Self-Healing by Property-Guided Structural Adaptation

Denise Ratasich, Thomas Preindl, Konstantin Selyunin, Radu Grosu
Technische Universität Wien
Vienna, Austria
Email: firstname.lastname@tuwien.ac.at

***Abstract*—Self-healing is an increasingly popular approach ensuring resiliency, that is, a proper adaptation to failures, in cyber-physical systems (CPS). A very promising way of achieving self-healing is through structural adaptation (SHSA), by replacing a failed component with a substitute component. We present a knowledge base modeling relations among system variables given that certain implicit redundancy exists in the system and show how to extract a substitute from that knowledge base using guided search. The result of our search, i.e., the substitute, is optimal w.r.t. a user-defined utility function considering properties of the system variables (e.g., accuracy). We demonstrate our approach - Self-Healing by Property-Guided Structural Adaptation (SH-PGSA) - by deploying it in a real-world CPS prototype of a rover whose sensors are susceptible to failure. We further show the increased runtime performance to find the optimal substitute by comparing it to related work.**

I. Introduction

Consider an autonomous car on the highway which tries to avoid collisions with the car in front of it. Typically, such a car uses range measurements (e.g., radar or laser) to emergency stop when a safety margin is violated. The safety margin will be chosen such that the car stops before crashing into the obstacle considering its current velocity and distance towards the car ahead. Observation components of a cyber-physical system (CPS) – such as the range finder of the autonomous car – may fail due to internal or external influences. Examples are timing or concurrency issues (e.g., late data from the range finder), hardware or software errors (e.g., missing communication link, platform/task crash), unexpected environmental conditions (e.g., rain) or inappropriate usage (e.g., incorrect mounting angle of the radar). Since these components provide inputs to CPS controllers, the CPS may fail (the car crashes into the car in front) or its performance, reliability or usability may considerably decrease.

Hence we desire the service delivery or functionality to persist even when facing unexpected failures in the underlying sensor network. In other words, the system is desired to be *resilient* to changes [1]. Possible fault-tolerance strategies [2] applied to our example are as follows: *i)* Trigger the emergency stop when the car detects a crash; *ii)* Switch to the acoustic parking system in case of a failure [3]; *iii)* Add redundant range finders and vote the nearest distance [4], [5]; *iv)* Design an adaptive Kalman filter to fuse related distance sensors [6], [7]. However, all these methods and many self-adaptation techniques [8] have to be considered during design time and will increase the complexity of the application, subsequently leading to a more sophisticated design and test.

A CPS is typically assembled from subsystems of different manufacturers, each incorporating its needed sensors and state estimation components. This often leads to implicit redundancy, i.e., components of possibly different subsystems observe related physical entities. We model the implicit redundancy in a knowledge base and react to unexpected behavior such as failures of observation components by adding a substitute component during runtime. We refer to this approach as *self-healing by structural adaptation* (SHSA).

In this paper, we propose a flexible and optimal SHSA. In particular, our novel contributions are:

- We introduce a simple model for self-healing by exploiting implicit redundancy that is able to incorporate user-defined performance measures.
- We introduce an optimal substitute-search algorithm, we call *self-healing by property-guided structural adaptation* (SH-PGSA), that takes advantage of the model and is guided by an associated performance measure.
- We present an exemplary utility function that returns the optimal substitute first. We compare the runtime performance of SH-PGSA with related work on an artificial, an automotive and a real-world application. The real-world application is an implementation on a rover prototype performing reliable collision avoidance.

The rest of the paper is organized as follows. Section II gives an overview of related work. Section III provides background information to SHSA. Section IV describes the knowledge base we use for SHSA. Section V presents SH-PGSA. Section VI states an utility function, provides experimental results and compares SH-PGSA to related work. Finally, Section VII concludes by summarizing the results we have achieved and outlines future work.

II. Related Work

The authors in [9], [8], [2] give an overview to self-adaptation, self-healing and fault-tolerance.

[10] introduces an ontology defining physical relations or semantic equivalences between variables that can be shared (e.g., laws of physics) and demonstrates SHSA which is referred to as *ontology-based runtime reconfiguration* (ORR). To execute SHSA on a specific application an instance of the ontology – that is a knowledge base – has to be created. We

extend the knowledge base with properties and utility theory. ORR substitutes a failed variable by depth-first search (DFS) traversing the ontology until a substitute is found with the root equal to the failed observation. This paper describes a guided search and an utility function returning the best substitute first.

Similar models are used in other areas of adaptation. For instance, the authors in [11] use ontologies and context information (cf. properties) to adapt sensor fusion. The SH-PGSA knowledge base is also related to Bayesian networks. In particular, it is a representation of a (deterministic) sensor model which may be part of a dynamic Bayesian network (DBN) [12]. The knowledge base of SH-PGSA is static, however, the variables and properties are actually *random variables* of a CPS which may be described by a probability distribution. Dynamic decision networks, that is an extension of DBNs with utilities and decision nodes, are already considered for self-adaptation [13], [14]. However, to the best of our knowledge, only ORR and SH-PGSA exploit implicit redundancy.

The trend goes towards web-based technologies adopted for CPS. For instance, the service-oriented architecture is applied to a CPS platform to enable self-adaptation [15], [16]. The resources and workflow models are adopted to self-manage the system, i.e., running services w.r.t. the system requirements and to increase efficiency [17], [18]. However, these methods are still not ready to be used in real-time or safety-critical systems since the approaches are typically non-deterministic or lack real-time performance. Less flexible, but predictable, methods switch between predefined configurations [3]. ORR and SH-PGSA separate static and runtime information. An upper bound for the execution time for the search in the static structure of the knowledge base can be derived.

Instead of structural adaptation the parameters or the software components themselves may be adapted, e.g., by changing the algorithm through parameters [19]. SH-PGSA exploits implicit redundancy of the variables exchanged between components to setup a substitute, i.e., applies outside of a component.

III. Self-Healing through Structural Adaptation

Self-healing is the process of detecting and recovering from failures in dynamic systems [8]. Contrast this to traditional fault-tolerance which is a design-time setup and cannot therefore cope with unexpected failures. Self-healing *adapts* the system during runtime in order to mitigate failures. In this paper we search for a substitute candidate (Sec. V) for the structural adaptation [19] which is performed on (sub-) system level considering a component- or service-oriented architecture.

A. Running Example

The running example is the autonomous car represented by a mobile robot which tries to avoid collisions with humans and objects. Our robot under test (Fig. 1) is equipped with a Jetson TK1 running the Robot Operating System (ROS) [15] and controlled via WiFi using a notebook. A ROS application can be distributed into several processes so-called *nodes* which

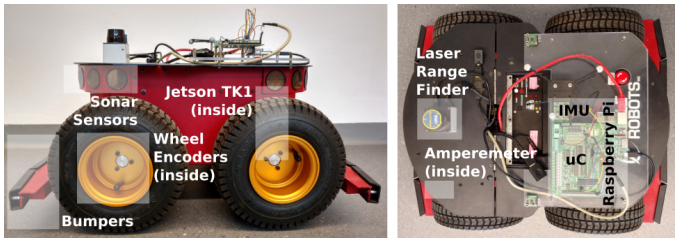

Fig. 1. Mobile robot equipped with its sensors and processing units.

may run on different machines. Nodes communicate via a message-based interface over TCP/IP. In particular, ROS nodes subscribe and publish to ROS topics (cf. named channels). ROS can start new nodes and reconfigure the communication flow of existing nodes during runtime and is therefore suitable for SHSA [16].

B. System Model

A CPS (e.g., our rover) consists of hardware and/or software components (e.g., Jetson TK1, ROS nodes) connected via a common network interface. We focus on adaptation in the software cyber-part of a CPS Z (cf. hardware reconfiguration or dynamic reconfiguration of FPGA). Hence, we assume that each physical component comprises at least one software component z (e.g., the driver of the laser range finder on the rover) and henceforth consider software components only.

A system $Z = \{z_1, .., z_l\}$ can be characterized by properties referred to as system features, or simply as *variables* V (e.g., velocity or weight of our rover). The values of system variables are communicated between components typically via message-based interfaces. Such transmitted data that is associated to a variable v, we denote as information atom [20], short *itom* v. A variable can be provided by different components simultaneously (e.g., p redundant sensors). $Itoms(\mathsf{v}) = \{v_1, .., v_p\}$ collects the itoms associated to the same variable v.

Each software component z executes a program P that uses input itoms I and provides output itoms O (Eq. 1).

$$z_i = (P, I, O) \qquad I = \{v_1^{(i)}, .., v_m^{(i)}\} \tag{1}$$
$$P\colon O = f_i(I) \qquad O = \{v_1^{(o)}, .., v_n^{(o)}\}$$

The CPS implements some functionality, a desired service (e.g., collision avoidance). The subset of components implementing the CPS' objectives are called *controllers* $Z_{ctrl} \subset Z$.

An itom v is *needed*, when v is input of a controller, that is, $\exists z \in Z_{ctrl} \mid v \in z.I$. A variable v is *provided* when at least one itom exists ($|Itoms(\mathsf{v})| \geq 1$).

Case Study (Fig. 2): The rover is tele-operated by the controller $z_{notebook}$ publishing the desired velocity v_{cmd}. The distance to the nearest obstacle v_{dmin} is evaluated by the component z_{dmin_calc} using the laser range finder as input. As soon as v_{dmin} falls below a safety margin (for simplicity a constant value) the robot's verified desired velocity v_{safe_cmd} is set to 0. The component z_{rover_uc} applies v_{safe_cmd} to the motors and provides actual velocity, estimated position by dead-reckoning and sonar range measurements.

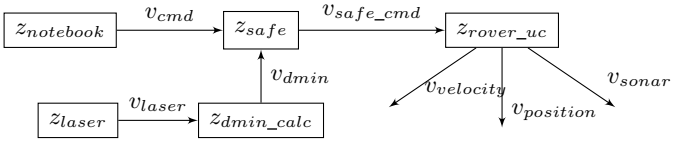

Fig. 2. ROS nodes (components) communicating over ROS topics (itoms).

C. Failure Model

An itom v has *failed*, when the itom deviates from the specification. We assume a monitor that detects such failures and notifies SH-PGSA of failed itoms. SH-PGSA immediately starts to substitute a failed itom. In our experiments we assume fail-silent itoms for simplicity, however, with a suitable monitor (e.g., which is able to shutdown erroneous components) also babbling idiots [2] (i.e., components that provide wrong itoms) can be handled. Failures can propagate from one component to another, i.e., a failed itom can lead to further failed itoms.

Case Study: In our failure scenario the laser range finder breaks. The itoms v_{laser} and v_{dmin} (due to error propagation) fail. However, the controller z_{safe} uses v_{dmin} to avoid crashes.

D. Problem Statement of SH-PGSA

SH-PGSA shall *substitute a needed but failed itom v_f by semantically equivalent itoms*. In other words, search (Sec. V) a suitable knowledge base (Sec. IV) to exploit implicit redundancy. SH-PGSA has to find the best (i.e., given a user-defined performance measure) function f_s using itoms I as input to calculate v_f, and instantiate a substitute component z_s (*substitute* for short) that's output is the itom $v_s \in Itoms(\mathsf{v}_f)$.

$$\begin{aligned} z_s &= (P, I, O) \qquad I = \{v_1^{(i)}, .., v_m^{(i)}\} \\ P\colon O &= f_s(I) \qquad O = \{v_s\} \quad v_s \in Itoms(\mathsf{v}_f) \end{aligned} \tag{2}$$

Note, that the substitute z_s and subsequently its output v_s should further satisfy the system's requirements regarding, e.g., safety. The interested reader is referred to [10], [21].

IV. Knowledge Base

This section defines the knowledge base used to describe implicit redundancy, and adds utility theory to assess possible substitute components.

A. Variables and Relations

Variables are related to each other. A relation $\mathsf{r} : \mathsf{v}_o = f(\mathsf{V}_I)$ is a function or program (e.g., math, pseudo code or executable python code) to evaluate an output variable v_o from a set of input variables V_I. The relations can be defined by the application's domain expert or learned (approximated) with neural networks, SVMs or polynomial functions (see [16]).

B. Structure

The knowledge base $K = (\mathsf{V}, \mathsf{R}, E)$ is a bipartite directed graph (which may also contain cycles) with independent sets V of variables and R of relations of a CPS. V and R are the nodes of the graph. Edges E specify the input/output interface of a relation. In particular, v_i is an input variable for r iff $\exists(\mathsf{v}_i, \mathsf{r}) \in E$ denoted as $\mathsf{v}_i \xrightarrow{e} \mathsf{r}$. v_o is the output variable of r iff $\exists(\mathsf{r}, \mathsf{v}_o) \in E$ denoted as $\mathsf{r} \xrightarrow{e} \mathsf{v}_o$. $Pred_Y(x)$ denotes the predecessors of a node x in graph Y.

There are no bidirectional edges, i.e., if $\mathsf{v}_i \xrightarrow{e} \mathsf{r} \rightarrow \nexists \mathsf{r} \xrightarrow{e} \mathsf{v}_i$. Hence a variable is either input or output to a relation, but never both. A relation can further have only one output variable, i.e., for $\forall j \neq i$ if $\mathsf{r} \xrightarrow{e} \mathsf{v}_i \rightarrow \nexists \mathsf{r} \xrightarrow{e} \mathsf{v}_j$.

The nodes alternate between variables and relations, i.e., variables are only connected to relations and vice-versa (bipartite). Note that relations have to be modeled as nodes, not edges, because a variable is typically related to more than one variable via a single relation.

C. Properties

Properties characterize an entity. A property p is identified by a name (e.g., velocity) and a corresponding entity (e.g., rover). The function $prop$ assigns a value to the property with name $name$ for a specific entity x. $p_{x,name} = prop(x, name)$ denotes the property value of a property $name$ of entity x. $Props(x) = \{p_{x,name} \mid \exists prop(x, name)\}$ collects the properties available for an entity x.

The knowledge base considers properties of relations, variables and itoms. The properties are used to distinguish and assess these entities of the knowledge base.

Case Study: Properties of itoms, e.g., the accuracy or sample rate of a sensor output, can be extracted from its datasheet or estimated by experiments. Properties of variables and relations may be the number of associated provided itoms or the computational costs respectively.

D. SH-PGSA Properties

Apart from system-related properties, SH-PGSA uses specific predicates defining its own concepts, such as when is a variable needed or provided.

$$prop(\mathsf{v}, \texttt{'need'}) = \begin{cases} True & \text{if } \exists\, z \in Z_{ctrl}, v \in Itoms(\mathsf{v}) \mid v \in z.I \\ False & otherwise \end{cases} \tag{3}$$

$$prop(\mathsf{v}, \texttt{'provided'}) = |Itoms(\mathsf{v})| > 0 \tag{4}$$

If an itom of a variable v is *directly* used by a controller z, the value of $p_{\mathsf{v},\texttt{'need'}}$ is $True$ (Eq. 3), otherwise $False$. $p_{\mathsf{v},\texttt{'provided'}}$ is $True$ if the variable v is provided by at least one itom (Eq. 4).

Note that all properties may change during runtime, e.g., a sensor may be disconnected (affects $p_{\mathsf{v},\texttt{'provided'}}$).

E. Substitution

A substitution s of v_{sink} is a connected acyclic sub-graph of the knowledge base with following properties: *i)* The output variable is the only sink of the substitution (acyclic + Eq. 6). *ii)* Each variable has zero or one relation as predecessor (Eq. 7). *iii)* All input variables of a relation must be included (Eq. 8; it follows that the sources of the substitution graph are variables only).

$$s = (\mathsf{v}_{sink}, \mathsf{V}_s, \mathsf{R}_s, E_s) \tag{5}$$
$$\mathsf{v}_{sink} \in \mathsf{V}_s, \mathsf{V}_s \subseteq \mathsf{V}, \mathsf{R}_s \subseteq \mathsf{R}, E_s \subseteq E$$
$$\forall x \in \mathsf{R}_s \cup \mathsf{V}_s \setminus \mathsf{v}_{sink} \quad \exists y \in \mathsf{R}_s \cup \mathsf{V}_s \mid \exists x \xrightarrow{e \in E_s} y \tag{6}$$
$$\forall \mathsf{v} \in \mathsf{V}_s \quad |Pred_s(\mathsf{v})| \leq 1 \tag{7}$$
$$\forall \mathsf{r} \in \mathsf{R}_s \quad \forall \mathsf{v} \in Pred_K(\mathsf{r}) \mid \mathsf{v} \in \mathsf{V}_s \tag{8}$$

A substitution s is *valid* if all sources are provided, otherwise the substitution is invalid. We denote the set of valid substitutions of a variable v as $S(\mathsf{v})$. Only a valid substitution can be instantiated (to a substitute) by concatenating the relations R_s to the function f_s which takes selected itoms I_s as input (e.g., best itoms of the source variables).

F. SH-PGSA Utilities

An utility function collects the preferences of a system. As a consequence, it can be used as the performance measure of the system [12]. We use utility functions to asses each valid substitution, i.e., to rank the substitutions to the best one. Itoms, variables, relations and substitutions have all an utility. The utility $u_x = u(Props(x))$ of an entity x is a function of all the entities' properties.

In the knowledge base, a substitution s has no separate properties, but only includes the properties of used itoms, variables and relations.

$$u_s = u(Props(s)) = u(p_1, .., p_p) \tag{9}$$
$$p_i \in Props(x), x \in \{I_s, s.\mathsf{V}_s, s.\mathsf{R}_s\}$$

The optimal or best substitution $s_{best} = \arg\max_{s \in S(\mathsf{v}_{sink})} u_s$ of a set of valid substitutions $S(\mathsf{v}_{sink})$ for a variable v_{sink} is the substitution with the highest utility.

G. Case Study

The SH-PGSA knowledge base of our running example is depicted in Fig. 3. When the laser range finder crashes, the itoms v_{laser} and v_{dmin} fails. The itom v_{dmin} is needed (input to controller z_{safe}) and therefore shall be substituted. Hence, we have to find a substitution with sink node v_{dmin} (see Fig. 4 for examples).

V. Implementation

A needed and failed itom is first mapped to a variable v_{sink} in the knowledge base. If v_{sink} is unprovided, Algorithm 1 searches for the best substitution s_{best} with sink node v_{sink}.

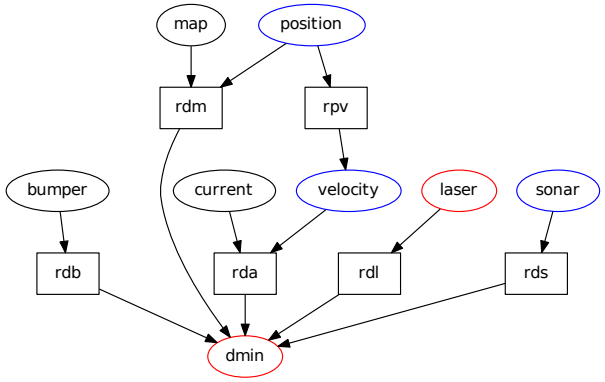

Fig. 3. Exemplary knowledge base of the mobile robot. Boxes are relations. Ellipses are variables (suppressed v and r for readability). Encoding: blue - provided variables; red - variables where itoms failed; black - unprovided variables.

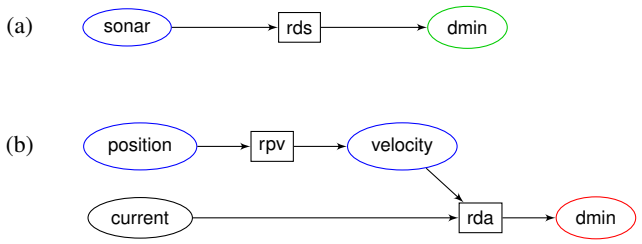

Fig. 4. (a) A valid substitution, because all source variables are provided. (b) The source variable $\mathsf{v}_{current}$ is not provided and leads to an invalid substitution.

Algorithm 1 SH-PGSA

```
1:  procedure SUBSTITUTE(v_sink)
2:      W ← { NewWorker(v_sink) }                ▷ set of workers
3:      while W ≠ ∅ do
4:          while W[0].HASNEXT() do
5:              W_new ← W[0].NEXT()              ▷ continue ...
6:              W ← W ∪ W_new                    ▷ and add new workers
7:              SortByUtility(W)
8:          end while
9:          if Successful(W[0]) then             ▷ best worker done
10:             return W[0].s
11:         end if
12:         W ← W \ W[0]
13:     end while
14:     return ∅                                 ▷ no substitution found
15: end procedure
```

SH-PGSA (Alg. 1) finds possible substitutions for v_{sink} similar to a breadth-first search. An object called *worker* is used to construct a substitution. The internal data of a worker comprises:

- unprovided variables V,
- possible relations per unprovided variable $R[\mathsf{v}] = Pred_K(\mathsf{v})\ \forall \mathsf{v} \in V$,
- (possibly incomplete) substitution s and
- the knowledge base and utility function.

A worker can be initialized by the unprovided variable(s) to substitute next, or relations per variable to proceed. The first worker starts from the sink node v_{sink} (Alg. 1:2). The best worker $W[0]$ is always on turn (by Alg. 1:7). The worker proceeds step by step (calling its method NEXT) by substituting its unprovided variables as long as the unprovided variables have relations as predecessors (checked in HASNEXT). The worker has finished successfully when its underlying substitution s is valid (Alg. 1:9). However, the worker may also abort, if an unprovided variable cannot be substituted. In this case the next worker will get on its turn (Alg. 1:12).

Algorithm 2 Worker's method HASNEXT

```
1:  function HasNext()
2:      if V = ∅ then
3:          return False          ▷ no unprovided inputs, done
4:      end if
5:      R ← ∅                     ▷ collect possible relations per variable
6:      for v ∈ V do
7:          R[v] ← Pred_K(v) \ s.R_s
8:          if R[v] = ∅ then      ▷ no substitution for v
9:              return False      ▷ stop worker
10:         end if
11:     end for
12:     return True               ▷ unprovided inputs can be substituted
13: end function
```

HASNEXT (Alg. 2) checks if there are unprovided variables V to substitute (Alg. 2:2) and collects all possible relations per variable $R[v]$ (Alg. 2:7). Each unprovided variable $\mathsf{v} \in V$ has to be substituted. If this is not possible, the worker aborts (Alg. 2:9). The substitution of the worker is then invalid.

If the worker can proceed (Alg. 2 returns $True$), the worker's method $Next$ is called (Alg. 3).

Algorithm 3 Worker's method NEXT

```
1:  function Next()
2:      C = Product(R)            ▷ create combinations of relations
3:      s.R_s ← s.R_s ∪ Best(C)   ▷ add best combination
4:      W ← ∅                     ▷ create workers for others
5:      for c ∈ C \ Best(C) do
6:          W ← W ∪ NewWorker(c)
7:      end for
8:      V ← ∅                     ▷ collect inputs for added relations
9:      for r ∈ Best(C) do
10:         V ← V ∪ Pred_K(r)
11:     end for
12:     V ← Unprovided(V)         ▷ filter vars to substitute next
13:     return W
14: end function
```

NEXT (Alg. 3) performs a combinatorial product C on the possible relations (Alg. 3:2) collected by HASNEXT. A combination is a set of relations $c = \{\mathsf{r}_1, .., \mathsf{r}_n\}$ with $|c| = |V|$ that selects a relation for each unprovided variable, i.e., $\mathsf{r}_i \in R[\mathsf{v}_i]$ for $i = 1..n$. The worker proceeds with the best combination $Best(C)$ (Alg. 3:3). The best combination is the combination $c \in C$ that leads to the highest utility when added to s. For other possible combinations, new workers are created[1] (Alg. 3:6) and returned. The unprovided inputs collected from the relations in $Best(C)$ are used to continue (Alg. 3:8-12). Figure 5 visualizes the NEXT step.

VI. EXPERIMENTS

SH-PGSA shall return the best substitution in minimal runtime. This section defines an exemplary flexible utility function that is optimal for the presented use cases and ensures that the first substitution returned is the best one (next two sections). Furthermore, the runtime of SH-PGSA is evaluated (w.r.t. size, connections, itoms in different knowledge bases and the use cases) and compared to ontology-based runtime reconfiguration (ORR) [10] and depth-first search (DFS). The experiments use the implementation of SH-PGSA[2] in Python and the ROS application[3] of obstacle avoidance. The evaluation is executed on a notebook with quad-core i7 2.1GHz and 8GB RAM.

A. Utility Function

Performance is a user-defined and application-dependent amalgamation of the various properties. We picked the optimal one for the presented use case. Equation 9 defines the utility of a substitution in general. However, we define the utility of a substitution hierarchically (Eq. 10) and in a way such that the utility can be computed efficiently.

$$u_s = \prod_{\mathsf{r} \in s.\mathsf{R}_T} u_\mathsf{r} \tag{10}$$

$$u_\mathsf{r} = \sum_{p_{x,name}} w_i \, u(p_{x,name}) \qquad x \in \mathsf{r} \cup Pred_K(\mathsf{r}) \tag{11}$$

We use normalized utilities (best utility $u_\top = 1$, worst utility $u_\bot = 0$). By requesting the utility of a relation to be less than one, we can penalize the number of relations. Note that u_s is monotonically decreasing with the number of relations. The product in Eq. 10 further enables an iterative calculation of the utility. Moreover, the best combination $Best(C)$ can be selected by considering the relations to be added only: $Best(C) = \arg\max_{c \in C} \prod_{\mathsf{r} \in c} u_\mathsf{r}$.

The utility of a relation is a weighted sum of utilities of properties of the relation and predecessor/input variables. In particular, the utility function of a relation uses the properties $p_{\mathsf{r},\texttt{'|vin|'}}$, $p_{\mathsf{r},\texttt{'|vunprovided|'}}$ and $p_{\mathsf{v},\texttt{'accuracy'}}$.

$$prop(\mathsf{r}, \texttt{'|vin|'}) = |Pred_K(\mathsf{r})| \tag{12}$$

$$prop(\mathsf{r}, \texttt{'|vunprovided|'}) \tag{13}$$
$$= |\{\mathsf{v} \in Pred_K(\mathsf{r}) \mid prop(\mathsf{v}, \texttt{'provided'}) = False\}|$$

$$prop(\mathsf{v}, \texttt{'accuracy'}) \tag{14}$$
$$= \arg\max_{v \in Itoms(\mathsf{v})} p_{v,\texttt{'accuracy'}}$$

[1]Note that the actual implementation does not instantiate a new worker before it is needed - to save memory.

[2]https://github.com/dratasich/shsa

[3]https://github.com/dratasich/shsa_ros

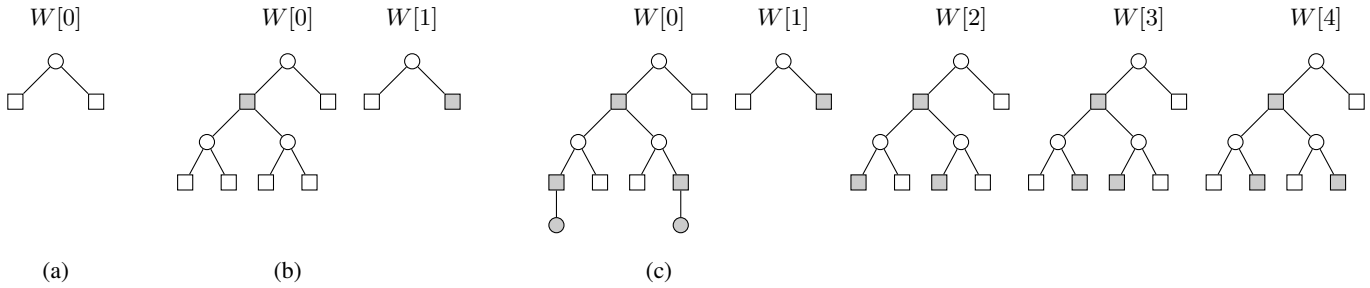

Fig. 5. Underlying substitutions of the workers executing NEXT() (encoding: provided variables and selected relations are filled). (a) Initial set of workers that is a single worker starting from the sink node. *W[0].*HASNEXT() identifies the two possible relations. (b) *W[0].*NEXT() chooses the left relation reaching two unprovided variables and returns another worker for the right relation. (c) *W[0].*NEXT() selects the best combination and returns additional three workers.

The utility functions of the properties are given in Equations 15 to 17.

$$u(p_{\mathsf{r},\texttt{'|vin|'}}) = \frac{1}{p_{\mathsf{r},\texttt{'|vin|'}}} \tag{15}$$

$$u(p_{\mathsf{r},\texttt{'|vunprovided|'}}) = \frac{1}{p_{\mathsf{r},\texttt{'|vunprovided|'}} + 1} \tag{16}$$

$$u(p_{\mathsf{v},\texttt{'accuracy'}}) = p_{\mathsf{v},\texttt{'accuracy'}} \tag{17}$$

The utilities of properties $p_{\mathsf{v},\texttt{'|vin|'}}$ and $p_{\mathsf{v},\texttt{'|vunprovided|'}}$ guide the search to relations with less variables to substitute. Provided variables with higher accuracy are preferred (the accuracy is directly specified as a value between 0 and 1, for simplicity). The weights have been selected such that our preferences match the results of SH-PGSA (cf. training of utility function by so-called preference elicitation).

B. First Substitution is Optimal

Considering the utility function described above: *The first substitution returned is the optimal one.* Outline of a proof: The best worker is always on turn (it has the highest utility) and therefore finishes first. Once it is finished, other workers cannot return a better substitution, i.e., with higher utility. Because an unfinished worker has unprovided variables to substitute, hence relations have to be added to finish, which decrease the utility (utilities of substitutions are monotonically decreasing by Eq. 10 and $u_\mathsf{r} \leq 1$).

C. Runtime Performance

Table I shows the measured execution time of the different algorithms to substitute the failed itom given a model.

The use cases under test are our rover and an automotive drivetrain which model is given in [10] (failure scenario: steering angle sensor breaks). In addition, we create random trees (constraints of Sec. IV-B) to evaluate the effect of the depth and branching factor of a knowledge base on the execution time of the substitute search (recall: more unprovided input variables give more possible combinations, i.e., possible substitutions). All paths lead to a valid substitution, i.e., all leaves are provided.

ORR and SH-PGSA (best) return as soon as a valid substitution is found. DFS searches the whole knowledge base and is therefore able to return all possible solutions including the optimal one. ORR is an efficient DFS implementation that stops at the first valid substitution without the overhead of calculating any utility or using objects introduced for SH-PGSA. It therefore leads to a fast result. However, SH-PGSA outperforms DFS to find the optimal substitution. Note that the setup of the substitute component typically takes much longer than the search (e.g., it takes about 500ms to start a ROS substitute [16] on the platform used for the experiments given in Table I).

TABLE I
AVERAGE TIME (IN MS OUT OF 100 EXECUTIONS) OF FIRST, BEST OR ALL SUBSTITUTIONS RETURNED GIVEN A MODEL.

Model	ORR	SH-PGSA (best)	DFS
Rover	0.10	0.21	0.53
Drivetrain	0.15	0.30	0.73
Balanced8 noU	1.56	823.22	21840

Balanced8 noU simulates following knowledge base: A balanced tree with constant branching factor $b = 2$ and depth $d = 8$ where nodes have equal(= no) utility. Note that a balanced graph with branching factor 2 and a depth of 8 nodes has already 32768 possible substitutions. SH-PGSA will perform a breadth-first search (BFS). Because the utilities of relations are equal, and decrease with depth the worker will change in every step. SH-PGSA (best) is in this case significantly slower than ORR.

However, when introducing random utilities and randomly provided itoms, varied depth and branching, SH-PGSA outperforms ORR (besides DFS) (Fig 6, Fig. 7; bar: standard deviation; 'x': max outliers).

Due to the combination of relations (Alg. 3), the number of possibilities increases exponentially with the number of unprovided variables, or in general with b^d (true for all presented algorithms). ORR is suitable to find a first substitution very fast in a knowledge base that has negligible differences in utility (constant branching factor, constant costs of nodes). However, SH-PGSA will find a solution which is also the best substitution faster in average graphs. This is due to the utility function that guides the search to less input variables and variables that are provided.

To increase the performance, the workers can be paral-

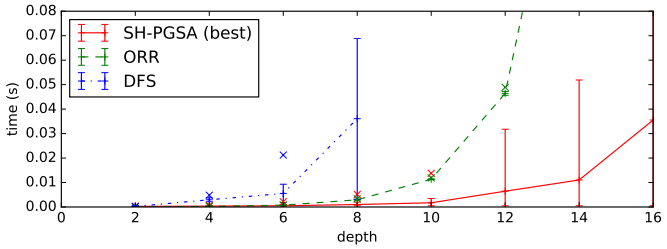

Fig. 6. Average time (in s out of 10 executions of 100 different models each with constant branching factor $b = 2$) until the first substitution is returned.

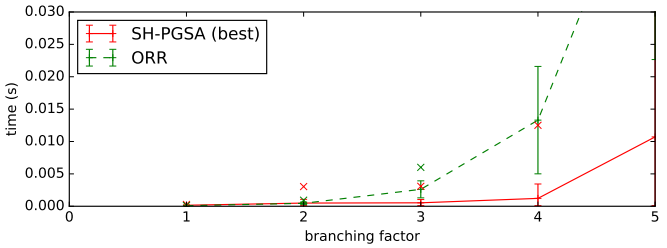

Fig. 7. Average time (in s out of 10 executions of 100 different models each, with constant depth $d = 8$) until the first substitution is returned.

lelized (i.e., separated into n threads working on the best n substitution possibilities, e.g., using Python coroutines). The utility function provides also means to further the decrease the runtime of the search, e.g., by including the distance to provided variables. With such a utility function, SH-PGSA can be compared to the A* algorithm which uses the distance to the target to guide the search and is a commonly used algorithm in path planning (note, a proper heuristic has to be defined to remain correct, i.e., still find the optimal solution for every input).

VII. Conclusion

Assemblies of CPS subsystems lead to more data which is often highly interrelated or even redundant. We present a knowledge base to model and an algorithm to exploit implicit redundancy in such systems to substitute failed observation components, considering system properties. The algorithm considers a user-defined flexible performance measure to guide the search. It has been shown that SH-PGSA significantly decreases the average execution time until a substitution is found and returns the optimal substitution first.

In ongoing and future work we will use the knowledge base for monitoring and sensor fusion. For instance, a subset of variables and relations may be chosen to compare signals against each other, or to fuse a system variable to decrease its variance. Furthermore, the knowledge base shall be able to handle state-based variables and the algorithm shall consider constraints, e.g., system requirements, on substitutes.

Acknowledgment

The research leading to these results has received funding from the IoT4CPS project partially funded by the "ICT of the Future" Program of the FFG and the BMVIT.

References

[1] J.-C. Laprie, "From Dependability to Resilience," in *Dependable Systems and Networks (DSN 2008), 38th Annual IEEE/IFIP International Conference*, 2008.

[2] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. New York: Springer, 2011.

[3] R. Adler, D. Schneider, and M. Trapp, "Engineering Dynamic Adaptation for Achieving Cost-Efficient Resilience in Software-Intensive Embedded Systems," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, March 2010, pp. 21–30.

[4] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, April 1962.

[5] D. Kim and R. Voyles, "Quadruple adaptive redundancy with fault detection estimator," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, Aug 2017, pp. 542–547.

[6] R. Mehra, "On the identification of variances and adaptive kalman filtering," *IEEE Transactions on Automatic Control*, vol. 15, no. 2, pp. 175–184, April 1970.

[7] H. Mitchell, *Multi-Sensor Data Fusion - An Introduction*. New York: Springer, 2007.

[8] H. Psaier and S. Dustdar, "A survey on self-healing systems: approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, Jan 2011.

[9] P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, Eds., *Self-aware Computing Systems: An Engineering Approach*, ser. Natural Computing Series. Springer, 2016.

[10] O. Höftberger, "Knowledge-based Dynamic Reconfiguration for Embedded Real-Rime Systems," Ph.D. dissertation, Technische Universität Wien, 2015.

[11] E. Martí, J. García, and J. M. Molina, "Adaptive sensor fusion architecture through ontology modeling and automatic reasoning," in *2015 18th International Conference on Information Fusion (Fusion)*, July 2015, pp. 1144–1151.

[12] S. Russel and P. Norvig, *Artificial Intelligence - A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey: Pearson Education, 2010.

[13] N. Bencomo, A. Belaggoun, and V. Issarny, "Dynamic decision networks for decision-making in self-adaptive systems: A case study," in *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2013, pp. 113–122.

[14] H. C. Lee and S. W. Lee, "Decision Supporting Approach under Uncertainty for Feature-Oriented Adaptive System," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 3, July 2015, pp. 324–329.

[15] Open Source Robotics Foundation, Inc., "Robot Operating System Wiki - Introduction," Available at http://wiki.ros.org/ROS/Introduction, accessed 2017-10-05, 2017.

[16] D. Ratasich, O. Höftberger, H. Isakovic, M. Shafique, and R. Grosu, "A Self-Healing Framework for Building Resilient Cyber-Physical Systems," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2017, pp. 133–140.

[17] C. Legat and B. Vogel-Heuser, *An Orchestration Engine for Services-Oriented Field Level Automation Software*. Cham: Springer International Publishing, 2015, pp. 71–80.

[18] W. Dai, V. N. Dubinin, J. H. Christensen, V. Vyatkin, and X. Guan, "Toward Self-Manageable and Adaptive Industrial Cyber-Physical Systems With Knowledge-Driven Autonomic Service Management," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 2, pp. 725–736, April 2017.

[19] B. H. C. Cheng et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer Verlag, 2009, pp. 1–26.

[20] H. Kopetz, "A Conceptual Model for the Information Transfer in Systems-of-Systems," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, June 2014, pp. 17–24.

[21] T. Amorim, D. Ratasich, G. Macher, A. Ruiz, D. Schneider, M. Driussi, and R. Grosu, *Runtime Safety Assurance for Adaptive Cyber-Physical Systems: ConSerts M and Ontology-Based Runtime Reconfiguration Applied to an Automotive Case Study*. Hershey, PA: IGI Global, 2017, pp. 137–168.