Formal Rules to Produce Object Notation for EXPRESS Schema-Driven Data

V. A. Semenov*,b,c,* (http://orcid.org/0000-0002-8766-8454), S. V. Arishin**, (ORCID: 0000-0001-6128-7082), and G. V. Semenovd,*** (ORCID: 0000-0003-4725-7666)

a Ivannikov Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia
b Moscow Institute of Physics and Technology, Institutskii per. 9, Dolgoprudny, Moscow oblast, 141700 Russia
c National Research University Higher School of Economics, ul. Myasnitskaya 20, Moscow, 101978 Russia
d St. Petersburg National Research University of Information Technologies, Mechanics, and Optics (ITMO University), Kronverkskii pr. 49A, St. Petersburg, 197101 Russia

*e-mail: sem@ispras.ru
**e-mail: arishin@ispras.ru
***e-mail: georgii.v.semenov@gmail.com

Received December 5, 2021; revised January 16, 2022; accepted January 22, 2022

Abstract—Currently, product data management (PDM) systems are widely employed to implement complex multidisciplinary projects in various industrial domains. These systems allow teams of designers, engineers, and managers to remotely communicate via a network, as well as exchange and share product information. To integrate CAD/CAM/CAE applications with PDM systems and ensure their interoperability, a dedicated family of STEP standards (ISO 10303) was developed. STEP defines the object-oriented programming language EXPRESS used to formally specify information schemas, as well as file formats designed to store and transfer product data driven by these schemas. These formats are SPF (clear text encoding) and STEP-XML. Presently, with the development and widespread use of web technologies, the JSON format becomes increasingly popular as it is well suited for object-oriented data exchange and storage, as well as because its syntax is simple and easy to parse. This paper investigates the applicability of JSON to unambiguous representation, storage, and interpretation of product data. On the assumption that product data can be described by arbitrary information schemas in EXPRESS, we propose formal rules for producing JSON notation. Examples that illustrate the proposed rules are provided. Results of our computational experiments confirm the advantages of JSON over SPF and STEP-XML, as well as encourage its wider use for integration of software applications.

DOI: 10.1134/S0361768822070076

1. INTRODUCTION

In recent decades, product data management (PDM) systems have been widely employed to implement complex interdisciplinary projects in various industries: aerospace, defense, automotive, shipbuilding, electronics, and construction. These systems allow teams of designers, engineers, and managers to remotely communicate via a network, as well as exchange and share product information obtained using computer-aided design, manufacturing, and engineering (CAD/CAM/CAE) applications. To integrate these applications and ensure their interoperability, a special international standard for exchange of product model data (STEP) was developed [1]. Its objective is to provide a mechanism, common for different industries, that is capable of describing product data throughout the life cycle of a product, from design to analysis, production, quality control, inspection, support, and maintenance. Due to its nonspecificity, such description is suitable not only for file data exchange but also for multi-user work with archives and databases. It is noteworthy that many other industrial standards like, for example, P-LIB (ISO 13584), CIS/2 (CIMSteel Integration Format), POSC/CAESAR (ISO 15926), PSL (ISO 18629), and IFC (ISO 16739), which were designed to achieve similar objectives, borrow the methodology and fundamental parts of STEP.
STEP is divided into several parts, each part was published separately. In the context of the problem under consideration, three of them—parts 11, 21, and 28—are most important. Therefore, in this paper, we confine ourselves to the detailed consideration of only these parts. The object-oriented data modeling language EXPRESS (part 11) provides the basic description method [2, 3]. Part 21 defines a format for clear text encoding known as the STEP physical file (SPF) [4]. Part 28 uses the extensible markup language (XML) to represent both EXPRESS schemas and data driven by these schemas [5]; this format is known as STEP-XML.

With the current progress in web technologies, the JSON format becomes increasingly popular as it is suitable for object-oriented data exchange and storage [6]. JSON is a lightweight key-value data exchange format, which does not depend on programming languages while being easy to parse and generate by software applications. In addition, it is easy to read for humans. In terms of product data processing, these properties make it more similar to SPF. Moreover, JSON is widely employed for data exchange among web applications, in particular, applications that support AJAX [7]. It is noteworthy that JSON is used in web applications together with XML. However, the use of XML leads to a significant decrease in the performance of web services due to the low efficiency of data reading and parsing operations. Based on performance indicators such as the number of sent objects, the average transmission time for one object, and CPU/memory usage, it was shown that data represented in the JSON format are processed much more efficiently than XML data [8].

Despite the fact that JSON is more preferable than XML for web application development, as well as the fact that its readability is close to that of SPF, the use of object notation for the unambiguous representation, storage, and interpretation of product data has not yet been thoroughly investigated. The purpose of this work is to fill this gap. The paper is organized as follows. Section 2 briefly describes the EXPRESS language with the main focus being placed on the categorization of its basic datatypes. Section 3 provides an example of an information schema in EXPRESS, as well as describes the data driven by this schema in comparison with the SPF and JSON formats. Formal JSON production rules for data driven by arbitrary EXPRESS schemas are summarized and illustrated in Section 4. Section 5 is devoted to the computational experiments and comparative analysis of the proposed JSON-based format versus SPF and STEP-XML. In Conclusions, we outline the prospects for the standardization of this format and its use for integration of industrial software applications, primarily for building information modeling.

2. BRIEF OVERVIEW OF EXPRESS

EXPRESS is a declarative data modeling language that provides formal rules for description of real-world objects and their characteristics. Many design principles of EXPRESS are borrowed from Ada, Algol, C, C++, Eiffel, Euler, Icon, Modula-2, Pascal, PL/I, SQL, ER, and UML. Some original features are implemented to make EXPRESS more suitable for data modeling tasks. In addition, the graphical representation for a subset of EXPRESS constructs, called EXPRESS-G, is developed and standardized.

A schema is the main construct of EXPRESS, which is why all definitions are introduced when declaring a schema. The schema includes definitions of datatypes and constraints imposed on their instances. The datatypes are divided into the following categories: simple datatypes, aggregate datatypes (collections), enumeration datatypes, select datatypes, defined datatypes, and entity datatypes. These categories are illustrated in Fig. 1: the predefined datatypes

![Fig. 1. Datatypes available in the EXPRESS language.](image_url)
are shown in bold and the user-defined datatypes are shown in normal font. The arrows indicate the generalization/specialization relationships among the datatypes. In the latest version of the EXPRESS standard, dynamic behavior for entities can be defined as a response to certain events. These language features are not essential for the problems under consideration, which is why the further discussion of the standard is limited to its original version [3].

Simple datatypes define atomic units of data in EXPRESS. Simple datatypes are REAL, INTEGER, NUMBER, STRING, LOGICAL, BOOLEAN, and BINARY.

Enumeration datatypes and selects are declared using keywords ENUMERATION and SELECT, respectively. The set of ENUMERATION values is a set of names called enum elements. Two different ENUMERATION datatypes can contain the same enum element. In this case, references to enum elements must be provided with datatype identifiers in order to be unambiguously interpreted. The set of SELECT values is a union of the sets of values of all named datatypes specified in the definition of SELECT. Thus, the SELECT datatype is a generalization of each of the named datatypes.

Aggregates in EXPRESS describe various collections. They are declared by specifying the basic type of their elements. The number of elements can vary and be restricted by specification. EXPRESS provides four main types of aggregates defined using the following keywords:

- **BAG** defines an unordered collection that permits duplicate elements;
- **SET** defines an unordered collection of unique elements;
- **LIST** defines an ordered collection of elements that can be accessed by their position in a sequence (LIST declaration can be extended with the UNIQUE constraint, which implies a sequence of unique elements);
- **ARRAY** defines an ordered collection of fixed size with indexed access to its element (the number of elements in an ARRAY collection can be fixed by a value or variable).

Aggregates in EXPRESS are one-dimensional. Multi-dimensional data structures can be represented by an aggregate the basic type of which is another aggregate. This approach allows one to set any number

---

**Fig. 2.** Graphical representation of the ActorResource schema in EXPRESS-G.
of dimensions. For instance, a linear algebra matrix can be defined as an array of arrays of real numbers.

Defined datatypes declared using keyword TYPE include named and constrained datatypes. The set of values of the constrained datatype is a subset of values of the basic type that satisfy logical constraint WHERE.

Generic datatypes are generally used to declare parameters for functions and procedures of EXPRESS schemas. The GENERIC datatype is a generalization of all datatypes. The AGGREGATE datatype is a generalization of all aggregate datatypes (BAG, SET, LIST, and ARRAY) and their specializations.

Entity datatypes are declared using keyword ENTITY. The entity datatype is defined in terms of attributes, each of which is characterized by its name and set of values. Attributes can take values from a specified set taking into account the constraints defined for an individual attribute or for a combination of attributes. Generally, attribute values are explicitly provided by the implementation when constructing instances of entities. An exception is made for attributes declared with keyword OPTIONAL. If an attribute does not have a value, then it is considered undefined (“?”). Keyword DERIVED indicates that the attribute value should be computed in a specified way. Keyword INVERSE indicates that the attribute value is a set of entities associated with a given entity in a particular attribute role.

EXPRESS allows entity datatypes to be specified using subtype/supertype relationships. In this case, the subtype inherits the properties (i.e., attributes, behavior, and constraints) of its supertype. An attribute declared in a supertype can be redeclared in a subtype with the refinement of its set of values. For instance, attribute NUMBER in a supertype can be changed to REAL or INTEGER in a subtype, an optional attribute can be changed to a mandatory attribute, an explicit attribute can be changed to a computable attribute, and an unordered collection attribute can be changed to an ordered collection attribute.

Complex inheritance is implemented using constraints ONEOF, ANDOR, and AND, which relate direct subtypes. The ONEOF constraint means that the subtypes included in its list are mutually exclusive. Each constructed entity must have exactly one of the listed subtypes and cannot have any other subtype (e.g., as a result of constructing composite instances). If subtypes are not mutually exclusive and entities can be instances of more than one of the listed subtypes, then the subtypes are related using the ANDOR constraint. If entities can be divided into multiple groups of mutually exclusive subtypes (i.e., several ONEOF groups), with the members of each such group being constructed as composite instances of several subtypes, then they are related using the AND constraint.

Finally, when defining entity datatypes, a set of admissible values is generally refined. For this purpose, special constraints are used, each of which sets one of the following properties:

- the admissible number of elements for an attribute of the aggregate datatype;
- the dependence between attribute values or the constraint on attribute values for a given entity, which are called the WHERE domain rules;
- the condition for the uniqueness of attribute values on an extent of the entity datatype (for all entities of this type), which is called the UNIQUE rule;
- the constraint on the number of entities involved in the inverse attribute of an INVERSE entity;
- the dependence among extents of several entity datatypes, which is defined as a global rule (RULE).

The principles and constructs of the EXPRESS language considered above are sufficient for further discussion. More detailed information about this language can be found in the corresponding standards [2, 3].

3. EXAMPLE OF AN EXPRESS SCHEMA AND SCHEMA-DRIVEN DATA

As an example, we consider schema ActorResource. Its specification is shown in Listing 1, while its EXPRESS-G diagram is depicted in Fig. 2. The schema defines a common domain for key entities Person, Organization, Address, PostalAddress, TelecomAddress, and OrganizationRelationship, as well as for several auxiliary datatypes ActorSelect, AddressTypeEnum, Label (string type), and ActorRole (defined string type).

Address is declared as an abstract supertype for entity datatypes PostalAddress and TelecomAddress. This means that Address is constructed only through its subtypes. PostalAddress and TelecomAddress inherit both the attributes and constraints of their shared Address supertype. These are explicit attributes Purpose and UserDefinedPupose, inverse attributes OfPerson and OfOrganization, as well as rule WR1. According to this rule, the value of UserDefinedPupose cannot be null if Purpose takes a USERDEFINED enum element. In addition, PostalAddress defines attribute AddressLines, while TelecomAddress defines attributes TelephoneNumbers, FacsimileNumbers, ElectronicMailAddresses, and WWWUrls, as well as its own WR1 rule.
TYPE ActorSelect = SELECT (Organization, Person);
END_TYPE;

TYPE AddressTypeEnum = ENUMERATION OF (OFFICE, HOME, USERDEFINED);
END_TYPE;

TYPE Label = STRING(255);
END_TYPE;

TYPE ActorRole = Label;
END_TYPE;

ENTITY Address
ABSTRACT SUPERTYPE OF (ONEOF(PostalAddress, TelecomAddress));
   Purpose : AddressTypeEnum;
   UserDefinedPurpose : OPTIONAL STRING;
INVERSE
   OfPerson : SET OF Person FOR Addresses;
   OfOrganization : SET OF Organization FOR Addresses;
WHERE
   WR1 : (Purpose <> AddressTypeEnum.USERDEFINED) OR
       ((Purpose = AddressTypeEnum.USERDEFINED) AND
        EXISTS(UserDefinedPurpose));
END_ENTITY;

ENTITY PostalAddress
SUBTYPE OF(Address);
   AddressLines : LIST [1:?] OF Label;
END_ENTITY;

ENTITY TelecomAddress
SUBTYPE OF(Address);
   TelephoneNumbers : OPTIONAL LIST [1:?] OF Label;
   FacsimileNumbers : OPTIONAL LIST [1:?] OF Label;
   ElectronicMailAddresses : OPTIONAL LIST [1:?] OF Label;
   WWWUrls : OPTIONAL LIST [1:?] OF Label;
WHERE
   WR1 : EXISTS (TelephoneNumbers) OR EXISTS (FacsimileNumbers) OR
       EXISTS (ElectronicMailAddresses) OR EXISTS (WWWUrls);
END_ENTITY;

ENTITY Organization;
   Id : INTEGER;
   Name : Label;
   Description : OPTIONAL STRING;
   Roles : LIST [1:?] OF UNIQUE ActorRole;
   Addresses : LIST [1:?] OF UNIQUE Address;
INVERSE
   IsRelatedBy : SET OF OrganizationRelationship FOR RelatedOrganizations;
   Relates : SET OF OrganizationRelationship FOR RelatingOrganization;
   Engages : SET OF Person FOR EngagedIn;
UNIQUE
Entities Person and Organization have their own sets of explicit and inverse attributes, as well as impose constraints on them in the form of corresponding rules. Entity datatype OrganizationRelationship is defined in the schema to establish qualified attribute relationships between Organization entities. To specify one-to-many relationships, attributes RelatingOrganization and RelatedOrganizations are used; the former has type Organization, while the latter is defined as a non-empty set of entities of the Organization type. In addition, using the inverse attributes IsRelatedBy, Relates, and Engages of an Organization entity, it is possible to determine the instances and relationships in which this entity is involved.

For brevity, we omit further explanations because the remaining part of the specification uses the same constructs.

Let us now consider a set of data that are driven by the EXPRESS schema described above and are represented in the ASCII SPF format (see Listing 2). The data file is organized as a set of strings, each of which represents an instance of the entity datatype and is passed within the exchange structure. The entity datatype of the instance must be defined in the EXPRESS schema specified in the file’s header. In our example, the schema is ActorResource and the instances are entities of datatypes Organization, Person, PostalAddress, TelecomAddress, and OrganizationRelationship, which are defined by this schema. Each instance is represented exactly once in the exchange structure and has a unique identifier. The sequence order of the instances in the exchange structure does not matter. The instances can be referenced by their identifiers from anywhere in the exchange structure.

#11 = Organization(1203, 'Automobile Inc.', 'In cars we trust.');
('Supply Chain Manager', 'Executive Manager', 'Sells Manager'), (#31, #34));

#12 = Organization(1204, 'Wheels Inc.', 'We do wheels.', ('Executive Manager', 'Sells Manager'), (#32, #35));

#13 = Organization(1205, 'Motor Inc.', 'Motors are essential.', ('Executive Manager', 'Sells Manager'), (#33, #36));

#31 = PostalAddress(.OFFICE., $, ('9292 Automobile Dr.', 'Mc Lean', 'VA 22101'));

#32 = PostalAddress(.USERDEFINED., 'Sells Department', ('1172 Wheel Avenue', 'Fresno', 'CA 93711'));

#33 = PostalAddress(.USERDEFINED., 'Sells Department', ('1119 Motor Road', 'Reno', 'NV 89501'));

#34 = TelecomAddress(.OFFICE., $, ('678-762-2354', '678-762-2355'), $, ('automobile@cars.com'), ('http://www.cars.com/automobile'));

#35 = TelecomAddress(.USERDEFINED., 'Product Information', ('775-201-8669', '775-761-2384'), $, ('wheels@cars.com'), ('http://www.cars.com/wheels'));

#36 = TelecomAddress(.USERDEFINED., 'Product Information', '609-639-9256', '201-213-0598'), $, ('motor@cars.com'), ('http://www.cars.com/motor'));

#51 = OrganizationRelationship('Consumer', $, #11, (#12, #13));

#52 = OrganizationRelationship('Supplier', $, #12, (#11));

#53 = OrganizationRelationship('Supplier', $, #13, (#11));

#61 = Person(901, 'Pringle', 'Andrew', $, ('Mr'), $, ('Supply Chain Manager', 'Executive Manager'), (#34), (#11));

#62 = Person(902, 'Martinez', 'Bill', $, ('Mr'), $, ('Executive Manager', 'Sells Manager'), (#35), (#12));

#63 = Person(903, 'Ackley', 'Chris', $, ('Mr'), $, ('Executive Manager', 'Sells Manager'), (#36), (#13));

Listing 2. Sample dataset driven by the ActorResource schema and represented in the SPF format.

The sequence order of parameters in the string of an instance is the same as the order in which the corresponding attributes are defined in the declaration of an entity. In this case, the attributes defined in supertypes precede the attributes of subtypes. The value of each parameter must correspond to the datatype of an attribute.

In the dataset under consideration, the PostalAddress instance with unique identifier #31 has the following parameters: the first parameter “.OFFICE.” is a value of the Purpose attribute declared in the Address supertype, the second parameter “$” is an undefined value of the UserDefinedPurpose attribute also declared in the Address supertype, and the third parameter “('9292 Automobile Dr.', 'Mc Lean', 'VA 22101’)” is a value of the AddressLines explicit attribute declared in the PostalAddress entity as a list of strings. It should be noted that the exchange structure contains only the stored values of explicit attributes, whereas the derived and inverse attributes declared using keywords DERIVED and INVERSE are ignored. It is assumed that they can always be computed by the application from explicit attributes, which is why there is need to include them in the exchange structure.

4. JSON PRODUCTION RULES

JSON is a text format for representing and serializing object-oriented data; it is based on the ECMAScript programming language, which, in turn, is an extension of JavaScript [6]. Data in this format are described using four primitive types (string, number, Boolean, and null) and two structured types (object and array). An object is an unordered collection of zero or more key–value pairs, where keys are strings and values are any supported datatypes. An array is an ordered sequence of zero or more values. A string is a sequence of zero or more Unicode characters.

This section is devoted to the generalization and systemization of the formal rules used to match the datatypes described in EXPRESS with the JSON datatypes. Such mapping should enable the unambiguous representation, storage, and interpretation of...
product data in the JSON format, as well as in the SPF and STEP-XML formats defined in current international standards.

The first two rules are general ones and do not depend on the user-defined datatypes of EXPRESS. The first rule specifies the representation of optional attributes declared in the schema by using keyword OPTIONAL. These attributes are not required to have values in instances of objects. Undefined values are encoded in JSON with “null”, which corresponds to the dollar sign (“$”) in the SPF format. Defined values are represented in accordance with the rules described in the following subsections.

The second rule specifies the qualification of EXPRESS datatypes. The qualification is necessary when the value of an attribute cannot be unambiguously interpreted without explicitly specifying its datatype. Usually, it is required for attributes of the SELECT type. Datatype qualification is also required in compound instances of entity datatypes declared with subtype/supertype constraints ONEOF, ANDOR, and AND. To avoid misinterpretation, simple instances of entity datatypes should be provided with datatype names. For clarity and readability, all datatype names are written using only lowercase letters.

```
SCHEMA IllustrationResource;

  TYPE Url = STRING; END_TYPE;
  TYPE Png = BINARY; END_TYPE;
  TYPE Pixels = LIST[1:?] OF INTEGER; END_TYPE;
  TYPE Image = SELECT (Url, Png, Pixels); END_TYPE;

  ENTITY Title;
    text : STRING;
  END_ENTITY;

  ENTITY Picture;
    figure : Image;
  END_ENTITY;

  ENTITY Illustration SUBTYPE OF (Title AND Picture);
  END_ENTITY;

END_SCHEMA;

[
  {
    "_ oid": "#1",
    "type": "Picture",
    "figure": {
      "type": "Pixels",
      "value": [0, 255, 255, 128, 128]
    }
  },
  {
    "_ oid": "#2",
    "type": "Illustration",
    "_ prototype": [
      {
        "type": "Title",
        "text": "Book"
      }
    ]
  }
]
Let us illustrate this rule by using the IllustrationResource schema and dataset shown in Listing 3. This schema defines main entities Title, Picture, and Illustration, as well as several auxiliary datatypes. Datatype Image is a select declared as a generalization of datatypes Url, Png, and Pixels, each providing an alternative way of image representation. Therefore, when a Picture instance is constructed and its “figure” attribute is initialized, the assigned value must be supplemented with the name of datatype Url, Png, or Pixels. In this dataset, the Picture instance with identifier #1 is defined as having the “figure” attribute that is initialized by a list of integers and is qualified as “Pixels”. It can be seen from the example that, in such cases, attribute values should be represented as nested JSON objects with their own “types” and “values”.

The IllustrationResource schema also defines complex entity datatype Illustration, which is a subtype of Title and Picture. This means that Illustration instances are composed of simple instances of the corresponding supertypes. In the example considered above, the Illustration instance with identifier #2 is defined as consisting of Title and Picture instances. Supertype instances must be represented as JSON objects and be qualified with “type”. The objects are included in the external JSON array defined as the “_prototype” of a complex instance. In this case, identifiers are not assigned to simple instances.

It should be noted that the sequence order of attributes in any JSON object does not matter. However, it is recommended to set attributes “_oid”, “type”, and “_prototype” first; the subsequent attributes are arranged in the order in which entity attributes are defined in the original schema. This allows product data presented in the JSON format to be processed more efficiently in accordance with the proposed rules.

4.1. Rules for Representing EXPRESS Datatypes Simple, Constructed, and Aggregate

The following rules are defined to represent simple datatypes of EXPRESS in JSON.

- Datatype NUMBER in EXPRESS corresponds to the numeric datatype in JSON. Numbers are represented in the same way as in most of the programming languages: using the decimal base with digits. A number has an integer component, which can be preceded by an optional minus sign (“–”) and can be followed by a fractional part and/or an exponential part.
  - Datatype INTEGER is regarded as an integer subset of the JSON numeric datatype, and its instances are represented as sequences of one or more digits without leading zeros, which can be preceded by the minus sign.
  - Datatype BOOLEAN in EXPRESS corresponds to the Boolean datatype in JSON.
  - Datatype LOGICAL corresponds to predefined enumeration of strings “true”, “false”, and “unknown”.
  - Datatype STRING corresponds to the string type in JSON, which is defined by the agreements adopted for C-family programming languages. The string begins and ends with quotation marks. All Unicode characters can be enclosed in quotation marks, except for the characters that must be escaped: quotes, backslashes, and control characters (from U+0000 to U+001F).
  - Datatype BINARY corresponds to the string type in JSON the values of which are encoded in accordance with the Base64 standard [9].

Datatype ENUMERATION in EXPRESS must be represented as a JSON string that corresponds to one of the enum element names. Datatype SELECT is mapped to JSON datatypes in the same way as the datatypes listed in its expression. Values of these datatypes, including nested selects, are encoded in accordance with the rules considered above and must be supplemented with the “type” field to qualify types of values.

Aggregate datatypes BAG, SET, LIST, and ARRAY are represented as JSON arrays. It is allowed to use heterogeneous elements of different datatypes. Each element in the JSON array must be encoded in accordance with its datatype declared in the EXPRESS schema. The sequence order of elements in the encoding must be preserved for ordered aggregates LIST and ARRAY. If the aggregate is empty, then it must be represented as an empty JSON array, rather than as an undefined value denoted by keyword “null”.

Listing 3. Example of EXPRESS datatype qualification in the JSON format.
Finally, the value of an attribute of the ENTITY datatype, which is an association to the corresponding object of this type, must be represented as a JSON string that includes the identifier of the instance of the object. Recall that each instance is represented in the exchange structure exactly once.

4.2. Rules for Representing EXPRESS Datatype Entity

The product data exchange structure in the JSON format is represented as an array of objects, each of which corresponds to an instance of the entity datatype in EXPRESS. Each object is represented as a set of key–value pairs.

The object representation must meet the following requirements.

• The string identifier of an object with key “_oid” must be unique within the exchange structure. For certain target applications, global uniqueness of identifiers can be declared. This attribute is mandatory for all instances of EXPRESS entity datatypes that are not simple instances of complex entity datatypes. The underscore used in the keyword should prevent conflicts of names with possible definitions of the basic schema. For consistency with SPF, integers preceded by # can be used as identifiers.

• The type of an object must be represented as a JSON string value with key “type”. The value must correspond to the name of the entity datatype defined in the basic schema. This attribute is mandatory for all objects.

• The prototype of an object with key “_prototype” is represented as an array of nested JSON objects. This attribute is defined only for composite instances of complex ENTITY datatypes declared using subtype/supertype constraints ONEOF, ANDOR, and AND. In these cases, each simple instance of a supertype must be represented as an object of a prototype array. Instances of supertypes are not assigned object identifiers. More information on constructing instances of complex entities can be found in the EXPRESS standard [3].

• Each object attribute corresponds to an explicit attribute of an instance of the EXPRESS entity datatype. As a key, the attribute name defined in the basic schema is used. Attribute values must be represented in accordance with the rules described in the previous subsections. Datatype qualification should be carried out in all cases where values cannot be unambiguously interpreted. There are no constraints on the order in which attributes are specified. Mandatory attributes not declared as OPTIONAL in the definition of the entity datatype must be specified in the structure of a JSON object. Missing optional attributes are considered undefined. Attributes declared as DERIVED and INVERSE in the schema are ignored.

4.3. Example of Rule-Based Object Notation Production

Let us illustrate the proposed formal rules on the ActorResource schema and dataset from Section 3.

```json
[
  {
    "_oid":"#11",
    "type":"Organization",
    "id":1203,
    "name":"Automobile Inc.",
    "description":"In cars we trust.",
    "roles": ["Supply Chain Manager", "Executive Manager",
              "Sells Manager"],
    "addresses": ["#31",="#34"]
  },
  {
    "_oid":"#12",
    "type":"Organization",
    "id":1203,
    "name":"Automobile Inc.",
    "description":"In cars we trust.",
    "roles": ["Supply Chain Manager", "Executive Manager",
              "Sells Manager"],
    "addresses": ["#31",="#34"]
  },
  {
    "_oid":"#13",
```
Listing 4. Sample dataset driven by the ActorResource schema and encoded in the JSON format.
lAddress, and TelecomAddress with the same attribute values. To match the datasets, the identifiers of the objects were set to be the same as in the SPF file, even though the rules permit more general methods.

It should be noted that the ActorResource schema is used for clarity and compactness of data representation. It is part of the EXPRESS specification of Industry Foundation Classes (IFC) [10], which is an open international standard that defines approximately a thousand entity datatypes and several thousand auxiliary types for representation of architectural and building construction data. Obviously, without the generalization and systematization of the formal rules for datatype mapping from EXPRESS to JSON, as well as without the qualification of product data representations, the processing and interpretation of the product data by different applications would not be possible.

5. VALIDATION OF THE PROPOSED FORMAL RULES

The proposed rules for product data representation were validated in the process of developing a software application for their JSON serialization and three-dimensional visualization. Using this application, a series of computational experiments were carried out and a comparative analysis with alternative product data representations in the SPF and STEP-XML formats was conducted.

As test data for the computational experiments, IFC data of various complexity were used (see Fig. 3). The data were publicly available and obtained by exporting from Revit 2019 into the SPF format (as applied to the IFC schema, it is often referred to as the IFC format). The experiments were carried out on a computer with Intel Core i7-7700 3.60 GHz and 32 GB of RAM.

In the first experiment, the sizes of the IFC files were compared upon conversion to the selected formats (see Table 1) and upon zip-archiving (see Table 2). It can be seen that the size of the files in the JSON format is, on average, two times smaller than their size in the STEP-XML format and approximately two times larger than that in the SPF format. The archiving evened out the difference in sizes; however, the general regularity holds: the SPF files are lighter than the JSON files, and the JSON files are lighter than the XML files.

In the second experiment, the CPU times required for parsing the IFC data in the selected formats were compared. For this purpose, JSON and XML parsers

### Table 1. Size of the original IFC data represented in the SPF, JSON, and STEP-XML formats

| IFC dataset | SPF, Kb | JSON, Kb (as % of SPF) | STEP-XML, Kb (as % of SPF) |
|-------------|--------|------------------------|-----------------------------|
| a           | 952    | 2195 (231%)            | 4375 (460%)                 |
| b           | 2516   | 6054 (241%)            | 11556 (459%)                |
| c           | 26274  | 56613 (215%)           | 113199 (431%)               |
| d           | 40816  | 89794 (220%)           | 175689 (430%)               |
| e           | 93160  | 237564 (255%)          | 379806 (408%)               |
| f           | 173307 | 430514 (248%)          | 746400 (431%)               |

Fig. 3. Visualization of the IFC data used in the computational experiments.
as part of the Qt environment 5.15.1 were used. The results are shown in Table 3, indicating a significant speedup when using JSON over STEP-XML.

Thus, the computational experiments confirm the advantages of the JSON representation generated using the proposed formal rules over the standard STEP-XML representation. However, it should be noted that, when JSON data parsing is combined with semantic control, e.g., when product data are received by the PDM system and a check for compliance with the original EXPRESS schema is required, the gain can be eliminated by a relatively high cost of semantic control. In other cases where semantic control is not required, e.g., when the PDM system client receives validated data, parsing speed becomes an important factor. With a significant decrease in the amount of data transferred in the distributed environment of the PDM system, the advantages of representing product data in the JSON format become obvious. Finally, it should be noted that the standard SPF format offers approximately the same size of archived files; however, it has other significant disadvantages that are due to the complexity of data processing and interpretation by web applications.

Table 3. CPU time for parsing the product data in the JSON and STEP-XML formats

| IFC dataset | JSON CPU time, ms | STEP-XML CPU time, ms | Speedup (JSON versus STEP-XML) |
|-------------|------------------|-----------------------|--------------------------------|
| a           | 38               | 146                   | 3.84                           |
| b           | 129              | 564                   | 4.37                           |
| c           | 824              | 2772                  | 3.36                           |
| d           | 1281             | 4205                  | 3.28                           |
| e           | 3124             | 9025                  | 2.89                           |
| f           | 5626             | 31864                 | 5.66                           |

6. CONCLUSIONS

In this paper, we have investigated the possibility of using the JSON format to represent product data driven by information schemas in the EXPRESS language. We have proposed the formal rules for producing unambiguous, non-redundant, and software-independent object notation for product data, as well as have illustrated the application of these rules by several examples. The rules have been validated in the process of developing the software application for serialization and three-dimensional visualization of product data. Using this application, the computational experiments have been carried out to confirm the advantages of the JSON format over the standard SPF and STEP-XML formats, which can encourage its wider use for integration of industrial software applications. We expect that the efforts aimed at standardization of the proposed rules and file format generated using them, primarily in the field of building information modeling, will contribute to this purpose.

It should be noted that the proposed rules define the representation of product data driven by an EXPRESS schema, rather than the representation of the schema itself. In some works [11], there were attempts to represent IFC schemas as JSON documents; however, these attempts should be considered rather specific and therefore limited, considering the variety of EXPRESS constructs. In particular, the presence of algebraic specifications for DERIVED attributes and various kinds of WHERE, UNIQUE, and GLOBAL rules does not allow one to fully represent information schemas in JSON and use them, e.g., for data validation [12]. That is why that form of representation has not been considered in this paper.

CONFLICT OF INTEREST

The authors declare that they have no conflicts of interest.
REFERENCES

1. ISO 10303, Industrial automation systems and integration: Product data representation and exchange.
2. ISO 10303-11:2004, Industrial automation systems and integration: Product data representation and exchange, Part 11: Description methods, The EXPRESS language reference manual.
3. ISO 10303-11:1994, Industrial automation systems and integration: Product data representation and exchange, Part 11: Description methods, The EXPRESS language reference manual.
4. ISO 10303-21:2016, Industrial automation systems and integration: Product data representation and exchange, Part 21: Implementation methods, Clear text encoding of the exchange structure.
5. ISO 10303-28:2007, Industrial automation systems and integration: Product data representation and exchange, Part 28: Implementation methods, XML representations of EXPRESS schema and data.
6. Bray, T., The JavaScript Object Notation (JSON) data interchange format, Internet Engineering Task Force (IETF), Request for comments: 8259, 2014.
7. Peng, D., Cao, L., and Xu, W., Using JSON for data exchanging in web service applications, J. Comput. Inf. Syst., 2011, vol. 7, no. 16, pp. 7552–7569.
8. Nurseitov, N., Paulson, M., et al., Comparison of JSON and XML data interchange formats.
9. Base64. https://en.wikipedia.org/wiki/Base64.
10. ISO 16739-1:2018, Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries.
11. Afsari, K., Eastman, Ch.M., and Castro-Lacouture, D., JavaScript Object Notation (JSON) data serialization for IFC schema in web-based BIM data exchange, Autom. Constr., 2017, vol. 77, pp. 24–51.
12. Semenov, V., Ilyin, D., Morozov, S., and Tarlapan, O., Effective consistency management for large-scale product data, J. Ind. Inf. Integr., 2019, vol. 13, pp. 13–21.

Translated by Yu. Kornienko