Received 25 April 2024; accepted 2 June 2024. Date of publication 12 June 2024;
date of current version 3 July 2024. The review of this article was arranged by Associate Editor Paulo Leito.

Digital Object Identifier 10.1109/OJIES.2024.3413568

Formal Verification of Nonfunctional Requirements of Overall Instrumentation and Control Architectures

POLINA OVSIANNIKOVA [1] (Graduate Student Member, IEEE), ANTTI PAKONEN [2], DMITRY MUROMSKY [4], MAKSIM KOBZEV [4], VIKTOR DUBININ [4], AND VALERIY VYATKIN [1,3] (Fellow, IEEE)

[1]Department of Electrical Engineering and Automation, Aalto University, 02150 Espoo, Finland
[2]VTT Technical Research Centre of Finland Ltd., 02044 Espoo, Finland
[3]Department of Computer Science, Electrical and Space Engineering, Luleå Tekniska Universitet, 97187 Luleå, Sweden
[4]Independent Researcher 440000, Russia

CORRESPONDING AUTHOR: POLINA OVSIANNIKOVA (e-mail: polina.ovsiannikova@aalto.fi)

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

ABSTRACT The design of safety-critical cyber–physical systems requires a rigorous check of their operation logic, as well as an analysis of their overall instrumentation and control (I&C) architectures. In this article, we focus on the latter and use formal verification methods to reason about the correctness of an I&C architecture represented with an ontology, using the example of a nuclear power plant design. A safe nuclear power plant must comply with the defense-in-depth principle, which introduces constraints on the physical and functional components of the I&C systems it consists of. This work presents a method for designing nonfunctional requirements using function block diagrams, its definition using logical programming, and demonstrates its implementation in a graphical tool, FBQL. The tool takes as input an ontology representing the I&C architecture to be checked and allows visual design of complex nonfunctional requirements as well as explanation of the results of the checks.

INDEX TERMS Function block diagrams (FBDs), instrumentation and control (I&C) architecture, logical programming, nonfunctional requirements, ontology, safety-critical systems.

I. INTRODUCTION

A safe design of a critical cyber–physical system assumes both correct logic of operation and conformity of the operational components to their nonfunctional requirements. In safety-critical technological areas, the latter also means fulfilling the defense-in-depth (DiD) principle [1], [2], [3]. According to the DiD principle, a successful instrumentation and control (I&C) architecture design has multiple redundant and diverse structures with components of different DiD levels as physically and functionally independent of each other as feasible [2], [4]. Thus, for example, in the nuclear power industry, to prevent failures from propagating into accidents and accidents propagating to radioactive releases, a nuclear facility shall have successive protective barriers.

To assess requirements based on the DiD principle, one needs to dive into at least two layers of information about the various aspects of the design, i.e., physical and functional. The physical layer includes the equipment used, its providers, technologies, and safety classes. This layer stores information about the structure of the architecture and physical interfaces between its components along with their physical locations. The functional layer determines the information flow, that is, the functions that the equipment implements, the required actions, and the measurements. Now, this variety of knowledge of the architecture and the fact that it is often scattered across different sources [5] form obstacles to the manual assessment of compliance with the DiD principle.

Pakonen and Mätäsniemi [5] gave a start to the formal verification of the nonfunctional requirements of nuclear I&C overall architecture. Here, it is represented with an OWL ontology [6] (Web Ontology Language), and the checks are performed using SPARQL (SPARQL Protocol and RDF Query Language) [7] queries. The ontology combines both layers of information; therefore, it is possible to check the

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

properties related to the physical and functional components. Although objects that satisfy a particular query can easily be found, the shortcoming of this approach lies in reasoning about more complex properties where the overall result is influenced by the results of the assessment of multiple criteria. Moreover, in this case, the result is not a logical variable but a number whose value depends on the individual requirements satisfied and the weight they have in the result.

To address this problem, we propose a method for developing and verifying complex requirements systems (or design principles) using function block diagrams (FBDs)[1] for the description of overall I&C architectures stored in a knowledge base. Having a design principle implemented in the form of FBD, we can not only relieve the analyst of manual labor of gathering the data across the scattered documents and its assessing, but we also provide reasoning mechanisms about the result of the assessment, which shed light on the result received.

FBDs are a common programming language in safety-classified nuclear automation systems (e.g., TELEPERM XS, Spinline, AC160). Although FBD is not the only language used for this purpose, using it to define nonfunctional requirements should reduce the barrier for control engineers to adopt our method.

We present our method formally using Prolog [9]—a declarative programming language with native reasoning capabilities and high expressive power. Here, information about our I&C architecture is stored in a Prolog knowledge base, and individual queries together with dependencies between them are formulated in the form of Horn clauses. Prolog has a lot of similarities with the declarative ontology languages OWL, SWRL, and SPARQL, but is more appropriate to describe concepts due to its elegant syntax and clear semantics.

Then, we present a graphical tool created using an object-oriented programming language. This tool allows for a visual building of requirement FBDs, their execution, and an explanation of the results. In the backend of the tool, an I&C architecture is represented as an ontology, and individual queries are formulated using the graphical SPARQL language. Such formulations are encapsulated in a special kind of function blocks (FBs)—SPARQL FB, which can be added to an FBD of a design principle.

The design of cyber–physical systems critical to safety is a complex process that involves multiple entities. The formulation of complex design principles with which such systems must comply may involve different actors, including governmental organs, as, for example, in the nuclear domain. After the requirements are formulated, various actors participate in checking the system against these requirements. They include domain specialists who can check whether I&C architecture achieves the formulated goals, engineers who specialize in particular parts of the architectures and design them using diverse methods, and analysts (might also be hired externally) who possess skills in the application of different techniques to I&C architecture checking, however, may not have extensive domain knowledge or knowledge about the particular system being verified. The assessment process is further complicated by confidentiality agreements; for example, engineers may know only about the subsystems they develop, and external specialists might not be allowed to check anything else than an abstract design.

We claim that the method and the tool presented in this article assist all the actors that have to check and/or design complex nonfunctional requirements. Thus, in the following, we call our users engineers, analysts, or domain specialists.

To demonstrate the method and its implementation, we take both the properties and the running example from Pakonen and Mätäsniemi's [5] work because it makes the demonstration more straightforward and practice-oriented.

The rest of the article is organized as follows. In Section II, we give a more detailed introduction to the DiD principle. In Section III, we discuss the applicability of FBDs for the formulation of complex design principles and introduce an attribute-criteria (AC) design pattern for this purpose. Section IV describes a system and a design principle used for experimental evaluation. Section V gives a definition of the method using the logical programming language, Prolog. The graphical tool for formulation design principles using the proposed design pattern is presented in Section VI. Section VII describes the current state of the art, while Section VIII discusses the applicability scope of the method. Finally, Section IX concludes this article.

II. DID PRINCIPLE

The nonfunctional requirements we consider in this article are the design principles that are derived from the DiD principle. One of the domains where the DiD principle is frequently applied is the nuclear industry. Therefore, to make the introduction of the principle more descriptive, we will heavily use examples related to the nuclear power plant safety context.

The DiD principle is a strategy to prevent critical failure situations, that is, damage to the reactor or radioactive contamination, by using successive layers of safety mechanisms. Therefore, failures in one layer of defense must not lead to consecutive failures in another layer. To achieve this, ideally, the design should include full physical and functional separation between the different DiD layers, together with multiple redundancies of the protection systems belonging to a single layer. However, in reality, this is impractical; therefore, designs involve acceptable compromises in dependencies [10].

I&C systems responsible for the safety functions must be failure tolerant. The typical requirement is *single failure tolerance*, which means that the system must be able to perform its function even if any single component designed for the function fails.

On the overall I&C architecture level, we also have to consider the propagation of failures from one system to

[1] FBD is one of the graphical programming languages officially supported by IEC 61131-3 [8]. However, the standard itself is not used much in the nuclear domain, therefore, it makes sense to consider a more general version of this language, as described in Section III.

another, and the possibility of the loss of an entire I&C system altogether.

- A *consequential failure* refers to "a failure caused by a failure of another system, component, or structure or by an internal or external event in the facility"[2].
- A *common cause failure*—a failure of several structures, systems, or components due to a common event or cause. Typically, a failure-tolerant system consists of several redundant (but often identical) subsystems, all susceptible to similar failure modes.

Our nonfunctional requirements are heavily motivated by the DiD principle and maintenance of the safe operation of the plant. We start our experiments with the group of requirements from IAEA [10], which focus on the independence of the I&C systems from different layers and therefore are related to the separation (physical, electrical, functional, independence of communication and supply systems), diversity, and redundancy design aspects. In general, when verifying these particular properties, we seek a violation or a counterexample; therefore, competency queries are negated versions of the requirements. The answers are then entities with wrong relations or with wrong properties, where an *entity* is either an I&C system, its component, an interface, or a function.

However, verification of design principles, such as diversity, cannot be restricted to competency questions, as it requires the assessment of multiple criteria to derive the answer. For this purpose, NRC [11] states a score-based approach, where the total score of the design principle, diversity, is computed based on subscores of its attributes, which values, in turn, depend on the particular criteria fulfilled by the architecture. Therefore, we introduce another category of requirements, which are expressed as systems of interdependent rules, reasoning about which we have to produce results both in the form of architectural entities and unsatisfied rules.

III. MODELING THE DESIGN PRINCIPLES USING FBDS

In this section, we describe the concept and design patterns to model the design principles using FBDs. These requirement formulation principles can then be implemented using various approaches and programming languages, and one of them we show further in the article.

A. FBD

FBDs in IEC 61131-3 are intended to implement PLC programs. Our main goal, in turn, is to formulate descriptive nonfunctional requirements. Thus, for example, we do not require the possibility for our FBs to be represented using structured text, our FBs might have internal variables, but it depends on a particular implementation whether they will. We also do not assume any global scope that can be freely edited by the user (except maybe knowledge base connection settings) and do not impose any constraints on the graphical representation of FBs, as this also depends on the goals of a particular implementation. Thus, we view FBDs in a broader scope than they are defined within the IEC 61131-3 standard, and, in this section, we provide our definitions for both, FBs and FBDs themselves.

Our goal is to assist in the formulation and checking of complex nonfunctional requirements which, informally, are questions of the type "How is the system designed?". This requires a single "scan" of a knowledge base, and thus a single evaluation of an FBD. Therefore, in this version of the method, we do not need FBs with memory or internal states based on internally defined variables.

An *FB* is a graphical notion of a function that transforms a tuple of its inputs into a tuple of its outputs according to its predefined algorithm. Thus, each FB has input and output interfaces that correspond to its sets of input and output variables. The outputs of one FB *A* can be connected to the inputs of another FB *B*, which means that the input variables of *B* are assigned with the output values of *A*. Outputs can pass their values to multiple inputs, whereas inputs can have only one incoming connection. The input interface variables have constant values unless they are connected.

Two types of FBs exist, i.e., *basic* and *composite*. The function and interfaces of the *basic FBs* cannot be changed graphically; such blocks are atomic.

The functions of *composite blocks* in turn, are determined by nets of interconnected FBs (basic or composite) they encapsulate and must include at least one FB. The input variables of a composite FB can be connected to the inputs of its internal FBs (thus, the input interface variables of a composite FB act as input variables on the nested level to which their FB belongs and as output variables for the internal net of their composite FB). The outputs of a composite FB must be connected to the outputs of internal FBs or to the inputs of the composite FB to which they belong. The output interface variables of a composite FB, then, opposite to input interface variables, act as output variables on the nesting level to which their FB belongs and as input variables in the internal net of their composite FB.

An *FBD*, then, is a composite FB of the highest hierarchy level (i.e., not nested in any other FB) that has a finite number of nested FBs in every included composite FB.

The input and output variables of every nested FB (also called component FB) in an FBD form its set of variables. An *assignment* is a particular value of a particular variable that belongs to an FBD, and an *assignment of a basic FB* is a set of assignments of input and output variables of a basic FB. An *assignment of composite FB or an FBD* is a set of assignments for all their corresponding variables, including the variables of their nested FBs.

Definition 3.1 (Correct assignment of a basic FB): Having a basic FB B with a set of input variables $u_1, \ldots u_k$, a set of output variables $u_{k+1}, \ldots u_m$, and a transformation function f, the *assignment of B is correct* if the formula $v(u_1), \ldots, v(u_k) = f(v(u_{k+1}), \ldots, v(u_m))$, where $v(u_i), i \in [1, m]$ is a value of variable u_i, is valid.

Definition 3.2 (Correct assignment of a connection): Having a connection between two variables u_1 and u_2 of two different FBs of any type, basic or composite, *an assignment*

of a connection is correct if the formula $v(u_1) = v(u_2)$ is valid.

Definition 3.3 (Correct assignment of a composite FB of an FBD): The function of a composite FB with a finite number of nesting levels (as only composite FBs with a finite number of nesting levels are allowed in an FBD) is determined by its net of interconnected FBs. Since the number of nesting levels is finite, such a composite FB can be flattened into a net of basic FBs, which can be represented as a tuple (B, C) of basic FBs and connections between them. Then, *an assignment of a composite FB of an FBD is correct* if the assignments of all FBs from B and all connections from C are correct.

Based on the definitions provided in this section, *a correct assignment of an FBD* is defined similarly to a correct assignment of a composite FB of an FBD.

The FBD execution semantics is defined by each particular implementation method individually. Each execution defined by the execution semantics of each implementation method must always produce a correct assignment of an FBD. In this article, we consider method definition using logical programming (and thus its implementation in Prolog) and its implementation using imperative object-oriented language C++. In the first case, the FBD execution semantics corresponds directly to the Prolog execution semantics (outlined in Section V-E). In the second case, we implement the execution semantics using the same programming language, C++, and describe it in detail in Section VI-D.

B. BASIC BLOCK TYPES

Different implementations of the FBDs may define different sets of basic FBs or even provide means for their customization or manual creation. In the end, such sets are determined by the design principles modeled. Overall, implementations are not obliged to but are expected to have a set of basic FBs representing simple arithmetic (i.e., plus, minus, multiplication, modulo) and logical (i.e., conjunction, disjunction, negation) and comparison (i.e., greater, equal, less) functions.

A necessary basic FB type that we define for a design principle FBD is a *REQUEST basic FB*, which makes a request to the knowledge base where information about the I&C architecture is stored. Such a block is generic, as various knowledge bases and FBD implementations require different connection means and request syntax; however, generally, such blocks are supposed to encode individual queries, i.e., request the details that are necessary to derive whether an individual specification is true. Thus, the input interface of a REQUEST basic FB is empty, as the knowledge base is external to an FB, while its output interface contains at least one output. In the situation where multiple knowledge bases are to be addressed, REQUEST FB might accept the specific knowledge base connection rules as input; however, we claim that it is less error-prone to describe the connection to the external knowledge base externally, for example, in the settings of the implementation.

C. AC DESIGN PATTERN

In this article, we propose modeling nonfunctional requirements using the AC design pattern.

In the context of industrial safety-critical cyber–physical systems, due to their complexity, the fact of whether the system complies with a design principle is often expressed not with a logical variable but with a number, an estimate to what degree the principle is followed, we also call it a *score* of the system in relation to the design principle being checked. In addition, as was mentioned earlier, a maximum value is not always desirable, as in some cases this will increase the costs and decrease the maintainability of such a system.

The basic brick of our design pattern is a QUERY composite FB [see Fig. 1(a)]. Such blocks formulate reverse facts about the system using basic REQUEST FBs to check whether some requirements are satisfied. For example, if the requirement is for subsystems of some system not to have interfaces with each other, the REQUEST FB will search for such subsystems that have interfaces with each other and are parts of one system. Therefore, QUERY FB checks whether there is a proof that the query has a counterexample. The result of the REQUEST FB is then cast to a Boolean value and then inverted. In the text, the requirements that are represented with QUERY composite FBs are called *individual queries*.

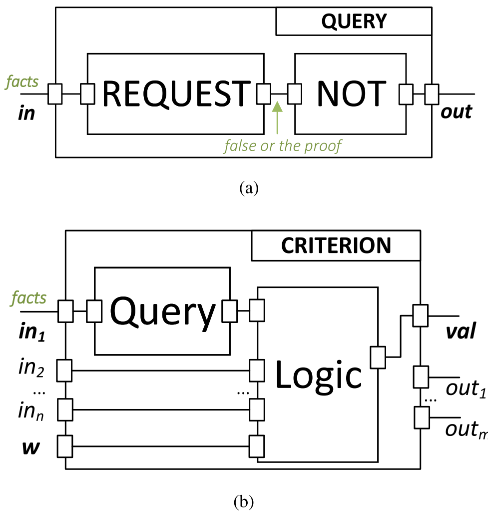
In general, a design principle is formulated with a set of *rules* that the system must follow to obtain the maximum score and the weights of these rules that determine how critical it is to follow the rule with respect to the design pattern analyzed. The rules, in turn, can directly correspond to individual queries and seek particular design faults, or they can be represented as predicates over several individual queries. A rule can only be followed or not; therefore, the result of the evaluation of some rule in a system is a Boolean value.

An *attribute* of a design principle is a subarea evaluated separately from the principle, which is represented by a set of weighted rules or *criteria* [see Fig. 1(b)]. In other words, an attribute is a group of rules united by their meaning and weighted according to their contribution to a particular component of a design principle [see Fig. 1(c)].

The score of a design principle is then defined as a function of weighted attributes since different attributes also contribute differently to the overall estimation.

Fig. 2 shows an example of the basic usage of the components of the defined design pattern. The design pattern can be used as presented or hierarchically, meaning that the design principle can be subdivided into design subprinciples that can be evaluated separately, and their score can also be weighted and aggregated into the score of a main design principle.

This design pattern was partially motivated by the way the diversity principle is evaluated in NRC [11]. CR-7007 [11] describes the diversity strategy evaluation for nuclear I&C. Here, in the beginning, sets of criteria are evaluated based on facts about the I&C architecture. These criteria partially correspond to the negated competence questions (CQs) from Pakonen and Mätäsniemi's [5] work. At the next level,



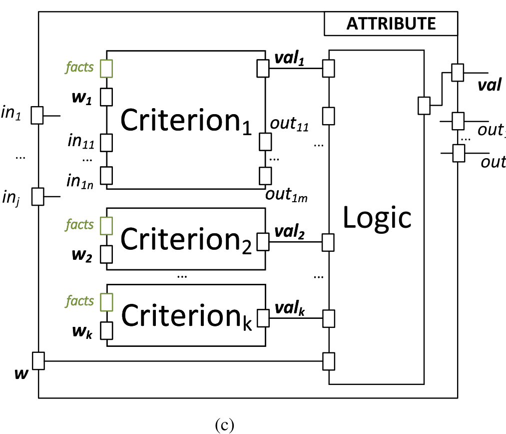


FIGURE 1. FBs `QUERY`, `CRITERION`, and `ATTRIBUTE`—the constituents of the AC design pattern. Rectangles on the sides of the FBs correspond to the input and output variables of the blocks, light green text and arrow are used to highlight the data types, the right upper corners of the composite FBs contain their names, while names of the internal blocks are written in the middle of the corresponding rectangles. The variables in bold and their designated connections are the required variables for the design pattern, and the remaining variables and connections are optional. The diagram should be read from left to right. (a) Query function block. To understand whether the requirement holds, we negate the meaning-reverse requirement and get the proof of the presence or the absence of a counterexample. (b) Criterion function block utilises the requirement and additional Boolean logic to derive the result. The `Logic` function block can be replaced with interconnected Boolean operators or omitted if the criterion corresponds to its competence query. (c) Attribute function block utilises the criteria results and additional Boolean logic to derive the result. The `Logic` function block can be replaced with interconnected Boolean operators.

attributes are assigned values depending on which criteria are satisfied and their weights. In the end, the attributes values are used to calculate the total score that can be used to compare the efficacy of the selected diversity strategy with the strategies used in other nuclear facilities.

D. APPLICATION OF EXPLANATION TECHNIQUES

In this article, we define an FBD and FBs similarly to Ovsiannikova et al.'s [12] work, except that here we do not predefine a set of basic blocks and do not assume support for the DELAY basic blocks (even though we still can insert such blocks in an FBD, and, for example, program the loops to execute iteratively some parts of an FBD, we chose to omit it as rare nonfunctional properties require it). We have also introduced a new type of basic block, a query block. Our design patterns use basic and complex blocks, the native elements of an FBD. This gives us the opportunity to explain the values of the FBD variables (input, output, as well as input and output variables of its internal FBs) after its execution is complete using the explanation techniques from Ovsiannikova et al.'s [12] work, with two adjustments. First, our counterexample now consists of only one step—a single execution of an FBD. Second, the explanation algorithm for the query block should be implemented. In general, its output is explained by its input from the knowledge base; however, such output is external to an FBD and depends on the concrete implementation whether the data can be accessed. Since, in explanation, we can use only assignments internal to FBD and we have only one counterexample step, we consider the outputs of query blocks as constants and do not explain them further.

Now, the definition of an explanation target is similar to that in Ovsiannikova et al.'s [12] work, considering the two adjustments—it is a value of a particular variable of an FBD. The explanation is also intuitively defined as a set of assignments that precede the explanation target in the order of logical inference and unambiguously define the value of the explanation target.

The example of an explanation for a nonfunctional requirement (a part of a diversity principle of NRC [11]) formulated using the design pattern introduced in this section is shown in Fig. 3. Here, we can see that Criterion 1.3 got a nonnegative valuation due to the main requirement of Criterion 1.1 being satisfied.

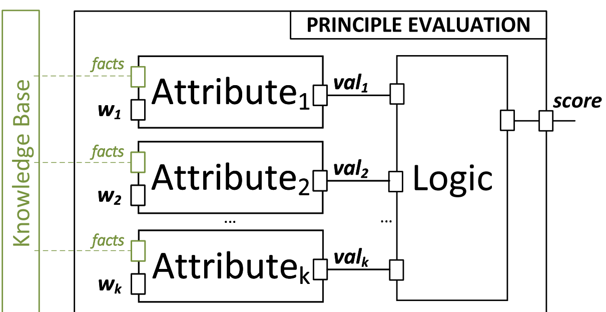


FIGURE 2. Basic application of an AC design pattern for a principle evaluation. The logic can be represented by one composite FB or a net of several FBs. The diagram can also be enriched with additional inputs, outputs, and constants. The connection to the knowledge base is shown with green dashed lines as the knowledge base is not an FB (internal REQUEST FBs encode the access to it).

IV. CASE STUDY SYSTEM

Similarly to Pakonen and Mätäsniemi's [5] work, as a running example, we used a semifictional model of a nuclear reactor, U.S. EPR (European Pressurised Water Reactor), whose

FIGURE 3. **Application of an AC design pattern to a diversity design principle from NRC [11] (the logic for the first attribute is shown). The explanation paths for the output of** `Criterion 1.3` **are shown in bold blue. As we can see, although** `Q1.3` **is the main request for this criterion, it obtained a nonnegative result due to** `Q1.1`**.**

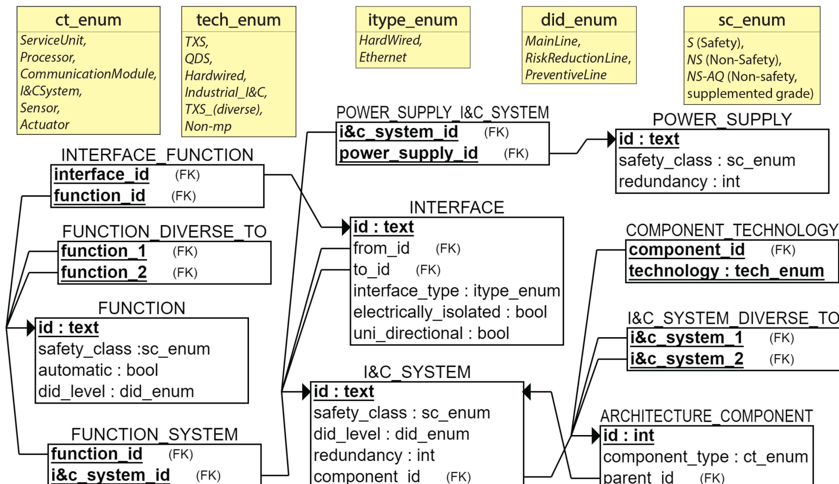


FIGURE 4. **Relational scheme of the information about U.S. EPR design entities, used as a running example. The yellow boxes on the top show the domains of the variables of the types in bold. The white rectangles correspond to entities with their properties listed on the side in the form** `<property_name> : <property_type>` **unless the property refers to another one. To save space, we omit completeness and consistency data constraints.**

safety I&C architecture is described in NP. U.S. Areva [13]. The publicly available documentation lists 24 I&C systems arranged into three lines (or layers) of defense: preventive, main, and risk reduction. It also proposes a methodology for the evaluation of I&C design according to the DiD principle, which addresses the Nuclear Regulatory Commission (NRC) guidance. The design was never commissioned, and publicly available documents omit the information required to verify some particular properties [5].

In the current work, in addition to an ontological representation of overall I&C architecture from Pakonen and Mätäsniemi's [5] work, it was convenient to also translate it into a relational model to define the approach using logical programming. This enabled the automatic generation of a set of Prolog facts about the architecture. The translation of the structure of the ontology into Prolog, that is, class and property hierarchies, was done manually in the form of the corresponding predicates. Although a tool can be developed to automate this process, we do not focus on it in the current work, as this was only required for a more precise definition of the method with Prolog. Fig. 4 shows the types of entities, their properties, and relations that we used to formulate the nonfunctional requirements. From this scheme, we can see that the architecture consists of *components* of different types and is implemented using various *technologies*. Each component can be part of an *I&C system*, including I&C systems themselves. Every I&C system, in turn, might have more than one *function* (which can be different from each other) and several *power supplies*. Some I&C systems are connected to wired or data *interfaces* to perform their functions.

As a source of a design principle, we use NRC [11], which provides a detailed description of the evaluation of the system against the diversity principle, one of the constituents of DiD.

V. METHOD DEFINITION USING LOGICAL PROGRAMMING

Logic programming (LP) is a formalism for programming and knowledge representation introduced in 1974 by R. Kowalski. In logical programming languages, the program is expressed using facts and rules, while the computation is done through logical inferences. As a programming paradigm, it was originally implemented in Prolog [14]. The areas of application of Prolog include symbolic computations to understand natural languages (for which it was originally designed), text parsing, and areas of artificial intelligence. In Prolog, LP is used both for programming and for knowledge representation.

We chose logical programming as our method definition language for the following reasons. First, one of its features is backtracking, that is, it is possible to receive all the alternations of the proof (in case their number is finite). This means that we can find all the possible violations of the properties formulated. Next, it fits problems that involve entities (not from a data structure point of view, but ideas that can be expressed using terms) and the relations between them [15]. The program text describes the task domain and the computation is driven by the Prolog internal algorithms, which makes the creation of domain-specific languages (DSL) straightforward. Then, the declarative style of query formulation assists in the search for deviations in I&C architecture, as we often know exactly *what* the deviation is in terms of the DSL of our knowledge base, but it may not be obvious *how* to find it. Another advantage is its ability not only to deduce but also to compute. Thus, we can formulate requirements that require additional computation directly in the knowledge base and trace them using goal queries, thus directly integrating the architectural properties into its model.

Prolog also implements a closed-world assumption, which means that the lack of information implies its falsity. This becomes especially important when the architecture design is unfinished and it is critical to discover the missing parts.

In this work, we consider pure Prolog, where the program is composed of Horn Clauses, a subset of first-order logic. A Horn Clause is a disjunction, containing at most one nonnegative literal, and in Prolog, such clauses are written in the form of implications. By the presence of positive and negative literals, Horn clauses can be divided into three types. A *definite clause* consists of both positive and negative literals and represents *rule* in a program. The consequent of a rule is a *head* and an implicant is a *body*. A *unit clause* is a clause that contains a single positive literal and no negative ones; this clause is called a *fact*. A *definite goal* is a clause with an empty consequent [16], to check whether some conclusions can be logically deduced from the program, we formulate *queries* with this type of clause. We will also talk about *compound terms*, which are functors with a finite set of arguments listed in parentheses. The number of arguments is called the term's arity and is used to denote a term, e.g., $\texttt{f}(a_1, \ldots, a_n)$ is denoted as `f/n`. The variables in Prolog start with a letter in uppercase and, essentially, are placeholders for terms.

A. SYSTEM ENCODING

The first step in the implementation of the AC design pattern is the appropriate encoding of the system. According to a database interpretation of the logic [17], [18], a logic program consisting solely of ground unit clauses (or facts with no variables) is considered as a database [16]. Therefore, the conversion between our relation scheme of knowledge about the I&C architecture from Fig. 4 to Prolog is straightforward.

In Fig. 4 we can distinguish two types of tables, the first of which corresponds to entities described in the architecture, i.e., interface, I&C system, function, and power supply, hereafter, *entities*, and the second of which is created to model many-to-many relations. If we organize knowledge into a database according to Fig. 4 and focus on entities, we can consider the rows of the represented tables as objects of types corresponding to the names of the tables with properties specified by their columns. We start by encoding each entity with compound terms, where the functor represents the name of the table, and its argument is the identifier of the object. Thus, we encode the knowledge "there exists an object of type t with identifier u" with `t/1`, i.e., `t(u)`.

Next, each property is encoded with the compound term `t_p/2`, e.g., `t_p(u, v).`, where `p` is the name of the column, `u` is the unique identifier of the object of type `t` and `v` is the column value.

Our many-to-many relations are motivated by the fact that one entity might be in relation with multiple entities of the same or another type. One way to formulate such a relation can be with the use of *lists* in Prolog; e.g., if the entity of type t_1 and the identifier u is in some relation r with a set of entities e of type t_2, we can add to the program the following fact $\texttt{t}_1\texttt{_r_t}_2\texttt{(u, e)}$. Now, if we are to find two entities in some relation, in order to disprove a requirement, we will have to add the code for processing each such list to the Prolog program. This will also affect the formulation of the property as it will require complementing it with imperative constructions. Therefore, we use another approach and encode such relations similarly to how they are encoded in the database. Assuming that the entity of type t_1 and identifier u is in some relation r with a set of entities E of type t_2, we add the following set of facts $\forall e \in E : \texttt{t}_1\texttt{_r_t}_2\texttt{(u, e}_\texttt{u}\texttt{)}.$, where e_u is the unique identifier of the entity e.

Translating the relational representation of the data to the Prolog facts, we cannot assume that the data are complete and consistent, as the architecture might still be under development with missing information. For the same reason, we allow the columns to be empty (except for unique identifiers); thus, even incomplete information can be analyzed, avoiding false negative answers of the reasoner due to the properties that have not been modeled yet. On the other hand, all missing information can also be found by checking specific queries, for example, $t_1(U) \wedge \neg t_1_p_1(U, _)$. checks whether there are entities U of type t_1 that lack the description of their property p_1. Thus, the closed-world assumption of Prolog is also useful during the consistency check of the model.

The last step in our conversion would be to add the missing hierarchies of classes, data properties, and relations of the ontology. This is useful when we want to generalize several types of relations and make queries that include the parent of a hierarchy to search for all the children, as in the following part of the SPARQL `SELECT` query:

```
  ?supportRel rdfs:subPropertyOf* :sup-
ports.
  ?equipment ?supportRel ?systemA.
```

In Prolog, for such relations, we add recursive rules. For example, for a class hierarchy, with root a and children C, the following rule is created: $a(U) \leftarrow \vee_{c \in C} c(U)$, where U is a Prolog variable for the unique identifier of an entity (note that in the Prolog syntax, an implication is written reverse, therefore "←," and is replaced with sign ": -"). For example, this can be used when we know that some system is a power supply or a heating, ventilation, and air conditioning system and want to conclude that it is a support system. SPARQL property paths are modeled with recursion in Prolog and used when we want to check if there is a path from one entity to another following the specific property. For example, if a device is a component (including subcomponents) of a system.

B. INDIVIDUAL QUERIES ENCODING

The first type of answer that we look for is whether there exist entities with particular properties in a particular relation that violate the property. A Prolog query (or a goal) is an implication without a consequent. In our domain, it is a description of a violation state using compound terms from the knowledge base and Prolog constructions, including operators $\wedge$, denoted as a comma, $\vee$ denoted as a semicolon, negation as failure `not/1`, with relations of equality (=) and nonequality (\=) between variables. For example, SPARQL query 2.1 (1) from Pakonen and Mätäsniemi's [5] work

```
 SELECT ?fromSystem ?toSystem ?interface
 WHERE {
  ?interface :interfaceFrom ?fromSystem.
  ?interface :interfaceTo ?toSystem.
  ?fromSystem :hasSafetyClass USEPR:S.
  MINUS {?toSystem :hasSafety-
Class USEPR:S}
  MINUS
  {?interface :electricallyIso-
lated true}
 }
```

Can be defined with Prolog as follows:

```
  interface(I, S1, S2, false),
  component_safety_class(S1, SC1),
  component_safety_class(S2, SC2),
  (SC1 = 'S'; SC2 = 'S'), SC1 \= SC2.
```

Here, the line `interface(I, S1, S2, false)` means that we are looking for such interfaces between two systems that are not electrically isolated.

Such a Prolog goal corresponds to REQUEST FB or an individual query in terms of our design pattern.

As a result, we get a set of substitutions for query variables that represent a failure found in the architecture. Unnecessary variables can be hidden from the answer by different means. For example, we can add a rule for a query with a reduced set of arguments to the program code using its head as a goal. Alternatively, it is possible to form the desired answer with a built-in procedure `print/1`, for example, we can complete our previous example with `print(('Interface' = I, 'System From' = S1, 'System To' = S2)), nl, fail.` if we are not interested in the safety classes of the systems.

To model a QUERY FB from our design pattern, we have to negate this Prolog goal (or individual query) with `not/1`, which will give us the answer to whether undesired relations are present in the architecture.

C. CRITERIA AND ATTRIBUTES ENCODING

Criteria and attributes are modeled as Prolog rules `cr/4`, which essentially represent the criteria and state to which attribute they belong, for example, Criterion 1.3 from Fig. 3 is encoded as follows:

```
  cr(design, diff_tech, S1, S2) :-
  different_technologies(S1, S2).
  cr(design, diff_appr_within_tech, S1,
S2) :-
  different_approach(S1, S2).
 cr(design, diff_architectures, S1, S2)
:-
   (different_architectures(S1, S2);
   cr(design, diff_tech, S1, S2);
   cr(design, diff_appr_within_tech,
  S1, S2)).
```

In this example, `different_technologies/2`, `different_approach/2`, and `different_architectures/2` are individual queries, which, following the logic from NRC [11] are assembled into Prolog rules that represent CRITERION FBs. The first argument in all the statements means that all the criteria belong to a `design` attribute. Criteria and attribute weights are encoded as separate predicates and are used further in code

```
 attribute_weight(design, 1).
  cr_weight(design, diff_tech, 0.5).
     cr_weight(design, diff_appr_within_
tech,
  0.333).
   cr_weight(design, diff_architectures,
0.167).
```

To obtain the final score, the criteria, attributes, and their weights are aggregated according to the logic specified in the evaluation of the design principle.

D. REASONING

Now, the system is ready to be evaluated. To not only obtain the score but to get an insight into its causes, we have to formulate a special term. We omit the Prolog code for the aggregation of the satisfied criteria and the summation of their weights (predicates `cr_satisfied/3` and `sum/4`) and show only the main query

```
 q(S1, S2) :- cr_satisfied(S1, S2, Cri-
teria),
  sum(S1, S2, Sum, Scores),
  print
  ('Criteria satisfied' = Criteria,
  'Scores by attribute' = Scores,
  'Total score' = Sum),
  nl, fail.
```

Execution of the goal `q('PS', 'DAS').` yields the following result for our example:

```
'Criteria satisfied' =
 [(design,diff_architectures),
 (design,diff_tech),
 (equip_manuf,diff_man_design),
 (equip_manuf,same_man_diff_design)],
'Scores by attribute' =
 [(design,0.667), (equip_manuf,0.175)],
'Total score' = 0.8420000000000001
```

This is an I&C architecture evaluation result based on two attributes that each includes three criteria. We can see that the criterion that checks the use of different approaches within the same technology is not satisfied, which reduces the total score of the overall design principle (here, the diversity) evaluation. Assuming that the whole design principle is encoded as shown, the engineer could consider how far the total score is from the maximum value with the acceptable error, and apply fixes to address the criteria that were not satisfied. With this code, we can change the logic of the design principle, for example, add additional criteria, build a more complex system of attributes, and change weights. Then, having this list of satisfied and unsatisfied rules, their structure, and attributes valuations, we obtain an assignment of an FBD which can then be analyzed using, e.g., graphical user interface tools.

E. EXECUTION SEMANTICS

One benefit of using Prolog as an implementation language is that the logic of the design principle is encoded declaratively, meaning that the analyst does not have to consider the execution semantics, as it is outsourced to the Prolog engine.

The computation is mainly based on two principles, i.e., unification and resolution refutation. Informally, unification stands for a process of discovering such a substitution for the variables in the formulas that makes the formulas equal. The resolution refutation is aimed at finding such a substitution of query variables that, applied, leads to a contradiction between the negated query and the program.

VI. IMPLEMENTATION AS A GRAPHICAL TOOL

In this section, we present a graphical tool implementing the presented method. The tool FBQL Editor presented in the current section was implemented using an imperative object-oriented language, C++. This allowed us to create means for users to define their custom basic blocks and explanation rules for them. The knowledge base is presented in a declarative form of ontology.

In the FBQL Editor, we distinguish the following three types of FBs: basic, composite, and SPARQL. Basic FBs perform computational functions; for example, they can implement logical operations (AND, OR, NOT), arithmetic operations (addition, subtraction, multiplication, division, etc.), and comparison operations (equal, not equal, greater, less, etc.). Basic FB functionality is defined using the JavaScipt language. In FBQL Editor we predefined a standard set of basic FBs that correspond to the most common arithmetic and logical operators. Although we believe that such a set can be sufficient for most of the usage scenarios, the tool is open-source, and the initial set can be extended according to one's needs. The functions and interfaces of basic FBs can be changed only textually. The functionality of composite FBs is determined by the corresponding nets of FBs of all three types, which can be created and edited in the graphical editor. The GUI includes the main diagram area and a list of available FBs. FBs libraries can be uploaded from a file or created manually. The user can create, update, or delete basic and composite FBs, both individually and within the library. The functionality of SPARQL FBs is defined by the corresponding visual queries based on the restricted SPARQL language. A special graphical editor has been developed for the creation and editing of this kind of FBs.

A. SYSTEM ENCODING

This implementation assumes that the overall I&C architecture is stored in an ontology.

An *ontology* is a vocabulary for a shared discourse domain, which defines classes, individuals, and the relationships between them for a given domain [19]. Ontology languages such as OWL [6] (which serves as the foundation of *Semantic Web* [20]) provide machine-interpretable structures to express semantics. OWL is an extension of resource description framework (RDF) [6], and a *knowledge base* based on RDF triples can use *reasoners* to infer, for example, subsumption or classification. SPARQL [7] serves as a query language for RDF graph patterns.

In Pakonen and Mätäsniemi's [5] work, the authors experiment with a semantic web-based approach to analyse nonfunctional requirements related to DiD. The U.S. EPR is used as a case study. The Protégé [21] ontology editor serves as a graphical medium for processing SPARQL queries. This is the ontology that we use in the experiments with the tool.

FIGURE 5. **Graphical representation of a SPARQL query for CQ 2.1 (1) from Pakonen and Mätäsniemi's [5] work. Rectangles with sharp and rounded corners allow one to define variables and constants, respectively. The outer rectangle marks the boundaries of a single graph pattern. Construction** `ORDER BY` **is omitted in the diagram because it does not affect the items of the result set.**

B. INDIVIDUAL QUERIES ENCODING

Since our I&C architecture is represented with an ontology, individual queries should contain the request to the knowledge base in an acceptable format, which is SPARQL in our case. To model a REQUEST FB, we developed a graphical language that we internally translate into SPARQL queries and make a request to the Apache Jena Fuseki SPARQL Server [22]. The implementation of REQUEST FB is called SPARQL FB in the FBQL Editor. It has no input and one numerical output. The reasons why we decided to develop a graphical version of SPARQL instead of having a plain textual editor are as follows. First, having the graph pattern visualized makes it easier to navigate complex relations. Second, this reduces the number of errors caused by typos and misplaced relations in queries. Finally and mainly, the visual SPARQL query design requires no special knowledge of SPARQL syntax or RDF prefixes; all domain experts have to do is to "draw" a graph pattern they want to search for in the ontology and add the necessary Boolean constraints. We claim that eliminating the need to learn SPARQL can positively influence the adoption rate. Our graphical version of SPARQL is a prototype, and the user experience is a subject for improvement. For example, drop-down menus can be introduced for choosing appropriate relationships or individuals, or, e.g., an intelligent autocomplete. In the following, we describe the syntax of our graphical language, while its semantics are fully inherited from SPARQL.

Graphical SPARQL syntax: The graphical language can be translated into a subset of SPARQL. We allow the formulation of graph patterns using graphic analogues for RDF triples with variables and literals, as well as filtering the results using Boolean constraints utilising FILTER statements. In addition to WHERE clauses, it is allowed to use the MINUS and UNION clauses to perform graph operations. Fig. 5 provides an example of a SPARQL query for a CQ 2.1 (1) from Pakonen and Mätäsniemi's [5] work and shows the formulating of multiple graph patterns with their relations. Fig. 6 contains a visual SPARQL query for CQ 3.3 from Pakonen and Mätäsniemi's [5] work that contains a Boolean predicate in its

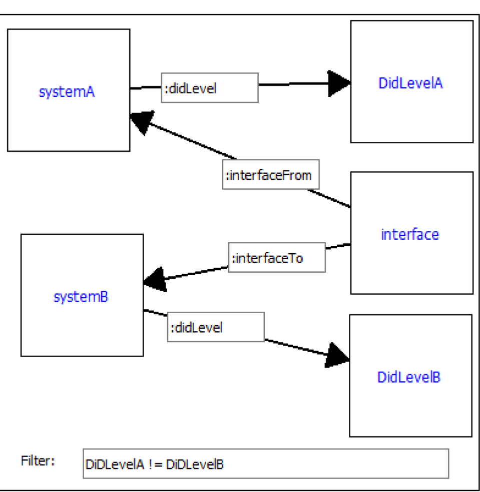


FIGURE 6. **Graphical representation of SPARQL query for CQ 3.3 from Pakonen and Mätäsniemi's [5] work. A Boolean predicate is added to the** `FILTER` **construction of the SPARQL query.**

"Filter" construction. The query 2.1 (1) is formulated using SPARQL as follows:

```
SELECT ?fromSystem ?toSystem ?interface
 ?description
WHERE {
 ?interface :interfaceFrom ?fromSystem.
 ?interface :interfaceTo ?toSystem.
 ?interface rdfs:comment ?description.
 ?fromSystem :hasSafetyClass USEPR:S.
 MINUS {?toSystem :hasSafety-
Class USEPR:S}
 MINUS {?interface :electricallyIso-
lated true}
}
ORDER BY ?interface
```

The query 3.3 is formulated using SPARQL as follows:

```
SELECT ?systemA ?DidLevelA ?systemB
 ?DidLevelB ?interface
WHERE {
```

```
   ?systemA :associatedWithDidLevel ?Di-
dLevelA.
   ?systemB :associatedWithDidLevel ?Di-
dLevelB.
   ?interface :interfaceFrom ?systemA.
   ?interface :interfaceTo ?systemB.
   FILTER (?DidLevelA != ?DidLevelB)
  }
```

Currently, graph patterns inside the "Filter" constructions cannot be formulated graphically. However, the tool has the option of calling a textual editor, where the user can complete the translated graphical query with necessary graph patterns or, e.g., `ORDER BY` constructions using the SPARQL syntax. Using graphical SPARQL together with the textual editor, it is possible to formulate all queries from Pakonen and Mätäsniemi's [5] work.

SPARQL FBs implement REQUEST FB from AC design pattern. To get an individual query, we connect the negation basic FB to the output of SPARQL FB and wrap it into a composite block. Note that at this stage of the development of query/computing FBD, the limitation is that SPARQL FB has a single numerical output. If the result of computing the corresponding SPARQL query is empty, then `0` is issued; otherwise the number of fitting tuples found. In the future, we are going to expand the possibilities of using the results of computing SPARQL queries. For example, SPARQL FB could output several integral numerical indicators (rather than one, as in the current version).

C. ATTRIBUTES AND CRITERIA ENCODING

Criteria and attributes are represented with composite FBs, which can include both basic and composite FBs. In the following, we describe the means for the creation of basic and composite FBs.

1) BASIC FBS CREATION

Basic FBs can be created manually using the editor shown in Fig. 7. The transformation function is encoded using JavaScript. The built-in variables are arrays of `inputs`, `outputs`, and `causes`. `inputs` and `outputs` represent arrays of input and output variables (or interface variables) of a basic FB correspondingly. The user may create additional internal variables to define the transformation function. Since we use JavaScript (a dynamically typed language) for the FB logic encoding, the variables do not have type constraints. The interface variables of a basic FB can be addressed by their indexes in the arrays. The output explanation logic can be integrated into the FB script using an array of `causes`. By adding the corresponding indexes of the input variables to `causes`, the user can determine which variables influence the output variables of the FB in each particular evaluation situation (or by using some common rule as shown in Fig. 7). Each basic block must have a name.

In our implementation, basic FBs may have a finite number of inputs (defined by the number of incoming connections) and outputs (defined in the transformation script). There is no defined structure for the script and the user may write it in their preferred coding style as long as it is a valid JavaScript code, which can be checked by clicking the button "Test Script" of the editor. The default library of basic FBs implements logical, comparison with zero, and arithmetic operators with a single output.

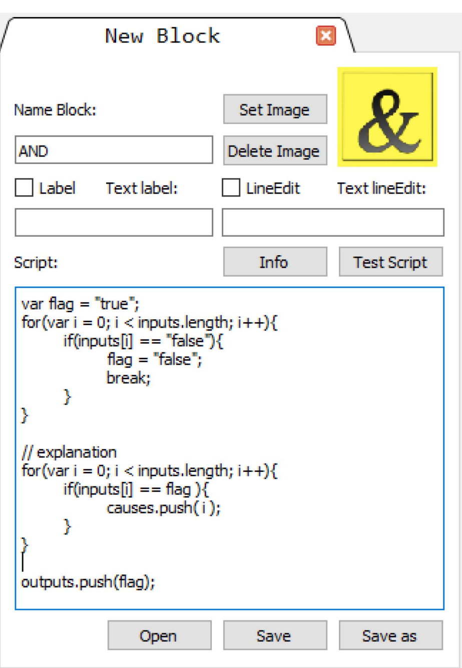


FIGURE 7. Basic FBs editor. The JavaScript script encodes the transformation function of a basic FB. "Label" and "LineEdit" check boxes and fields are used to create basic FBs with user input, e.g., to define constants. More can be found in the documentation of the tool [23].

2) COMPOSITE FBS CREATION

Composite FBs are created using the main diagram area and comply with the description of composite FBs in Section III-A. Each composite FB has a name. Table 1 shows the elements that can be arranged and interconnected to represent the internal network of FBs of a composite FB, Fig. 8 shows the example of the internal diagram of a composite FB. The execution semantics of composite blocks is discussed further.

D. EXECUTION SEMANTICS

The execution semantics of our FBDs is synchronous, so that the output values of each block are functions of its input values, and the signals are propagated through connections instantly.

Each execution cycle of an FBD is independent of the previous executions, which means that any execution is preceded by the deletion of the previously obtained assignments (except for the assignments of constant values). If the diagram was not modified between two subsequent execution cycles, then the

TABLE 1. Building Constructs of a Composite FB or an FBD

Symbol	Type	Semantics
AND &	Basic Function Block	Basic FB "AND"
SUM b sum a	Composite Function Block	Composite FB "SUM" with two inputs (`a` and `b`) and one output (`sum`)
CQ2.1.1 SPARQL	SPARQL Function Block	SPARQL Function Block "CQ2.1.1"
input_1	Input variable	Input variable `input_1`
output_1	Output variable	Output variable `output_1`

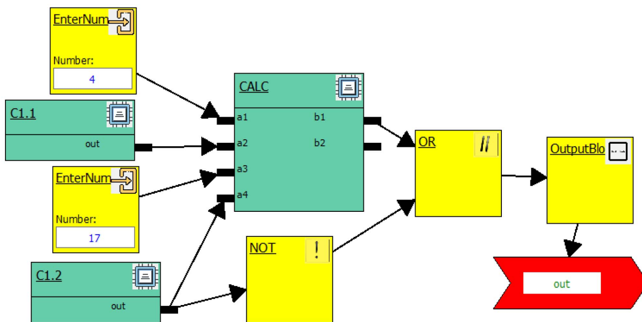


FIGURE 8. Example of a composite block internal diagram designed in FBQL editor. The diagram is formed by interconnected basic and composite FBs, the block has no inputs and one output `out`**.**

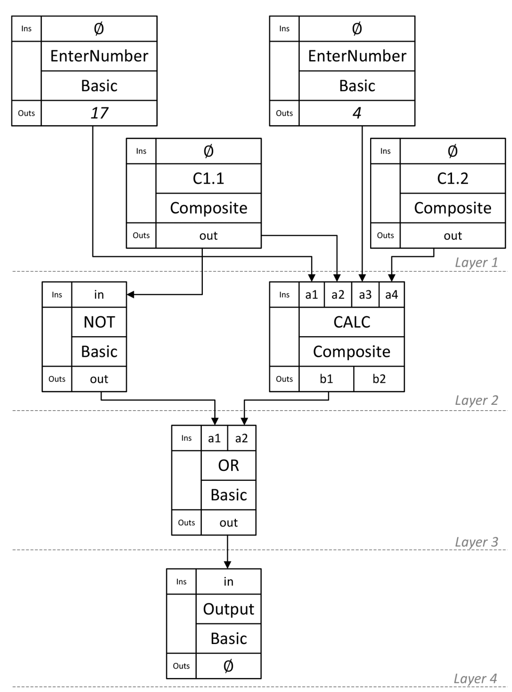


FIGURE 9. FBs of a composite FB from Fig. 8 graphically distributed by the execution layers they belong to. Numbers in italics are user-defined constants.

FBD assignments obtained after each of the executions will be the same.

The execution order of the FBs is determined by the structure of the FBD to which they belong. The basic rule is that an FB can be executed only after all its inputs are calculated during the current execution cycle. Thus, the first FBs to execute are the ones with no input (e.g., REQUEST FBs) or the ones whose all inputs are assigned with constant values.

Following this execution rule, the FBs of an FBD can be divided into *execution layers*—tuples (i, B), where $i \in N$ is the order of the layer and B is a set of FBs of the FBD that can be executed in parallel. For example, Fig. 9 shows the distribution of the execution layers of FBs that make up a composite FB from Fig. 8. We show the distribution for the initial internal diagram of the FB. The internal diagram also contains composite FBs that can be represented with their internal nets. Since we forbid infinite nesting of composite FBs, each composite FB and an FBD, in general, in the end, can be unfolded into a net of basic FBs. This unfolding is the first step of our execution algorithm. It is followed by building the execution layers over the unfolding result and their subsequent ordered execution.

In our FBDs, we do not allow loops in general. This is explained by the fact that we model requirements, which are evaluated over an ontology during a single cycle. Our requirements modeled using FBDs are enriched queries to an I&C architecture knowledge base (queries with postprocessing) rather than a control logic.

E. REASONING AND ITS IMPLEMENTATION

The reasoning is based on the concept of dependence of FB output on input. Let us assume that each FB has only one output and several inputs. In this case, it can be represented as a function $y = f(x_1, x_2, \ldots, x_n)$.

Each of the arguments x_i affects the result of y (in other words, the value of the function depends on each of the arguments). Conceptually, if some of the arguments did not affect the result of the function, they could be safely removed from the set of arguments. This type of dependence of the output y on the inputs $x_1, x_2, \ldots, x_n$ follows from the definition of the function.

Now, let us consider the concept of dependence of the output value of y on the assignment of the input variables. Let $w = f(d_1, d_2, \ldots, d_n)$, where $d_1, d_2, \ldots, d_n$ are the specific

assignments of the inputs and w is the specific assignment of the output.

We say that the input value d_i does not affect the result w if any change applied solely (that is, maintaining the same context) to d_i does not lead to a change in the result. Otherwise, we say that the output value w directly depends on the input value d_i in a given context (in other words, the input value d_i individually influences the output value w).

Let us call the input value d_i *individually dominant* among the input parameters $d_1, d_2, \ldots, d_n$, if it completely determines the output value w. In this case, the remaining values can be changed, but the result w will remain. We define such a dependence of the output value w on the input value d_i as a *individual dominance dependence* (context-independent).

Following the same logic, if there is a group of input values $d_k, d_m, .., d_s$ that completely determines the output value w, that is, the remaining values can be changed, but the result w remains, such a dependence of the output value w on the group of input values $d_k, d_m, \ldots, d_s$ is a *group dominance dependence* (also context-independent). As previous, here, the context also refers to the input values that are not included in the group. Several dominance groups can exist for one output value. If the group includes all input values, then this dependency is a *full-group dominance dependency* (or a *full-group dependency*). Since in this case there is no context as such, the dependence directly determines the result.

For explanation purposes, we consider only the dominance relations (individual and group) between an output and the inputs. The assignment whose value affects the output is an *active assignment*. An input assignment is active if:

1) there exists a change in the value of this assignment, which leads to a change in the output value or;
2) the value of this assignment directly determines the output value.

For each FB type, it is necessary to formulate particular rules for determining active assignments.

Reasoning is implemented according to the algorithm from Ovsiannikova et al.'s [12] work, which was also used to calculate the explanation in Oeritte, presented in the same article. Here, we use the same idea of backtracking, that is, starting from the explanation target (a variable value of interest), moving in the opposite direction to the information flow, and determining assignments that influence the target (active assignments). The difference from the existing application of the algorithm (in Oeritte) here is that a set of basic FBs can be created manually, which means that their individual explanation functions are determined by the user.

The explanation function explicitly encodes which assignments are active depending on the current output and input values. Thus, for example, for a basic FB that encodes the disjunction $y = x_1 \vee x_2$, the explanation function includes the following rules. If $y = T$, then the active assignments are those whose values are T and if $y = F$, both input assignments fully determine the output value.

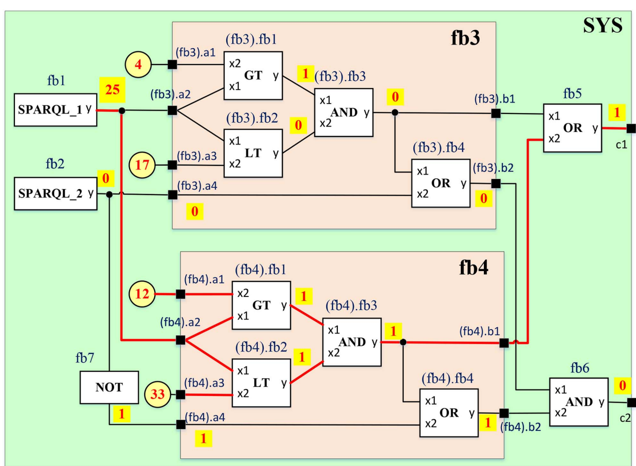


FIGURE 10. Explanation example shown in an unfolded internal diagram of a composite block SYS for the explanation target `c1` (the diagram was created using other than FBQL Editor visual redactor). The connections between the assignments that influence `c1` are shown in bold red. The block has two composite blocks (`fb3` and `fb4`) which are also shown unfolded. Red numbers on the yellow rectangle and round backgrounds show the valuations of the variables and predefined constants correspondingly.

As shown in Fig. 7, the user should use variable `causes` to store the active assignments determined (as they *cause* the output value).

Fig. 10 shows the result of the explanation for the output `c1` of a composite block `SYS`. The internal diagram contains two composite FBs and four basic FBs. As we can see, the result of `fb2` did not influence the target.

VII. RELATED RESEARCH

A considerable amount of scientific work describes approaches for modeling nonfunctional requirements. In our state-of-the-art analysis, we searched for existing visual languages that have built-in capabilities to assist domain experts in the formulation of DiD requirements. Special attention was paid to the techniques applicable to evaluating complex I&C architectures.

Most of the works consider integrating nonfunctional requirements into the software development process. Even though our nonfunctional requirements were decoupled from the ones that software must comply with, we found several visual techniques that partially motivated our research. For example, Fei and Xiaodong [24] discussed the integration of software architecture during analysis, and the described hierarchy of properties resembles our AC design pattern. Then, Gross and Yu [25] proposed the formulation of nonfunctional requirements of different types in the form of different design patterns, while we propose a design pattern to formulate various requirements.

Supakkul and Chung [26] extended the UML language for the formulation of functional requirements with nonfunctional requirements framework and integrate two notations. Hammani [27] also addressed the software engineering domain and points out the diversity of nonfunctional requirements; such requirements can often be decomposed into several subrequirements. Their extended feature model, as our FBD, among all, depicts how the requirements can be expressed quantitatively using variables of a system. Bernardi et al. [28] presented dependability-specific modeling and analysis in MARTE, which extends UML with concepts for modeling and quantitative analysis of real-time and embedded systems.

Other articles in the software engineering domain address nonfunctional requirements in model-driven development [29], Cysneiros et al. [30] integrated them into conceptual models to deal with process-oriented properties, Botella et al. [31] developed a special language and use UML to implement its constructs, Ernst et al. [32] proposed a quality-based visualisation scheme for nonfunctional aspects, which is layered on top of functional artifacts.

In the I&C research domain, most of the works propose the overall architectural development approaches, often without particular focus on the nonfunctional requirement formulation. Thus, Linnosmaa et al. [33] applied the architecture analysis and design language to overall nuclear I&C. Neyret-Thibault et al. [34] in turn, proposed a tool, STIMULUS, for the visual design of requirements, but for the functional design of I&C. The tool also has the capability of automatic observer creation for the model under development, which reflects our idea of being able to check the requirements automatically and immediately after formulation.

There are also a few works on automatic nonfunctional property checking; commonly, such works focus on one particular requirement or their group. For example, Singh [35] used the Petri Nets representation of safety critical systems to estimate their performance risk, Wakankar et al. [36] proposed an automated dependability analysis using model checking on the system architecture hierarchy, and Promyslov et al. [37] dealt with cybersecurity requirements. Sannier et al. [38] focused on modeling requirements under the conditions of their frequent changes and mentions tools for their creation (SysML and Unispace); however, it keeps the creation of the architecture model and communication with it beyond the scope.

More recent works mainly consider the software engineering domain. For example, they include Alhaizaey and Al-Mashari's [39] work, which proposes a framework for reviewing nonfunctional requirements in agile requirements formulated as user stories or Bajammal [40] proposed a visual analysis of web applications' nonfunctional properties. In turn, Ernerstedt [41] addressed distributed systems and evaluates visual tools to verify cloud-deployed applications against nonfunctional properties.

Lubars et al. [42] assembled common challenges in the development of complex software systems and the management of their requirements, which also persist today in the domain of I&C architectures. Our approach is flexible enough to be tuned to the needs of different industries and different approaches within the industry. The tool-assisted approach simplifies the process of evaluating the system in the situation of changing requirements. Resolution of conflicts between different requirements can also be implemented using FBDs; for instance, the cost optimization dimension can be added to the existing FBD that evaluates the diversity of the architecture.

Our approach can also be used with other methods and tools, for example, executable UML. In this case, the knowledge base should be figured out; for example, a relational database can be queried instead of a set of Prolog statements or an ontology.

VIII. DISCUSSION

We developed a method that 1) enables computer-aided graphical formulation and a quantitative analysis of nonfunctional properties or design principles that often can be satisfied only to some extent and 2) reasoning over the obtained evaluation result. The method of formulating design principles with FBDs can be implemented using diverse instruments, and we provided the example implementation in the FBQL Editor graphical tool.

We demonstrated the method and the tool using a particular ontology that represents I&C architecture and formulated a diversity design principle using the proposed pattern. However, the applicability scope of the method is not restricted by this example. Domain experts, engineers, and analysts can maintain the knowledge they have about the system in ontologies and formulate various design principles or complex nonfunctional properties. The use of the proposed method can be justified if several requirements that can be answered by whether they hold or not for the system are combined with additional logic, for example, weighting or prioritization.

The reasoning principles of the method are described with Prolog, while the extended FBD language is used for query definition.

In our tool implementation, we use ontology as a knowledge base and SPARQL queries to formulate the requests, which are similar to Prolog, but better supported by modern software tools.

IX. CONCLUSION

The contribution of this article is three-fold. First, we proposed a method and a design pattern to formulate complex design principles using FBD. This is a continuation of Pakonen and Mätäsniemi's [5] work where the authors argued that ontologies are convenient for developing and maintenance of I&C architectures and showed how to formulate what we call individual queries, using SPARQL. Together with

Ovsiannikova et al.'s [43] work, this series of work establishes the basis for developing complex I&C architectures, their verification against nonfunctional properties, and subsequent debugging of failures.

Second, we defined the method using the logical programming language, Prolog. Finally, we demonstrated the implementation of the proposed method within the graphical tool, equipped with a graphical editor for the intuitive formulation of SPARQL queries, which relieves users from the need to learn SPARQL in the event that their knowledge base is represented with an ontology.

The FBQL Editor graphical tool is available online [23]. The tool is in the prototype stage and its experimental evaluation was limited to the project team. In future work, the tool is intended to be used by more engineers.

Future work also includes improving the user experience of the tool and enriching the graphical implementation of SPARQL. In addition to general tool enhancements, we plan to integrate algorithms from Ovsiannikova et al.'s [43] work to provide a magnifying view of the knowledge base and to highlight particular areas in I&C architectures that may contain flaws. Then, it will be possible to create a method to determine the optimal fix to the architecture for the criteria provided by the analyst.

REFERENCES

[1] J.-E. Holmberg, "Defense-in-depth," in *Handbook of Safety Principles*. Hoboken, NJ, USA: Wiley, pp. 42–62, 2017.

[2] STUK, "Safety design of a nuclear power plant," Radiation and Nuclear Safety Authority YVL Guide B.1, 2019. [Online]. Available: https://www.stuklex.fi/en/ohje/YVLB-1

[3] A. Mosteiro-Sanchez, M. Barcelo, J. Astorga, and A. Urbieta, "Securing IIoT using defence-in-depth: Towards an end-to-end secure industry 4.0," *J. Manuf. Syst.*, vol. 57, pp. 367–378, 2020, doi: 10.1016/j.jmsy.2020.10.011.

[4] WENRA, "Safety of new NPP designs - study by reactor harmonization working group RHWG," Western European Nuclear Regulators' Association, *Tech. Rep.*, 2013. [Online]. Available: https://www.wenra.eu/sites/default/files/publications/rhwg_safety_of_new_npp_designs.pdf

[5] A. Pakonen and T. Mätäsniemi, "Ontology-based approach for analyzing nuclear overall I&C architectures," in *Proc. 47th Annu. Conf. IEEE Ind. Electron. Soc.*, 2021, pp. 1–7, doi: 10.1109/IECON48115.2021.9589078.

[6] W3C, "OWL 2 web ontology language document overview (second edition)," *The World Wide Web Consortium*, 2012. [Online]. Available: https://www.w3.org/TR/owl2-overview/

[7] W3C, "SPARQL 1.1 query language" The world wide web consortium, w3c recommendation, 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[8] International Electrotechnical Commission, International Standard IEC 61131-3:2013: Programmable Controllers. Part 3: Programming Languages. IEC, 2013.

[9] W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*. Berlin, Germany: Springer, 2003.

[10] IAEA, "Exploring semantic technologies and their application to nuclear knowledge management," *Int. At. Energy Agency, Nucl. Energy Ser. NG-T-6.15*, 2021. [Online]. Available: http://www-pub.iaea.org/MTCD/Publications/PDF/P1899_web.pdf

[11] NRC, "Diversity strategies for nuclear power plant instrumentation and control systems" United States Nuclear Regulatory Commission, NUREG CR-7007, 2009. [Online]. Available: https://www.nrc.gov/docs/ML1005/ML100541256.pdf

[12] P. Ovsiannikova, I. Buzhinsky, A. Pakonen, and V. Vyatkin, "Oeritte: User-friendly counterexample explanation for model checking," *IEEE Access*, vol. 9, pp. 61383–61397, 2021, doi 10.1109/ACCESS.2021.3073459.

[13] Areva NP. U.S. EPR Final Safety Analysis Report. 2013. [Online]. Available: https://www.nrc.gov/reactors/newreactors/design-cert/epr/reports.htm

[14] L. S. Sterling and E. Y. Shapir, *The Art of Prolog*, 2nd ed. Cambridge, MA, USA, MIT Press, 1994.

[15] I. Bratko, *Prolog Programming for Artificial Intelligence*, London, U.K.: Pearson, 2001.

[16] J. W. Lloyd, *Foundations of Logic Programming*. Berlin, Germany: Springer, 2012.

[17] J. Minker, "Logic and databases: A 20 year retrospective," in *Proc. Int. Workshop Log. Databases*, 1996, pp. 1–57.

[18] H. Rybiński, "On first-order-logic databases," *ACM Trans. Database Syst.*, vol. 12, no. 3, pp. 325–349, 1987.

[19] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl. Acquisition*, vol. 5, no. 2, pp. 199–220, 1993, doi: 10.1006/knac.1993.1008.

[20] N. Shadbolt, T. Berners-Lee, and W. Hall, "The semantic web revisited," *IEEE Intell. Syst.*, vol. 21, no. 3, pp. 96–101, Jan./Feb. 2006, doi: 10.1109/MIS.2006.62.

[21] M. A. Musen, and Protégé Team "The Protégé project: A look back and a look forward," *AI Matters*, vol. 1, pp. 4–12, 2015, doi: 10.1145/2757001.2757003.

[22] A. J. Fuseki. [Online]. Available: https://jena.apache.org/documentation/fuseki2/

[23] FBQL editor. [Online]. Available: https://github.com/FBQLEditor/FBQLEditor

[24] Y. Fei and Z. Xiaodong, "An XML-based software non-functional requirements modeling method," in *Proc. 8th Int. Conf. Electron. Meas. Instrum.*, 2007, pp. 2-375–2-380, doi: 10.1109/ICEMI.2007.4350695.

[25] D. Gross and E. Yu, "From non-functional requirements to design through patterns," *Requirements Eng.*, vol. 6, no. 1, pp. 18–36, 2001.

[26] S. Supakkul and L. Chung, "A UML profile for goal-oriented and use case-driven representation of NFRs and FRs," in *Proc. 3rd ACIS Int. Conf. Softw. Eng. Res. Manage. Appl.*, 2005, pp. 112–119, doi: 10.1109/SERA.2005.19.

[27] F. Z. Hammani, "Survey of non-functional requirements modeling and verification of software product lines," in *Proc. IEEE 8th Int. Conf. Res. Challenges Inf. Sci.*, 2014, pp. 1–6, doi: 10.1109/RCIS.2014.6861085.

[28] S. Bernardi, J. Merseguer, and D. C. Petriu, "A dependability profile within MARTE," *Softw. Syst. Model.*, vol. 10, pp. 313–336, 2011.

[29] D. Ameller, X. Franch, and J. Cabot, "Dealing with non-functional requirements in model-driven development," in *Proc. 18th IEEE Int. Requirements Eng. Conf.*, 2010, pp. 189–198, doi: 10.1109/RE.2010.32.

[30] L. M. Cysneiros, J. C. S. do Prado Leite, and J. de Melo Sabat Neto, "A framework for integrating non-functional requirements into conceptual models," *Requirements Eng.*, vol. 6, pp. 97–115, 2001.

[31] P. Botella, X. Burgues, X. Franch, M. Huerta, and G. Salazar, "Modeling non-functional requirements," in *Proc. Jornadas de Ingenieria de Requisitos Aplicada JIRA*, vol. 2001, 2001.

[32] N. A. Ernst, Y. Yu, and J. Mylopoulos, "Visualizing non-functional requirements," in *Proc. 1st Int. Workshop Requirements Eng. Visual.*, 2006, Art. no. 2, doi: 10.1109/REV.2006.10.

[33] J. Linnosmaa, A. Pakonen, N. Papakonstantinou, and P. Karpati, "Applicability of AADL in modelling the overall I&C architecture of a nuclear power plant," in *Proc. 46th Annu. Conf. IEEE Ind. Electron. Soc.*, 2020, pp. 4337–4344, doi: 10.1109/IECON43393.2020.9254226.

[34] M. Neyret-Thibault, T. Lemattre, and G. Robin, "Model-based verification of I&C specifications-184," NPIC&HIMIT, USA, Jun. 11–15, 2017.

[35] P. Singh and L. Singh, "Verification of safety critical and control systems of nuclear power plants using petri nets," *Ann. Nucl. Energy*, vol. 132, pp. 584–592, 2019, doi: 10.1016/j.anucene.2019.06.027, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306454919303457

[36] A. Wakankar, A. Kabra, A. Bhattacharjee, and G. Karmakar, "Architectural model driven dependability analysis of computer based safety system in nuclear power plant," *Nucl. Eng. Technol.*, vol. 51, no. 2, pp. 463–478, 2019, doi: 10.1016/j.net.2018.10.019, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1738573318300536

[37] V. G. Promyslov, K. V. Semenkov, and G. V. Promyslov, "Practical method of the I&C system security architecture design using graph models," *IFAC-PapersOnLine*, vol. 55, no. 9, pp. 227–232, 2022, doi: 10.1016/j.ifacol.2022.07.040, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896322004256

[38] N. Sannier, B. Baudry, and T. Nguyen, "Formalizing standards and regulations variability in longlife projects. A challenge for model-driven engineering," in *Proc. Model-Driven Requirements Eng. Workshop*, 2011, pp. 64–73, doi: 10.1109/MoDRE.2011.6045368.

[39] A. Alhaizaey and M. Al-Mashari, "A framework for reviewing and improving non-functional requirements in agile-based requirements," in *Proc. 18th Iberian Conf. Inf. Syst. Technol.*, 2023, pp. 1–7, doi: 10.23919/CISTI58278.2023.10211956.

[40] M. Bajammal, "Automated visual analysis of non-functional web app properties," Ph.D. dissertation, Univ. Brit. Columbia, Vancouver, BC, Canada, 2022. [Online]. Available: https://open.library.ubc.ca/collections/ubctheses/24/items/1.0413754

[41] J. Ernerstedt, "An evaluation of tools for verifying non-functional requirements for cloud deployed applications," *Master's thesis*, Fac. Sci. Technol., Dept. Comput. Sci., Umeå Univ., Umeå, Sweden, 2023.

[42] M. Lubars, C. Potts, and C. Richter, "A review of the state of the practice in requirements modeling," in *Proc. IEEE Int. Symp. Requirements Eng.*, 1993, pp. 2–14, doi: 10.1109/ISRE.1993.324842.

[43] P. Ovsiannikova, A. Pakonen, and V. Vyatkin, "Automatic generation of repair suggestions for overall I&C architecture represented with an ontology," in *Proc. IEEE 28th Int. Conf. Emerg. Technol. Factory Automat.*, 2023, pp. 1–8, doi: 10.1109/ETFA54631.2023.10275557.

POLINA OVSIANNIKOVA (Graduate Student Member, IEEE) received the B.Sc. degree in software engineering and the M.Sc. degree in applied mathematics and computer science from ITMO University, Saint Petersburg, Russia, in 2016 and 2018, respectively, and the Ph.D. degree from Aalto University, Espoo, Finland, in 2023.

Her research interests include formal verification and its industrial applicability, methods for aiding in requirements analysis for I&C systems, and automation technologies for vertical farming.

ANTTI PAKONEN received the M.Sc. (Tech.) degree in I&C systems from the Helsinki University of Technology, Espoo, Finland, in 2004.

He is currently a Senior Scientist and Project Manager with the VTT Technical Research Centre of Finland Ltd., Espoo, Finland, where he has been employed, since 2002. His research interests include I&C software engineering, I&C architecture evaluation, practical application of model checking in industrial applications, and knowledge management.

DMITRY MUROMSKY received the B.Sc. degree in software engineering from the University of Penza, Russia, in 2023.

He is a Software Engineer, who actively participates in research and development of software for information protection and information security. His research interests include graphic editors and networked systems.

MAKSIM KOBZEV received the B.Sc. degree in software engineering from the University of Penza, Russia, in 2023.

He is a Software Engineer. He participates in the development of software systems for the collection, processing, and analysis of data in the field of education. His research interests include finite state machine models and their transformations, human–machine interface design methods, and graphical query languages based on SPARQL.

VIKTOR DUBININ received the diploma and Ph.D. degrees in computer engineering and the Dr.Sc. degree in computer science from the University of Penza, Penza, Russia, in 1981, 1989, and 2014, respectively.

From 1981 to 1989, he was a Researcher; from 1989 to 1995, he was a Senior Lecturer; and from 1995 to 2015, he was an Associate Professor with the University of Penza. In 2011, he held a visiting Researcher position with The University of Auckland, Auckland, New Zealand, and from 2013 to 2019, he was with the Luleå University of Technology, Luleå, Sweden. From 2015 to 2022, he was a Professor with the Department of Computer Science, University of Penza. His research interests include formal methods for specification, verification, synthesis, and implementation of distributed and discrete event systems.

Dr. Dubinin was a recipient of DAAD grants to work as a Guest Scientist with Martin Luther University Halle-Wittenberg, Halle, Germany, in 2003, 2006, and 2010.

VALERIY VYATKIN (Fellow, IEEE) received the Ph.D. degree in applied computer science from the Taganrog State University of Radio Engineering, Taganrog, Russia, in 1992, the Dr.Eng. degree in electrical engineering from the Nagoya Institute of Technology, Nagoya, Japan, in 1999, and the Habilitation degree in engineering from the Ministry of Science and Technology of Sachsen-Anhalt, Magdeburg, Germany, in 2002.

He is currently the Chaired Professor with the Luleå University of Technology, Luleå, Sweden, and a Full Professor with Aalto University, Helsinki, Finland. He was a Visiting Scholar with Cambridge University, Cambridge, U.K., and had permanent academic appointments with New Zealand, Germany, Japan, and Russia. His research interests include dependable distributed automation and industrial informatics, software engineering for industrial automation systems, artificial intelligence, distributed architectures, and multiagent systems applied in various industry sectors, including smart grids, material handling, building management systems, data centres, and reconfigurable manufacturing.

Dr Vyatkin was a recipient of the Andrew P. Sage Award for the Best IEEE Transactions Paper in 2012. He has been Chair of the IEEE IES Technical Committee on Industrial Informatics, since 2016, and the Vice President of IES for Technical Activities for the term 2022–2025.