2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems

Planning with Utility and State Trajectory Constraints in Self-Healing Automotive Systems

Benjamin Klöpper, Shinichi Honiden
National Institute of Informatics (NII)
Tokyo, Japan
{*kloepper,honiden*}*@nii.ac.jp*

Jan Meyer, Matthias Tichy
S-Lab, University of Paderborn
Paderborn Germany
{*jmeyer,mtichy*}*@s-lab.upb.de*

***Abstract*—Planning is an important method in self-adaptive systems. Existing approaches emphasize the functional properties of the systems but do not consider possible alternative adaptations resulting in system functionality with different grades of quality. In compositional adaptation, the adaptation process should identify not only a feasible system configuration, but a good one. In safety-critical systems such as cars, the adaptation process has to fulfill special requirements. The sequence of reconfiguration activities has to maintain constraints over the entire state trajectory defined by the adaptation process, e.g., that certain processes are always running or even a minimal number of redundant instances. At the same time, in modern cars, many optional processes, such as learning of the engine model or optimization of control processes, improve the performance of the car. Possible optimization objectives are fuel consumption, driving comfort, and wear. Thus, this paper introduces a model of a self-adaptation process by reconfiguration, which considers the quality of alternative configurations. Furthermore, a planning process is introduced that generates a sequence of reconfiguration activities, which result in good configuration. The introduced process can be used to maintain the basic system functionality and also to select the currently most appropriate task implementations and optional tasks to run in a recovered system, e.g. after hardware failures.**

***Keywords*-Planning; Self-Healing; Automotive System**

I. INTRODUCTION

The new paradigm of self-adaptive software aims to minimize the human effort in setting up and maintaining software systems. The system modifies its behavior in response to changing operating environments [1], if the software is not accomplishing its tasks, or when better functionality or performance is possible [2]. Thus, a self-adaptive system must be able to answer two questions without human interaction. The first question is what good behavior is regarding the current operation conditions and the user preferences. The second question is how the desired behavior can be achieved. Regarding compositional adaptation, the questions can be stated more precisely:

1) What is a good system configuration?
2) How can a good system configuration be achieved?

In safety-relevant systems, such as mechatronic products or cars, not only the final configuration should meet the specification. Because these systems usually cannot stop operation for reconfiguration at an arbitrary time, each intermediate system state or configuration that results from adaptation activities has to meet the safety requirements.

Planning is an important method in the context of compositional adaptation (e.g. cf. [3], [4], [5], [6]). Planning provides a sequence of reconfiguration actions that transforms the given system configuration into a desired new configuration. Most planning approaches in self-adaptive systems rely on planning models, where a goal state is characterized as a set of state conditions, which are either true or false. Such planning processes must consider every configuration that satisfies the goal state constraints as equivalent. Hence, the goal state must be defined precisely enough that any feasible goal state corresponds to a *good* system configuration. Consequently, the approaches by Kramer et al. ([3] [4]) rely on so-called high-level goals. Satzger et al. [5] and Arshad et al. [6] assume that the user or administrator specifies the goal conditions. Furthermore, all approaches either explicitly assume that the system is stopped during reconfiguration (it is the case for [6]) or do not explicitly consider how the functionality of the adaptive system can be maintained during reconfiguration.

Automotive systems are an excellent and challenging application example for self-adaptive systems. First, automotive systems are safety-critical systems and cannot be stopped during operation to perform a reconfiguration. This is especially the case if the self-adaptation process should respond to failure situations, e.g. in a self-healing reaction to a hardware crash. Second, in modern cars, various components aim to optimize the system behavior, e.g., regarding such criteria as fuel or energy consumption, driving comfort and pleasure, emissions, etc. These *optimization* processes are *not required* to maintain the basic functionality of the car, and at the same time their contribution to system behavior depends on the application situation. Recent approaches, such as self-optimizing systems [7], evaluate the contribution or utility of such processes depending on the current and future application situations.

In this paper we introduce a planning model defined in the Planning Domain Definition Language that includes information about the quality or utility of components and state trajectory constraints. The planning model is designed for the challenging application scenario of self-healing in automotive systems. The planning process is demonstrated by using an *off-the-shelf* planning system. Hence, the main contributions of this paper are the following:

- A formal model of adaptation by reconfiguration, considering safety constraints and component utility
- A self-healing scenario for automotive systems
- A planning model with safety constraints and utilities
- Experimental validation of the planning approach

The reconfiguration planning process searches for a good configuration regarding the current situation and a corresponding plan that supports concurrent actions. It also ensures that all relevant tasks are executed at any time. To improve the dependability of the systems, the planning process is able to consider redundant component instances. For specific, safety-critical tasks, a minimum number of redundant instances can be defined and maintained. The issues of concurrent actions [6], runtime reconfigurations and redundancy [8] have been considered before, but not in this combination. Alternative components and configuration quality have also not been considered before.

II. Adaptation by Reconfiguration

To maintain the scope of the paper, we will focus on a specific class of systems. This system class is suitable for many applications, in particular for controlled systems. In such systems, a process is usually not executed to calculate a specific result, but is continuously active in order to manage the system behavior. Hence, we consider a class of systems with the following:

- A set of tasks T, which carry out system activities
- A set $MT \subset T$ of mandatory tasks, which maintain the basic system functionality and must always be active
- A set of components C_t for each Task $t \in T$, which implement the activities
- A set of computation nodes N, which can be used to execute the components
- A set of system-wide resources (e.g. network related) GR, a node resource types NR and a set of node properties NP
- Three functions $cap_G : GR \to \mathbb{N}$, $cap_N : N \times R \to \mathbb{N}$ and $provi : N \to \mathcal{P}(P)$ defining the available capacity of global resources and the resources and properties of a node
- Three functions $con_G : C \times GR \to \mathbb{N}$, $con_N : C \times NR \to \mathbb{N}$ and $need : C \to \mathcal{P}(P)$ defining the resources occupied and properties required by a component

A configuration of a system is defined by an assignment of components to nodes. One component may be execute redundantly on several nodes to improve the dependability of the system. Hence, a system configuration is described by

$$conf : \bigcup_{t \in T} C_t \to \mathcal{P}(N).$$

If a component is not active in a configuration, then $conf(c_t) = \emptyset$ follows. To react to changes in the operating environment (e.g. failed hardware) an adaptation process transforms an initial configuration $conf_i$ into a new and better final configuration $conf_f$, with a number of intermediate configurations $conf_j$. A feasible configuration asserts that all mandatory tasks run (1), no node (2) or global resource (3) exceeds its capacity constraints, and all components have the required properties on the nodes they run (4):

$$\forall t \in MT : \exists c_t : conf(c_t) \neq \emptyset \tag{1}$$

$$\forall n \in N : \forall r \in R \sum_{c \in C | n \in conf(c)} con_N(c) \leq cap(n, r) \tag{2}$$

$$\forall gr \in GR : \sum_{c \in C} |conf(c)| \cdot con_G(c) \leq cap(gt) \tag{3}$$

$$\forall c \in C : \forall n \; conf(c) : need(c) \subseteq provi(c) \tag{4}$$

We assume that the overall performance of the system can be evaluated by the utility function:

$$U_t = \sum_{j=1}^{t} \max_{c_i \in C_j | conf(c_i) \neq \emptyset} (\sum_{j=1}^{pi} (w_{ij} \cdot v_{ij}(pi_{ij}))) \tag{5}$$

where t is the number of tasks and pi is the number of performance indicators of the *i-th* active component. v_{ij} is a value function, mapping specific values of performance indicators to a unified utility measure and w_{ij} denotes the relative importance regarding the global utility function. A systematic procedure to derive v_{ij} and w_{ij} is the *Value Tree Analysis* (VTA), which we previously applied to self-optimizing mechatronic systems [7]. Equation (5) implicitly assumes that redundant instances of a task (which may have a lower utility) do not contribute to overall utility. This assumption is reasonable, if hardware and software failures are considered unusual events, because the redundant tasks will only become active in the case of such failures. Obviously, the inner sum can be used to determine the possible contribution of any component c_i to the overall utility of the system:

$$U(c_i) = (\sum_{j=1}^{pi} (w_{ij} \cdot v_{ij}(pi_{ij}))) \tag{6}$$

III. Application Example

As an application example, we consider a self-healing process in an automotive application. We assume that it is possible to run redundant instances of software components for the same task and that software components can be deployed and started on electronic control units (ECU) at runtime. Today, automotive manufacturers configure the ECU's functions statically at design time, but online reconfiguration is becoming of interest and will be achieved soon. An example of manually defined reconfigurations can be found in [9]. The ability of online reconfiguration can improve the availability by redundant software instances or result in lower costs by flexible usage of computational resources. The AUTOSAR standard [10] with its standardized interfaces and the run-time environment (RTE) is the first step towards reconfigurable systems.

Under the assumption, that relevant system tasks run redundantly in the system, a hardware failure does not result in failure of such critical tasks. Hence, the adaptation can be executed under *soft real-time constraints*: the earlier a reconfiguration plan is provided, the earlier the systems implements the improved behavior. The general operation of the system is not affected. Obviously, in unlikely cases where all nodes fail, which execute a specific task, the operation is affected immediately. This case is beyond the scope of the introduced self-healing process and has to be handled by classical dependability methods for technical systems (e.g., cf. [11]).

A good example of a safety-relevant automotive subsystem is an adaptive cruise control (ACC) [12]. Its functionality is to accelerate the car to the driver specified velocity, if no obstacle is detected. If an obstacle is detected, the car is first decelerated and then the velocity is controlled in order to maintain an adequate gap between the car and the obstacle (usally another car). To provide the functionality of the ACC, several interacting tasks have to be executed simultaneously. The tasks can be divided into mandatory tasks, such as Object Detection which must be always active, and normal tasks such as the Comfort Control. Comfort Control implements two optional functionalities. First, it uses information provided by the navigation systems to reduce the travel speed before curves and turns. Second, Comfort Control reduces the travel speed according to speed limits. The second functionality can be achieved with less computation and memory. Hence, two alternative components implement Comfort Control: one component implements both functionalities, and the other one is a light weight component limited to the second functionality. In any case, the driver has the final responsibility to control and adjust the speed in accordance with the traffic situation. All tasks are executed on electronic control units, which correspond to the computational nodes in the systems.

For the application example, an actual ACC system is simplified to a certain extent. We assume that every control unit is connected to a communication bus. The sensor is also directly connected to the bus, and so the sensor data is available at any control unit.

IV. Reconfiguration Planning Process

This section introduces a planning model for self-healing in automotive systems. The planning model will encompass information about the utility of components and state trajectory constraints. The information about utility will enable a self-healing process, that identifies an optimal new system configuration. State trajectory constraints are used to make it easier to define reliable planning domains. The design of the planning domain is specific for the application domain, although the general modeling concepts can be transferred to various applications. In particular, the handling of resource constraints requires specific knowledge of the executing system, so each productive environment requires a specifically designed or modified planning domain. Especially in consideration of safety-relevant properties, this implies a challenge: the human designer of planning domains has to ensure that safety-relevant properties are maintained by the generated plans. In many cases this is possible by specific design of the action preconditions and effects. However, checking if a set of actions maintains the required properties for every possible planning problem is not trivial and in some cases may be impossible or results in too restrictive action definition, suppressing feasible adaptation processes. The modeling feature of state trajectory constraints can be used to solve this issue and to maintain the safety-relevant properties of the system during the reconfiguration process. Mainly, the constraints will force the planner to generate a plan that maintains certain properties in every intermediate state or configuration.

To use an *off-the-shelf* planning system for experimental validation, the Planning Domain Definition Language (PDDL) is used. This domain independent modeling language for planning problems is mainly used in the biannual *International Planning Competition*, and several planning systems use PDDL as the input format. Since PDDL 2.1 [13] numerical state variables and temporal planning with *durative actions* are supported. To exploit the distributed nature of the considered system with various nodes, we will apply *temporal planning*, where actions have a specific duration and hence concurrency becomes possible. State trajectory constraints were introduced in version 3.0 [14] of PDDL. The following subsections introduce the planning domain[1] for the self-healing process, while the goal definition as an essential part of the problem is explained in the following section.

[1]Domain file can be found at http://homepages.upb.de/kloepper/reconf.pddl

A. State definition

The first part of a PDDL domain definition describes the structure of the states considered in the planning problem. The structure of the states is defined by the objects, predicates and functions. The *self-healing domain* encompasses six types of objects: task, component, node, property, resource and cpu. The type *cpu* is an addition to the formal model from section II and is required to determine the time required to start a component on a node. The *self-healing domain* has two subsets of predicates. The first subset captures the static properties, which will not be affected by the actions of the domain (lines 2-4, domain 1). The second subset describes the current deployment situation and will be affected by actions (lines 5-8, domain 1).

Domain 1. *Static Domain Properties*

```
1  (:predicates
2    (implements ?c − component ?t − task)
3    (providesProperty ?n − node ?p − property)
4    (requiresProperty ?c − component ?p − property)
5    (deployedTo ?c − component ?n − node)
6    (runningOn ?c −component ?n −node)
7    (running ?c −component)
8    (deployed ?c − component))
```

To start any component or to transfer a component to another node, the executables of the component must be available in the memory of a node $(deployedTo\ ?c\ ?n)$. The fact, that a specific component is running on a node n (and hence $n \in conf(c_t)$) is expressed by $(runningOn\ ?c\ -\ component\ ?n\ -\ node)$. The PDDL formalism prohibits the generation of new objects that were not specified in the planning problem. Thus, the tuple $(component,\ node)$ identifies such instances and any component can be deployed or run at most once on each node.

Numerical properties of objects or of relations between objects are expressed in *functions*. Naturally, the available capacities in the system (cap_G) and locally on nodes (cap_N) and the required resources of any component are expressed as functions (lines 2-9 in domain 2 show examples).

Domain 2. *Functions*

```
1  (:functions
2    (availableResource ?k − node ?r − resource)
3    ...
4    (busload)
5    ...
6    (requiresMemory ?c −component)
7    (requiresResource ?c −component ?r −resource)
8    (setupTime ?c −component ?cpu −cpu)
9    ...
10   (repNo ?t − task)
11   (minRep ?t − task)
12   (contribution ?t − task)
13   (copNo ?c − component)
14   (insNo ?c − component)
15   (utility ?c − component))
```

The next functions (lines 10-15, domain 2) model the resource requirements of components. We assume a worst-case busload for each component, calculated by sum of the required communication times, if a component is connected to every possible communication partner. Obviously this results in pessimistic plans and configurations, but this simplification makes the planning problem more tractable.

Besides the capacities and resource requirements, six more functions describe properties of the current configuration. The functions either refer to the running number of instances $(repNo,\ insNo)$, the number of copies $(copNo)$ or the utility of the configuration $(utility,\ contribution)$. Utility denotes how much a component contributes to the system behavior, if it is the best active instance of its task. The utility of the best active task is assigned to the function $contribution$ and hence denotes how a task contributes to the system behavior.

B. Changing the State – Reconfiguration Actions

Actions describe how the self-healing process can change the configuration. Below, a typical action from the domain is shown in PDDL notation. The action describes the abilitiy to transfer the executables of a component from one node to another. The *parameter* clause defines the objects involved in the action. Any object that is later referred to in the action body, has to be defined in the parameter clause (with the exception of objects defined in quantifiers). The *duration* of an action can be defined by a real valued constant, or can be calculated by an arbitrary numerical expression, if its variables are not accessed by any action effects. In case of the transfer action, the duration depends on the size of the component and the reserved transfer load.

Domain 3. *Action Transfer Component*

```
(:durative−action transfer−to−persistentMemory                              1
 :parameters (?c −component ?k −node                                        2
             ?source −node ?t −task)                                        3
 :duration (= ?duration (/ (componentSize ?c) (transferload)))              4
 :condition (and                                                            5
            (at start(replicaOf ?c ?t))                                     6
            (at start(deployedTo ?c ?source))                               7
            (over all(deployedTo ?c ?source))                               8
            (at start(not (deployedTo?c ?k)))                               9
            (at start(> (available_transferload 0)))                        10
 )                                                                          11
 :effect (and                                                               12
           (at end(deployedToStorage ?c ?k))                                13
           (at start(assign (available_transferload 0)))                    14
           (at start(decrease (availableStorageMemory ?k)                   15
             (componentSize ?c))                                            16
           )                                                                17
           (at end(isDeployed ?c))                                          18
           (at end(increase (copNo ?c) 1))                                  19
           (at end(assign (available_transferload transferload)))))         20
```

The *condition* clause defines the boundary conditions, which have to be fulfilled during the execution of the action. Contrary to classical (non-temporal) planning, these conditions cannot be exhaustively defined based on the state before the start of an action. Instead, certain conditions may be required at the beginning of an action (*at start*), at the end (*at end*) or during the execution of an action ((*over all*)). A good example is the availability of the components at the source node: As long, as the transfer proceeds, the executables cannot be removed from the memory of source nodes (lines 7-8, domain 3). The effect of the action is defined in the same way. The component is finally available at the persistent memory of the goal node *at the end*. Limited to the expressions *at start, at end, over all*, all required resources have to be allocated at the beginning of action (lines 14-17, domain 3), otherwise the planner may generate inconsistent resource allocations. In total, the *dynamic reconfiguration domain* encompasses five more actions for which we give short informal descriptions.

Domain 4. *Domain Actions*

Start *(?c1 ?c2 −component, ?n −node, ?t −task)*
Conditions*: ?c1 is an instance of ?t, sufficient free resources on ?n, no other instance of ?t running on ?n at the start and during the action, ?c2 is the best running implementation of ?t at the end*
Effects*: Resources required by ?c are blocked on ?n, ?c1 running on ?n, one more instance of ?c1 and ?t is running, (utility ?c2) is assigned to (contribution ?t).*

Stop *(?c1 ?c2 −component, ?n −node, ?t −task)}*
Conditions*: ?c1 runs on ?n, no other implementation of ?t runs on at the start and during the entire action on ?n, ?c2 is the best running implementation of ?t at the end*
Effects*: Resources reserved by ?c1 released, ?c1 not running on ?n, one instance less of ?c1 and ?t, (utility ?c2) is assigned to (contribution ?t)*

StopLast *(?c1 −component, ?n −node, ?t −task)}*
Conditions*: ?c1 runs on ?n, at the end of the action no other instance of ?t is running somewhere in the system*
Effects*: Resources reserved by ?c1 released, ?c1 not running on ?n, one instance less of ?c1 and ?t, 0 is assigned to (contribution ?t)*

Swap *(?c1 ?c2 ?c3 −component, ?n −node, ?t −task)*
Conditions*: ?c1 runs on ?n, ?c2 runs not on ?n, ?c3 is the best running implementation of ?t*
Effects*: Resources reserved by ?c1 released, resource required by ?c2 blocked, ?c1 not running on ?n, ?c2 running ?n, (utility ?c3) is assigned to (contribution ?t)*

Remove *(?c −component, ?n1 −node)*
Conditions*: Component ?c is deployed to node ?n1*
Effects*: ?c is not deployed to ?n1, the storage blocked by ?c is released*

The action definitions have to ensure a proper evolvement of the functions of the domain. In particular, the number of redundant task instances $repNo\ ?t$ is an issue. Given that the initial value of $repNo\ ?t$ is correct, all actions have to maintain the following conditions: increase $repNo\ ?t$ only if a new pair $(task,\ node)$ is added and decrease $repNo\ ?t$ only if pair $(task,\ node)$ is removed.

Remark 1. $repNo\ ?t$
Since these conditions refer to the effects of a single action, they can be easily maintained by proper combination of preconditions and effects. The effect $increase((repNo\ \ ?t)1)$ *is only allowed if the action also contains the effect* $isRunning(?c\ \ ?k)$ *in combination with the precondition* $instanceOf(?c\ \ ?t)$ *and* $not(exists\ (?c2\ -component)(and(isRunning\ ?c2\ ?k)$ $(instanceOf(?c2\ ?t))))$. *For this reason, the action* swap *is introduced, which does not change* $(repNo\ ?t)$.

C. State Trajectory Constraints

The state trajectory constraints are conditions that must be fulfilled by the entire sequence of states visited during the execution of a plan. They are expressed through temporal modal operators over first order formulae involving state predicates [14]. In the self-healing automotive system, we consider the following global constraints:

Domain 5. *State Trajectory Constraints*

```
(:constraints
 (and
  (forall(?t −task) (always (> (repNo ?t) (minRep ?t))))
  (forall(?c −component) (always (> (copNo ?c) 0)))
  (forall(?n −node ?r −resource)
     (always (>= (availableResource ?n ?r) 0)))
  (forall(?n −node) (always(>=(availableStorageMemory ?n)0)))
  (forall(?n −node) (always
     (>= (availableWorkingMemory ?n) 0)))))
```

The constraints in lines 3 and 4 ensure, that there is always a sufficient number of task instances and component copies, while lines 5-9 model the capacity constraints.

V. The Self-Healing Process

Figure 1 shows a diagram of the self-healing process, where we assume the availability of a self-healing infrastructure offering monitoring, diagnosis, and execution of reconfiguration actions. The monitor component is responsible for monitoring the system state and detecting failure situations, in particular hardware failures. If such a failure situation is recognized, a corresponding recovery planning problem is generated, that aims to restore a system state that fulfills the safety and redundancy constraints of the system. This planning problem and the planning domain (described in section IV) are used as input for the planning system. The resulting plan – if one can be found – maintains the state trajectory constraints and achieves a configuration that fulfills the redundancy properties. This plan is immediately carried out by an executor, that is able to conduct the start, stop, and transfer of components on the various nodes.

The configuration resulting from the initial plan is evaluated in terms of the utility function in equation (5). The

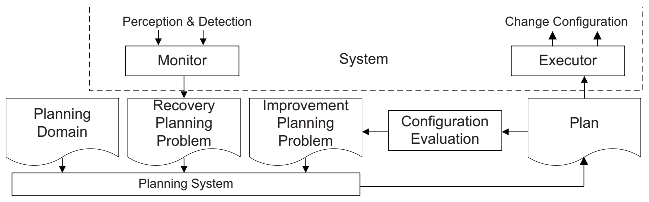

Figure 1. Diagram of the Self-Healing Process

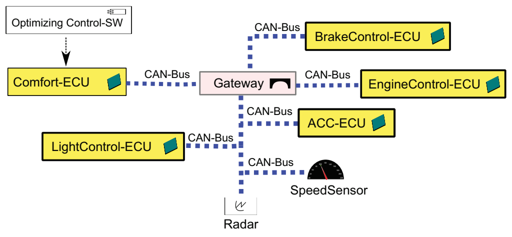

Figure 2. ACC Overview

result is used to generate a new planning problem, this time with the new goal of increasing the system utility.

A. Generating Planning Problems

In a failure situation and in particular after a hardware crash, the redundancy constraints of several tasks may be violated. The system functionality is usually still maintained, as redundant instances carry out the affected tasks. But the system functionality is at risk, as an additional failure may remove the remaining instances. Hence, the first step aims to recover a system state that maintains the redundancy constraints. As the constraints are already violated, they cannot be applied unmodified in the recovery planning problem. To generate a feasible planning problem, for each task that does not have a sufficient number of instances, the number of minimum redundant instances is set to the number of current redundant instances. Hence, the planner is not allowed to deactivate any additional component. Given a set of tasks $\{T1, T2, ...TN\}$ that are below their minimum number of redundant instances (n_i), the goal of the recovery problem is to restore the minimal number of redundant instances of $\{T1, T2, ...TN\}$ while maintaining the modified state trajectory constraints.

However, the planning system is not aware of the meaning and importance of utility and the contribution function. Hence, the new system configuration may be suboptimal in terms of system utility. The evaluation component calculates the current utility of the system and generates a new planning problem, this time with the original values for $(minRep?t)$ for all tasks.

$contribution?t$ is the utility of the best active component implementing task $?t$ or 0, if no such component is active. Thus, the sum overall contributions equals the utility function 5. To improve the system performance, the utility of a new configuration has to be greater than utility of the current configuration. The state trajectory constraints ensure that no plan is generated that violates the redundancy constraints either during the reconfiguration or the final configuration.

B. Alternative Search Strategies

In [8] we introduced two alternative search strategies to find a good or even optimal system configuration regarding a numerical evaluation function. Both strategies iteratively increase the goal value of the numerical evaluation function until no plan can be found. The first strategy s_i assumes that any plan that improves the system state is immediately applied. Hence, in each iteration the initial state of the planning problem is changed. The second strategy s_f assumes that only the final found plan is applied to systems and hence starts each iteration from the original initial state.

Remark 2. *Optimality of* s_f
With the assumption that the planning algorithm is complete, s_f *will find an optimal system configuration regarding the utility function in equation (5) and the initial state after first recovery.*
Because the planning algorithm is complete, it will find in each iteration i *a goal state with* $utility_i > utility_{i-1}$*, if there is such a state. Consequently, if in iteration* t *no plan can be found, then there is no such goal state with* $utility_t > utility_{t-1}$ *and hence* $utility_{t-1}$ *is the optimal utility value and the corresponding goal state and plan are found in iteration* $t-1$.

The strategy s_i on the other hand can only find local optima, as in each iteration the search commits itself to a subset of the search space. Nevertheless, s_i might be beneficial for practical applications, where time and memory limitations have to be considered. The commitment to new initial states and hence subsets of the solution space reduces the size of the search space in each iteration significantly. Hence, if computation time and memory are limited, s_i might outperform s_f. In general, the more computation time and memory are constrained, the more suitable is s_i.

VI. Use Case and Experiments

This section describes an illustrative use case of the self-healing process in automotive systems and the results from an experimental validation. SGPlan5 [15] was used as a planning system, a planner that supports the required language features.

A. Use Case: Failing ACC

This section describes a planning process for self-healing after an hardware crash in the ACC example. In figure 2, the system topology is presented, only one exemplary software component deployment to an ECU is shown. The deployment of all software components and the redundant elements is shown in table 1 (top of page 8). All software components

START	ACTION	DURATION
0.001:	(TRANSFER ECC CE ECE)	[2.0000]
0.002:	(STOPLAST OPT1 CE OPTT)	[1.0000]
2.003:	(TRANSFER VCC CE ECE)	[2.0000]
4.004:	(TRANSFER DC1 CE ECE)	[0.8000]
4.805:	(TRANSFER IC ECE LCE)	[1.0000]
5.806:	(START IC IC ECE IT ECU)	[1.0000]
5.807:	(TRANSFER ECC BE)	[2.0000]
7.808:	(START ECC ECC BE ECT ECU)	[1.0000]
7.809:	(TRANSFER IC BE LCE)	[1.0000]
8.810:	(START DC DC CE DT ECU)	[1.0000]
9.811:	(START TGC TC CE TGT ECU)	[1.0000]
10.812:	(START VCC VCC CE VCT ECU)	[1.0000]

Table II
RECOVERY PLAN WITH START TIMES AND DURATIONS

Node	Memory	Storage	Property	
A	916k - 1024k	768k - 1024k	32 bit	
B	768k - 812k	352k - 768k	32 bit	
C	256k - 512k	256k - 512k	32 bit	
Component	Memory	Size	Property	Factor
I	300k - 460k	200k - 400k	32 bit	1.0
II	192k - 256k	100k - 190k	16 bit	0.75
III	192k - 256k	100k - 190k	16 bit	0.55
IV	15k - 64k	15k - 32k	8 bit	0.55
V	5k - 10k	4k - 8k	8 bit	0.1

Table III
NODE AND COMPONENT CLASSES FOR PROBLEM GENERATION

except the optimizing control have two instances in the system. This redundancy guarantees the correct operation of the system, even if one ECU fails. The optimizing control is implemented by two alternative software components (full and light), of which one consumes less memory but provides also slightly poorer performance. As all other instance are mandatory and implemented by only a single component type, only the optimization tasks contribute to the system utility ($util_{full} = 20$,$util_{light} = 10$).

The first step of the self-healing process is the determination of a recovery plan that is immediately applied to the system. According to the description in section II, the minimum number of redundant instances of the affected tasks were reduced in order to obtain a feasible planning problem. Given the problem file with the initial state, the goal, state and state trajectory constraints, SGPlan5 returned the recovery plan shown in table 2. First, the components are transferred to the ECU where they will be started. Since the Comfort ECU does not provide sufficient memory to run optimize (full), indicator, and velocity control at the same time, optimize (full) is stopped in order to achieve a feasible system configuration (recovery configuration cf. table 1 on page 8). As the only task providing a utility value is stopped, the current configuration has a new utility of 0. Consequently, the goal is defined by any configuration that possesses a utility larger than 0. With this goal, SGPlan5 generates a plan transferring the small instance of optimized control to the brake ECU and starting it there (final configuration cf. table 1 on page 8).

B. Experimental Validation

To give the approach broader validation, randomly generated self-healing problems from specific classes of nodes, tasks, and components were used. Table 3 shows the node and component classes. For each instance in a specific problem, values are drawn from the intervals using an equal distribution. The factor is multiplied with the utility assigned to the task class, ensuring that more resource consumption results in higher utility. Task classes are assigned subsets of implementing component classes; we considered five classes with three alternative components and two classes with two alternative components. For validation, problem instances with 4 to 6 nodes (100 each) were generated and tested.

Table 4 shows the planning times for immediate recovery depending on the (original) number of nodes and components, the two main factors of the combinatorial complexity of the reconfiguration planning process. The actual planning time depends on a large number of factors, e.g. the availability of memory on nodes with the required properties, available storage memory, distribution of physical copies, etc. These numerous factors make a compact statistical analysis difficult. Hence, table 4 shows four different performance indicators: the average, maximum, minimum planning time and the 0.75 quartile. As was expected, there is a clear correlation between the number of nodes and components and the planning time. The most critical value may be considered the maximal recovery time of about 5 seconds for 6 nodes, 21 tasks and 54 components (from the last line of table 4). Considering, that the systems due to redundant mandatory tasks remain functional, this recovery time seems to be acceptable. Figure 3 shows an aggregated comparison of the two search strategies for problem instances with four nodes. As assumed, the search strategy assuming immediate plan application (s_i) is relatively better with short deadlines, where as the complete search strategy s_f is able to find better configurations if the deadline is rather long. A detailed analysis brings up interesting figures: for 7% of the problems, the first improvement was found within 1 second (61% within 2 seconds, 81% within 3 seconds and 84% within 4 seconds). For 16% of the problem instances no plan improving the initial utility after recovery could be found. The average improvement for the s_i increases from 0.78% (1 second) within 5 seconds to 13%.

VII. RELATED WORK

This section reviews similar and complementing approaches regarding reconfiguration, planning with objective functions, and self-healing in automotive systems.

A. Reconfiguration Planning

Planning is an important method in reconfiguration problems. All approaches we found are based on the qualitative

Software component	Brake ECU (BE)	EngineControl ECU (ECE)	Light ECU (LE)	ACC ECU (ACCE)	Comfort ECU (CE)
Brake Control (BCC)	I,R_1,R_2				I,R_1,R_2
Light Control (LCC)	I,R_1,R_2		I,R_1,R_2		
Brake Light (BLC)	I,R_1,R_2		I,R_1,R_2		
TimeGap Control (TGC)	I,R_1,R_2	R_1,R_2		L	
Engine Control (ECC)	R_1,R_2	I,R_1,R_2		L	
Velocity Control (VCC)		I,R_1,R_2		L	R_1,R_2
Object Detection (DCC)	R_1,R_2	I,R_1,R_2		L	
Indicator (IC)			I,R_1,R_2	L	R_1,R_2
Full Optimizing Control (OPT1)					I
Light Optimizing Control (OPT2)			R_2		

Table I
SYSTEM CONFIGURATIONS
(I = RUNNING IN INITIAL STATE, L = LOST AFTER CRASH, R_1 = RUNNING AFTER FIRST RECOVERY, R_2 = RUNNING AFTER IMPROVEMENT)

Nodes	Components	Avg.	Max.	Min.	0.75 Quartil
4	23-26	68	129	28	88
4	30-32	105	252	27	156
5	39-41	221	604	40	282
5	44-46	517	1699	130	602
6	47-48	1013	2988	258	1311
6	50-52	1039	3294	269	1104
6	53-55	1250	4697	322	1700

Table IV
RECOVERY TIMES [ms]

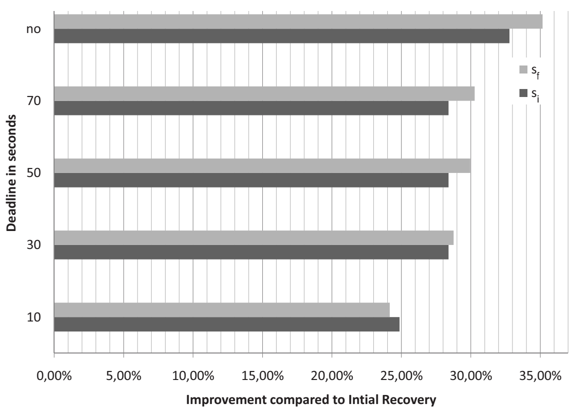

Figure 3. Comparison of Search Strategies

conditions the final configuration has to fulfill. Arshad et. al [6] introduced deployment and reconfiguration planning based on the PDDL planning formalism. Their system Planit creates plans for the initial deployment of components in a distributed computing environment. Capacity constraints such as memory are not considered and it is assumed that the files required to start a component are available on every computer. Any deployment in which all components are running is considered as feasible and Planit searches for the plan that minimizes the time until all components are running. A similar approach, but one considering resource constraints and limited availability of components on distributed nodes is introduced in [16]. In [8], we suggest a PDDL domain for the consideration of runtime restrictions in the reconfiguration problem: the generated plan can be executed while the reconfigured system is in operation. The system maximizes the robustness of the overall system based on a *minimal fault tolerance* over all tasks. Alternative implementations of tasks was not considered. Satzger et al. [5] introduced automated planning approaches for organic computing systems. Instead of autonomously reasoning about the utility of the achieved behavior, the user or administrator specifies goal state conditions, which must be met in the final configuration. Intermediate configurations, which are visited during the configuration process, are not explicitly considered. It remains unclear if the example system, a robot assembly cell, remains in operation during reconfiguration. Kramer et al. ([4], [3]) introduced a three layer architecture that separates functions with respect to their time scale. On the first layer (goal management), plans are generated according to predefined goals. This layer has the largest time scale. The second layer (change management) links the generated plans to the execution layer, which has the finest time scale. In accordance with the detected status, the change management selects an precalculated plan. Similar to the approach of Satzger et al., the architecture requires externally defined *high-level goals*.

B. Planning Domain Definition Language

Since version 2.1, PDDL features *plan metrics* [13]. A plan metric specifies how a generated plan will be evaluated for a particular problem. The consideration of the plan metric by the planning system is optional (e.g., a system considers the metric for one problem instance but ignores it for another one). Any arithmetic expression can be used to specify a metric. Preferences, introduced in version 3.0 [14], improve the expressiveness metrics. A preference is a label for condition, that is preferred to be true, e.g., that a certain component is active in the final configuration or that all nodes have always at least 20% free memory. The function ($is - violated\ preference$) counts how often a preference is violated (e.g., two nodes drop below 20% free memory at the same time) and this numerical function can be integrated into the plan metric. Obviously, these modeling features are ideal for the reconfiguration problem. Unfortunately, we found in the experiments that the planning process is much less tractable, metrics are often ignored and planning time increases significantly. In particular, a bigger number of preferences (around five) resulted in unsolved planning problems, which could swiftly be solved if no metric or preferences was used. Hence, we decided to stick to the optimization strategy originally introduced in [8]. PDVer [17] is a tool to verify properties of PDDL domains, e.g. the evolvement of numerical variables. Systematic verification tools are useful for larger planning domains, where the number of actions grows so large, that important conditions cannot be directly checked by the domain developer.

C. Self-Healing Automotive Systems

As previously mentioned, we focus on a special aspect of self-healing in automotive systems: the planning or more generally decision making process on how to achieve a new and valid configuration. Other approaches for self-healing in automotive systems are more comprehensive and capture also aspects like monitoring, diagnosis and execution of reconfiguration or healing mechanisms. At the same time, these approaches lack an advanced decision making process.

A good example of a comprehensive self-healing architecture for automotive systems is given by Dinkel et al. [18]. The architecture is based on a classification scheme for automotive systems and constantly carries out four actions: health-monitoring, root-cause-diagnosis, planning of changes, and the execution of action. The main focus is on the system monitoring and diagnosis. The decision making about self-healing activities is rule-based, by previously defined situation-action schemes provided by the system developer. Complex sequences of reconfiguration activities, which might be necessary to maintain redundancy requirements, are unlikely to be provided in such a rule base. Hence the contribution of this paper might be considered a supplement to such architectures.

Another example of an automotive middleware our approach could supplement was developed in the DySCAS project ([19], [20]). The middleware and reconfiguration mechanism focus on the recognition of errors and degrade the functionality, so that minimal functionality is available in case of failures. The presented approach, in contrast is able to generate situation-specific configurations, which maintain as much and as useful functionality as possible.

Trumler et al. [21] introduced architecture for self-configuration and self-healing in automotive systems based on a combination of the AUTOSAR standard and the organic middleware. Like the approach presented in this paper, the self-healing process based on the organic middleware tries to optimize a metric. However, the metric is not defined in terms of system utility, but by utilization of resources. In the case a service or a node crashing, the introduced self-healing process tries to rebuild a valid configuration with the remaining hardware and software resources. Redundant execution of software is not explicitly considered, nor are physical copies of the executables.

None of the existing approaches for self-healing in automotive systems distinguishes between mandatory and optional systems tasks, or is able to handle alternative implementation with different trade-offs of resource consumption and delivered quality. Consequently, the approaches are not able to identify the best possible configuration after a hardware failure. Furthermore, state constraints that must be maintained during the entire reconfiguration process are not explicitly addressed and hence can hardly be ensured.

VIII. Limitations and Further Work

The limitations of the planning model and process presented in this contribution are mainly in the expressiveness of the planning domain and the availability of utility information about the system. Regarding the *first issue*, one major limitation is the detailed modeling of component dependencies and communications. It is desirable that the information flow between components is explicitly considered. An explicit modeling makes the approach suitable for more general communication infrastructures than bus systems. This possibility is limited more by the scalability of the planning systems, than by the expressiveness of the planning language. Hence, future efforts have to aim to improve the scalability of the planning process. In general we found, that more specific goal states is, the larger the problem instances and the more complex the models that can be solved. Thus, a possible extension is the definition of precisely defined goal configurations instead of quantitative expression such as number of instances or minimal utility. If the planning aspect is ignored, the identification of an optimal system configuration is a combinatorial optimization problem. Obviously, there remains the question if a reconfiguration process exists, which leads to the optimal configuration. Nevertheless, in particular population-based neighborhood searches such as *genetic algorithms* are promising for obtaining additional information about good candidate configurations. As a very important aspect, the constraints defining the feasibility of a configuration have not to be encoded in the optimization problem. This would result in various non-linear and even discontinuous functions. Instead, the feasibility could externally handled. For instance, standard SAT-Solver could be incoperated for this purpose. The *second issue* is the autonomous assessment of situation-dependent utility functions. Autonomous assessment does not refer to a process, where a software system generates preferences and utility from scratch, but one where the system valuates the utility function depending on the current perception without human interaction. A straightforward approach is to provide the system with quantitative metrics of system behavior during the design time. Here the integration of methods from decision support (as applied in [7]) and work on goal oriented design for self-adaptive systems ([22], [23]) is a promising direction. For a holistic domain design process, tools such as PDVer [17] shall be included.

IX. Conclusion

This paper introduced a planning model for generating plans for compositional adaptation and demonstrated its application in a self-healing scenario for automotive systems. The planning model exploits the current state-of-the-art artificial intelligence planning, in order to provide plans that generate a sequence of configurations that maintain safety related properties, such as minimum number of active redundant instances. This ability enables flexible online reconfiguration, while maintaining the basic functionality of the self-adaptive systems. Furthermore, the planning model considers alternative implementations of system tasks, in order to use trade-offs regarding resource consumptions and level of functionality. We introduced two alternative search strategies to find optimal or good final system configurations. The first strategy is theoretically optimal, while the second one is better suited for practical application. With an adaptation mechanism that is able to consider quantitative aspects of the functional behavior of systems, a cornerstone for self-adaptive systems with new levels of autonomy is set. For future work, we envision an architecture for autonomous decision-making in self-adaptive systems, further minimizing the required human interaction.

Acknowledgment

B. Klöpper is a visiting researcher at NII and scholarship holder of the German Academic Exchange Service (DAAD)

References

[1] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, and E. L. Wolf, “An Architecture-Based Approach to Self-Adaptive Software,” pp. 54–62, 1999.

[2] R. Laddaga, “Self adaptive software problems and projects,” *Proceedings of the Second International IEEE Workshop on Software Evolvability*, vol. 0, pp. 3–10, 2006.

[3] D. Sykes, W. Heaven, J. Magee, and J. Kramer, “From goals to components: a combined approach to self-management,” in *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. ACM, 2008, pp. 1–8.

[4] J. Kramer and J. Magee, “Self-Managed Systems : an Architectural Challenge Self-Managed Systems : an Architectural Challenge,” in *International Conference on Software Engineering*, 2007, pp. 259–268.

[5] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, “Using automated planning for trusted self-organising organic computing systems,” in *Autonomic and Trusted Computing*. Springer, 2008, pp. 60–72.

[6] N. Arshad, D. Heimbigner, and A. Wolf, “Deployment and dynamic reconfiguration planning for distributed software systems,” *Software Quality Journal*, vol. 15, no. 3, pp. 265–281, 2007.

[7] B. Klöpper, C. Romaus, A. Schmidt, H. Voecking, and J. Donoth, “Defining plan metrics for multi-agent planning within mechatronic systems,” in *Proceedings of ASME 2008 International Design Engineering Technical Conference & Computers and Information in Engineering Conference*, 2008.

[8] C. Danne, V. Dück, B. Klöpper, and M. Tichy, “Considering Runtime Restrictions in Self-Healing Distributed Systems.” in *Proceedings of the 21st International Conference on Advanced Networking and Applications*. IEEE Computer Society Press, 2007, pp. 228–235.

[9] B. Becker, H. Giese, S. Neumann, M. Schenck, and A. Treffer, “Model-based extension of autosar for architectural online reconfiguration,” in *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems*, 2009, pp. 123–137.

[10] “Autosar specification,” Autosar GbR, 2009. [Online]. Available: www.autosar.org

[11] B. Bertsche, *Reliability in Automotive an Mechanical Engineering*. Berlin: Springer-Verlag, 2004.

[12] H. Winner, S. Hakuli, and G. Wolf, *Handbuch Fahrerassistenzsysteme - Kapitel Adaptive Cruise Control -*. Vieweg, 2009.

[13] M. Fox and D. Long, “PDDL2.1: an extension to PDDL for expressing temporal planning domains,” *Journal of AI Research*, vol. 20, no. 1, pp. 61–124, 2003.

[14] A. Gerevini and D. Long, “Preferences and soft constraints in PDDL3,” in *Proceedings of ICAPS workshop on Planning with Preferences and Soft Constraints*, 2006, pp. 46–53.

[15] C. Hsu, B. Wah, R. Huang, and Z. Chen, “New Features in SGPlan for Handling Soft Constraints and Goal Preferences in PDDL3.0,” in *Fifth International Planning Competition*, 2006.

[16] T. Kichkaylo, a. Ivan, and V. Karamcheti, “Constrained component deployment in wide-area networks using AI planning techniques,” in *Proceedings International Parallel and Distributed Processing Symposium*, 2003.

[17] F. Raimondi, C. Pecheur, and G. Brat, “PDVer, a Tool to Verify PDDL Planning Domains,” in *Proceedings of ICAPS’09 Workshop on Verification and Validation of Planning and Scheduling Systems*, 2009.

[18] M. Dinkel, S. Stesny, and U. Baumgarten, “Interactive Self-Healing for Black-Box Components in Distributed Embedded Environments,” in *Kommunikation in Verteilten Systemen 2007*, 2007, pp. 243–254.

[19] R. Anthony, A. Rettberg, D. Chen, I. Jahnich, G. de Boer, and C. Enkelin, “Towards a dynamically reconfigurable automotive control system architecture,” *International Federation for Information Processing (IFIP)*, vol. 231, pp. 71–84, 2007.

[20] R. Anthony, A. Leonhardi, C. Ekelin, D. Chen, M. Trngren, G. de Boer, I. Jahnich, S. Burton, O. Redell, A. Weber, and V. Vollmer, “A future dynamically reconfigurable automotive software system,” *Proceedings of the IESS*, 2007.

[21] W. Trumler, M. Helbig, A. Pietzowski, B. satzger, and T. Ungerer, “Self-configuration and self-healing in autosar,” *14th Asia Pacific Automotive Engineering Conference*, 2007.

[22] H. Nakagawa, A. Ohsuga, and S. Honiden, “Constructing self-adaptive systems using a kaos model,” in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2008, pp. 132–137.

[23] M. Moradini, L. Penserini, and A. Perini, “Towards goal-oriented development of self-adaptive systems,” in *SEAMS08,*, 2008, pp. 9 – 16.