Automatic generation of repair suggestions for overall I&C architecture represented with an ontology

Ovsiannikova, Polina; Pakonen, Antti; Vyatkin, Valeriy

Published in:
2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)

DOI:
10.1109/ETFA54631.2023.10275557

Published: 15/09/2023

Document Version
Peer reviewed version

Automatic generation of repair suggestions for overall I&C architecture represented with an ontology

Polina Ovsiannikova*, Antti Pakonen†, and Valeriy Vyatkin*‡
*Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
‡Department of Computer Science, Computer and Space Engineering, Luleå Tekniska Universitet, Sweden
†VTT Technical Research Centre of Finland Ltd., Espoo, Finland
Email: polina.ovsiannikova@aalto.fi, antti.pakonen@vtt.fi, valeriy.vyatkin@aalto.fi

***Abstract*—We present an approach for suggesting possible fixes to an overall I&C nuclear architecture during its design phase. Despite the I&C architecture, in our case, being represented with an ontology, we do not aim to change the properties of an ontology per se. Instead, we focus on the subset of ABox triples that do not contain terminological elements in either subject, predicate, or object parts. Such a subset we call the *design artifacts*. When the ontology is filled with the design artifacts, the analyst runs the check of non-functional requirements using SPARQL queries. The requirements associated with the queries that returned results do not hold. The goal of the current work is to provide support for the next stage when the analyst has to change the design artifacts so that the queries no longer return results. Our method is based on representing the results of queries as graphs, intersecting them, and finding the minimal changes that prevent the results from being mapped on their (or other) queries.**

***Index Terms*—non-functional requirements, I&C architecture, safety-critical systems, ontology, design repair.**

I. Introduction

Nuclear power plants (NPP) contain safety-critical systems that require a rigorous check before commissioning. The instrumentation and control (I&C) systems play a major role, and the requirements targeted at them can be divided into functional and non-functional. Functional requirements specify what the systems should do, and verification can reveal flaws in the control logic and help eliminate program bugs. However, when analyzing the *overall I&C architecture*, we are more concerned with non-functional requirements. Here, we need to analyze whether design aspects like separation, independence, fault tolerance, safety classification, and diversity are implemented in a way that the principle of *Defense-in-Depth (DiD)* is properly applied.

Architectural flaws are very expensive to fix in later design changes, so it is crucial to ensure that key requirements are achieved, and can be justified [1]. During the design process, the architecture has to be constantly re-evaluated, as new details become available [2]. Even during the operation phase, events like the Fukushima accident may cause the need to reassess acceptance criteria and re-evaluate existing architectures [1]. All the above facts underline the need for proper tool support in designing and analyzing the overall I&C architecture.

In this work, we concentrate on non-functional requirements and continue the investigation started in [3], where the overall I&C architecture is modeled using ontologies. We found ontology to be a suitable means for describing the architecture of an NPP due to its intrinsic power for domain representation as well as the human readability of the resulting model, which is especially useful for domain experts.

Our case study is based on a nuclear power plant I&C architecture with three DiD layers but the method is not restricted to the nuclear domain. The DiD concept calls for the I&C systems on the different layers to be as independent as each other as feasible—to prevent operational occurrences from escalating into postulated accidents, and then further into radioactive release. As the case study illustrates, total independence of the DiD layers is not practically possible, but certain compromises are needed [2]. Our model has 24 I&C systems of varying safety classes, communicating and sharing equipment within and across the DiD layers.

Informally one can view an ontology as a directed graph, where the nodes are the individuals (e.g., systems, details, interfaces, safety classes), interconnected with the edges representing the relations between them. The non-functional requirements are then formulated based on the competency questions listed in [3] and written in the form of SPARQL, an ontology query language.

After the query is executed, we can obtain two possible outcomes, i.e., no results were found, which means that the requirement holds or a set of individuals and relations between them, indicating where exactly the design principle flaws. The next step is to address the issue. However, the information model of a nuclear facility I&C architecture is usually complex, and manual correcting of the design properties requires a deep dive into all the requirements that must be satisfied as well as the existing design nuances. Here, we focus on the subset of ABox triples that do not contain terminological elements in either subject, predicate, or object parts. Such a

This work was supported by the Finnish Research Programme on Nuclear Power Plant Safety 2018-2022 (SAFIR 2022).

subset we call the *design artifacts* or the *design*. Therefore, the main objective of the current work is to introduce the method and the tool for the automatic repair of design issues discovered with a batch of non-functional requirements in the design artifacts representing an overall nuclear I&C architecture.

The tactics for eliminating the results of a batch of queries can be plentiful. However, we noticed that a problematic system or a relation between individuals is often repeated in most of the results. Thus, our method is based on the intersection of the graph representations of the results of the queries and suggesting changes for such intersections. Intersections are parts of several query results simultaneously, so introducing one fix might remove the issue from several subgraphs found, being a minimal correction.

II. Preliminaries

An *ontology* is a vocabulary for a shared domain of discourse [4], defining classes, individuals, and relationships between them. Languages like OWL [5] provide a formal means of expressing semantics in a machine-processable format, allowing automated reasoning over information stored in a *knowledge base*. A knowledge base consists *TBox* statements about general properties of concepts and *ABox* statements about assertions on individual objects [6].

In literature, when it is spoken about the repair of ontologies or knowledge bases, in the majority, authors mean repairment of its terminological part (TBox) and elimination of unwanted consequences [7]. Nevertheless, we considered such papers in our review, even though our goal is to fix the facts about nuclear I&C design (ABox) rather than domain vocabulary, which is used to express it. One of the reasons was that, in our work, we search for a solution that solves several issues simultaneously, which can be true for TBox repair methods as well.

Perhaps the most inspiring works for the current one are [7] and [8]. In [7], root justifications are used for ontologies repair and the authors discuss a root unsatisfiable concept. When such a concept is repaired, all the concepts that are unsatisfiable because of this one are repaired automatically as well. In our work, we search for the intersections of the results of the queries, because if such an intersection is changed, each of the results that include such an intersection is automatically changed too.

In [8], heuristic repair methods introduce minor changes to a knowledge base to fix the unsatisfied constraints, which is one of the basic ideas of our approach as well. Here, the authors created an approach to maintain the consistency of a knowledge base of a configurator while allowing changes in product definitions and user requirements.

Other examples of fixing knowledge base structure modeling errors are [9]–[14]. Work [11] is dedicated to fixing TBox and unsatisfiable concepts, while [12] focuses on integrity constraints. In [9], several tools are introduced for ontology repair and enrichment, one of them is [14] which helps in understanding undesired entailments (for instance, unsatisfiable classes or inconsistencies) and deals with the class learning problem. Another tool presented is RDFUnit [15] which proposes test-driven Linked Data quality assessment.

The incompleteness of a probabilistic knowledge base is considered in [10]. The core of the method here is to uncover missing relationships using goal-directed data mining. In [13] the authors propose an approach to mitigate inconsistencies in the prioritized knowledge base with the data collected from different sources. This paper is relevant to the previous design stage of nuclear I&C architecture, where the information might be scattered across various documents.

There exists literature that provides approaches for working with a form of the data presented in the knowledge bases. For example, in [16], having base facts (ontology), soft and hard rules, the authors infer new facts about the data (or increase certainty of the existing ones). The technique is also useful when acquiring knowledge from various sources of non-equal reliability. Such data problems as inconsistency, comprehensibility, and redundancy are addressed in [17]. [18] presents the approach for linking data between different ontologies that represent the same or similar domains.

Another domain where ontologies and knowledge bases overall are widely used is the maintenance and repairment of implemented mechatronic systems, e.g., [19], discusses the idea of a knowledge management system for the maintenance, repair, and service of the manufacturing system. For the automotive industry, a specific ontology is developed to find the root causes of malfunctions in [20], while the way to infer repairs from the existing ontology is presented in [21].

As we can see, there is literature on ontologies repair and debugging, maintaining data consistency, and gathering data from different domains of various certainty, as well as using ontologies (or designing them) for (future) repairment process of implemented systems. Meanwhile, only a few articles give us an idea of how the objects and the relations between them can be addressed.

III. Ontology-based design artifacts repair

Within the scope of the current paper, we work with the representation of design artifacts in an ontology-based knowledge base. The objective is not to restructure the classes in the ontology, but to restructure (or "fix") the artifacts that are represented by instances. Based on requirements for the overall I&C architecture design, we compose SPARQL queries to search for particular instances in a particular relation, and if a SPARQL query returns a non-empty result, it means that the counterexample for the initial requirement is found.

Our goal is, given the results of the executed SPARQL queries and an ontology, to suggest changes to the instances and relations in the knowledge base, or, more precisely, to the ABox assertions in the RDF graph, so that the SPARQL query patterns no longer produce results (counterexamples).

Before applying any change it is necessary to check if the modified graph will still be structurally valid when the change is introduced and whether it will still satisfy the domain requirements. Thus, if we consider the example from [3], changing the safety class of some subsystem should not lead

to emerging of an information flow from the lower level safety class to the higher. Another example is if some system should have exactly two support systems and it has two, we should not try to delete one of them. Here, we propose several approaches to how the changes can be validated.

A. SPARQL query

SPARQL is a semantic query language developed for retrieving and manipulating data stored in Resource Description Framework (RDF) format. The RDF data model expresses information as graphs consisting of *triples* that are formed by a subject, a predicate, and an object [22]. Generally, a subject and an object can be represented by any of the individuals, classes, properties, and their values, data types that belong to ontology or variables. SPARQL query is executed against such an RDF dataset, which essentially is a collection of graphs.

SPARQL supports several query forms for various data manipulations, where the form of our interest is a SELECT query, more precisely, its WHERE part. Such a query is aimed at retrieving the raw data from the database and may contain conjunction and disjunction of triples together with various constraints and aggregation functions over the variables present in triples. In this work, we omit the aggregation functions as they have no influence on whether the requirement holds, however, we still work with the parts of such queries that contain graph patterns or constraints.

Hereinafter, we distinguish a *query* from a *SPARQL query*. While the meaning of a SPARQL query remains, a *query* is a subset of ABox statements over instances or variables from the WHERE part of a SELECT SPARQL query, represented as a Boolean formula:

$$Q = \bigwedge_{i=1}^{l} p_i(x_i) \bigvee_{j=l+1}^{m} p_j(x_j) \bigwedge_{k=1}^{n} c_k(x_k), (x_i, x_j, x_k \in N \times N),$$

where N is a set of entities included in the query triples, p is a predicate over a pair of instances or variables ($p(x) = T$ is a triple), c is a constraint over a pair of instances or variables, m is a total number of triples and n is a total number of constraints.

For example, borrowing an example from [3]:

```
SELECT ?sysA ?DidLevelA ?sysB
  ?DidLevelB ?interface
WHERE {
  ?sysA :associatedWithDidLevel ?DidLevelA.
  ?sysB :associatedWithDidLevel ?DidLevelB.
  ?interface :interfaceFrom ?sysA.
  ?interface :interfaceTo ?sysB.
FILTER (?DidLevelA != ?DidLevelB)
},
```

where `?<name>` is a variable, `:<predicate>` is a prefix followed by a predicate name, as follows:

```
T1 = :associatedWithDidLevel(?sysA, ?DidLevelA),
T2 = :associatedWithDidLevel(?sysB, ?DidLevelB),
T3 = :interfaceFrom(?interface, ?sysA),
T4 = :interfaceTo(?interface, ?sysB),
C1 = (?DidLevelA != ?DidLevelB),
```

and the whole query is then $Q = \bigwedge_{i=1}^{4} T_i \wedge C_1$.

When a SPARQL query is executed, its result, when obtained, consists of a set of bindings or substitutions for the variables declared in the SPARQL query and selected (in the SELECT part) for the output. Therefore, we can represent each SPARQL query result using the formula of its corresponding query Q by replacing the variables with their bindings. At this stage, it is important that all the variables declared in the WHERE part of the SPARQL query are listed in its SELECT part, as unlisted variables will not appear in the results and therefore will not be substituted.

Assume, for the set of results of a SPARQL query, based on its query Q, we inferred a set of formulas $Q'_1, ..Q'_n$ (the overall result can then be represented as $R = \bigvee_{i=1}^{n} Q'$). If it is possible to replace instances in Q'_i with variables in such a way that all conjuncts or any of the disjuncts of Q'_i will repeat corresponding constitutes of Q, it means that Q'_i *can be mapped* to Q.

B. Change and Fix

To repair the design, we have to introduce such changes that none of Q'_i can be mapped to Q. A *change* is an addition or deletion of a predicate over two instances or an individual to/from the existing triple of Q'_i.

A *fix*, then, is a set of changes for a set of triple $\mathcal{T}'$ from Q'_i that, when applied, eliminate the mapping of each $T' \in \mathcal{T}'$ on their corresponding triples from Q.

For example, a fix "replace individual I with individual I'" in $Q' = P_1(A, I) \wedge P_2(B, I)$ is a set of the following changes:

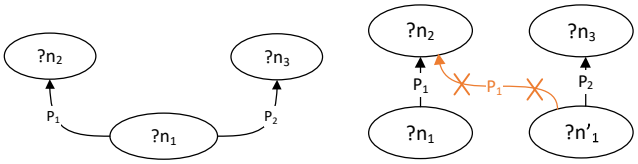
- Delete predicate $P_1(A, I)$
- Delete predicate $P_1(B, I)$
- (if I' is not part of the ontology) Add I'
- Add predicate $P_1(A, I')$
- Add predicate $P_1(B, I')$

C. Single result repair

We start with adjusting the design artifacts for a single result. The idea is to propose such a set of possible fixes for a single result $Q'_i \in Q'_1, ..., Q'_n$ that, when applied, it can no longer be mapped to the Boolean representation Q of the initial SPARQL query.

To infer a set of possible fixes, we have to consider various kinds of triples of the query Q. The triples we address are ABox assertions without TBox elements.

We start with the broadest example. Consider a query consisting of a single triple $Q = p(n_1, n_2)$, where p, n_1, n_2, – variables. Such a query searches for any instances in any relation, with any properties. Assume, in our ontology, we have such instances. Now, there are only three conceptually unique ways to prevent the result to be mapped to the formula of Q, i.e., remove the predicate over two instances, remove some of the instances from the knowledge base, or remove the predicate from the ontology. We will not consider the third option as it assumes changing the high-level structure of the ontology.



(a) Initial query represented as a graph. (b) Introducing new node n'_1 to break the implicit constraints.

Fig. 1. A graph, representing an example of breaking the implicit constraint over a SPARQL query formula. Note, that the new node should not have a property P_1 with value n_2, otherwise the constraint will remain satisfied, but with the new entity.

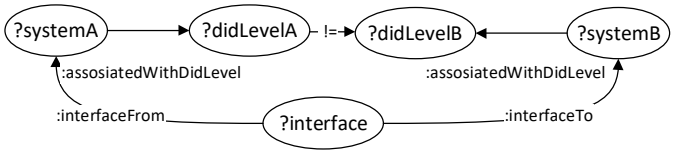


Fig. 2. Representation of an example property from Section III-A as a graph.

If the triple contains constants, e.g., $Q = P(n_1, N_2)$, where P and N_2 are constants, in addition to the previous method, the triplet can be fixed by replacing (1) the constants in the predicates with instances that match the domain of the ones from the initial triple, or (2) predicates themselves with other ones, that have similar ranges and domains. For example, if P is a predicate over two instances of class A, then, we can replace such a predicate only with a predicate whose domains and ranges include class A. Another change that can be introduced to such a triplet is reordering the instances in the predicate if they both are of the same class and turning it into $P(N_2, n_1)$. However, if $n_1 = N_2$, the triple can still be mapped to the initial one.

If there are several triples in Q, we can take a look at the whole formula. Assume, our query is $Q = P_1(n_1, n_2) \wedge P_2(n_1, n_3)$ (this way, for example, we can encode T_3 and T_4 from Section III-A). Here, we can see that there exists an implicit constraint over the whole formula which stands for having the same variable n_1 at the first place of both P_1 and P_2. Thus, we can choose either P_1 or P_2 and replace n_1 with n'_1 from the same domain as n_1 in Q'_i. An example of such a fix is depicted in Fig. 1.

Now, when we search for instances for replacement of the existing ones, the domain, from which we pick a new instance might be huge, especially, if we replace an individual. Therefore, in our suggestions, we provide only M first closest individuals from the same domain. We measure the distance by counting the predicates over individuals that involve an individual to replace and a candidate individual at the same place with the same second individual. We also take into account whether the classes of the two individuals are children of the same parent class in the structure of the ontology. The individuals with the higher scores are then suggested for replacement. The strategy for replacing predicates over two instances is very similar, i.e., we compare parents in the property trees, domains, and ranges of the properties.

To summarize the type of fixes that can be applied to a triple of Q'_i:

- **Singe triple without constants:** remove either of the instances or the predicate over two instances.
- **Single triple with constants:** change any of the constants to the ones from the same domain or, in case the predicate is constant in the triple, replace it with the one whose domains and ranges include domains of the corresponding instances.
- **Multiple triples with implicit constraints over the structure of the formula:** falsify the implicit constraint by introducing new instances into the pattern found and reformulating the existing predicates.

D. Multiple results repair

When we have multiple results that are mapped to various queries, we can follow two approaches: (1) fix each result individually and check if such a fix does not introduce faults to the other requirements, and (2) fix the results in a batch. In this section, we will speak about the second option.

Dealing with non-functional requirements, we consider some of them as critical to be satisfied and some of them as desirable or non-critical. Commonly, total independence of the DiD layers is not feasible, so it is up to the designer to decide which compromises need to be made. Accordingly, an expert decision is needed when determining which requirements should have priority (and the decision can be specific to the plant, or the national regulatory guides, and therefore not universally applicable).

Now, to check a batch of requirements, the user marks the ones that are critical. The task is to satisfy all the critical requirements and a maximum count of non-critical ones. For this, we merge the Boolean formulas of the results of the corresponding queries into one and check, how, by applying the least amount of fixes from Section III-C we can achieve the result. The common formula, in this case, is $Q = \bigwedge Q_i^c \bigvee Q_j^d$, where Q_c is a query for a critical requirement and Q_d is a query for a desirable requirement. At this stage, we can also check if the critical requirements are concurrent by encoding the formulas into SAT and running the solver. If no solutions are found, the requirements are concurrent.

Having a single formula with all the results, some triplets and individuals can be repeated. Therefore, to find a minimal set of fixes, firstly, we generate them for such intersections, as a single fix, in this case, can remove several issues at once.

IV. Implementation

In our implementation, we substitute the variables in the initial queries with the bindings from the SPARQL query results and infer a graph for each of them. When representing the results as a graph, each triple we view as two nodes connected to each other with an edge, which is labeled by a predicate. In the terminology of the ontologies domain, a node can, then, represent an individual, a class, a value of a property, or a datatype.

If there is a UNION operator in the formula, we infer all the graphs that have their bindings and view them as separate

Algorithm 1: Proposed fixes generation algorithm

```
   Data: set Q of initial SPARQL queries, set R of
         substitutions for all queries, O – ontology
   Result: set of possible fixes for intersections F
1  G ← generateGraphsFromResults(Q, R, O);
2  I ← inferIntersections(G, O);
3  F ← ∅;
4  for (N, E, Ω) ∈ I do
5      for n ∈ N do
6          if ¬(n belongs to negated triple) ∧(n is constant)
             then
7              F ← F ∪ replaceConstantNode(n, O);
8          end
9          F ← F ∪ deleteNode(n, O);
10     end
11     for e ∈ E do
12         if e is negated then
13             F ← F ∪ unNegateEdge(e);
14         else
15             F ← F ∪ replaceEdge(e, O);
16             F ← F ∪ removeEdge(e, O);
17         end
18     end
19     if len(N) > 2 then
20         F ← F ∪ rebuildGraph((N, E, Ω), O);
21     end
22 end
23 return F
```

queries with their own fixes. The MINUS operator is processed as a negated predicate. It can contain only variables declared in WHERE and SELECT and constants. Thus, for example, the query from Section III-A, can be represented as a graph as in Fig. 2.

To find the minimal fix, we, first, intersect all the results graphs. An *intersection* is a tuple (N, E, Ω), where N stands for nodes, E denotes edges, and Ω – a set of SPARQL queries results that include all the nodes and all the edges from the intersection.

Then, we check if any of the fixes can be applied to those intersections that belong to most of the queries. When fixes for all intersections are found, we search for the fixes for the rest of the results that do not belong to any intersection, following the logic for the single result repair from Section III-C.

Knowing that the approach would benefit the graphical user interface, we implemented it as a command line tool to demonstrate the work of the algorithm first.

At a high level, the implemented algorithm is shown in Algorithm 1. Here, line 20 represents the generation of the last fix from Section III-C. It happens only when there are more than two nodes in the intersection. In the method `rebuildGraph` we check if there are nodes that are referenced in several triples. Then, for one of the edges, we try to replace such a node with the most similar one from the ontology.

During any replacement procedure, we can occasionally infer a node, which will cause the emergence of a subgraph that also maps to the query. There are several approaches to prevent this. First, we check existing result graphs. If a new subgraph matches any of the existing graphs, then the node is declined. Then, the introduction of a new node in a particular pattern might cause some other subgraph to be mapped to some query (in other words, the non-erroneous subgraph might start having an issue). To prevent this, we can re-verify the ontology with the change and, if any new results are found, they are added to the set of current results, and the process is repeated.

V. Case Study

A. U.S.EPR I&C architecture

Our case study is derived from [3], where the authors modeled the overall I&C architecture of the US version of the European Pressurized Water Reactor (U.S.EPR) as an OWL knowledge base. The U.S. Nuclear Regulatory Commission has published excerpts of the U.S.EPR Final Safety Analysis Report (FSAR) online[1]. In addition to the data on I&C systems, interfaces, functions, and input/output points mentioned in the FSAR, the authors of [3] added identifiers and data properties based on their own invention[2].

The excerpt of the overall I&C architecture and the association of the different I&C systems with DiD layers are shown in Fig. 3. The deliberate design optimizations that violate the total independence of the DiD layers include:

1) The Main Line systems PS and SAS need to share actuators with Preventive Line and Risk Reduction Line systems (as each system cannot have its own Reactor Coolant Pumps, etc.). The prioritization logic (PACS) has an important role in ensuring that the Main Line systems can always handle accidents.
2) Not every system has its own user interface, but systems in different DiD layers can be monitored and controlled through PICS.

[1]https://www.nrc.gov/reactors/new-reactors/design-cert/epr/reports.html
[2]The OWL files are available at: https://doi.org/10.5281/zenodo.5010644

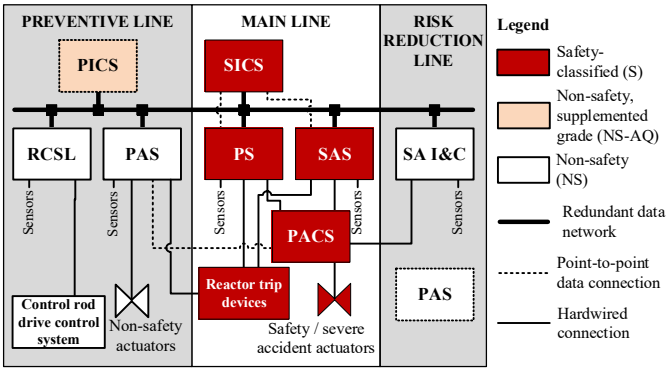


Fig. 3. The overall I&C architecture of the U.S.EPR, and the Defense-in-Depth layers
PAS = Process Automation System
PACS = Priority and Actuator Control System
PICS = Process Information and Control System
PS = Protection System
RCSL = Reactor Control, Surveillance and Limitation System
SA I&C = Severe Accident I&C
SAS = Safety Automation System
SICS = Safety Information and Control System

TABLE I
RESULTS OF EXECUTION OF A SPARQL QUERY BASED ON CQ 3.1

Result Id	System	SC of the system	SS	SC of the SS
1.1	DAS	NS-AQ	12UPS	NS
1.2	SCDS	S	UPS	NS
1.3	PAS	NS-AQ	12UPS	NS
1.4	PAS	NS-AQ	UPS	NS
1.4	PAS	NS-AQ	NUPS	NS

TABLE II
RESULTS OF EXECUTION OF A SPARQL QUERY BASED ON CQ 5.3

Result Id	From system	From system SC	To system	To system SC	Interface
2.1	DAS	NS-AQ	SICS	S	IF01
2.2	RCSL	NS	PAS	NS-AQ	IF21
2.3	TG_I&C	NS	PAS	NS-AQ	IF38
2.4	DAS	NS-AQ	PACS	S	IF06
2.4	PAS	NS-AQ	PACS	S	IF25
2.4	RCSL	NS	PICS	NS-AQ	IF20
2.4	TG_I&C	NS	PICS	NS-AQ	IF39
2.4	DAS	NS-AQ	RTB	S	IF03

Failure tolerance within individual systems is based on redundancy. Main Line systems PS and SAS, for example, have four redundant subsystems in physically separate divisions. The Diverse Actuation System (DAS) is not shown in Fig. 3, as it is actually a separate, hardwired subsystem of PAS. DAS can perform diverse reactor trip functions upon common cause failure of software-based PS and SAS.

The case study has some limitations. First, as many of the details are omitted from the published FSAR, we are missing hundreds of safety functions, and potentially thousands of input/output points, from the knowledge base. The scale, therefore, is not that of a full, real-world model of a nuclear plant's overall I&C architecture. Second, the DiD concept is simpler than the current Western European approach [1], which has more DiD layers. Third, when writing the SPARQL queries, the authors in [3] deliberately specified DiD requirements that the U.S.EPR concept was not intended to fulfill.

B. I&C architecture checking and repair suggestions generation

To demonstrate our approach we chose two SPARQL queries from [3]. The first one is derived from the Competency Question [23] (CQ) 3.1 (hereinafter, SPARQL query based on CQ 3.1 is addressed as SPARQL query 1) which is related to communication independence and checks if there exist interfaces between systems with different safety classes, where the information flows from lower to higher safety class. The second one is related to safety classification, CQ 5.3 (hereinafter, SPARQL query based on CQ 5.3 is addressed as SPARQL query 2) searches for I&C systems, whose support systems are of a lower safety class.

Execution of these SPARQL queries produces 5 and 8 results correspondingly (Tables I and II). We assigned identifiers to the results to address them further in the text.

Both SPARQL queries retrieve information about the safety classification of the systems, therefore, we can notice that safety classes are repeated in the results. We can also spot that issues often tackle almost identical sets of I&C systems.

The workflow we propose starts with the analysis of the intersections between the graphs of the results of the SPARQL queries. Then, the user chooses the intersections of interest and processes the fixes that are proposed for the intersections chosen, i.e., accepts them as is or changes other elements of the design artifacts based on the findings. This process repeats until the result satisfies the user for most of the results of the SPARQL queries. Finally, the fixes are applied to the results that were not addressed during the previous steps. In this case study, we do not consider intersections that belong to a single query result as the principles of applying fixes are the same as for the subgraphs that belong to several results.

Practically, the workflow starts with running the tool, and getting a set of intersections of the results of the queries, the first 10 (out of 28) are provided in Table III. Hereinafter we will use graph-related notions, where individuals are nodes and predicates are edges. Table III shows that all the results except for one contain an I&C or support system with safety class NS-AQ, 9 results contain systems with safety class NS and 8 with both of them. Here, we can already get an idea that the design issues with respect to the checked properties are most likely to happen with systems of these two safety classes.

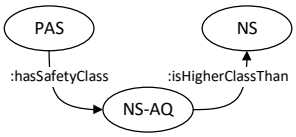
The I&C system that is most involved in the issues is PAS with DAS following it. There is no pair of the systems that belong to more than one result, but changes in subgraphs containing these two systems eliminate 10 issues out of 13.

After the analysis of the intersections is done, we proceed with the fixes suggestion phase and get 42 fixes in total for the first 10 queries. The first two most repeated subgraphs consist of a single node, which is a variable, therefore, the only fix we can generate is to remove those nodes, which means removing safety classes NS-AQ or NS from the systems in the results. Such a fix is not applicable to our system, however, it indirectly suggests that the safety classes of the systems found in the results could be reassigned.

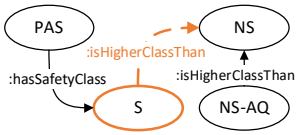
The next intersection (number 3) involves 8 results and consists of two nodes NS and NS-AQ with the edge `:isHigherThan`. The group of suggestions for it, other than deleting nodes or edges, recommends changing the edge from `:isHigherThan` to `:isLowerThan`. Again, changing the edge itself between safety classes might be too radical of a change, but the analyst might get the idea that 8 issues can be fixed if the systems involved in each of these results will exchange their safety classes. Such a fix is particularly interesting because it implies the existence of another kind of a fix. We can also revert the edge if the domain and the range of the corresponding property of the ontology are identical.

Adding PAS to the previous intersection in intersection 4 leads to appearing of the fix to remove PAS altogether to eliminate 5 results, which might signal that its design solution with respect to safety classes is problematic.

With PAS, we also obtained a suggestion telling us to replace `:hasSafetyClass` with

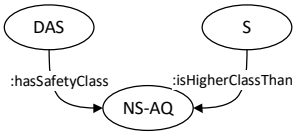


(a) Intersection 6

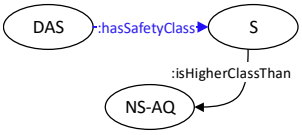


(b) An attempt to rebuild the intersection, which fails, because the same pattern emerges.

Fig. 4. An example of unsuccessful fix of type *rebuild intersection* that was filtered out.



(a) Intersection 9



(b) A successful attempt to rebuild the intersection.

Fig. 5. An example of successful fix of type *rebuild intersection*.

`:associatedWithDidLevel`, which is spurious. Such a fix could be eliminated if the properties related to different aspects of the domain were grouped under different parents.

We did not get any rebuild intersection suggestions as any proposed node instead of NS-AQ emerges the same intersection pattern (Fig. 4). On the other hand, a valid rebuild intersection suggestion is given for the second intersection with DAS (intersection 9). The tool proposes to replace the edge `:hasSafetyClass` from DAS to NS-AQ with the edge from DAS to S, which indeed eliminates 3 issues (Fig. 5).

Overall, we can see that the consistent analysis of the intersections and their possible fixes reduces the time spent on analyzing the design issues as the tool pinpoints the areas in the described domain that should be addressed first.

VI. Discussion

As it was stated in the beginning, we work solely with the design artifacts of nuclear I&C architecture, meaning, the graphs we consider for adjustment, when generating fixes, do not contain terminological elements. Therefore, the more organized the domain vocabulary, the more suitable fixes are generated. We recommend following an object-oriented approach during the design of the ontology structure, i.e., build the hierarchies to group similar properties and classes, define the domains and ranges of the properties precisely, and build the structure logic around the objects and their relations. Generally, the more accurate the ontology, the more meaningful the results.

In our work we did not change the given ontology, however, we noticed, that properties can be grouped in various ways and some do not assume the interchangeability of the children under the same parent in the hierarchies. Due to this, fixes that include replacing the edges with similar ones might not be valid from the design point of view. This can be mitigated either by introducing the variable *replaceable* for each property and checking this variable in the algorithm or by overall analysis of the scenarios where the property is used. The latter demands developing a heuristic algorithm as, in the industrial-sized design ontology, the analysis of each and every edge is time-consuming.

Experiments showed that the intersections included in the maximum amount of results commonly consist of the fewest amount of nodes and edges, thus, receiving fewer but more targeted fixes. This happens when, for example, we query the classes represented by the reduced amount of individuals compared to the other classes, or when the query contains constants. In the latter scenario, these constants will most probably be such an intersection and the first suggestion will be to delete or change it. Such a seemingly little fix, however, might lead to a large set of changes to be introduced in the ontology as it requires deleting all the incoming and outgoing edges to/from the node representing the constant, adding a new node, and restoring the edges using the new node.

When fixing results in batch, the requirements that the queries are based on are marked as either critical or desirable. Ideally, there should be a more flexible scale for assigning priority to different requirements.

VII. Conclusion and Future Work

The method and the tool would benefit from a Graphical User Interface (GUI). For example, the user can have a graph made of intersected results of the queries, with intersections listed next to it together with SPARQL queries and their results included in each of the intersections. Then, the list of fixes can be grouped by the intersections the fixes address. The users, then, might choose which SPARQL queries, queries results, or intersections they wish to consider when generating fixes. There also should be an option to state how many similar nodes to propose in case the replacement fix is found. The graph should be browsable and the changes that the user might try to apply should first be depicted in such a graph with the possibility to manually adjust nodes and edges.

Generally, this method and the tool (especially, when equipped with the GUI) provide a magnifier view into the problem area of the domain artifacts encoded in an ontology. With this approach, the user does not have to browse the whole ontology manually in search of interconnections between individuals and minimal fixes.

Other topics are related to repairing the design artifacts. One considers experimenting with different types of fixes, for example, edge reversing, or enhancing the existing ones, as, for example, a fix of type rebuild intersection to search for more targeted and suitable replacements. The system can learn user preferences and if the fix of deleting the edge or node was never accepted, its priority when suggesting will be lower. Another important aspect of a nuclear design is its cost. In the future, it would be also interesting to experiment with inferring the cost of the changes and sort the fixes accordingly. Finally, there is the task of creating a validation algorithm that will not require the rechecking of the requirements over the whole ontology.

TABLE III
INTERSECTIONS BETWEEN THE QUERIES RESULTS (NODES AND EDGES ARE NAMED AFTER PROPERTIES AND INDIVIDUALS)

Id	Results count	Results intersected	Fixes count	Nodes	Edges
1	12	1.1, 1.3, 1.4, 1.5, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8	1	NS-AQ	-
2	9	1.1, 1.2, 1.3, 1.4, 1.5, 2.2, 2.3, 2.6, 2.7	1	NS	-
3	8	1.1, 1.3, 1.4, 1.5, 2.2, 2.3, 2.6, 2.7	4	NS, NS-AQ	(NS-AQ, isHigherClassThan, NS)
4	6	1.3, 1.4, 1.5, 2.2, 2.3, 2.5	4	PAS, NS-AQ	(PAS, hasSafetyClass, NS-AQ)
5	5	1.2, 2.1, 2.4, 2.5, 2.8	1	S	-
6	5	1.3, 1.4, 1.5, 2.2, 2.3	7	PAS, NS, NS-AQ	(NS-AQ, isHigherClassThan, NS), (PAS, hasSafetyClass, NS-AQ)
7	4	1.1, 2.1, 2.4, 2.8	4	DAS, NS-AQ	(DAS, hasSafetyClass, NS-AQ)
8	4	2.1, 2.4, 2.5, 2.8	4	S, NS-AQ	(S, isHigherClassThan, NS-AQ)
9	3	2.1, 2.4, 2.8	8	S, DAS, NS-AQ	(DAS, hasSafetyClass, NS-AQ), (S, isHigherClassThan, NS-AQ)
10	2	1.1, 1.3	8	12UPS, NS-AQ, NS	(12UPS, hasSafetyClass, NS), (NS-AQ, isHigherClassThan, NS)

REFERENCES

[1] WENRA, "Safety of new NPP designs - study by reactor harmonization working group RHWG," Western European Nuclear Regulators' Association, Tech. Rep., 2013.

[2] IAEA, "Approaches for overall instrumentation and control architectures of nuclear power plants," International Atomic Energy Agency, Nuclear Energy Series NP-T-2.1, 2018. [Online]. Available: http://www-pub.iaea.org/MTCD/Publications/PDF/PUB1821_web.pdf

[3] A. Pakonen and T. Mätäsniemi, "Ontology-based approach for analyzing nuclear overall I&C architectures," in *The 47th Annual Conference of the IEEE Industrial Electronics Society (IECON 2021)*, 2021.

[4] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[5] W3C, "OWL 2 Web Ontology Language Document Overview (second edition)," The World Wide Web Consortium, W3C Recommendation, 2012. [Online]. Available: https://www.w3.org/TR/owl2-overview/

[6] D. Calvanese, G. De Giacomo, and M. Lenzerini, "Ontology of integration and integration of ontologies." *Description Logics*, vol. 49, no. 10-19, p. 30, 2001.

[7] K. Moodley, T. Meyer, and I. J. Varzinczak, "Root justifications for ontology repair," in *Web Reasoning and Rule Systems*, S. Rudolph and C. Gutierrez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 275–280.

[8] G. Schenner and A. Falkner, "Ideas for removing constraint violations with heuristic repair," in *Papers from the Workshop at ECAI 2002 Workshop n 4*, 2002, p. 27.

[9] S. Hellmann, V. Bryl, L. Bühmann, M. Dojchinovski, D. Kontokostas, J. Lehmann, U. Milošević, P. Petrovski, V. Svátek, M. Stanojević, and O. Zamazal, *Knowledge Base Creation, Enrichment and Repair*. Cham: Springer International Publishing, 2014, pp. 45–69.

[10] D. J. Stein III, S. B. Banks, E. Santos Jr, and M. L. Talbert, "Utilizing goal-directed data mining for incompleteness repair in knowledge bases," in *Proceedings of the Eighth Midwest Artificial Intelligence and Cognitive Science Conference*, 1997, pp. 82–85.

[11] T. Scharrenbach, R. Grütter, B. Waldvogel, and A. Bernstein, "Structure preserving tbox repair using defaults," in *CEUR Workshop Proceedings*, vol. 573. University of Zurich, 2010, pp. 384–395.

[12] G. Feuillade, A. Herzig, and C. Rantsoudis, "Knowledge Base Repair: From Active Integrity Constraints to Active TBoxes," in *33rd International Workshop on Description Logics (DL 2020) co-located with KR 2020)*, S. Borgwardt, T. Dresden, Germany, T. Meyer, CAIR, U. of Cape Town, and S. Africa, Eds., Rhodes, Online event, Greece, Sep. 2020, pp. 1–10.

[13] G. Hamdi, A. Telli, and M. N. Omri, "Recursive algorithms to repair prioritized and inconsistent dl-lite knowledge base," 2019.

[14] J. Lehmann and L. Bühmann, "Ore - a tool for repairing and enriching knowledge bases," in *The Semantic Web – ISWC 2010*, P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–193.

[15] D. Kontokostas, M. Brümmer, S. Hellmann, J. Lehmann, and L. Ioannidis, "Nlp data cleansing based on linguistic ontology constraints," in *European Semantic Web Conference*. Springer, 2014, pp. 224–239.

[16] T. Meiser, M. Dylla, and M. Theobald, "Interactive reasoning in uncertain rdf knowledge bases," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 2557–2560.

[17] C. Fürber and M. Hepp, "Using sparql and spin for data quality management on the semantic web," in *Business Information Systems*, W. Abramowicz and R. Tolksdorf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–46.

[18] A.-C. Ngonga Ngomo, M. A. Sherif, and K. Lyko, "Unsupervised link discovery through knowledge base repair," in *The Semantic Web: Trends and Challenges*, V. Presutti, C. d'Amato, F. Gandon, M. d'Aquin, S. Staab, and A. Tordai, Eds. Cham: Springer International Publishing, 2014, pp. 380–394.

[19] S. Wan, J. Gao, D. Li, and R. Evans, "Knowledge management for maintenance, repair and service of manufacturing system," in *International Conference on Manufacturing Research*. Southampton Solent University, 2014.

[20] M. Sanseverino and F. Cascio, "Model-based diagnosis for automotive repair," *IEEE Expert*, vol. 12, no. 6, pp. 33–37, 1997.

[21] X. Fang and K. Fang, "Knowledge-based auto repair diagnosis system and application," in *Proceedings of the 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013)*. Atlantis Press, 2013/07, pp. 33–36.

[22] W3C, "SPARQL 1.1 Query Language," The World Wide Web Consortium, W3C Recommendation, 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[23] D. Wiśniewski, J. Potoniec, A. Ławrynowicz, and C. M. Keet, "Analysis of ontology competency questions and their formalizations in SPARQL-OWL," *Journal of Web Semantics*, vol. 59, p. 100534, 2019.