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

Georgii Semenov
Faculty of information technologies and programming
ITMO University
Saint-Petersburg, Russia
georgii.v.semenov@gmail.com

Abstract*—Recently, product data management systems (PDM) are widely used to conduct complex multidisciplinary projects in various industrial domains. The PDM systems enable teams of designers, engineers, and managers to remotely communicate on a network, exchange and share common product information. To integrate CAD/CAM/CAE applications with the PDM systems and ensure their interoperability, a dedicated family of standards STEP (ISO 10303) has been developed and employed. The STEP defines an object-oriented language EXPRESS to formally specify information schemas as well as file formats to store and transfer product data driven by these schemas. These are clear text encoding format (STEP P21) and XML-based format (STEP P28). Nowadays, with the development and widespread adoption of Web technologies, the JSON language is getting increasingly popular due to it being apropos for the tasks of object-oriented data exchange and storage, as well as its simple, easy to parse syntax. The paper explores the topic of the suitability of the JSON language for the unambiguous representation, storage and interpretation of product data. Under the assumption that the product data can be described by arbitrary information schemas in EXPRESS, formal rules for the producing JSON notation are proposed and presented. Explanatory examples are also provided to illustrate the practical use of the rules.

Keywords*—object-oriented data modeling, EXPRESS, STEP, JSON, Web services, interoperability

I. INTRODUCTION

In recent decades, product data management (PDM) systems have been widely used to implement complex multidisciplinary projects in such industries as aerospace, defense, automotive, shipbuilding, electronics, and construction. The PDM systems is an instrument for teams of designers, engineers and managers to remotely communicate on a network, to exchange and share common product information obtained through computer-aided design, manufacturing and engineering applications (CAD/CAM/CAE). PDM systems such as ProjectWise, Windchill, Teamcenter, Enovia have got a certain popularity thanks to advanced concurrent engineering facilities which reduce the time and costs of designs in specific industries [1, 2].

To integrate the software applications and ensure their interoperability, a dedicated international STandard for the computer-interpretable representation and Exchange of Product data (STEP) was developed [3]. Its objective is to provide a general industry-neutral mechanism capable of describing product data throughout the entire product life cycle from design to analysis, manufacturing, quality control, inspection, support and operation. The neutral nature of this description makes it suitable not only for the neutral file exchange, but also for being used as a basis for the sharing, archiving and implementation of the product databases. It is noteworthy that many other industry standards, such as P-LIB (ISO 13584), CIS/2 (CIMSteel Integration Standard, where CIMSteel stands for Computer Integrated Manufacturing of Constructional Steelwork), POSC/CAESAR (ISO 15926), PSL (ISO 18629) and IFC (ISO 16739), developed for similar purposes, borrow the model-driven methodology and fundamental parts of the STEP.

The STEP standard defines the object-oriented data modeling language EXPRESS as an underlying description method (part 11). Among the other implementation methods, the part 21 and the part 28 represent an interest for further consideration. The part 21 defines the format for writing data to a flat text encoding file, also known as the STEP Physical File format (SPF) [4]. The part 28 relies on the eXtensible Markup Language (XML) to represent schemas using the EXPRESS language and the data that is governed by EXPRESS schemas.

Nowadays, with the development and widespread adoption of Web technologies, the JavaScript Object Notation (JSON) language is getting increasingly popular due to it being apropos for the tasks of object-oriented data exchange and storage. JSON is a key-value style lightweight data exchange format which is independent of any programming language and which is easy for machines to parse and generate while it is also human readable. As applied to product data, these features bring it closer to the SPF format. At the same time, JSON is extensively adopted as a default data exchange format by various Web applications, specifically Asynchronous JavaScript and XML (AJAX) Web services. It is noteworthy that JSON is used in Web applications along with XML. However, the performance of Web services has shown a significant decrease when using XML data because of the low efficiency of reading and parsing XML data during the execution of services [5]. Based on the measurement of performance metrics such as the number of objects sent, average time per object transmission, CPU and memory utilization, it has been proved that JSON is significantly faster and has higher efficiency than XML [6, 7]. Several researches have shown that JSON deserialization on mobile platforms is few orders faster than XML [8, 9]. It therefore follows that XML should not be used for large datasets because of its long deserialization time and larger size relatively to the same datasets represented in JSON.

Being designed to process large engineering datasets, PDM systems must meet the requirements of efficiency and scalability, while providing open interfaces to access the data. Having several standard interfaces implemented, it is advisable for PDM systems to operate with light-weight and easily parsed data format.

Although JSON is a promising candidate for such purposes, so far there is a lack of studies on representing product data using the JSON. This paper highlights the topic of the suitability of the JSON language for the unambiguous presentation, storage and interpretation of product data driven

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

by EXPRESS schemas. Section 2 provides a brief overview of the EXPRESS language with the emphasis on underlying data types. An example schema formally specified in EXPRESS is presented in Section 3. An explanatory dataset driven by the specified schema and presented in both SPF and JSON formats is provided too. Formal rules to produce JSON notation for product data driven by arbitrary EXPRESS schemas are proposed and shortly illustrated in Section 4. In Conclusions, the perspectives of the JSON notation produced according to the rules are outdrawn in the context of the STEP standardization.

II. Brief overview of EXPRESS language

EXPRESS is a conceptual schema language which consists of the constructs supporting an unambiguous object-oriented data model definition, the specification of dynamic aspects and constraints on both the data and dynamics. A lot of languages such as Ada, Algol, C, C++, Eiffel, Euler, Icon, Modula-2, Pascal, PL/I, SQL, ER and UML have contributed to EXPRESS. Some facilities have been invented to make EXPRESS more suitable for the representation of information models. A graphical representation for a subset of the EXPRESS constructs, called EXPRESS-G, has been also developed and standardized [10].

The top-level EXPRESS construct is the schema so that all declarations occur as a part of a schema declaration. The schema defines both datatypes and constraints on the instances with the corresponding data types. Datatypes are divided into the simple datatypes, enumeration and selection datatypes, defined datatypes, entity datatypes, aggregation datatypes, and generalized datatypes. These datatype categories are shown in Figure 1. Predefined datatypes available in EXPRESS are highlighted by bold italic type. Constructs for user-defined types are in regular type. The arrows show the generalization/specialization relationships between datatypes. In the latest edition of EXPRESS standard, an entity may be additionally defined in terms of its behavior, described by the events it responds to and the way it responds. These facilities are left beyond the scope of the paper, and therefore the original version of the EXPRESS standard [11] will be discussed upon further consideration.

Simple datatypes define the domain of the atomic data units in EXPRESS, which cannot be further subdivided into smaller elements recognized by EXPRESS. The simple datatypes are REAL, INTEGER, NUMBER, STRING, LOGICAL, BOOLEAN and BINARY.

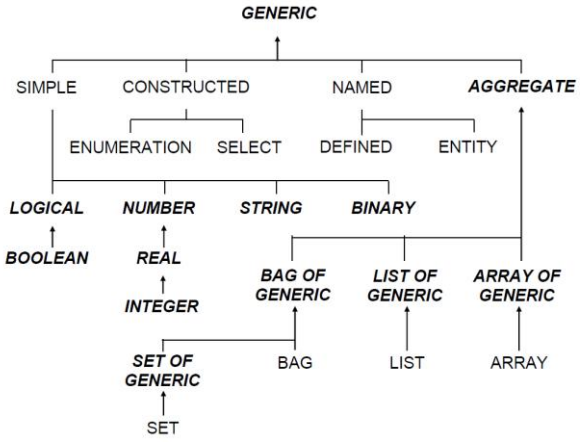

Fig. 1. Datatype categories available in EXPRESS.

Enumeration and selection datatypes are declared using the ENUMERATION and SELECT keywords respectively. An ENUMERATION datatype has a set of names as its domain. The names are referred as enumeration items which represent the values of the enumeration datatype. Two different ENUMERATION datatypes may contain the same enumeration item. In this case, any reference shall be pre-qualified with the datatype identifier to ensure that the reference is unambiguous. A SELECT datatype domain is the union of the domains of the named datatypes. The SELECT datatype is a generalization of each of the named datatypes in its select items.

The domains of the aggregation datatypes are the collections of values of a given base datatype. These base datatype values are called elements of the aggregation collection. The number of elements in a collection may vary and may be constrained by the specification of bounds. There are four kinds of aggregation datatype generators: BAG, SET, LIST, and ARRAY. Each kind of aggregation datatype generator attaches different properties:

- BAG generates a datatype containing an unordered collection;

- SET generates a datatype containing an unordered collection of unique elements;

- LIST generates a datatype containing an ordered collection of elements (a sequence) that can be accessed by their position within the sequence. The LIST generator can be modified with the UNIQUE constraint, thereby implying an ordered collection of unique elements;

- ARRAY generates a datatype containing a collection of elements with associated indexes of an INTEGER type, where indexes identify the element. The number of elements in an ARRAY collection is either fixed by the index values or variable, in which case it is unconstrained.

EXPRESS aggregations are one-dimensional. Data structures usually considered to have multiple dimensions (such as mathematical matrices) can be represented by an aggregation datatype whose base type is another aggregation. Thus, aggregation datatypes can be nested to an arbitrary depth, allowing any number of dimensions to be represented. For example, a matrix for linear algebra can be defined as an array of arrays of reals.

Defined datatypes declared by the TYPE keyword include aliased and constrained datatypes. The domain of the constrained defined datatype is a subset of the domain of the underlying type, as constrained by the WHERE clause.

Entity datatypes represent objects of interest. The entity datatype with the ENTITY declaration is defined in terms of its attributes, which are characterized by a name and a value space. The attributes can take values from the specified domains with the constraints taken in account on these values defined individually or in combination. Typically attribute values are explicitly supplied by an implementation in order to create an entity instance. An exception is made for the attributes declared with the keyword OPTIONAL meaning that the attribute need not have a value. If the attribute has no value, the value is said to be indeterminate. The DERIVED keyword indicates that the attribute value is computed in some manner. The INVERSE keyword indicates that the attribute value consists of the entity instances which use the entity in a particular role.

EXPRESS allows the specification of entities as subtypes of other entities, where a subtype entity is a special form of its supertype. This establishes an inheritance (i.e., subtype/supertype) relationship between the entities in which the subtype inherits the properties (i.e., attributes, behavior, constraints) of its supertype. An attribute declared in a supertype can be redeclared in a subtype. The attribute is kept in the supertype but the domain for this attribute is governed by the redeclaration given in the subtype. The declaration may be changed in different ways, but the attribute redeclared in the subtype shall have a domain narrower than or equal to the attribute being redeclared. For example, a NUMBER datatype attribute in the supertype may be changed to a REAL or to an INTEGER datatype in the subtype, an optional attribute — to a mandatory attribute, an explicit attribute — to a derived attribute, an unordered collection type attribute — to an ordered collection attribute, etc.

Two or more direct subtypes of a supertype may be allowed to have overlapping instantiations. The ONEOF, ANDOR, AND constraints are used to specify the relationship within a group of direct subtypes. The ONEOF constraint states that the elements of the ONEOF list are mutually exclusive. None of the elements may be instantiated with any other element in the list. Each element shall be a subtype expression which may resolve to a single subtype of the entity datatype. If the subtypes are not mutually exclusive, that is, an instance of the supertype may be an instance of more than one of its subtypes, then the relationship between the subtypes shall be specified using the ANDOR constraint. If the supertype instances are categorized into multiple groups of mutually exclusive subtypes (i.e., multiple ONEOF groupings) indicating that there is more than one way to completely categorize the supertype, then the relationship between those groups shall be specified using the AND constraint. The AND constraint should only be used to relate groupings established by other subtype/supertype constraints.

Generalized datatypes are used to specify a generalization of other datatypes and their application is very narrowed. The GENERIC datatype is a generalization of all datatypes. The AGGREGATE datatype is a generalization of all aggregation datatypes. The general aggregate datatypes are generalizations of aggregation datatypes which relax some of the constraints normally applied to the aggregation datatypes.

Finally, the domain of the datatype used in the attribute declaration is set by constraints. Each constraint represents one of the following properties of the entity:

- limits on the number, kind and organization of values of the attributes, which are specified in the attribute declarations;

- required relationships between attribute values or limits on the attribute values for a given instance, which appear in the WHERE clause and are referred as domain rules;

- required relationships between attribute values over all instances of the entity datatype, which appear in the UNIQUE clause, where they are referred to as uniqueness constraints, the INVERSE clause, where they are referred to as cardinality constraints, or GLOBAL rules;

- required relationships between instances of several entity types, which appear rather in GLOBAL rules than in entity declaration itself.

The above-mentioned constructs and features of the language are sufficient for further consideration. More details can be found in the EXPRESS standard revisions [11].

III. Examples of EXPRESS schema and schema-driven dataset

Let us consider an example of the schema with the specification shown in Figure 2 and corresponding EXPRESS-G diagram presented in Figure 3. The ActorResource schema declaration defines a common scope for a collection of related entity and other datatype declarations. These are the entities Person, Organization, Address, PostalAddress, TelecomAddress, OrganizationRelationship, as well as the selection ActorSelect, the enumeration AddressTypeEnum, the string type Label, and the aliased string type ActorRole.

```
SCHEMA ActorResource;

  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
      UR1 : Id;
```

```
    WHERE
      WR1 : SIZEOF( QUERY( temp <* Engages | 'Director' in
temp.Roles ) ) = 1;
  END_ENTITY;

  ENTITY OrganizationRelationship;
      Name                 : Label;
      Description          : OPTIONAL STRING;
      RelatingOrganization: Organization;
      RelatedOrganizations: SET [1:?] OF Organization;
    WHERE
      WR1 : SIZEOF( [RelatingOrganization] *
RelatedOrganizations ) = 0;
  END_ENTITY;

  ENTITY Person;
      Id           : INTEGER;
        FamilyName   : OPTIONAL Label;
        GivenName    : OPTIONAL Label;
        MiddleNames  : OPTIONAL LIST [1:?] OF Label;
        PrefixTitles : OPTIONAL LIST [1:?] OF Label;
        SuffixTitles : OPTIONAL LIST [1:?] OF Label;
        Roles        : LIST [0:?] OF UNIQUE ActorRole;
        Addresses    : LIST [1:?] OF UNIQUE Address;
        EngagedIn    : SET OF Organization;
      UNIQUE
        UR1 : Id;
      WHERE
        WR1 : EXISTS(FamilyName) OR EXISTS(GivenName);
    END_ENTITY;

  END_SCHEMA;
```

Fig. 2. Specification of ActorResource schema in EXPRESS language.

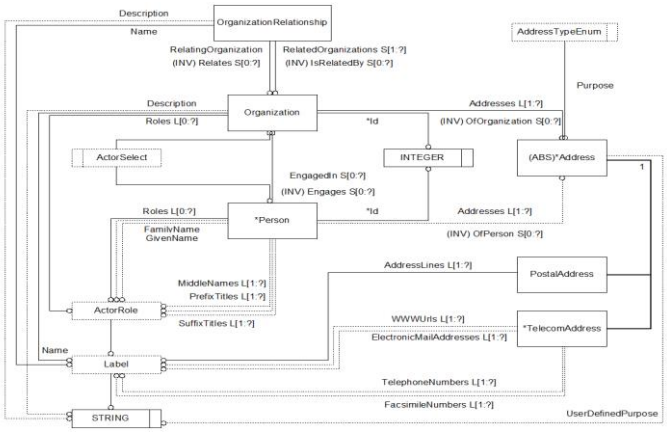

Fig. 3. EXPRESS-G diagram of ActorResource schema

The Address entity is declared as an abstract supertype of the PostalAddress and TelecomAddress entities. It means that the Address entity can be instantiated only through its subtypes. PostalAddress and TelecomAddress instances inherit both attributes and constraints from their mutual supertype Address. These are the explicit attributes Purpose, UserDefinedPurpose, the inverse attributes OfPerson, OfOrganization and the domain rule WR1. According to the rule, the value of the UserDefinedPurpose attribute cannot be indeterminate if the attribute Purpose takes the USERDEFINED enumeration item. Additionally, the PostalAddress defines the attribute AddressLines, and the TelecomAddress — the attributes TelephoneNumbers, FacsimileNumbers, ElectronicMailAddresses, WWWUrls and own domain rule WR1.

The Person and Organization entities define own sets of explicit and inverse attributes, as well as domain and uniqueness rules. The OrganizationRelationship entity is specially introduced into the schema in order to be able to set up qualified, attributed relations between Organization instances. These one-to-many relations can be established through the attributes RelatingOrganization and RelatedOrganization of the proper collection cardinality. Using the inverse attributes IsRelatedBy, Relates, Engages defined by the Organization entity, it becomes possible to find out in which relations particular Organization instances are involved.

For brevity, we omit further clarifications since the remained part of the schema specification uses similar constructs and is quite transparent to understanding.

As an example, let us consider an EXPRESS schema-driven dataset in Figure 4 presented as a data section of the corresponding ASCII SPF file. The data section contains entity instances to be transferred between applications as an exchange structure. Each instantiated entity must correspond to one EXPRESS schema specified in the header section of the file. In our example, the specified schema is ActorResource, and instantiated entities are Organization, Person, PostalAddress, TelecomAddress, and OrganizationRelationship all belonging to the schema. Each entity instance is represented at most once in the exchange structure and must have an instance name that is unique within the exchange structure. The entity instances need not be ordered in the exchange structure. An instance name may be referenced before it is defined.

The order of the instance parameters in the exchange structure are the same as the order of the corresponding attributes in the entity declaration. Each parameter list first encodes the values of the inherited explicit attributes of all supertype entities and, then the explicit attributes of the leaf entity datatype. The form of each parameter must correspond to the attribute datatype. In the presented example dataset, the PostalAddress instance with unique name #31 has the following parameters: the first parameter .OFFICE. is the value of the attribute Purpose declared in the supertype Address, the second parameter $ is the indeterminate value of the attribute UserDefinedPurpose also declared in the Address supertype, the third parameter ('9292 Automobile Dr.', 'Mc Lean', 'VA 22101') is the value of the explicit attribute AddressLines declared in the PostalAddress entity as a list of the Label strings. It is worth mentioning here that the derived and inverse attributes are not mapped to 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));
```

Fig. 4. Sample dataset driven by ActorResource schema and represented in SPF format.

IV. Formal rules to produce object notation

JSON is a text format for the serialization of structured data. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard [12]. JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays). An object is an unordered collection of zero or more name/value pairs (members), where a name is a string and a value is a string, number, boolean, null, object or array. An array is an ordered sequence of zero or more values. A string is a sequence of zero or more Unicode characters.

Let us systemize the rules how instances of data types defined in EXPRESS language can be mapped to the JSON document. Such mapping must conform to an unambiguous representation, storage and interpretation of product data regardless of format used (SPF, STEP XML or JSON).

Two general rules can be formulated independently of the predefined datatypes available in EXPRESS and from user-defined datatypes declared in EXPRESS schemas. The first rule regulates the representation of the explicit attributes declared in the schema entities as OPTIONAL. Such attributes are not required to have a value in the entity instance. When the optional value is supplied in an entity instance, it shall be represented according to the rules below. When the optional value is not supplied in an entity instance, the missing attribute value shall be encoded by JSON "null" keyword. It is worth mentioning that the dollar sign "$" is used for the same purpose in the SPF file format.

The second general rule specifies the qualification of EXPRESS datatypes. The qualification is necessary in cases where an attribute value cannot be unambiguously interpreted using the attribute definition. Typically, such cases occur for attributes of selection datatypes. The datatype qualification is also necessary for overlapping instantiations of complex entities declared with the subtype/supertype constraints ONEOF, ANDOR, AND. To avoid misinterpretations, simple entity instances shall be supplied with the qualified entity datatype. For definiteness, all the datatype names will be converted to the corresponding lowercase letter names. The qualification may be desirable in cases where software applications require additional validation of the interpreted data.

To clarify this rule, let us consider a sample dataset and the driving schema IllustrationResource both presented in Figure 5. The schema defines the entities Title, Picture and Illustration as well as a few auxiliary datatypes. The selection datatype Image is declared as a generalization of the defined datatypes Url, Png, Pixels each of which corresponds to a particular way of the representation of images. Therefore, when a Picture instance is created and its attribute “figure” is initiated, the type of the assigned value shall be additionally qualified as Url, Png, or Pixels. In the sample dataset, the Picture instance with the object identifier “#1” is specified as having the attribute “figure”, initiated by a given list of integers and qualified as “Pixels”. As seen from the example, in such cases the attribute values shall be represented as nested JSON objects with own “type”, “value” members.

The schema defines the Illustration datatype as a complex entity being a subtype of the Title and Picture entities. It implies that complex Illustration instances are composed of simple instances of the corresponding supertype instances. In the sample dataset, the Illustration instance with the object identifier “#2” is specified as being composed of the Title and Picture instances. The supertype instances shall be represented as JSON objects and qualified using the “type” member. The objects are included in an internal JSON array that is identified as a “_prototype” member of the complex instance. Simple instances are not given the object identifiers due to the uselessness. It is worth mentioning that the order of attributes in any JSON object does not matter, however, it is a good practice to keep it by setting the attributes “_oid”, “type”, and “_prototype” before others.

```
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"
           },
           {
           "type": "Picture",
           "figure": {
               "type": "Url",
               "value":
"http://www.cars.com/automobile/picture.jpg"
           }
           }
        ]
     }
]
```

Fig. 5. An example of EXPRESS datatype qualification in JSON format.

A. Mapping of EXPRESS simple, constructed and aggregation data types

EXPRESS simple, constructed and aggregation data types should be mapped to JSON primitive types in accordance with the following rules:

- NUMBER datatype shall be mapped to JSON number datatype. The representation of numbers is similar to the one used in most of the programming languages. A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed. A fraction part is a decimal point followed by one or more digits. An exponent part begins with the letter E in uppercase or lowercase, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits. Limits on the range and precision of numbers should be regulated by software implementations and are beyond of the consideration;
- A value of INTEGER datatype shall be treated as specified for an integer component of the JSON number datatype: a sequence of one or more digits with no leading zeros, optionally preceded by a minus sign "-";
- A value of REAL datatype shall be represented in the same way as the EXPRESS datatype NUMBER;
- BOOLEAN datatype shall be mapped to JSON datatype boolean. The BOOLEAN values TRUE and FALSE shall correspond to the JSON boolean values "true" and "false" respectively;
- A value of LOGICAL datatype shall be treated as a predefined enumerated datatype with the values encoded by the strings "true", "false" and "unknown" which correspond to the LOGICAL values TRUE, FALSE, and UNKNOWN;
- A value of STRING datatype shall be represented as JSON string similar to the conventions used in the C family of programming languages. A string begins and ends with quotation marks. All Unicode characters may be placed within the quotation marks, except for the characters that must be escaped: quotation mark, reverse solidus, pound symbol and the control characters (U+0000 through U+001F);
- A value of BINARY datatype shall be represented as JSON string of Base64 encoded character sequence;
- A value of ENUMERATION datatype shall be represented as JSON string corresponding to the one of the names of the enumeration items as declared in the EXPRESS schema. All uppercase letters in the names shall be converted to the corresponding lowercase letters to match the JSON text style. This conversion is done as opposed to the SPF format in which the names of the enumeration items are converted to uppercase letters and delimited by a full stop symbol ".";
- A value of ENTITY datatype, being the reference to an entity instance, shall be represented as a JSON string containing the instance name. The referenced entity instance must be presented once in the JSON document and its name must be unique within the document;
- BAG, SET, LIST, ARRAY datatypes shall be mapped to JSON array datatype. It is not required for array element values to belong to the same data type. Within the JSON array, each element shall be encoded as specified above in accordance with its data type declared in the EXPRESS schema. The ordering of the elements within the encoding must be maintained for the ordered aggregations LIST and ARRAY. Nested aggregations are represented as multi-dimensional JSON arrays in such a way that the inner-most array, the array containing only instances of the element type, shall correspond to the right-most aggregation specifier in EXPRESS statement of the nested datatype. If the aggregate is empty, it shall be represented as an empty JSON array, rather than as an indeterminate value designated by JSON "null" keyword;
- SELECT data type shall be mapped to JSON datatypes in the same way as the named datatypes listed in its EXPRESS statement. Values of the named datatypes, including nested selections, are encoded according to the rules above and must be supplied with type qualifiers to refine the datatypes of the selected values.

B. Mapping of EXPRESS entity data types

The JSON document containing product data can be thought as an array of JSON objects each corresponding to an EXPRESS ENTITY instance. Each object structure is represented as a set of members (name/value pairs).

A value of ENTITY instance shall be represented as JSON object with the following members:

- the identifier "_oid" represented as JSON string. Its value must be unique, at least among the objects within the document. Depending on the application context, the value may be globally unique. This member is mandatory for all instances not being supertype instances (see the explanations below). The predefined member name with the preceded underscore symbol should prevent naming conflicts with the underlying schema definitions. The values preceded by "#" symbol can be used to match the SPF format style;
- the datatype "type" shall be represented as JSON string. Its value must exactly match the name of the ENTITY datatype to which the instance belongs. This member is mandatory for all instances;
- the prototype "_prototype" shall be represented as an array of nested JSON objects. This member appears if only the instance belongs to complex ENTITY datatype being a specialization of several ENTITY datatypes under the subtype/supertype constraints ONEOF, ANDOR, AND (for more details on complex entity instantiations, see internal and external mappings of the EXPRESS standard [11]). In such cases each supertype instance shall be represented as JSON object of the prototype array according to the rules specified. Supertype instances are not provided with the object identifiers due to the uselessness;
- the attribute members. Each explicit attribute of the ENTITY instance shall be represented exactly by one attribute member independently on whether it was declared in the entity definition as OPTIONAL. The name of the attribute member must exactly match the name of its datatype as defined by the EXPRESS schema. The value of the attribute member shall be represented according to the rules of the mapping of simple, constructed and aggregation data types specified in the previous subsection. The datatype qualification must be applied in cases where the value cannot be unambiguously interpreted using the attribute definition. No restrictions on the order of the attribute members are imposed. DERIVED and INVERSE attributes shall not be represented as members.

C. An example of the employment of the formal rules

To illustrate the formal mapping rules, let us focus on the ActorResource schema and the sample dataset presented in Section III. It can be seen that the dataset presented in the SPF format is semantically equivalent to the JSON document encoded accordingly to the specified rules and shown in Fig. 6.

The document contains the same set of the objects of the Organization, OrganizationRelationship, Person, Postal Address and TelecomAddress types. Their attribute values are the same as in the original dataset. The object identifiers were intentionally assigned in the same way as in the SPF file to simplify the comparison.


```
[
   {
      "_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",
      "type":"Organization",
      "id":1203,
      "name":"Automobile Inc.",
      "description":"In cars we trust.",
      "roles":["Supply Chain Manager", "Executive
Manager", "Sells Manager"],
      "addresses":["#31", "#34"]
   },
   {
      "_oid":"#61",
      "type":"Person",
      "id":901,
      "familyName":"Pringle",
      "givenName":"Andrew",
      "middleNames":null,
      "prefixTitles":["Mr"],
      "suffixTitles":null,
      "roles": ["Supply Chain Manager", "Executive
Manager"],
      "addresses":["#34"],
      "engagedIn":["#11"]
   },
   {
      "_oid":"#51",
      "type":"OrganizationRelationship",
      "name":"Consumer",
      "description":null,
      "relatingOrganization":"#11",
      "relatedOrganizations":["#12","#13"]
   },
   {
      "_oid":"#31",
      "type":"PostalAddress",
      "purpose":".OFFICE",
      "userDefinedPurpose":null,
      "addressLines":["9292 Automobile Dr.", "Mc
Lean", "VA 22101"]
   },
   {
      "_oid":"#34",
      "type":"TelecomAddress",
      "purpose":".OFFICE",
      "userDefinedPurpose":null,
      "telephoneNumbers":["678-762-2354", "678-762-
2355"],
      "facsimileNumbers":null,
      "electronicMailAddresses":
["automobile@cars.com"],
      "WWWUrls":["http://www.cars.com/automobile"]
   }
]
```


Fig. 6. Sample dataset driven by ActorResource schema and encoded in JSON format.

It is important to note that the rules introduced define how to present product data driven by EXPRESS schemas as

JSON documents. It also seems possible to present EXPRESS schemas themselves as JSON documents. However, it has been shown that full mapping is too complicated even for a particular schema, such as Industry Foundation Classes (IFC) [13]. It is explained by the availability of algebraic specifications of the INVERSE, DERIVED attributes as well as various kinds of WHERE, UNIQUE, and GLOBAL rules. Incomplete alternative representations of EXPRESS schemas are not of great interest because they cannot be used for product data validation purposes [14].

CONCLUSIONS

Thus, the general rules for producing JSON notation for EXPRESS schema-driven data are formulated and presented with explanatory examples. The rules allow to produce an unambiguous, non-redundant and implementation-neutral data representation in the JSON file format. The universality of the rules in relation to arbitrary schemas formally specified in the EXPRESS language facilitates their widespread adoption in Web services dedicated to manage product information models in various industries.

The directions of future research involve practical aspects of the implementation of the proposed rules as a part of the existing and emerging PDM systems to accelerate exchange of complex engineering data and improve processing of long transactions.

REFERENCES

[1] R.D. Barad. PDM: the essential technology for concurrent engineering. Spvryan's International Journal of Engineering Sciences & Technology (SEST), 2015, vol. 2, no. 3, pp. 1-8.

[2] J. Osborn. Survey of concurrent engineering environments and the application of best practices towards the development of a multiple industry, multiple domain environment. Clemson University TigerPrints, 2009.

[3] ISO 10303. Industrial automation systems and integration — Product data representation and exchange.

[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] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta. Comparison of JSON and XML Data Interchange Formats: A Case Study. Proceedings of the ISCA 22nd International Conference on Computer Applications in Industry and Engineering, 2009.

[6] D. Peng, L. Cao, W. Xu. Using JSON for data exchanging in web service applications. Journal of Computational Information Systems, 2011, vol. 7, no. 16.

[7] A. Šimec, M. Magličić. Comparison of JSON and XML Data formats. Proceedings of Central European Conference on Information and Intelligent Systems, 2014, pp. 272-275.

[8] I. Jørstad, E. Bakken, T. A. Johansen. Perfomance evaluation of JSON and XML for data exchange in mobile services. Proceedings of the International Conference on Wireless Information Networks and Systems, 2008, pp. 237-240.

[9] A. Sumaray, S. Kami Makki. A Comparison of Data Serialization Formats For Optimal Efficiency on a Mobile Platform. Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, 2012, pp. 1-6.

[10] ISO 10303-11:2004. Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.

[11] ISO 10303-11:1994. Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.

[12] T. Bray. The JavaScript Object Notation (JSON) data interchange format. Internet Engineering Task Force (IETF), 2014.

[13] K. Afsari, C. M. Eastman, D. Castro-Lacouture. JavaScript Object Notation (JSON) data serialization for IFC schema in web-based BIM data exchange. Automation in Construction, 2017, vol. 77, pp. 24-51.

[14] V. Semenov, D. Ilyin, S. Morozov, O. Tarlapan. Effective consistency management for large-scale product data. Journal of Industrial Information Integration, 2019, vol. 13, pp. 13-21.