced. Second, the components scheduled for re-deployment have to be carefully selected. If running components must be re-deployed, those components should be selected for re-deployment first which have working redundant copies due to an application of a fault-tolerance pattern. If this requirement is not fulfilled, it will lead to a further reduction of redundancy or it will even result in a failure to provide the required service. Thus, our approach aims at minimizing the negative effects of component failures (including component migration) in combination with the recovery time.

3.1 Reducing Solving Time

In order to describe system requirements, we build sets of properties attached to each node or link and sets of constraints describing requirements of particular components or connectors. To speed up the constraint solving, we can exploit that there are two types of constraints. The first one corresponds to properties whose values are changed during the solving process. An example for this type of properties is the amount of free memory provided by a particular node. It naturally decreases, if a component, which allocates memory, is deployed on this node. Linear restrictions for this kind of relationships are defined as follows.

**Definition 1.** For each node/link $j$ and property $u$, the variable $u_{\text{max}}$ holds the initial property value (e.g. maximum amount of memory), $u_{i,j}$ holds the constraint on the property specified for
the i-th component/link (e.g. memory allocation). Based on these variables, the following constraint must hold \( \forall j: \sum_{i=1}^{m} x_{i,j} u_{i,j} \leq u_{j} \).

The initial value of the property provides an upper bound for the sum of all constraint values corresponding to it. This kind of constraints is called cumulative.

The other type of constraints corresponds to properties whose values do not depend on the number of components/connectors deployed on the corresponding resource. It could be the operating system type or specific scheduling policies provided by a node. The compliance of these constraints depends only on the relation between the property value of a particular node/link and a constraint value of a particular component/connector. This type of constraints is called non-cumulative.

The compliance is independent of the constraint solving process, as e.g. a constraint concerning the operating system type for the node executing a component \( c \) (see equation 1) is independent from any other variable than \( x_{c,j} \).

\[ \forall j: (x_{c,j} = 1) \rightarrow (j.os = \text{linux}) \] (1)

The boolean expression \( n12.os = \text{linux} \) is even constant as the node \( n12 \) obviously does not change its operating system during the constraint solving process. Therefore, if \( n12.os = \text{linux} \) evaluates to false, the variable \( x_{c,n12} \) is always 0 due to constraint 1.

For this reason, we decided to evaluate these non-cumulative constraints prior to the actual constraint solving process. This phase is called pre-solving. The pre-solver removes variables and constraints from the model, if one of the corresponding non-cumulative constraints cannot be satisfied. In the above example, the variable \( x_{c,n12} \) and the constraint \( (x_{c,n12} = 1) \rightarrow (n12.os = \text{linux}) \) for the node \( n12 \) are removed from the constraint model. The variable \( x_{c,n12} \) is replaced by 0 in all occurrences of this variable in other constraints. This process helps us to decrease the number of variables and restrictions dramatically (see Section 4). Thus, the amount of time required for the constraint solving is heavily reduced. This pre-solving has to be done only once prior to the initial deployment and the pre-solved model is reused during online self-repair. Thus, the additional overhead for pre-solving is negligible.

### 3.2 Selecting Components for Migration

As described in Section 2, it might be necessary to migrate running components in order to free up a suitable node for redeployment of the failed components. The selection which components should be migrated is rather important as the damage implied by migrating components should be minimal.

Thus, we propose an algorithm which first only considers the failed components for deployment. If that deployment submodel is not solvable, it expands the submodel with selected running components. The selection is in principal based on the check, whether a redundant copy of the component remains, if a running component is migrated. The redundant component copies stem from applications of fault-tolerance patterns. Therefore, this algorithm either avoids unnecessary migrations of components or if migrations are necessary migrates only components whose migration do not affect the service provision of the system. As the submodels are typically small compared to the complete model, we expect this submodel expansion approach to compute feasible deployments very fast.

Alternatively when solving the complete deployment constraint model rather than only submodels, it is possible to give the constraint solver an objective for minimization which captures the damage associated with migrating a component. Thus, the constraint solver automatically tries to keep the number of component migrations small in order to minimize the objective. As the complete constraint model must be solved and in addition the optimal solution must be found, we expect this to take much more time than the submodel expansion approach.

#### 3.2.1 Submodel Expansion

![Figure 2: Submodel expansion process](image-url)

**Submodel not solvable.**

1. **Initial situation:**
   - **Broken:**
     - Submodel:
       - Consider: Consider later:
         - Submodel not solvable.
   - **Working:**
     - Submodel:
       - Consider:
         - Consider later:

2. **Members of same FT- Pattern instance**
   - **Not related**

3. **Members of same FT- Pattern instance**
   - **Not related**

4. **Members of same FT- Pattern instance**
   - **Not related**

5. **Members of same FT- Pattern instance**
   - **Not related**

6. **Members of same FT- Pattern instance**
   - **Not related**

7. **Members of same FT- Pattern instance**
   - **Not related**

**Figure 2:** Submodel expansion process

Figure 2 shows the behavior of the algorithm based on an example. Initially, the components \( a, b, c \) experience a failure. The other components are working. In each step of the algorithm, we have three sets of components. The first set Submodel includes all components which must be redeployed. In the first step, this set contains only the three failed components. The Consider queue contains the components which are currently not scheduled for migration, but may be considered for migration in later iterations of the algorithm.

In our example, the Submodel is not solvable (step 1), i.e. there exists no deployment of the components in the Submodel which satisfies all deployment constraints. In order to find a deployment, it is necessary to enlarge the Submodel by an additional running component. This gives the constraint solver additional degrees of freedom to find a suitable deployment as this additional component may be migrated to a different node. Thus, the head element of the Consider queue is checked for addition to the Submodel set of components. This check tests whether the component is member of the same fault tolerance pattern as a component in the set Submodel in order to avoid a further reduction of redundancy which would lead to higher damage. In our example, the head element component \( d \) is member of the same fault tolerance pattern instance as component \( c \). Thus, the check evaluates to true and the head element is put into the Consider Later queue (step 2). If not (as in step 3 the component \( e \)), the head element is added to the Submodel set and the constraint solver is executed to solve the new model (step 4). In our example, the Submodel is still not solvable, thus, components \( f \) and \( g \) are checked for adding to Submodel one after another.
(step 5). As both components and components b and e are member in the same fault tolerance instance, the components are added to queue ConsiderLater (step 6). Now, the queue Consider is empty and in order to redeploy the broken components we have to consider the components in the ConsiderLater queue, too. Thus in this special case, we may further reduce the redundancy in order to recover from failure. We add component d to the Submodel and start the constraint solver. Now, the Submodel is solvable and the repair action comprises a redeployment of the components a to e to nodes computed by the constraint solver.

### 3.2.2 Objective Function

The constraint solver has the ability to compute a solution which is optimal for a given minimization objective. We can use this minimization objective in order to penalize the migration of running components. Thus, the constraint solver computes a damage optimal new deployment. Considering the component structure in Figure 3, we attach a damage to each component which captures the damage value if this running component is selected for migration. Components C1 and C5 are single-point of failures as well as the combination of C2, C3, and C4 and, thus, the damage 13 is attached to them. Migration of a single component of C2, C3, C4 implies the damage value 1, whereas the migration of two of these three components is penalized by a damage value of 4.

![Figure 3: Damage calculation of sample component structure](image)

The objective expression is the sum of the damage values of all migrated components. A new deployment of component C2 to another node results in the decision variable $x_{c2,n2}$ getting the value 0. Thus, we must add the damage value for component C2 iff $(1 - x_{c2,n2}) = 1$. The objective of the constraint solver is to minimize the following expression for the given deployment of this simple component structure:

$$\min 13(1 - x_{c1,n1}) + 4(1 - x_{c2,n2})(1 - x_{c3,n3})(1 - x_{c4,n4}) + 2(1 - x_{c2,n2})(1 - x_{c3,n3}) + 2(1 - x_{c3,n3})(1 - x_{c4,n4}) + 2(1 - x_{c2,n2})(1 - x_{c4,n4}) + (1 - x_{c2,n2}) + (1 - x_{c3,n3}) + (1 - x_{c4,n4}) + 13(1 - x_{c5,n5})$$

Obviously, migration of either the important C1 and C5 components or multiple components of C2, C3, and C4 leads to a higher value of the above expression. In contrast, a migration of the single component C4 leads to a small value. This expression can be adapted in accordance to fault tolerance patterns to capture for example a 2-out-of-3 voting behavior.

### 4. SIMULATION EXPERIMENT

In order to validate the approach for an optimized self-repair presented above, we did simulations using the ILOG solver 5.2 software on a 3GHz-P4 Linux system with 1GB RAM.

In our experiments, we used a model containing 36 nodes with 114 links and 72 components with 99 connectors. The model contained a set of properties (5 node-specific and 2 link-specific properties). A half of them are cumulative, another half are non-cumulative properties. Furthermore, there was a set of constraints corresponding to these properties on each component and connector. The redundancy aspects of the model were described by 14 application of the triple-modular-redundancy pattern.

#### 4.1 Effect of Pre-solving

The first experiment aimed at analyzing the influence of pre-solving on the solving time and memory allocation during the solving process. Table 1 shows the model size and the time and memory used by the constraint solver for computing an initial deployment. Pre-solving reduces the model size by 50%. The computation time is reduced by nearly 90%.

<table>
<thead>
<tr>
<th></th>
<th>w/o pre-solving</th>
<th>w/ pre-solving</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>17046</td>
<td>8577</td>
<td>50%</td>
</tr>
<tr>
<td>constraints</td>
<td>21 377</td>
<td>9 018</td>
<td>58%</td>
</tr>
<tr>
<td>time (ms)</td>
<td>14 580</td>
<td>16 10</td>
<td>89%</td>
</tr>
<tr>
<td>memory (mb)</td>
<td>17</td>
<td>9</td>
<td>47%</td>
</tr>
</tbody>
</table>

**Table 1: Effect of Pre-solve**

#### 4.2 Effect of Submodel Expansion

The second experiment shows the effect of our submodel expansion approach in comparison to a redeployment of the complete model. We simulated a series of random node failures which lead to failures of the components executed on these nodes. In each experiment, we simulated a crash failure of a single randomly selected node based on the initial deployment. Table 2 shows that our submodel expansion approach easily beats a newly solved constraint model both in terms of computation time and damage due to failed and migrated components in each experiment.

<table>
<thead>
<tr>
<th></th>
<th>complete</th>
<th>expansion</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>13630</td>
<td>40</td>
<td>99.7%</td>
</tr>
<tr>
<td>damage</td>
<td>775</td>
<td>7</td>
<td>99.1%</td>
</tr>
<tr>
<td>time</td>
<td>14 890</td>
<td>29</td>
<td>98.7%</td>
</tr>
<tr>
<td>damage</td>
<td>97</td>
<td>3</td>
<td>99%</td>
</tr>
<tr>
<td>time</td>
<td>13 790</td>
<td>10</td>
<td>99.9%</td>
</tr>
<tr>
<td>damage</td>
<td>4</td>
<td>5</td>
<td>-25%</td>
</tr>
<tr>
<td>time</td>
<td>13 660</td>
<td>40</td>
<td>99.7%</td>
</tr>
<tr>
<td>damage</td>
<td>34</td>
<td>34</td>
<td>0%</td>
</tr>
</tbody>
</table>

**Table 2: Effect of Submodel Expansion**

In those experiments, the average number of variables and constraints were 8577 and 9 018 for the complete model, but 946.5 and 13 177.75 for the expanded submodel. The memory used by the constraint solver dropped from 9MB for the complete model to 1MB for the expanded submodel. The submodel expansion algorithm expanded the submodel once or twice in the experiments.

#### 4.3 Effect of Objective Minimization

Finally, we evaluate the effect of using an objective function for the constraint solver in order to force the constraint solver to minimize the damage associated with repair actions. We made the same series of experiments as in the previous section. Table 3 shows that the constraint solver is able to compute repair actions which are
only a little bit better than the repair actions computed by our sub-model expansion approach. Though, it needs much more time. In experiment 1, we shut down the constraint solver as it did not return a solution after 1 hour. In experiment 3, the constraint solver is able to compute a significantly better result with a damage value of 1 but needs much more computation time than our submodel expansion approach.

<table>
<thead>
<tr>
<th>complete</th>
<th>expansion</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>damage</td>
<td>time</td>
</tr>
<tr>
<td>1</td>
<td>N/A</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>56060</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>14920</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>16430</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 3: Effect of Objective Minimization

In conclusion, our approach performed nearly as good for minimizing the damage of self-repair actions as the optimal solution returned by the constraint solver, but in a fraction of the time.

5. RELATED WORK

Pinello et al. present a deployment approach [10] which uses constraints to describe a mapping between a system graph and a platform graph. Fault-tolerance constraints over actor replicas are used to guarantee that the system works in case of fault patterns [5]. There is no special approach presented for online-reconfiguration.

Dearle et al. [4] propose a declarative constraint-based deployment language. The Autonomic Deployment and Management Engine (ADME) 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. Similar to our approach in [15], they first try to solve a small submodel consisting of only the failed components. If this submodel is not solvable, they extend it as required. They do not use pre-solving nor do they present, how exactly they extend the submodel. There is no quantitative evaluation given.

Mikic-Rakic et al. present in [8] a number of heuristic algorithms to find good deployments fast. The algorithms maximize the availability of a certain system deployment. As they only consider the initial deployment, there is no special support for online-reconfiguration given. If the presented algorithms are used for online-reconfiguration, the computed deployments will probably lead to arbitrary component migrations and, thus, higher damage costs.

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). If the planner finds a plan to the goal state, it may retry finding other plans in the remaining time. The approach presented in [2] is more general than ours as it allows for arbitrary actions. Though, due to its general nature the planner may need more time to find a rather optimal plan.

In [1], experimental results for computing initial deployments are presented. In those experiments, the number of components, nodes and constraints are less than those in our experiments. Though, our pre-solved model is solved faster by the constraint solver than the models presented in [1]. As different hardware for the experiments are used and in [1] a plan optimized for a minimal start time is computed while our initial deployment is not optimized in any way, the results of the experiments cannot be directly compared.

6. CONCLUSION AND FUTURE WORK

We presented an approach for computing optimal self-repair actions. Our approach is based on a mapping of deployment constraints to the model of a standard constraint solver. Based on this model, a pre-solving step reduces the model complexity by removing deployment variables and according constraints which are not satisfiable. During self-repair, our submodel expansion approach avoids unnecessary component migration in order not to induce further component unavailability while also computing self-repair actions very fast. We presented an experimental evaluation which shows that our approach efficiently computes deployment modifications which realize requested repair steps in a nearly optimal manner concerning the minimization of damage and repair-time. Our approach is also usable for reconfiguration management like administrator induced addition or replacement of components.

We are currently finishing the tool, which supports the graphical specification of fault tolerance patterns and deployment constraints. The frontend to the constraint solver is already finished which supports the improvements presented above. The integration of the constraint solver frontend and a communication and monitoring framework is ongoing work.

The sorting of components in the Consider queue based on the associated damage implied by migration is a possible improvement. This would result in an early migration of less important components, while more important components would only be migrated late during self-repair. In addition, the effect of changes to the number of components which are added to the submodel in one expansion step will be further evaluated.

Special repair rules somewhat similar to those presented in [3] but based on graph transformation systems [11] are evaluated whether they can complement our approach.

7. REFERENCES


