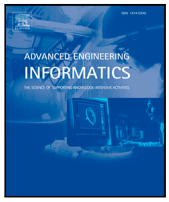
Advanced Engineering Informatics 57 (2023) 102099

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei



Full length article

Modeling and validating temporal rules with semantic Petri net for digital twins

Han Liu [a,b], Xiaoyu Song [c], Ge Gao [a,d,*], Hehua Zhang [a,d], Yu-Shen Liu [a,d], Ming Gu [a,d]

[a] Tsinghua University, Haidian, Beijing, China
[b] Digital Horizon Technology Co., Ltd., Shenzhen, China
[c] Portland State University, Portland, OR, United States
[d] Beijing National Research Center for Information Science and Technology (BNRist), Haidian, Beijing, China

ARTICLE INFO

Dataset link: https://github.com/highan911/CBIMS.SPN

Keywords:
Digital twin
Building information model (BIM)
Petri net
Semantic web
SPARQL

ABSTRACT

The semantic web has become an important resource of domain knowledge in the construction industry, and there is a need to model the temporal states and check the transition rules of digital twins in the semantic web. Related studies have added timestamps to describe an RDF graph that varies in time, but digital twin applications require a formal representation of the temporal states and transition rules other than simple timestamps. Related studies also focused on the interaction between temporal and semantic models, but there are still challenges in the two-way sharing of knowledge in the runtime of the temporal model. In this paper, the Semantic Petri Net (SPN) is proposed as a method to represent the temporal states and rules in RDF and SPARQL so that a runnable temporal model can be implemented on a common SPARQL engine, and the state change rules can be checked with access to the rich domain knowledge provided by the semantic web. An application case is presented for modeling and simulating the constraints in the process of a construction project.

1. Introduction

Digital twin is a technology representing real-world elements, environments, systems, state changes, and so forth in virtual data models. With the development and integrated applications of technologies such as Geographic Information System (GIS), Building Information Model (BIM), and Internet of Things (IoT), in recent years, the digital twin technology has become a frontier concept in the intersection field of information technology and industry. The semantic web is a common framework for sharing machine-readable knowledge and data based on the World Wide Web. By integrating the spatio-temporal data with the knowledge provided by the semantic web, digital twins can be used for real-time monitoring and coordination and supporting decision-making and design optimization [1–4].

In the construction industry, the information on building projects and assets is represented as Resource Description Framework (RDF) graphs. Human knowledge about the domain is usually represented as shared ontologies in the semantic web, which explicitly defines the classification of concepts, datatypes, properties, and relationships involved in this domain. An ontology is usually represented in the Web Ontology Language (OWL) [5,6], which supports the formal representation of the ontology in RDF. The shared ontologies allow computer programs to access the knowledge for implementing various rule-based applications on the domain data in RDF. Widely used languages for accessing the knowledge in the semantic web include SPARQL [7] for querying the linked data, SHACL [8] for checking the conformance of the data, and SWRL [9] for reasoning on the data.

Specifically, in the construction industry, the languages mentioned above are used to ensure the conformance of the BIM data with the standards and regulations [10–14]. While at present, most of the rules and constraints are for static models that do not change in time. In related studies like stSPARQL [15], the timestamps and periods are included in the language for querying and checking the building objects and properties that vary over time. However, digital twin applications usually have more complicated temporal states than timestamps and periods. There are usually rule constraints to determine whether a state change is allowed. Such state change rules are closely related to the classifications, properties, and relationships, so there is a need for an integrated temporal model that can make use of the domain ontologies and other RDF resources to model the temporal states and processes, and to represent the rule constraints for state changes.

In other domains, such as the manufacturing industry and information technology industry, there are studies on modeling the temporal

* Corresponding author at: Tsinghua University, Haidian, Beijing, China.
E-mail addresses: liuhan@digital-h.cn (H. Liu), gaoge@tsinghua.edu.cn (G. Gao).

https://doi.org/10.1016/j.aei.2023.102099
Received 16 December 2022; Received in revised form 6 July 2023; Accepted 7 July 2023
Available online 9 August 2023

states and state change rules. The Petri Net [16,17] methods are commonly used for modeling the states and constraints for concurrent systems. A Petri Net model is clearly defined and runnable on computers, so it can be used for implementing automatic agents on computers or for simulating the behavior of real-world concurrent systems. Several powerful mathematical tools can be applied to the Petri Net models for analyzing the concurrent systems to validate the features of the system, such as the accessibility in the state space and the stoppability of the system [18].

Related studies also focused on the interaction between temporal models and semantic web applications for enhancing the temporal modeling and validation with the information provided by the semantic models. The studies tried to describe the temporal models in ontologies [19–21], to generate the temporal models according to the RDF resources [22,23], and to transfer information between the temporal models and semantic web applications [24,25]. However, there are still challenges in the two-way knowledge sharing in the runtime of a temporal model. On the one hand, the transition rules in the temporal models can be evaluated with access to the vast information from the RDF graphs. On the other hand, the temporal model structures, rules and its current state are also in the form of RDF that can be accessed by other semantic web applications.

In this paper, based on the Petri Net method, the Semantic Petri Net (SPN) is proposed as a novel temporal modeling method. The temporal states and transitions are defined in OWL, and the temporal transition rules are represented in SPARQL statements. As a result, runnable SPN can be implemented on common SPARQL engines like Apache Jena [26] and dotNetRDF [27]. It is possible for the SPN to access the vast semantic information in the RDF graphs based on the domain ontology such as ifcOWL [28] in the runtime of the temporal model, and the current state of the SPN is also queryable from a SPARQL endpoint, which enables the two-way knowledge sharing between the SPN and other applications on the semantic web.

The main contributions are listed as follows.

- The definition of the SPN structures and rules for modeling temporal states are proposed (see Section 3).
- The compatibility of SPN and the method to search the state space are discussed (see Section 4).
- An application case is presented for modeling and simulating the state change rules in the construction industry (see Section 5).

2. Related work

2.1. Modeling and querying the semantic information in the construction industry

In the domain of construction industry, the physical building elements are always in spatial relationships with other elements or spaces. The elements are also usually linked to schedules, processes, or sensor records in construction projects. So the integration of spatio-temporal information with semantic data is an interest of research in order that the semantic queries and inferences can be performed for rule-based applications. Related studies have been performed in several disciplines.

IFC (Industry Foundation Classes) [29] is an open standard commonly used for buildings and infrastructures. The IFC schema involves the representation of the building element types and instances, relationships, properties, geometries, materials, and so on. The ifcOWL [28] is the full representation of IFC schema in OWL, which is used in digital twin applications such as sensor data integration [30,31], and indoor tracking and navigation [32,33]. There are studies based on ifcOWL with simplified semantic graph structure and enriched 3D spatial relationships [14,34,35] for supporting practical semantic queries and inferences.

Rather than setting up a large integrated ontology like ifcOWL, recent studies have focused on the composition of multiple ontologies for an application in the construction industry. Each ontology provides a module for modeling a particular aspect of information [36]. Related studies include the Digital Construction Ontologies [37], the Brick schema [38], and several ontologies maintained by the Linked Building Data Community Group (LBD CG) in the World Wide Web Consortium (W3C). In a construction or asset management project, the building product types can be defined in an ontology with fine-grained classification, such as the Building Product Ontology (BPO) [39], the Building Element Ontology (BEO) [40], and the Distribution Element Ontology (MEP) [41]. Other aspects of information can be modeled in different domain ontologies, such as the Building Topology Ontology (BOT) [31,42] for spatial structure components and relationships, the Ontology for Property Management (OPM) [43,44] for modeling the properties in different levels of complexity, and the Ontology for Managing Geometry (OMG) [45] and File Ontology for Geometry formats (FOG) [46] for geometry representation and exchange.

In addition to semantic models, the extensions on the query languages are also performed for various applications in the engineering domain. GeoSPARQL [47] is an Open Geospatial Consortium (OGC) standard for representing geographic information data and spatial relationships in RDFS/OWL, and for enabling SPARQL to support spatial relationship queries. BimSPARQL [13] is another extension of SPARQL, focusing on the queries for 3D geometrical and topological relationships between the building elements. In the time dimension, stSPARQL [15] introduces timestamps and periods based on GeoSPARQL, which can be used for querying the data records that are variable in location and over time. The triple patterns are extended into quad-patterns in stSPARQL with the last term as the valid period. Additional predicates and functions are also defined for comparing the timestamps and periods.

However, in the management of construction projects and assets, there are usually more complicated temporal states than just timestamps and periods. The construction and installation processes of building elements are usually composed of a sequence of states. The state changes are usually with dependencies and constraints, such as resources and machines, approvals, preprocess dependencies, safety conditions, etc. Several related studies involve the modeling of the temporal states and constraints in the construction industry. In the IFC schema, the processes and dependencies can be modeled using “IfcProcess” and “IfcControl” with assigned building products, resources, and actors, which is simple and informative for human users. The OPM [43,44] introduces the “Level 3 properties” with several property states that may change over time. The historical property state changes can be recorded and queried with the distinguish of “current” and “outdated” property states. The Digital Construction Ontologies [37] introduces process modeling involving various activities, such as movement, management issues, and construction operations of elements. The activities can be linked to various constraints, such as constraints for the starting condition, the ending effect, the machine requirement, and the time duration.

The above-mentioned related studies have proposed various methods for representing the elements, relationships, and constraints in construction projects, which provide methodologies and technologies for the semantic applications on digital twins. As the complexity of the states and constraints in the system increases, formal methods for modeling the temporal states and constraints are needed to accurately describe the system, in order that the process model is both informative for human users and executable for automatic processing, validation, and analysis applications.

2.2. Methods for modeling and validating temporal states

In the Information Delivery Manual (IDM) [48,49] standard for supporting BIM data exchange, the Business Process Modeling Notation

(BPMN) [50,51] is introduced for modeling the process maps of data exchange between multiple stakeholders in the construction project. The BPMN method represents process maps composed of events, activities, and gateways, and it is commonly used in project management tasks in various domains. BPMN is suitable for modeling workflows with multiple participants and is designed to provide human-readable instructions, including the details of the activities, the conditions on gateways, and the messages between the participants. However, BPMN is not designed to implement a model that automatically runs on computers.

The Finite State Machine (FSM) [52,53] methods are popular for modeling temporal states and state change conditions in electronic industry and information technology domains. FSM represents a temporal model with a finite set of states, an input alphabet, and a global transition function defining the next state on each input. FSM is suitable for modeling the life cycle of automata in response to various signals or events, such as string parsers or communication protocols.

The Petri Net methods are also widely used in modeling and analyzing temporal states, especially for concurrent systems with multiple participants. In a concurrent system that involves resource allocation and waiting between multiple threads, there are risks of conflicting accesses to shared resources, and a deadlock may occur. The Petri Nets are used to model the concurrent systems for simulating the execution of the system. Mathematical tools such as state space generation and place invariant analysis [54–56] can be used in analyzing the system in order to validate the reliability of the concurrent system against the deadlocks and other issues. In related studies, the BPMN models are transformed into Petri Nets, so that the tools for analyzing and validating the system can be applied [57–60].

Among the variants of the original Petri Net method, the Colored Petri Nets (CPN) [18,61] is a method with both formality and ease of use. A Petri Net represents the concurrent system as a graph with "places" to store tokens, "transitions" to consume and generate tokens, and "arcs" as the connection between places and transitions. CPN is a type of high-level Petri Net with rules on places, transitions, and arcs. Each token in a CPN can have an attached data value named a "color", which must be defined in a finite "colorset". A CPN can be equivalently unfolded to a low-level Petri Net without colors, which enables the mathematical tools to analyze the CPN. It is known that the extended CPN with an infinite colorset is Turing-complete [62].

Real-world engineering projects and digital twin systems usually have multiple participants, and state changes may occur concurrently. In this paper, the CPN is chosen as the basis of the research, and the following part of the paper will try to integrate the CPN method with the rich semantic information from the engineering domain.

2.3. Interaction between temporal models and semantic web applications

Semantic web technology can provide rich information about domain knowledge and engineering instances, and the interrelated information is essential in supporting the automation of judgments and processes. Related studies have focused on the interaction between temporal models and RDF resources, in order that the knowledge in the semantic web can be accessed to enhance the ability of temporal models. The studies can be classified into the following topics.

Describing the static structure of a temporal model using an ontology. In such studies, ontologies are established to describe the components and structures in the temporal models. The behavior of the temporal models can also be translated into the axioms in the ontologies for supporting reasoning and validation. The described temporal models include FSM [19], BPMN [20,63,64] and Petri Nets [21,65,66]. In such studies, the contents of the ontologies are mainly about the temporal models themselves, but not much domain knowledge is included.

Knowledge-based generation of temporal models. Based on the knowledge graph of a specific domain, a temporal model is generated to perform as a runnable agent for implementing a particular task such as a classifier [22] or a workflow simulator [23,67]. In such studies, domain knowledge is introduced in initializing the temporal models but is no longer referred to in the runtime.

Accessing and querying the temporal states from semantic web applications. In Knowledge-driven FSM [25] and DARPA Agent Markup Language (DAML) [68], both the static structure and the dynamic state changes of a temporal model are written into the RDF graph. In such studies, the rules to drive the temporal model are still disjoint with the RDF graph. But such methods can be used to support decision-making in knowledge-based applications by querying the current state of the temporal model or by querying the candidate adjacent states from the RDF graph.

Involving domain knowledge in the runtime of the temporal models. The Petri Nets over Ontological Graphs (PNOG) [24] allows the tokens to be with a hierarchical classification system. The classification of the tokens is represented in an RDF graph, which can be accessed in the runtime of the Petri Net. The synonyms and hyponyms rules among the tokens are also supported in the PNOG. However, only the classifications of the tokens are supported, but the properties and relationships between the tokens are not included in the model.

A more detailed literature review of research on knowledge sharing between temporal models and semantic models can be found in the Refs. [69].

Compared with the previous studies, the motivation in this paper is to integrate the temporal model with the semantic model, in which the enriched semantic information can be used in validating the state change rules for driving the temporal model, and the temporal states are also in the form of RDF that is accessible to other semantic web applications. The idea is to implement a runnable Petri Net based on a semantic engine by defining the structure of the Petri Net in OWL and representing the state change rules in SPARQL. Since the proposed temporal model is established based on semantic technologies, the two-way sharing of knowledge between domain ontologies and temporal models in the runtime can be realized. The SPARQL statements can be used inside the temporal model as state change rules concerning domain knowledge, and also outside the temporal model as domain queries concerning current temporal states.

3. The semantic Petri net (SPN)

The definition of SPN is provided in Section 3.1. The representation of SPN structure components and rules in OWL and SPARQL are shown in Section 3.2. The implementation of the SPN toolkit is shown in Section 3.3.

3.1. The definition of SPN

The definition of SPN inherits from the CPN, with an external RDF graph **W** providing the relationships between the Petri Net structures and contents.

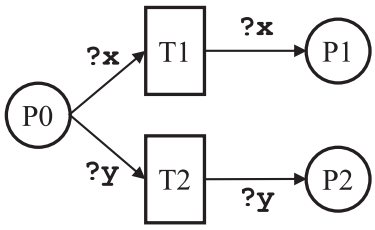
The SPN is defined as a tuple

$$\mathbf{S} = \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{T}, \mathbf{A}, \mathbf{N}, \mathbf{C}, \mathbf{G}, \mathbf{E}, \mathbf{I}; \mathbf{W} \rangle. \quad (1)$$

Σ is the "colorset" containing the token colors allowed in the Petri Net. In SPN, **Σ** is a finite set of RDF terms (including literals and URIs).

P, **T**, **A**, **N** are about the structure of an SPN, and **C**, **G**, **E**, **I** are about the rules in the SPN.

- **P** is the set of places.
- **T** is the set of transitions.
- **A** is the set of arcs.
- **N** is the assignment of each arc to link one place and one transition.
- **C** is the assignment of a subset of allowed colors to each place.



(a) The diagram of an example SPN for dispatching the tokens from "P0" to "P1" and "P2".

```
1    :T1  a spn:Transition;
2         spn:guardRule     :guard_1;
3         spn:hasArg        :arg_x.
4    :T2  a spn:Transition;
5         spn:guardRule     :guard_2;
6         spn:hasArg        :arg_y.
7    :P0  a spn:Place;
8         spn:colorRule     :color_0;
9         spn:initRule      :init_0.
10   :P1  a spn:Place.
11   :P2  a spn:Place.
12   :Ax0 a spn:ArcP2T;
13        spn:relPlace      :P0;
14        spn:relTransition :T1;
15        spn:hasArg        :arg_x.
16   :Ax1 a spn:ArcT2P;
17        spn:relPlace      :P1;
18        spn:relTransition :T1;
19        spn:hasArg        :arg_x.
20   :Ay0 a spn:ArcP2T;
21        spn:relPlace      :P0;
22        spn:relTransition :T2;
23        spn:hasArg        :arg_y.
24   :Ay1 a spn:ArcT2P;
25        spn:relPlace      :P2;
26        spn:relTransition :T2;
27        spn:hasArg        :arg_y.
```

```
28   :color_0 a spn:SPARQLRule;
29            spn:hasSPARQL "ASK {
30              ?TOKEN a ifcowl:IfcDoor }".
31   :init_0  a spn:SPARQLRule;
32            spn:hasSPARQL "SELECT ?TOKEN {
33              ?TOKEN a ifcowl:IfcDoor }".
34   :guard_1 a spn:SPARQLRule;
35            spn:hasSPARQL "ASK {
36              ?x ifcowl:overallHeight_IfcDoor ?h .
37              ?h express:hasDouble ?v .
38              FILTER(?v > 1800)}".
39   :guard_2 a spn:SPARQLRule;
40            spn:hasSPARQL "ASK {
41              ?y ifcowl:overallHeight_IfcDoor ?h .
42              ?h express:hasDouble ?v .
43              FILTER(?v <= 1800)}".
44   :arg_x   a spn:ArgDef;
45            spn:argName   "?x";
46            spn:argType   ifcowl:IfcDoor.
47   :arg_y   a spn:ArgDef;
48            spn:argName   "?y";
49            spn:argType   ifcowl:IfcDoor.
```

(b) The RDF representation of the structures (left) and the rules (right) of the example SPN.

Fig. 1. An example SPN for fetching the "IfcDoor" nodes and dispatching by "overallHeight".

- **G** is the set of guard rules, which assigns each transition with a rule that returns a boolean value, deciding whether this transition is enabled.
- **E** is the set of arc expressions, which assigns each arc with an expression that returns a multiset of tokens, deciding which tokens to consume (for place-to-transition arcs) or to generate (for transition-to-place arcs).
- **I** is the assignment of an initial multiset of tokens to each place.

The unit of the behavior of CPN is a "binding". A transition has a set of arguments shared in the guard rules and arc expressions. A binding is an assignment of values to the arguments, so that the guard rules are evaluated to return a boolean value, and the arc expressions are evaluated to return a multiset of tokens to consume in the input places or to generate in the output places. A transition is enabled when the guard rule returns true and there are enough tokens to consume in every input place. The SPN inherits the binding behavior of the CPN.

Based on the definition of the SPN, the SPN structure components are defined in OWL, and the SPN rules are represented in SPARQL. So SPN structures and rules can be saved and exchanged in several formats of the RDF data, and can be loaded and run on RDF engines such as Apache Jena [26] and dotNetRDF [27]. In the SPARQL grammar, an "ASK" query returns a boolean value, and a "SELECT" query returns RDF tokens, so the two types of SPARQL statements can be used in the guard rules and the arc expressions, respectively.

Fig. 1 provides an example SPN. The "IfcDoor" nodes are fetched from the RDF model and put into the place "P0". Depending on the "overallHeight" attribute of each node, either transition "T1" or "T2" is fired to dispatch the node to the place "P1" or "P2", respectively. In this example case, the guard rules on the transitions are evaluated for each input "IfcDoor" node to accept or reject the node to trigger the transition. This mechanism is useful in real-world scenarios with the guard rules composed according to the state change constraints, so that whether an input token is allowed to trigger the state change can be automatically checked.

Fig. 1(a) is the diagrammatic representation of the example SPN. The places are drawn as circles, the transitions are drawn as boxes, the arcs are drawn as directed arrows, and the tokens are drawn as little dots contained in the places. Fig. 1(b) is its representation in the Terse RDF Triple Language (TTL) [70] format. The left column of Fig. 1(b) is about the SPN structure components.

- Rows 1 to 6 are about two transitions. Each transition has an argument definition, and a guard rule to decide whether this transition is enabled.
- Rows 7 to 11 are about three places, in which the place "P0" is assigned a color rule and an initializing rule.
- Rows 12 to 27 are about four arcs. Each arc links one place and one transition, and has an argument definition.

Table 1
The definition of SPN structure components.

Class	Property	Range	Cardinality	Description
spn:Transition	spn:guardRule	spn:Rule	0:1	A boolean rule deciding whether the transition is enabled.
	spn:hasArg	spn:ArgDef	1:?	Definition of arguments used in guard rules and arc expressions.
spn:Place	ldp:contains	rdfs:Resource	0:?	Contained tokens.
	spn:colorRule	spn:Rule	0:1	A boolean rule deciding whether a token is allowed in the place.
	spn:initRule	spn:Rule	0:1	A rule assigning initial tokens.
spn:Arc	spn:relPlace	spn:Place	1:1	Related place.
	spn:relTransition	spn:Transition	1:1	Related transition.
	spn:arcExpr	spn:Rule	0:1	An arc expression deciding which tokens to consume or to generate.
	spn:hasArg	spn:ArgDef	1:?	Definition of arguments, must be a subset of the arguments of the related transition.

The right column of Fig. 1(b) is about the rules and arguments to drive the SPN.

- Rows 28 to 30 are about a color rule for the place "P0", which has an "ASK" rule in SPARQL that decides whether a token is allowed to be put into the place.
- Rows 31 to 33 are about an initializing rule for the place "P0", which has a "SELECT" rule in SPARQL to fetch the tokens from an external RDF graph into the place on initializing the Petri Net.
- Rows 34 to 43 are about two guard rules for transitions "T1" and "T2", respectively. Each guard rule has an "ASK" rule in SPARQL that decides whether the transition is enabled by a token when the token is assigned to the input argument of the transition.
- Rows 44 to 49 are about two argument definitions. Each argument has a name and a required argument type. The arguments are shared in the guard rules and the arc expressions.

Before the example SPN is initialized, two RDF graphs are loaded in a graph repository. One is the RDF graph for SPN structures and rules shown in Fig. 1(b), and another is the RDF graph about the components in a building model. On initializing the SPN, the tokens of the type "ifcowl:IfcDoor" are fetched from the building model into the place "P0" according to the initializing rule. When the SPN starts to execute, the system keeps accepting argument bindings and evaluating guard rules for each transition. When a transition is enabled, it may be "fired" to consume some tokens from the input places, and to generate some tokens to the output places. The guard rules of the two transitions are evaluated as filters to dispatch the tokens according to the value of the attribute "ifcowl:overallHeight_IfcDoor". Finally, the system may come to an end state when no more available bindings in the system can enable the transition.

3.2. Representation of SPN structures and rules in OWL and SPARQL

The components of the SPN structure are represented as an RDF graph, and the rules to drive the SPN are written in SPARQL queries and attached to the SPN structure nodes. The SPN ontology is established to define the classes and properties for representing the SPN structures and rules, with the namespace "spn:".

The classes and properties for representing SPN structure components are listed in Table 1.

Transitions. A transition is of the class "spn:Transition". A transition may have several arguments with the attribute "spn:hasArg", and each argument is an "spn:ArgDef" node with an argument name and a type definition on the argument. The arguments are shared in the guard rule of the transition, and the arc expressions of the input and output arcs of the transition. A transition may have a guard rule by "spn:guardRule", for checking whether the transition is enabled when a binding is provided. If the guard rule is empty, the transition is always enabled by default as long as enough input arguments are provided and enough tokens are in the input places.

Places. The SPN places are implemented based on the Linked Data Platform (LDP) standard [71]. LDP is recommended by W3C for maintaining dynamic relationships in linked data, with the definition of data node containers and the HTTP requests for manipulating the dynamic linked data. The "spn:Place" is defined as a subclass of "ldp:Container", so the contained tokens can be linked to the place with "ldp:contains". The LDP standard defines the range of "ldp:contains" as "rdfs:Resource", which means that any RDF term (including literals and URIs) can be placed into the container as a token. A place may have a color rule with "spn:colorRule", which returns a boolean value deciding whether a token is allowed in the place. A place may have an init rule with the relationship "spn:initRule", which defines a rule to fetch the initial tokens from another RDF graph.

Arcs. The "spn:Arc" is the abstract superclass for arcs, which must relate one place with "spn:relPlace" and one transition with "spn:relTransition". The "spn:Arc" has two subclasses:

- "spn:ArcT2P" is for arcs pointing from a transition to a place;
- "spn:ArcP2T" is for arcs pointing from a place to a transition.

An arc must have at least one argument with "spn:hasArg", and all arguments must also be assigned to the related transition. An arc may have an arc expression by "spn:arcExpr", which is calculated for obtaining a multiset of tokens according to the arguments. When the arc expression is null, it directly passes the tokens that are assigned to the arguments.

The SPN rules are closely related to the SPN structure components and are also defined as rule nodes in the SPN ontology. An SPN rule is formed as a tree structure in the RDF graph and is attached to an SPN component node. The SPN rule structure allows linking several relatively small SPARQL queries together to form a compound rule, which is helpful in debugging and maintenance in the software implementation of the SPN.

The classes and properties for representing SPN rules are listed in Table 2. The "spn:Rule" is the abstract superclass of all SPN rule nodes, which has the following subclasses:

- The "spn:SPARQLRule" contains a SPARQL statement.

Table 2
The definition of SPN rule components.

Class	Property	Range	Cardinality	Description
spn:SPARQLRule	spn:hasSPARQL	xsd:string	1:1	The SPARQL statement of the rule.
spn:ConstantRule	spn:hasValue	rdfs:Resource	1:1	The constant value of the rule.
spn:ArgRule	spn:hasArg	spn:ArgDef	1:1	The argument to be returned by the rule.
spn:CompoundRule	spn:operator	xsd:string	1:1	The logical operator. The allowed values are AND, OR, XOR, and NOT.
	spn:subRule	spn:Rule	1:?	The sub-rules connected by operator.
spn:ConditionRule	spn:if	spn:Rule	1:1	The "if" condition rule, which must return a boolean value.
	spn:then	spn:Rule	1:1	The "then" condition rule when "if" condition returns true.
	spn:else	spn:Rule	1:1	The "else" condition rule when "if" condition returns false.
spn:ArgDef	spn:argName	xsd:string	1:1	The argument name.
	spn:argType	rdfs:Class	0:?	Allowed argument types in a binding.

- The "spn:ConstantRule" contains a constant token or a multiset of tokens. Each token can be a URI or a literal value.
- The "spn:ArgRule" contains an argument definition. The rule returns the value that bound on the argument.
- The "spn:CompoundRule" contains multiple sub-rules connected by a logical operator (AND, OR, XOR, or NOT), and each sub-rule returns a boolean value. Specifically, there can be only one sub-rule with a NOT operator.
- The "spn:ConditionRule" contains an "if" sub-rule that returns a boolean value, and with the true and false conditions returned by the "if" sub-rule, the "then" and "else" sub-rules are evaluated, respectively.

The rule nodes compose a rule tree structure, in which each leaf node should be one of "spn:SPARQLRule", "spn: ArgRule" and "spn:ConstantRule". The SPARQL statements can be "ASK" queries returning a boolean value, or "SELECT" queries returning the queried tokens from the RDF graph.

The "spn:ArgDef" is the class for defining the name and allowed types of an argument, which can be assigned to the "spn:hasArg" property of transitions and arcs. The name of the argument is used as the input variable in the SPARQL statements. The argument type is allowed to be null, which indicates that the argument accepts any type of tokens.

3.3. The implementation of the SPN tool

The software structure of the implemented SPN tool is shown in Fig. 2. The descriptions of the modules are listed as follows.

RDF storage. In this paper, the dotNetRDF [27] toolkit is used to maintain the domain RDF graphs. The following RDF graphs are stored together in a repository supporting SPARQL queries.

- SPN graph: Containing the SPN structure components and the SPN rules.
- IFC schema graph: Containing an ontology for the IFC schema.
- IFC instance graph: Containing the IFC instances with properties and relationships.

LDP controller. The LDP controller is developed based on the Linked Data Platform specification [71], which maintains the containment of tokens in the LDP containers and has the operations to read and manipulate the containers. For the SPN places, an extension is added to support the multiplication of the same token in a place. If an identical token is stored multiple times in a place, an additional property "spn:multi_i" (in which "i" is an integer greater than 1) is added between the place and the token in addition to the "ldp:contains".

SPARQL engine. The SPARQL engine module queries the data in all the RDF graphs in the RDF storage, supporting the SPN runner to access the schema and instance data and check the rule constraints. The SPARQL engine module is based on the functions of the dotNetRDF toolkit [27]. On initializing the SPN and in the runtime of the SPN, the SPN runner keeps sending the SPARQL query strings, the current tokens, and the argument bindings to the SPARQL engine module. The SPARQL engine module injects the tokens and the argument bindings into the SPARQL query strings with "BIND" or "VALUES" clauses. Then the module executes the query to get the results to drive the SPN.

SPN runner. The SPN runner is the core module for the execution of the SPN. Supported by the modules mentioned above, the SPN runner has the following functions to run the simulation of the SPN.

- Initializing: Performing the initializing rule of each place to fetch RDF tokens into the places.
- Transition registration: Registering the transitions to the runner so they can be automatically triggered in the main loop of the runner. The user can choose to register all transitions or keep part of the transitions under the manual control of the user.
- Binding finder: Finding the enabled bindings for all transitions as candidates for firing the transitions and refreshing the candidates when the state of the system is changed.

With the functions mentioned above, The main loop keeps finding the bindings, checking the guard rules, and triggering the registered transitions. The main loop stops when no more enabled bindings are found in the system.

Interface. The interface has various connections between the SPN model and the external data, applications, and users. The following functions are included in the interface:

- BIM loader: Loading the BIM data model in an original format (such as IFC) and extracting instances, properties, and relationships from the original data.
- Events: Event handlers can be added to the transitions for events like "triggered" or "binding rejected". Such event handlers can be used for interaction with external applications.

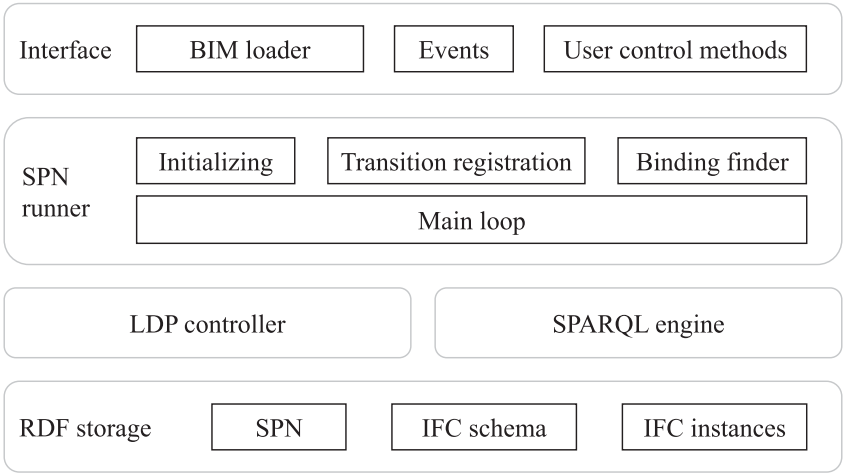


Fig. 2. The software structure of the SPN tool.

- User control methods: Public methods are used for manual control of the SPN, such as assigning a user-defined global argument and triggering a transition with a user-specified binding.

In the definition of Petri Net, each place contains an unordered multiset of tokens, and the transitions may be fired concurrently, so the SPN runner involves random behaviors. The different orders of triggering the transitions and the random selection of tokens when multiple tokens can enable the guard rule may lead to different results. The user may provide a sequence of bindings as an input for the SPN, and the SPN runner may also automatically run to search for a random sequence of states of the system. All the possible states form the "state space" of the Petri Net [18].

4. Discussions on analyzing the SPN

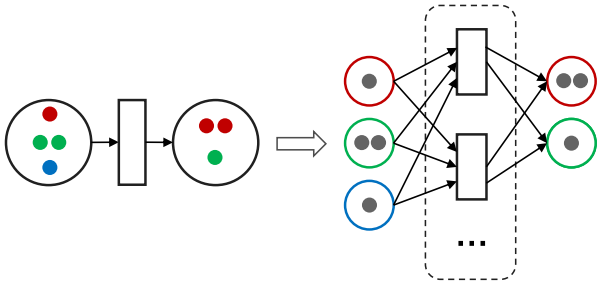
In this section, the issues for supporting the analysis of concurrent systems modeled in the SPN are discussed. In Section 4.1, the downward compatibility of SPN is discussed to ensure that the existing methods for analyzing low-level Petri Nets can also be applied to the SPN. In Section 4.2, the method to find the adjacent states according to a current state in the state space of the SPN is discussed for supporting simulation and analysis in the state space of the SPN.

4.1. The downward compatibility of SPN

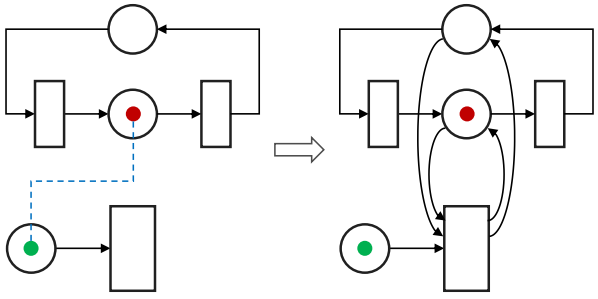
Compared with the variant methods, the original low-level Petri Net has only non-colored tokens and simple number constraints on the arcs. But a low-level Petri Net is mathematically simple and easy to be analyzed. The high-level Petri Nets (such as the CPN [18]) have more complex tokens and rules but are usually more friendly to engineers in modeling a real concurrent system.

The high-level Petri Nets keep the downward compatibility with the low-level Petri Nets, which means that a high-level Petri Net like CPN can be converted to an equivalent low-level Petri Net with "unfolding". Such equivalence is essential for ensuring that the complex high-level Petri Nets do not exceed the scope of the capability of the low-level Petri Nets, so that the methods for analyzing low-level Petri Nets can also be applied to the high-level Petri Nets.

In CPN, the tokens are unrelated, and the rules are static, so the actions in a far-away transition do not change the behavior of a local transition, which is essential for analyzing concurrent systems. Compared with the CPN, the SPN involves an RDF graph for providing the relationships between the tokens, and SPARQL rules are introduced for representing domain constraints. Due to the latent relationships between the tokens from the domain RDF graph, in SPN, the behavior



(a) Unfolding a CPN to non-colored Petri Net.



(b) Unfolding an SPN with latent connections to CPN.

Fig. 3. Examples of unfolding CPN and SPN.

of a local transition may be changed by the actions in a far-away transition. The benefit of introducing semantic relationships is that the structure of the SPN can be simplified without explicitly representing the coupled states. Nevertheless, the discussion on downward compatibility is necessary for ensuring that SPN does not exceed the scope of the capability of the CPN, and that the analysis methods for low-level Petri Nets are still applicable to SPN.

The conversion of the SPN to an equivalent CPN can be inspired by unfolding a CPN to lower-level Petri Nets. Fig. 3(a) shows an example unfolding of a CPN. A colored place is represented as multiple non-colored places corresponding to each allowed color of this place. A transition with conditions can be represented as multiple sub-transitions dealing with each condition, which can always be established by listing all possible bindings due to the "finite colorset".

In this section, a three-step unfolding process for SPN is proposed for obtaining an equivalent CPN, as shown in Fig. 3(b).

Step 1: Putting all variables of the domain RDF graph inside the SPN. This step ensures a "constant domain RDF graph" condition, in which the latent connections in the domain RDF graph are not changeable by external editing operations. This step can be done by representing all variable items in the domain RDF graph as LDP containers, and representing the editing operations as transitions connected to the containers, since the LDP method is recommended by W3C for maintaining variable members and relationships in an RDF graph [71].

Step 2: Splitting latent semantic connections into explicit arcs. The SPARQL rules in SPN may concern latent connected remote tokens. In the "constant domain RDF graph" condition, for each SPARQL rule, a finite set of concerned remote tokens can be listed, and then a finite set of concerned remote places which may contain such tokens can be listed. As a result, the latent semantic connections can be split into explicit arcs. For each remote place, a pair of arcs are added to fetch the remote tokens and send them back to keep the remote place unchanged.

Step 3: Implementing SPARQL rules in the CPN. Similar to the unfolding of CPN, due to the "finite colorset", the SPARQL rules can always be equivalently implemented in CPN, at least by listing all possible conditions of the bindings.

The three-step unfolding process shows the existence of an equivalent CPN for an SPN, which ensures that the analysis methods for CPN can also be applied to SPN. However, the unfolding is usually not necessarily performed before simulating or analyzing a high-level Petri Net. Since a low-level Petri Net has simple rules, the number of places and transitions may be much larger to represent an equivalent high-level Petri Net. There are also related studies about simplified unfolding methods [72] rather than listing the bindings, which is out of the scope of this paper. So in the rest of this section, the method of directly finding the adjacent states in the state space of the SPN is discussed.

4.2. Finding the adjacent states in the state space of the SPN

4.2.1. The reachability graph of Petri nets

The state space of a Petri Net is the representation of all possible states and state changes of the Petri Net. The state space is represented as a directed graph $\mathbf{R} = \langle \mathbf{M}, \mathbf{Y} \rangle$ named a reachability graph or a coverage graph, in which each node $\mathbf{m} \in \mathbf{M}$ is named a "marking", and each edge $\mathbf{y} \in \mathbf{Y}$ is named a "step". A marking $\mathbf{m}$ is a state of the whole Petri Net, which records the number of different tokens in every place at this moment. A step is an edge in the reachability graph. By triggering a transition t with the arguments in a binding $\mathbf{b}$, the marking $\mathbf{m}$ is changed to a new marking $\mathbf{m}'$, which is recorded as a step $\mathbf{y} = \langle \mathbf{m}, \langle t, \mathbf{b} \rangle, \mathbf{m}' \rangle$ in the reachability graph. The pair $\langle t, \mathbf{b} \rangle$ is named a "binding element", which is an assignment of binding arguments to a transition.

The state space analysis is a powerful tool in analyzing the Petri Net. The reachability graph can be used to validate the features of the Petri Net [18], such as the reachability between two markings, the existence of a deadlock, and the ability to return to a home marking, etc.

The core process in state space analysis is to find the steps to change a current marking to the adjacent markings. Each step corresponds to an "enabled binding element", which is an assignment of binding arguments that enables a transition, i.e., the guard rule is satisfied, and the input places have sufficient tokens to be consumed. The reachability graph can be generated by iteratively searching for the enabled binding elements. The found binding elements are also useful in the simulation of the SPN for providing the simulator with a list of candidate transitions to be triggered next.

A straightforward method to find the enabled bindings on a transition can be described as follows.

(a) For each argument of the transition, get a finite token set according to the range of the argument.
(b) List all possible combinations of the arguments, which forms an original set of candidate bindings.
(c) Check the guard rule for obtaining a subset of the bindings with a TRUE result.
(d) For each binding, evaluate the arc expression on each input arc of the transition, for obtaining a multiset of tokens to be consumed.
(e) Remove the binding if any of the corresponding input places have insufficient tokens to be consumed.

This straightforward method has a shortcoming in that the number of original bindings may be too large, and the checking through the input arcs and places may also be time-consuming. In related research on CPN, a method is proposed to efficiently find the binding elements by inferring the argument assignments according to the tokens in the input places, and to get the combination of argument assignments by matching the patterns between several arc expressions [73].

For SPN, since the SPARQL engine is powerful in finding the argument patterns from the RDF graph, it is possible to find the bindings by generating and performing SPARQL queries (see Section 4.2.2). While in SPN, the rules may be in a condition tree composed of "if-then-else" rules, which needs to be considered in composing the query to find the bindings (see Section 4.2.3).

4.2.2. Finding the enabled bindings for the SPN transitions

In SPN, each argument may have a type definition, which corresponds to a clause in a SPARQL query statement for finding a collection of tokens. By composing a query with the type definitions, an original binding set can be found in the RDF graph. Considering that the guard rules and arc expressions are also in SPARQL, the main idea in this section is to add more constraints to the query so that the number of the candidate bindings can be reduced.

In this section, Algorithm 1 is proposed for finding the enabled bindings for an SPN transition by performing integrated SPARQL queries.

The guard rule is usually an "ASK" statement, which returns a boolean result for a binding of the arguments. In row 4 of Algorithm 1, the guard rule is converted from the "ASK" statement into a "SELECT" statement, and the rule segment h_0 is composed by connecting the guard rule with the definition rules of the arguments, which is used for ensuring a binding that conforms to the guard rule and the argument types. In row 4, the symbols "+" and "$\sum$" represents the connection of rules, which can be performed either by joining the SPARQL clauses into one SPARQL statement, or by executing the queries in sequence with substituting the previous results.

For the arc expressions, the idea is to find the "inverse expression". An arc expression is a function that maps a binding to a multiset of tokens to be consumed in the input place. An inverse expression for an arc expression is a function that maps the content of the input place to a set of possible bindings.

An arc expression f and its inverse expression h can be defined as the mappings shown in Eqs. (2) and (3), respectively. In the definitions, $\mathbf{c} = \{c_1, \ldots, c_{|\mathbf{c}|}\}$ is a color set with different tokens, $\langle c_1 : n_1, \ldots, c_{|\mathbf{c}|} : n_{|\mathbf{c}|} \rangle$ is a multiset which records the number n_i of each token c_i, and $\mathbf{b} = \langle a_1 : v_1, \ldots, a_{|\mathbf{b}|} : v_{|\mathbf{b}|} \rangle$ is a binding which assigns a value v_i to each argument a_i.

$$f: \quad \mathbf{b} = \langle a_1 : v_1, \ldots, a_{|\mathbf{b}|} : v_{|\mathbf{b}|} \rangle \rightarrow \mathbf{s} = \langle c_1 : n_1, \ldots, c_{|\mathbf{c}|} : n_{|\mathbf{c}|} \rangle \tag{2}$$

$$h: \quad \mathbf{s} = \langle c_1 : n_1, \ldots, c_{|\mathbf{c}|} : n_{|\mathbf{c}|} \rangle \rightarrow \mathbf{B} = \{\langle a_1 : v_1, \ldots, a_{|\mathbf{b}|} : v_{|\mathbf{b}|} \rangle, \ldots\} \tag{3}$$

If an arc expression is in a simple (but commonly used) form, the inverse expression can be automatically generated. In our implementation, the following cases are handled.

(a) If an arc expression returns a constant value irrelevant to the arguments, it cannot provide any constraint for the argument assignments. Such expression is ignored so that the inverse expression only keeps the basic type constraints for the arguments.

Algorithm 1 FindEnabledBindings

Input: transition t, marking $\mathbf{m}$, RDF graph $\mathbf{W}$
1: $g \leftarrow$ GuardRuleOf(t)
2: $\mathbf{e} \leftarrow$ InputArcsOf(t)
3: $\mathbf{a} \leftarrow$ ArgumentsOf(t)
4: $h_0 \leftarrow$ ToSelectRule(g) + $\sum_{a \in \mathbf{a}}$ DefinitionRuleOf(a)
5: $\mathbf{h} \leftarrow \varnothing$
6: $\mathbf{e}_{\text{skip}} \leftarrow \varnothing$
7: **for** arc $e \in \mathbf{e}$ **do**
8: $f \leftarrow$ ArcExpressionOf(e)
9: $h \leftarrow$ TryInverse(f)
10: **if** $h \neq$ NULL **then**
11: $\mathbf{h} \leftarrow \mathbf{h} \cup \{h\}$
12: **else**
13: $\mathbf{e}_{\text{skip}} \leftarrow \mathbf{e}_{\text{skip}} \cup \{e\}$
14: **end if**
15: **end for**
16: bindings $\mathbf{B} \leftarrow \varnothing$
17: **for** rule $h \in \mathbf{h}$ **do**
18: $\mathbf{B} \leftarrow \mathbf{B} \cup$ ProcessQuery($h_0 + h, \mathbf{m}, \mathbf{W}$)
19: **end for**
20: **for** arc $e \in \mathbf{e}_{\text{skip}}$ **do**
21: $f \leftarrow$ ArcExpressionOf(e)
22: $p \leftarrow$ PlaceOf(e)
23: **for** binding $\mathbf{b} \in \mathbf{B}$ **do**
24: multiset $\mathbf{s} \leftarrow f(\mathbf{b})$
25: **if** NOT $\mathbf{s} \subseteq \mathbf{m}(p)$ **then**
26: $\mathbf{B} \leftarrow \mathbf{B} - \{\mathbf{b}\}$
27: **end if**
28: **end for**
29: **end for**
Output: enabled bindings $\mathbf{B}$

(b) If an arc expression is with a single argument, and returns the argument itself (or a collection of itself), then the inverse expression is a query for this token in the input place. When a token matches the argument type, and the number of the token in the place is greater than the multiplier, then this token is returned as a possible assignment to the argument.
(c) If an arc expression is with several arguments and returns a multiset of the arguments, then for each argument, a partial inverse expression is generated according to case (b). The full inverse expression is the connection of the partial inverse expressions. If the arc expression is a SPARQL rule, the partial clauses can be joined to form the SPARQL statement of the inverse expression.
(d) If an arc expression does not directly return the input arguments, but returns the tokens in the place that are linked with the input arguments, in the inverse expression, the links are used to find the possible assignments of the arguments according to the tokens in the place. Such an arc expression is usually a SPARQL rule, and the clauses representing the links can be copied into the SPARQL statement of the inverse expression.

In Algorithm 1, from row 7 to row 15, the loop is to find the inverse of the arc expression on each input arc. The function "TryInverse" is the sub-process for finding the inverse expression for an arc expression, in which the above-mentioned cases for finding inverse expressions are handled. Then from row 17 to row 19, each found inverse expression is connected with the rule h_0 to form a query, and the query is processed on the current marking $\mathbf{m}$ to obtain a set of candidate bindings. If the SPARQL statement of an arc expression is complex that the inverse expression cannot be automatically generated yet, it can be skipped for the moment, and the straightforward method to check the sufficiency of tokens can be performed later, as shown in the loop from row 20 to row 29.

4.2.3. Finding the inverse of a rule with condition tree

In SPN, there are rules in a condition tree structure. In an arc expression with conditions, the sub-rule assigned to the "if" branch returns a boolean result for a binding of the arguments. According to the boolean result, the "then" branch and the "else" branch are evaluated, respectively. The condition tree can be with several layers, and finally, each leaf node returns a multiset of tokens.

The condition tree can be split into several paths from the root to the leaf nodes. On each path, the branch nodes are several boolean rule nodes, and the final leaf node will return a multiset of tokens when all the previous boolean nodes return "TRUE". In this way, let f be an arc expression with a condition tree structure, and it can be split into a set of rule pairs, as shown in Eq. (4), in which "α_i"s are the conjunction of all branch rule nodes, and "β_i"s are the corresponding leaf nodes.

$$\text{Split}(f) = \{\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\} \tag{4}$$

On each split path, with an input binding $\mathbf{b}$, the part of the function can be represented as "f_i" in Eq. (5).

$$f_i(\mathbf{b}) = \begin{cases} \beta_i(\mathbf{b}) & \text{when} \quad \alpha_i(\mathbf{b}) = \text{TRUE} \\ \varnothing & \text{when} \quad \alpha_i(\mathbf{b}) = \text{FALSE} \end{cases} \tag{5}$$

The conditions $\{\alpha_1, \ldots, \alpha_k\}$ are exclusive to each other, and for each binding $\mathbf{b}$, at most one of the "f_i"s returns a non-empty multiset. As a result, the function f can be defined as the union of the function "f_i"s.

In finding the inverse of f, the inverses of the leaf rules are found first. Let ω_i be the inverse of β_i, then the inverse of f_i is the function h_i in Eq. (6).

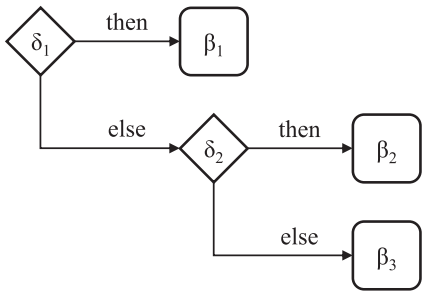
$$h_i(\mathbf{s}) = \{\mathbf{b} \mid \mathbf{b} \in \omega_i(\mathbf{s}) \ \wedge \ \alpha_i(\mathbf{b}) = \text{TRUE}\} \tag{6}$$

Function h_i calculates a set of bindings from a multiset of tokens $\mathbf{s}$ (which is the content of the input place) and keeps a subset of the bindings which can make α_i return "TRUE".

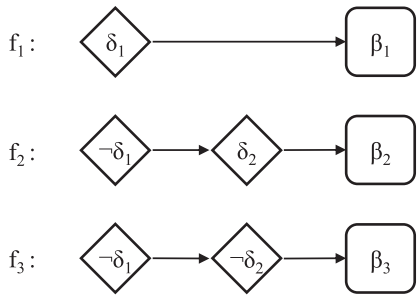
Since the "α_i"s are exclusive, the results of "$h_i(\mathbf{s})$"s are also exclusive. So let h be the inverse of f, it returns the union of the results of "h_i"s, i.e. $h(\mathbf{s}) = \bigcup_i h_i(\mathbf{s})$.

In composing each h_i in SPN rule nodes, it is considered that in the corresponding rule pair $\langle \alpha_i, \beta_i \rangle$ for composing f_i, each rule node in the path of α_i is usually an "ASK" statement in SPARQL (or the logical combination of several "ASK" statements). Each rule node in α_i uses a binding as input and returns a boolean result as a filter for the input binding. According to Eq. (6), in composing h_i, the function ω_i is first performed to find a set of bindings, then the filters are evaluated later to keep a subset of the bindings. So the h_i can also be composed as a chain of rule nodes, which starts with the ω_i node (usually a "SELECT" statement) for obtaining a set of bindings, and then followed by a chain of nodes in the corresponding α_i rule for filtering the bindings. Each node in an α_i rule is converted from an "ASK" statement into a "SELECT" statement.

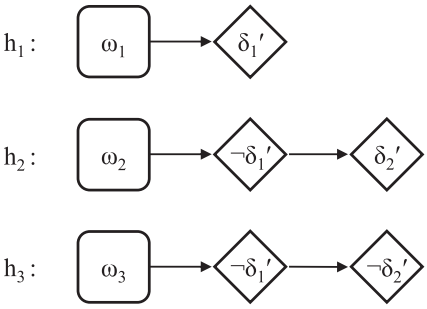
Fig. 4 shows an example of composing the inverse of an arc expression f with condition tree structure, in which the arrows represent the order to check the rule nodes. The expression f is presented in Fig. 4(a), in which the δ_1 and δ_2 are condition rule nodes in the "spn:if" positions, the β_1, β_2 and β_3 are leaf rule nodes returning a multiset of tokens. The split paths of f are shown in Fig. 4(b), which can be written as $\text{Split}(f) = \{\langle \delta_1, \beta_1 \rangle, \langle \neg\delta_1 \wedge \delta_2, \beta_2 \rangle, \langle \neg\delta_1 \wedge \neg\delta_2, \beta_3 \rangle\}$. Let ω_1, ω_2 and ω_3 be the inverses of $\beta_1, \beta_2, \beta_3$, respectively. Let δ_1' and δ_2' be the "SELECT" statements converted from δ_1 and δ_2, respectively. Then in Fig. 4(c), the composed inverse of each path starts with the corresponding ω_i rule, and the converted "SELECT" rules are appended for filtering the bindings queried by the ω_i rule.



(a) The structure of an arc expression f.



(b) The split paths of f.



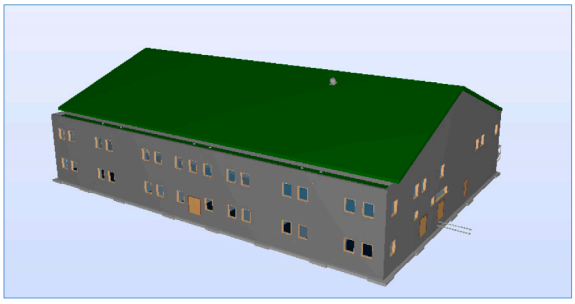
(c) The composed inverse for each path, where δ_i' is the converted "SELECT" statement of δ_i.

Fig. 4. An example of composing the inverse of an arc expression with tree structure.

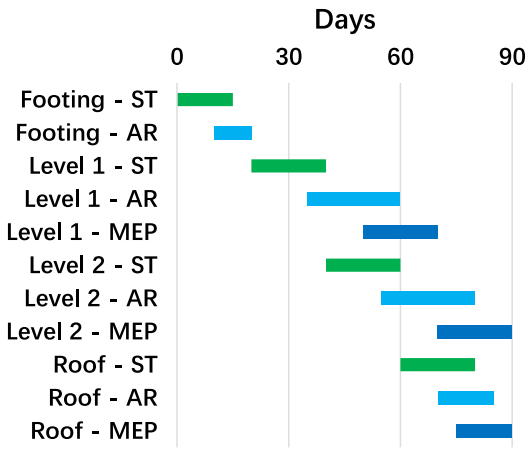
5. Application case

In this section, a case is provided to demonstrate the application of SPN in modeling state changes with semantic constraints in the construction industry. The case is about simulating the construction process of a building project, in which the state change of a building element may be constrained by the state of its host building element or its related spatial structure.

The construction process of an asset can be represented as a spatio-temporal model in which each building element changes the state from "uninstalled" to "installed". There are various dependency rules for deciding whether a state change is allowed, such as the amount of material and tools, the constraint on periods, the state of host objects and permission files. The state change dependencies may involve the properties of the building elements and the relationships between building elements. The enriched domain RDF graph can provide much information about the building elements, spaces, roles, and linked external documents. By modeling the state changes with an SPN structure, the dependency rules can be composed for involving the information from the domain RDF graph and supporting automatic checking and simulation for the construction process.



(a) The sample model.



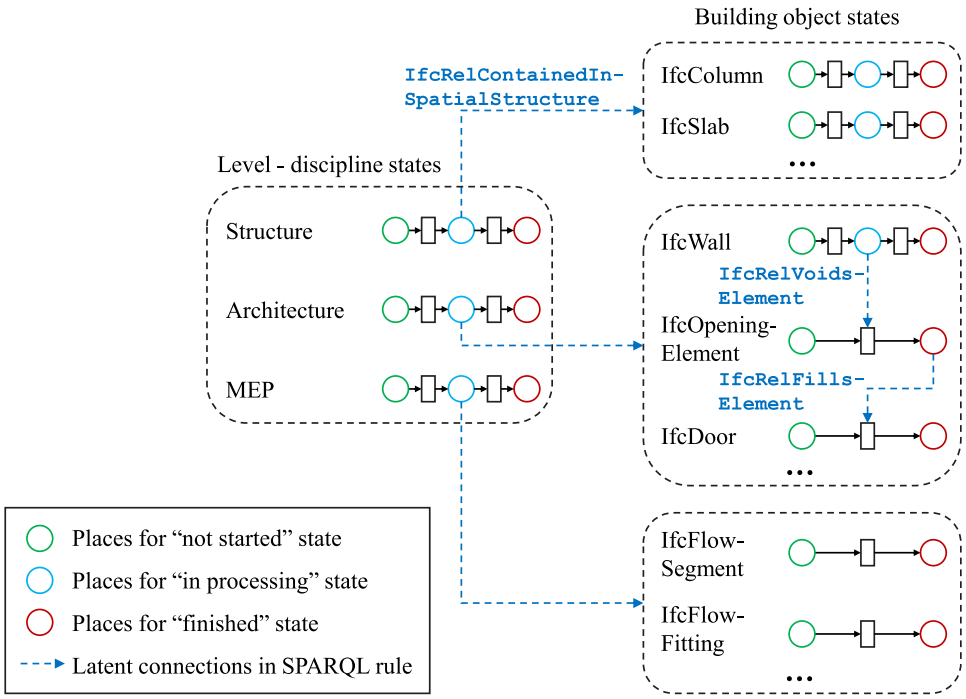
(b) The input schedule data for 4 levels and 3 disciplines.

Fig. 5. The input data in the SPN use case for modeling the construction process.

A sample model of an office building [74] is used in the experiment, as shown in Fig. 5(a). The input schedule data is about the starting and ending timestamps for the construction phases of the levels, in which each phase corresponds to a discipline, including structure (ST), architecture (AR), and mechanical-electrical-plumbing (MEP), as shown in Fig. 5(b). The input model is in IFC format. The building elements, properties, and relationships used in the application case are extracted from the model to form an RDF graph. The RDF graph can be in the ifcOWL schema, or some other RDF schemas that is suitable to represent the building elements and relationships. In this case, a simplified IFC4 schema in RDFS is used, which involves the classification of entities and the attribute names for the entities.

The SPN structure is set up for modeling the state changes in the construction process for each type of building object, as shown in Fig. 6(a). The states are represented as a series of places, and the dependencies are represented as the guard rules of the transitions. Small building components have two states (uninstalled and installed), and large components like walls and slabs have three states (not-started, in-processing, and finished). In the state change rules for the building elements, the concerned latent semantic connections between the building elements and spaces are marked as dashed arrows. The building elements are contained in the building levels with the "IfcRelContainedInSpatialStructure" relationships, and the levels are linked with the input schedules. The relationships between building elements are also involved. For example, the openings must be installed during the construction of the host walls, and the installation of windows and doors must be after the finish of the host openings and walls.

Fig. 6(b) shows the RDF representation of an example transition that ends the structure construction phase of levels. Rows 1 to 4 are the representation of this transition. The transition has an argument named "?level" shown in rows 5 to 6, which can be assigned a token from



(a) The SPN structure for modeling the states of level-disciplines and several object types in different disciplines.

```
1    :T_Level_End_Struct    a                  spn:Transition;
2                           :disciplineTag     "STRUCT";
3                           spn:guardRule      :cprule_0;
4                           spn:hasArg         :arg_0.
5    :arg_0         a                  spn:ArgDef;
6                   spn:argName        "?level".
7    :cprule_0      a                  spn:CompoundRule;
8                   spn:operator       "NOT";
9                   spn:subRule        :sprule_1.
10   :sprule_1      a                  spn:SPARQLRule;
11                  spn:hasSPARQL      "ASK
12       {
13           ?place a spn:Place.
14           ?place :disciplineTag ?dTag1.
15           ?SELF  :disciplineTag ?dTag2.
16           ?place :stateTag ?sTag.
17           FILTER (?dTag1 = ?dTag2 && ?sTag != 'END')
18           ?level ifc4:containsElements/ifc4:relatedElements ?elem.
19           ?place ldp:contains ?elem.
20       }".
```

(b) The RDF representation of an example transition "ending the structure construction phase of a level".

Fig. 6. The SPN model for the construction process.

the input place. Rows 7 to 9 are the guard rule node of the transition, which is the negation of another SPARQL rule shown in rows 10 to 20. The SPARQL clauses in rows 13 to 17 find the places with the same discipline tag as the transition ("?SELF"), and the state tags of the places are not "END". The SPARQL clauses in rows 18 to 19 find the elements related to the level token in the IFC data, and the elements are also contained in the places without an "END" state tag. If such elements are found in the query result, the rule ":sprule_1" returns true, and then the negation rule ":cprule_0" returns false, which means that the transition is not allowed to fire, and the structure construction phase of the level is not allowed to end.

On top of all the transitions and places for each building element type, there is a place recording the date value. The input schedule is for the combinations of level-discipline, and the schedule is represented as guard rules of the corresponding transitions. The guard rules require that the level-discipline token is not allowed to move to the next place until the date number is greater than the given value.

The construction process is usually constrained by several other issues. For example, the upper limit of concurrency may be constrained by the number of machines, the maximum operations per day may be constrained by the number of workers, and the minimum days required for a procedure may be according to the feature of the materials. In our experiment, such additional constraints are also included in the SPN rules.

- The rule of the upper limit of concurrency is represented in the guard rule considering the maximum number of tokens in the output place, which stands for the maximum number of building elements that can be in the same state in a moment.

- A counter place is set for recording the number of triggering of a transition on each day, and the counter place is checked in the guard rules for those transitions with the maximum operations constraint.
- A building element token is attached with a date tag for recording the time of the last state change, and the transition for moving the token out of the current place will check the date tag to ensure the minimum timespan staying in the current state.

On each day of the simulation, the SPN runner keeps finding the enabled binding elements and firing the transitions. Once a transition is fired, its number of fires is updated in the counter place. A transition keeps being fired on this day until there are no more tokens in the input place to pass the guard rule or it exceeds the maximum number of fires. The runner stops when no more enabled transitions exist in the SPN. Then the date number is updated, and the counter place is refreshed to be prepared for the next day.

The SPN structure in this experiment has 70 places and 43 transitions, and there are in total 135 rules in the SPN. On initializing the model, 4 levels and 7,169 building objects from the IFC model are dispatched to the starting places as tokens. With the input schedule and the constraints represented as the guard rules, the application runs to find a sequence of state changes that satisfies all the constraints. The simulation outputs a result of pass (all tokens move to the end places successfully) or not pass (the system stops with some of the tokens unable to move to the end places). The result shows whether the input schedule conflicts with the constraints for state changes.

The experiment is performed on a PC with a processor in 3.19 GHz and 64 GB of physical memory. In a recorded experimental result, with a proper configuration of the constraints, the 90-day simulation finishes in 250 s. During the simulation, the rules are checked 440,200 times, and the transitions are triggered 8,235 times in total to move the tokens into the end places. If the configuration of the constraint is improper, the simulation will stop on a certain day when the construction phase is unable to finish.

The record of the simulation can be outputted as one possible sequence of state changes of the Petri Net. Since the simulation of SPN has random behaviors, in a future case, when the constraints get more strict, only some of the possible sequences in the state space might pass the constraints. The current SPN tool has the ability to validate one sequence of state changes, and it also has the ability to find adjacent binding elements according to a current state. With such abilities, it would be an interest of future study to scan the state space of the system, in order to find a proper (or even optimal) sequence of binding elements, which can be output as a fine-grained schedule for every element.

6. Conclusion and future work

In this paper, the SPN is proposed as a novel method for modeling and validating temporal states directly based on OWL and SPARQL, which realizes the two-way sharing of knowledge between domain RDF graphs and temporal models in the runtime. The structure components of the SPN temporal model are represented as RDF graphs, and the state change rules are represented as SPARQL queries. The SPN runner is implemented based on the functions of common RDF repositories and SPARQL endpoints, which supports the integrated checking of state change rules involving both temporal states and semantic knowledge. The application case demonstrates the ability of SPN to model a process map, in which the state change rules are assigned to different collections of building elements, and such collections can be defined by RDF types or flexible SPARQL queries.

In our simulation, the transitions are triggered automatically on finding the enabled bindings. Considering a real digital twin application, with the interface connecting the SPN with other information systems, the transitions can be triggered by human commands or by sensor events. The guard rules are checked to ensure that the state changes are allowed, and the state change events may also interact with other information systems. The list of found enabled binding elements can also be used to provide recommendations on the next transitions according to a current state.

However, the implementation of SPN in this paper is only a rough attempt at integrating temporal models and semantic web applications. There are still limitations in the current implementation, and further study is needed to fulfill the ability of the Petri Net to model and analyze the temporal states in domain-specific applications. First, the current implementation of SPN supports validating one input sequence of binding elements or randomly selecting enabled binding elements for simulation. The algorithms to search the state space and to find possible or optimal sequences of state changes are still to be studied. Second, the discussions on the downward compatibility of SPN and the method of finding enabled bindings are still rough. Theoretical studies on the problems are still needed, in order that the mathematical tools can be introduced to analyze the behavior of a complex concurrent system modeled in SPN [18]. Third, in the experiments in this paper, the SPN structures and rules are composed in programming scripts, and a more user-friendly visual tool is needed to set up the process model and constraints, load the domain data, and perform the applications. Finally, the SPN method proposed in this paper is defined based on the LDP [71] and SPARQL [7], which is with flexibility to accept various RDF data, but lacks sufficient constraints for supporting decidable inferencing. Further discussions about the constraints on the SPN to support applications with inferencing can be a topic for future research.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The codes and data are provided in GitHub repository: https://github.com/highan911/CBIMS.SPN.

Acknowledgments

This work was supported in part by the 2019 MIIT Industrial Internet Innovation and Development Project “BIM Software Industry Standardization and Public Service Platform”.

References

[1] W. Shen, Q. Hao, H. Mak, J. Neelamkavil, H. Xie, J. Dickinson, R. Thomas, A. Pardasani, H. Xue, Systems integration and collaboration in architecture, engineering, construction, and facilities management: A review, Adv. Eng. Inform. 24 (2) (2010) 196–207.

[2] S. Kaewunruen, P. Rungskunroch, J. Welsh, A digital-twin evaluation of net zero energy building for existing buildings, Sustainability 11 (1) (2019) 159.

[3] Q. Lu, X. Xie, J. Heaton, A.K. Parlikad, J. Schooling, From BIM towards digital twin: Strategy and future development for smart asset management, in: International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing, Springer, 2019, pp. 392–404.

[4] J.M.D. Delgado, L. Oyedele, Digital twins for the built environment: learning from conceptual and process models in manufacturing, Adv. Eng. Inform. 49 (2021) 101332.

[5] D.L. McGuinness, F. van Harmelen, OWL web ontology language, 2004, Available from: https://www.w3.org/TR/owl-features/. (Accessed January 2022).

[6] W3C OWL Working Group, OWL 2 Web Ontology Language, 2012, Available from: https://www.w3.org/TR/owl2-overview/ (accessed January 2022).

[7] S. Harris, A. Seaborne, SPARQL 1.1 query language, 2013, Available from: http://www.w3.org/TR/sparql11-query/. (Accessed January 2022).

[8] H. Knublauch, D. Kontokostas, Shapes constraint language (SHACL), 2017, Available from: https://www.w3.org/TR/shacl/. (Accessed January 2022).

[9] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A semantic web rule language combining OWL and RuleML, 2004, Available from: https://www.w3.org/Submission/SWRL/. (Accessed January 2022).

[10] P. Pauwels, D. Van Deursen, R. Verstraeten, J. De Roo, R. De Meyer, R. Van de Walle, J. Van Campenhout, A semantic rule checking environment for building performance checking, Autom. Constr. 20 (5) (2011) 506–518.

[11] P. Pauwels, S. Zhang, Semantic rule-checking for regulation compliance checking: An overview of strategies and approaches, in: 32rd International CIB W78 Conference, 2015, pp. 619–628.

[12] T.H. Beach, Y. Rezgui, H. Li, T. Kasim, A rule-based semantic approach for automated regulatory compliance in the construction sector, Expert Syst. Appl. 42 (12) (2015) 5219–5231.

[13] C. Zhang, J. Beetz, B. de Vries, BimSPARQL: Domain-specific functional SPARQL extensions for querying RDF building data, Semant. Web J. (Preprint) (2018) 1–27.

[14] H. Zhang, W. Zhao, J. Gu, H. Liu, M. Gu, Semantic web based rule checking of real-world scale BIM models: a pragmatic method, in: International Congress and Conferences on Computational Design and Engineering, I3CDE, 2019, pp. 130–137.

[15] M. Koubarakis, K. Kyzirakos, Modeling and querying metadata in the semantic sensor web: the model stRDF and the query language stSPARQL, in: International Conference on the Semantic Web: Research & Applications, 2010, pp. 425–439.

[16] C.A. Petri, Kommunikation Mit Automaten (Ph.D. Thesis), University of Bonn, 1962.

[17] C.A. Petri, W. Reisig, Petri net, Scholarpedia 3 (4) (2008) 6477.

[18] K. Jensen, L.M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, Springer Science & Business Media, 2009.

[19] Y. Belgueliel, M. Bourahla, M. Brik, Towards an ontology for UML state machines, Lect. Notes Softw. Eng. 2 (1) (2014) 116.

[20] A. Annane, N. Aussenac-Gilles, M. Kamel, BBO: BPMN 2.0 based ontology for business process representation, in: 20th European Conference on Knowledge Management, Vol. 1, ECKM 2019, 2019, pp. 49–59.

[21] F. Zhang, Z.M. Ma, S. Ribarić, Representation of Petri Net with OWL DL ontology, in: 2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery, Vol. 3, FSKD, IEEE, 2011, pp. 1396–1400.

[22] J. Yim, J. Joo, G. Lee, Petri Net representation of ontologies for indoor location-based services, in: International Conference on Grid and Distributed Computing, Springer, 2011, pp. 423–430.

[23] D. Arena, D. Kiritsis, A methodological framework for ontology-driven instantiation of Petri Net manufacturing process models, in: IFIP International Conference on Product Lifecycle Management, Springer, 2017, pp. 557–567.

[24] J. Szkoła, K. Pancerz, Petri Nets over ontological graphs: Conception and application for modelling tasks of robots, in: International Joint Conference on Rough Sets, Springer, 2017, pp. 207–214.

[25] L.E.G. Moctezuma, B.R. Ferrer, X. Xu, A. Lobov, J.L.M. Lastra, Knowledge-driven finite-state machines. Study case in monitoring industrial equipment, in: 2015 IEEE 13th International Conference on Industrial Informatics, INDIN, IEEE, 2015, pp. 1056–1062.

[26] Apache Software Foundation, Apache Jena, 2019, Available from: http://jena.apache.org/. (Accessed January 2022).

[27] dotNetRDF Project, dotNetRDF, 2019, Available from: http://dotnetrdf.org/. (Accessed January 2022).

[28] J. Beetz, J.V. Leeuwen, B.D. Vries, IfcOWL: A case of transforming EXPRESS schemas into ontologies, Artif. Intell. Eng. Des. Anal. Manuf. 23 (23) (2009) 89–101.

[29] buildingSMART, Industry Foundation Classes IFC4 official release, 2013, Available from: https://standards.buildingsmart.org/IFC/RELEASE/IFC4/FINAL/HTML/. (Accessed January 2022).

[30] Z. Chevallier, B. Finance, B.C. Boulakia, A reference architecture for smart building digital twin, in: SeDiT Workshop at ESWC, 2020, pp. 1–12.

[31] M.H. Rasmussen, M. Lefrançois, G.F. Schneider, P. Pauwels, BOT: the building topology ontology of the W3C linked building data group, Semant. Web 12 (1) (2021) 143–161.

[32] J. Park, J. Chen, Y.K. Cho, Self-corrective knowledge-based hybrid tracking system using BIM and multimodal sensors, Adv. Eng. Inform. 32 (2017) 126–138.

[33] R. Hendrikx, P. Pauwels, E. Torta, H.P. Bruyninckx, M. van de Molengraft, Connecting semantic building information models and robotics: An application to 2D LiDAR-based localization, in: 2021 IEEE International Conference on Robotics and Automation, ICRA, IEEE, 2021, pp. 11654–11660.

[34] P. Pauwels, A. Roxin, SimpleBIM: From full ifcOWL graphs to simplified building graphs, in: EWork and EBusiness in Architecture, Engineering and Construction, CRC Press, 2017, pp. 11–18.

[35] P. Pauwels, T. Krijnen, W. Terkaj, J. Beetz, Enhancing the ifcOWL ontology with an alternative representation for geometric data, Autom. Constr. 80 (2017) 77–94.

[36] P. Pauwels, D. Shelden, J. Brouwer, D. Sparks, SahaNirvik, T.P. McGinley, Building and Semantics: Data Models and Web Technologies for the Built Environment, CRC Press, 2022, pp. 106–107.

[37] S. Törmä, Y. Zheng, Digital construction ontologies, 2021, Available from: https://digitalconstruction.github.io/v/0.3/index.html. (Accessed April 2023).

[38] Brick Consortium, Inc., Brick: A uniform metadata schema for buildings, 2020, Available from: https://brickschema.org/. (Accessed April 2023).

[39] A. Wagner, U. Rüppel, BPO: The building product ontology for assembled products, in: Proceedings of the 7th Linked Data in Architecture and Construction Workshop, LDAC 2019', Lisbon, Portugal, 2019, p. 12.

[40] P. Pauwels, Building element ontology, 2018, Available from: https://pi.pauwel.be/voc/buildingelement. (Accessed April 2023).

[41] P. Pauwels, Distribution element ontology, 2019, Available from: https://pi.pauwel.be/voc/distributionelement. (Accessed April 2023).

[42] M.H. Rasmussen, P. Pauwels, M. Lefrançois, G.F. Schneider, Building topology ontology, 2021, Available from: https://w3c-lbd-cg.github.io/bot/. (Accessed April 2023).

[43] M. Holten Rasmussen, M. Lefrançois, M. Bonduel, C. Anker Hviid, J. Karlshø j, OPM: An ontology for describing properties that evolve over time, in: CEUR Workshop Proceedings, Vol. 2159, 2018, pp. 24–33.

[44] M.H. Rasmussen, M. Lefrançois, Ontology for property management, 2018, Available from: https://w3c-lbd-cg.github.io/opm/. (Accessed April 2023).

[45] A. Wagner, M. Bonduel, P. Pauwels, OMG: Ontology for managing geometry, 2019, Available from: https://www.projekt-scope.de/ontologies/omg/. (Accessed April 2023).

[46] M. Bonduel, A. Wagner, P. Pauwels, FOG: File ontology for geometry formats, 2020, Available from: https://mathib.github.io/fog-ontology/. (Accessed April 2023).

[47] R. Battle, D. Kolas, GeoSPARQL: enabling a geospatial semantic web, Semant. Web J. 3 (4) (2011) 355–370.

[48] J. Wix, J. Karlshoej, Information Delivery Manual: Guide to components and development methods, 2010, Available from: https://standards.buildingsmart.org/documents/IDM/IDM_guide-CompsAndDevMethods-IDMC_004-v1_2.pdf. (Accessed January 2022).

[49] R. See, J. Karlshoej, D. Davis, An integrated process for delivering IFC based data exchange, 2012, Available from: https://standards.buildingsmart.org/documents/IDM/IDM_guide-IntegratedProcess-2012_09.pdf. (Accessed January 2022).

[50] OMG, Business process modeling notation version 1.0, in: OMG Final Adopted Specification, Object Management Group, Vol. 190, 2006.

[51] G. Decker, R. Dijkman, M. Dumas, L. García-Bañuelos, The business process modeling notation, in: Modern Business Process Automation, Springer, 2010, pp. 347–368.

[52] G.H. Mealy, A method for synthesizing sequential circuits, Bell Syst. Tech. J. 34 (5) (1955) 1045–1079.

[53] E.F. Moore, et al., Gedanken-experiments on sequential machines, Autom. Stud. 34 (1956) 129–153.

[54] W. Reisig, Petri Nets and algebraic specifications, in: High-Level Petri Nets, Springer, 1991, pp. 137–170.

[55] K. Yamalidou, J. Moody, M. Lemmon, P. Antsaklis, Feedback control of Petri Nets based on place invariants, Automatica 32 (1) (1996) 15–28.

[56] K. Wolf, Generating Petri Net state spaces, in: International Conference on Application and Theory of Petri Nets, Springer, 2007, pp. 29–42.

[57] K. Ryu, E. Yücesan, CPM: A collaborative process modeling for cooperative manufacturers, Adv. Eng. Inform. 21 (2) (2007) 231–239.

[58] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.

[59] R. Scherer, F. Kog, Transformation of business process models into Petri Nets for building process simulation, in: EWork and EBusiness in Architecture, Engineering and Construction: ECPPM, 2010, pp. 151–156.

[60] R.J. Scherer, S.-E. Schapke, A distributed multi-model-based management information system for simulation and decision-making on construction projects, Adv. Eng. Inform. 25 (4) (2011) 582–599.

[61] K. Jensen, Coloured Petri nets, in: Petri Nets: Central Models and their Properties, Springer, 1987, pp. 248–299.

[62] J.L. Peterson, A note on Colored Petri Nets, Inf. Process. Lett. 11 (1) (1980) 40–43.

[63] C. Natschläger, Towards a BPMN 2.0 ontology, in: International Workshop on Business Process Modeling Notation, Springer, 2011, pp. 1–15.

[64] M. Rospocher, C. Ghidini, L. Serafini, An ontology for the Business Process Modelling Notation, in: FOIS, 2014, pp. 133–146.

[65] D. Gašević, Petri Nets on the semantic web guidelines and infrastructure, Comput. Sci. Inf. Syst. 1 (2) (2004) 127–151.

[66] M. Bing-xian, X. Ying-lei, Integrating PNML with OWL for Petri Nets, in: 2009 2nd IEEE International Conference on Computer Science and Information Technology, IEEE, 2009, pp. 228–230.

[67] Y. Wang, X. Bai, J. Li, R. Huang, Ontology-based test case generation for testing web services, in: Eighth International Symposium on Autonomous Decentralized Systems, ISADS'07, IEEE, 2007, pp. 43–50.
[68] J. Hendler, Agents and the semantic web, IEEE Intell. Syst. 16 (2) (2001) 30–37.
[69] H. Cheng, Z. Ma, A literature overview of knowledge sharing between Petri Nets and ontologies, Knowl. Eng. Rev. 31 (3) (2016) 239–260.
[70] D. Beckett, T. Berners-Lee, Turtle - terse RDF triple language, 2014, Available from: https://www.w3.org/TR/turtle/. (Accessed January 2022).
[71] W3C, Linked data platform 1.0, 2015, Available from: https://www.w3.org/TR/ldp/. (Accessed January 2022).
[72] F. Liu, M. Heiner, M. Yang, An efficient method for unfolding Colored Petri Nets, in: Proceedings of the 2012 Winter Simulation Conference, WSC, IEEE, 2012, pp. 1–12.
[73] L.M. Kristensen, S.r. Christensen, Implementing Coloured Petri Nets using a functional programming language, Higher-Order Symb. Comput. 17 (3) (2004) 207–243.
[74] NIBS, Common building information model files and tools, 2012, Available from: https://www.wbdg.org/bim/cobie/common-bim-files. (Accessed January 2022).