ETSI ES 201 873-11 V4.10.1 (2023-05)

Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 11: Using JSON with TTCN-3

Reference
RES/MTS-20187311v4.10.1

Keywords
JSON, language, testing, TTCN-3

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

The present document is part 11 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document specifies the rules to define schemas for JSON data structures in TTCN-3, to enable testing of JSON-based systems, interfaces and protocols, and the conversion rules between TTCN-3 [1] and JSON [2] to enable exchanging TTCN-3 data in JSON format between different systems.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference/.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] IETF RFC 7159: "The JavaScript Object Notation (JSON) Data Interchange Format".

[3] ISO/IEC 10646:2017: "Information technology -- Universal Coded Character Set (UCS)".

[4] IEEE 754™: "IEEE Standard for Floating-Point Arithmetic".

[5] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] IETF draft-handrews-json-schema-validation-00: "JSON Schema Validation: A Vocabulary for Structural Validation of JSON".

[i.2] World Wide Web Consortium W3C® Recommendation: "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures".

[i.3] World Wide Web Consortium W3C® Recommendation: "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes".

[i.4] ETSI ES 201 873-8: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.5] ETSI ES 201 873-9: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.6] ETSI ES 202 781: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support".

[i.7] ETSI ES 202 782: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Performance and Real Time Testing".

[i.8] ETSI ES 202 784: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Advanced Parameterization".

[i.9] ETSI ES 202 785: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Behaviour Types".

[i.10] ETSI ES 202 786: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Support of interfaces with continuous signals".

[i.11] ETSI ES 202 789: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Extended TRI".

[i.12] ETSI ES 203 022: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language extension: Advanced Matching".

[i.13] ETSI ES 203 790: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Object-Oriented Features".

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI ES 201 873-1 [1] apply.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
IO	ASN.1 Information Object
JSON	JavaScript Object Notation
SUT	System Under Test
TTCN-3	Testing and Test Control Notation version 3
UCS	Universal Coded Character Set
USI	UCS Sequence Identifier
UTF-8	Unicode Transformation Format-8
XML	eXtensible Markup Language
XSD	XML Schema Definition

4 Introduction

An increasing number of distributed applications use the JSON to exchange data for various purposes like data bases queries or updates or event telecommunications operations such as provisioning. The JSON specification [2] defines the syntax and encoding for JSON types and defined literals, but no semantics is defined. JSON does not have a schema specification, like the XML Schema Definition Language used for XML documents (see [i.2] and [i.3]).

NOTE: Though an IETF draft proposal exists for JSON structural validation (see [i.1]), it has not reached the RFC status.

The core language of TTCN-3 is defined in ETSI ES 201 873-1 [1] and provides a full text-based syntax, static semantics and operational semantics. Other parts of the ETSI ES 201 873 series are defining its use with other specification languages like ASN.1 [5], IDL [i.4], or XSD [i.5] as shown in figure 1, while other documents as ETSI ES 202 781 [i.6], ETSI ES 202 782 [i.7], ETSI ES 202 784 [i.8], ETSI ES 202 785 [i.9], ETSI ES 202 786 [i.10], ETSI ES 202 789 [i.11], ETSI ES 203 022 [i.12] and ETSI ES 203 790 [i.13] specify language extensions and thus can define additional rules to the JSON/TTCN-3 mapping defined in the present document.

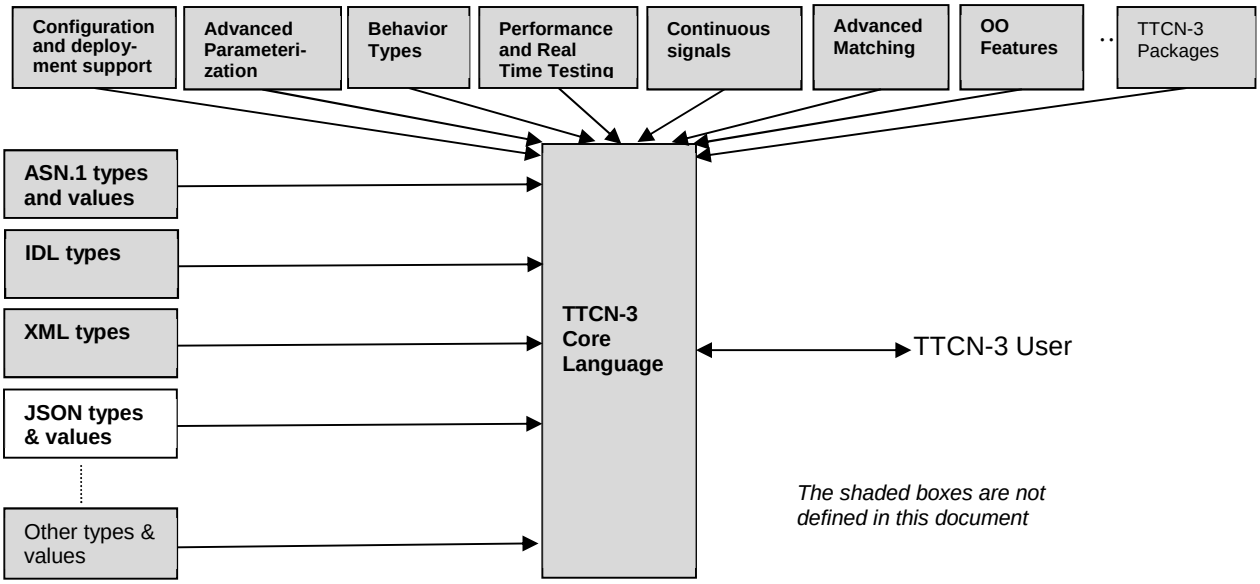

Figure 1: User's view of the core language and its packages

In the context of TTCN-3, JSON can be used for different purposes:

1) TTCN-3 can be used as a JSON Schema definition language that allows generating JSON values from TTCN-3 and consuming and evaluating received JSON values, i.e. enables testing of JSON-based interfaces and protocols.

2) To exchange type and data information between the TTCN-3 test system and systems written in other languages like Java, C, C++, Python, etc. In this way TTCN-3 test systems can be used as a subsystem of a more complex test system; for example, the TTCN-3 system receiving contents of messages to be sent to an SUT, encode and send a message, receive and process the response and report the result to the other system.

Consequently, there is a need to specify mappings between JSON and TTCN-3 for the above purposes.

5 Conformance and compatibility

For an implementation claiming to support the use of TTCN-3 as a JSON schema language, all features specified in clause 6 of the present document shall be implemented consistently with the requirements given in clause 6 and Annex B of the present document and in ETSI ES 201 873-1 [1].

For an implementation claiming to support the exchange of TTCN-3-based data between systems, and not supporting using ASN.1 with TTCN-3, all features specified in clause 7 of the present document, with the exception of the mapping of the objid type in clause 7.2.11 shall be implemented consistently with the requirements given in clause 7 and Annex B of the present document and in ETSI ES 201 873-1 [1]. Implementations claiming the support of using ASN.1 with TTCN-3, shall in addition support features in clause 7.2.11 and clause 8 of the present document.

The language mappings presented in the present document is compatible to:

- ETSI ES 201 873-1 [1], version 4.9.1.

If later versions of those parts are available and should be used instead, the compatibility of the rules presented in the present document shall be checked individually.

6 Using TTCN-3 as JSON Schema

6.1 Approach

JSON [2] defines a limited set of JSON types and literal values. The clauses below define the TTCN-3 types that can be used to specify a Schema for any JSON interface specification. The TTCN-3 types defined in the clauses below will allow to use the same set of values as JSON permits. Annex A provides a TTCN-3 module containing all TTCN-3 definitions specified in these clauses. The **JSON** module in Annex A shall either explicitly be present in TTCN-3 test suites or TTCN-3 tools shall support these types implicitly. This is left as a tool implementation option.

JSON in many cases allows different encoding options for the same value. These may be controlled by the JSON encoding instructions specified in Annex B. JSON encoding instructions may be added to TTCN-3 types and fields by using TTCN-3 variant attributes (see ETSI ES 201 873-1 [1], clause 27.5).

6.2 Validation of JSON Values

For further study.

6.3 Name conversion rules

The current version of the JSON specification [2] uses names to identify JSON object members only. The present document defines JSON to TTCN-3 name conversion rules that shall be used when using TTCN-3 to specify a schema for a JSON interface (see for example clause 6.4.4). When the JSON and the TTCN-3 names differ after applying the rules in this clause, the "name as …" encoding instruction shall be used to identify the exact JSON name. To ensure compatibility with future versions and automatic conversions, the rules specified in this clause should always be applied.

JSON names can be identical to TTCN-3 reserved words, can contain characters not allowed in TTCN-3 identifiers or allowed to be identical, when the corresponding TTCN-3 names are required to be unique, in which case the JSON names shall be processed by the rules below to obtain the corresponding TTCN-3 identifiers.

The following character substitutions shall be applied, in order that each character string being mapped to a TTCN-3 name, where each substitution (except the first) shall be applied to the result of the previous transformation:

a) any character except "A" to "Z" (Latin Capital Letter A to Latin Capital Letter Z), "a" to "z" (Latin Small Letter A to Latin Small Letter Z), "0" to "9" (Digit Zero to Digit Nine), and "_" (Low Line) shall be removed;

b) a sequence of two or more "_" (Low Line) characters shall be replaced with a single "_" (Low Line);

c) "_" (Low Line) characters occurring at the beginning or at the end of the name shall be removed;

d) if a character string starts with a digit (Digit Zero to Digit Nine), it shall be prefixed with an "**x**" (Latin Small Letter X) character;

e) if a character string is empty, it shall be replaced by "**x**" (Latin Small Letter X);

f) if the TTCN-3 name being generated is identical to a previously generated TTCN-3 identifier in the same scope, then a postfix shall be appended to the character string generated by the above rules. If a field name of a TTCN-3 structured type is clashing with a type's name used in the same structured type, the field's name shall be postfixed. The postfix shall consist of a "_" (Low Line) followed by an integer. This integer shall be the least positive integer such that the new identifier is different from the identifier of any previously generated identifier in the same scope (i.e. the first postfix applied by this mechanism is "_1"). TTCN-3 names that are one of the TTCN-3 keywords (see clause A.1.5 of ETSI ES 201 873-1 [1]) or names of predefined functions (see clause 16.1.2 of ETSI ES 201 873-1 [1]) after applying the postfix to clashing names, shall be suffixed by a single "_" (Low Line) character.

6.4 Mapping of JSON Values

6.4.1 JSON Numbers

JSON numbers are represented as base 10 decimal digits containing a mandatory integer component that can be prefixed with an optional minus sign, and can be followed by a fraction part, an exponent part or both. Leading zeros are not allowed. JSON does not distinguish numbers based on their value sets like integers and reals, like other languages do. No special values (as –infinity, infinity or NaN) are allowed.

In the general case, JSON numbers shall be mapped by using the following TTCN-3 type:

```
type float Number (!-infinity .. !infinity) with {
  variant "JSON:number"
  }
```

When the JSON interface specification requires a number to conform to the IEEE 754 [4] floating-point number specification, the IEEE 754 [4] floats useful types of clause E.2.1.4 of ETSI ES 201 873-1 [1] can be used in the context of JSON encoding, in which case, by default, the given useful type will constrain the value set and the encoding of the JSON value according to this clause. The JSON encoding instructions in this case can be applied to fields of IEEE 754 [4] useful types.

By default, i.e. without any encoding instruction applied, the form of the JSON representation of *JSON.Number* is a tool implementation option (i.e. the number of fraction digits, using the exponent part, etc.)

To make defining JSON Schemas in TTCN-3 easier, the present document, in addition to the generic mapping of JSON numbers, also specifies a TTCN-3 type that may be used where the interface specification allows only numbers without the fraction and the exponent parts:

```
type integer Integer (-infinity .. infinity) with {
    variant "JSON:integer"
}
```

Attempts to decode a JSON number value with either a fraction or an exponent part or both into this *JSON.Integer* type shall cause a decoding failure.

In addition to the generic encoding instructions like "normalize" and "name as …", the following specific instructions shall be applicable to types and fields of *JSON.Number* and the IEEE 754 [4] useful types:

- fractionDigits see clause B.3.5
- useMinus see clause B.3.6

NOTE: The beginning character of the exponent part can be both "e" and "E". This is not controlled by any of the encoding instructions but left as a tool implementation option.

and to types and fields of *JSON.Integer* types:

- useMinus see clause B.3.6

6.4.2 JSON Strings

A JSON string is a sequence of zero or more Unicode characters, enclosed in a pair of quotation mark characters ("""", **char**(U22)). Any characters may be escaped by the escape sequence: "\u<HHHH>", where <HHHH> represents four hexadecimal digits, but the icharacters: quotation mark ("""", **char**(U22)), reverse solidus ("\", **char**(U5C)) and all C0 control characters (**char**(U0) through **char**(U1F)) shall be escaped.

Alternatively, the short, two-character escape sequences defined in table 1 can be used to escape some of the characters.

Table 1: Short character escape sequences

Character's name	Character code	Short escape sequence
quotation mark	**char**(U22)	\"
reverse solidus	**char**(U5C)	\\
solidus	**char**(U2F)	\/
backspace	**char**(U8)	\b
form feed	**char**(UC)	\f
line feed	**char**(UA)	\n
carriage return	**char**(UD)	\r
horizontal tab	**char**(U9)	\t

By default, it is a tool implementation option which form of escaping is used, which may be overridden by the "escape as …" encoding instruction.

NOTE 1: Note that the JSON module in Annex A defines useful TTCN-3 constants for the characters listed above.

The following TTCN-3 type shall be used to map JSON strings to TTCN-3:

```
type universal charstring String with {
  variant "JSON:string"
}
```

NOTE 2: Though Unicode and ISO/IEC 10646 [3] do not necessarily contain the same set of characters at all points in time, JSON strings are expressed using the TTCN-3 universal charstring type.

In addition to the generic encoding instructions like "normalize" and "name as …", the following specific encoding instructions are applicable to **JSON.String** types:

- escape as … see clause B.3.7

EXAMPLE: String encoding examples

If:

```
const JSON.String c_string1 := <actual value> with {variant "escape as short"};
```

then:

<actual value>	JSON character sequence	UTF-8 serialization of the JSON value	Note
"abcd"	"abcd"	226162636422	
"ab\cd"	"ab\\cd"	2261625C5C636422	
"ab/cd"	"ab\/d"	2261625C2F636422	
"ab" & char(U7) & "cd"	"ab\u0007cd"	2261625C75303030 37636422	
"ab" & char(U7) & cu_ht & "cd"	"ab\u0007\tcd"	2261625C7530303037 5C74636422	

If:

```
const JSON.String c_string1 := <actual value> with {variant "escape as usi"};
```

then:

<actual value>	JSON character sequence	UTF-8 serialization of the JSON value	Note
"abcd"	"abcd"	226162636422	
"ab\cd"	"ab\u005Ccd"	2261625C7530303543636422	
"ab/cd"	"ab\u002Fcd"	2261625C7530303246636422	Escaped solidus character (escaping solidus is optional in JSON)
"ab" & char(U7) & "cd"	"ab\u0007cd"	2261625C7530303037636422	
"ab" & char(U7) & cs_ht & "cd"	"ab\u0007\u0009cd"	2261625C75303030375C7530303039636422	

If:

```
const JSON.String c_string1 := <actual value> with {variant "escape as transparent"};
```

then

<actual value>	JSON character sequence	UTF-8 serialization of the JSON value	Note
"abcd"	"abcd"	226162636422	
"ab\cd"	"ab\cd"	2261625C636422	Note that the resulting sequence is an invalid JSON encoding
"ab/cd"	"ab/cd"	2261622F636422	
"ab" & char(U7) & cs_ht & "cd"	"ab\u0007\tcd"	2261625C75303030375C74636422	Note that the BELL and HT C0 control characters are escaped by the encoder

6.4.3 JSON Arrays

JSON arrays can contain a sequence of zero or more JSON values, i.e. the array members may be of different JSON "types".

The following TTCN-3 type shall be used to map JSON arrays to TTCN-3:

```
type record of JSON.Values Array with {
    variant "JSON:array"
},
```

Where:

```
type union Values {
    JSON.String str,
    JSON.Integer int,
    JSON.Number num,
    JSON.Object obj,
    JSON.StrArray strArray,
    JSON.IntArray intArray,
    JSON.NumArray numArray,
    JSON.BoolArray boolArray,
    JSON.ObjectArray objArray,
    JSON.Array array,
    JSON.Bool bool,
    JSON.Null null_
} with {
    variant "asValue"
}
```

To make specifying JSON Schemas easier for values, when according to the interface specification a specific array can contain a sequence of values of the same JSON "type", also the TTCN-3 types below are specified:

NOTE: Use the below subsidiary types with due precaution. The syntax of TTCN-3 values based on the below helper type differs from the syntax of a **`JSON.Array`** value. Therefore, changes in an array description in a JSON interface specification can require changing the TTCN-3 code as well.

```
type record of JSON.String StrArray with {
    variant "JSON:array"
}

type record of JSON.Number NumArray with {
    variant "JSON:array"
}

type record of JSON.Integer IntArray with {
    variant "JSON:array"
    }

type record of JSON.Boolean BoolArray with {
    variant "JSON:array"
}

type record of JSON.Object ObjArray with {
    variant "JSON:array"
}
```

Where JSON.Object is:

```
type record Object {
    record length (1..infinity) of JSON.ObjectMember memberList optional
} with {
    variant "JSON:object"
}
```

And:

```
type record ObjectMember {
    JSON.String name, // shall contain an object member's name
    JSON.Types value_ // shall contain an object member's value
} with {
    variant "JSON:objectMember"
}
```

There is no type-specific encoding instruction defined for ***JSON.Array***, ***JSON.StrArray***, ***JSON.NumArray***, ***JSON.IntArray***, ***JSON.BoolArray***, ***JSON.ObjArray***, in addition to the generic ones like "normalize" and "name as …". In addition to the generic encoding instructions, the **JSON.Types** type uses the following instruction:

- asValue see clause B.3.10.

EXAMPLE: TTCN-3 Schema for a JSON array

Provided a JSON interface specification, allows a list of arbitrary JSON values e.g. as the value part of an object member. Its TTCN-3 schemata will be:

```
type JSON.Array MyValue;
```

And e.g. the TTCN-3 value:

```
const MyValue c_myValue := {
    { str := "abcd" },
    { num := 1.0 },
    { int := 42 },
    { intArray := { 1, 2, 3, 4, 5, 6 },
    { null_ := null_ }
};
```

will be encoded in JSON e.g. as (character sequence):

```
["abcd",   1e0,  42,   [ 1, 2, 3, 4, 5, 6 ], null ]
```

Note that as no additional instruction is specified for the "num" element, its encoding is a tool option.

6.4.4 JSON Objects

JSON object values consist of unordered sequences of zero or more object members, where each object member is constructed of a name-value pair. The JSON specification IETF RFC 7159 [2] does not require uniqueness of object member names within a JSON object.

JSON object specifications should be translated to TTCN-3 **`record`**-s. The name of the record shall be the product of applying clause 6.3 to the name of the object being translated, if this is specified, or be an arbitrary valid TTCN-3 identifier otherwise. The name of the type shall not be "`Object`", if this is the name of the JSON object being translated, the postfixing rules specified in item f) of clause 6.3 shall be applied to it.

Each member of the object being converted shall generate a record field, where the name of the field shall be the product of applying clause 6.3 to the name of the object member, but it shall not be "`order`": if this would be the name of the TTCN-3 field, the postfixing rules specified in item f) of clause 6.3 shall be applied to it, as if it was preceded by a field named `order`. The type of the field shall be a type defined in clause 6 of the present document, corresponding to the JSON type (or literal values allowed) of the object member. If the object member can carry values of different JSON types (or literals), the type of the field shall be a **`union`** of the TTCN-3 types required to represent all possible values of the object member being translated, where the union field names shall be the names of the field's type with lower cased first character.

Following this translation an **`optional`** TTCN-3 **`record of JSON.String`** field named `order` may be inserted as the first field of the record, and an **`optional`** field named `memberList` of the type **`record length (1..infinity) of JSON.ObjectMember`** shall be inserted as the last field of the record.

Possible name clashes caused by inserting the `memberList` field shall be resolved according to item f) of clause 6.3 with the exception that the additionally inserted field `memberList` shall be handled as if it was the first field of the record (i.e. if an object member with the name "memberList" exists, the object member's name shall be postfixed). Finally the "JSON:object" and if the `order` field is present the "useOrder" encoding instructions shall be attached to the generated record type. Any other encoding instructions may be attached to the record fields, as necessary for a correct translation.

The `memberList` field allows inserting any extra object members in instance definitions, and converters shall encode each `ObjectMember` as member of the JSON object being produced. At JSON to TTCN-3 conversion JSON object members, which cannot be decoded into any other field being produced from the member's name, shall be decoded as element of the `memberList` field.

Use of the `order` field is specified in clause B.3.12.

NOTE: Note that attaching the **optional** "implicit omit" attribute to TTCN-3 values and templates implementing object instances will allow skipping unused `order` and `memberList` fields in the instance definitions.

EXAMPLE: TTCN-3 Schema for a JSON object

Suppose that the specification defines a JSON object that shall carry the location information "Latitude", "Longitude", which are floating point numbers and may carry the "Precision" and "Address" information, where precision is to be provided as a number and the address is another JSON object containing the city, street and house number information, where city and street are JSON strings, house number is an integer value.
The schema of this specification can be described in TTCN-3 e.g. as:

```
module MyObjectSchema {

    import from JSON all;

    type record Coordinates {
        record of JSON.String order optional,
        JSON.Number Latitude,
        JSON.Number Longitude,
        JSON.Number Precision optional,
        Address Address_1 optional,
        record length (1..infinity) of JSON.ObjectMember memberList optional
    } with {
        variant "JSON:object";
        variant "useOrder"
        variant(Address_1) "name as 'Address'";
      }

    type record Address {
        record of JSON.String order optional,
        JSON.String city,
        JSON.String street,
        JSON.Integer house_no_,
        record length (1..infinity) of JSON.ObjectMember memberList optional
    } with {
        variant "JSON:object";
```

```
        variant "useOrder";
        variant (house_no_) "name as 'house no.' "
    }


} with { encode "JSON" }
```

And the TTCN-3 templates:

```
    template Coordinates t_coordinates := {
        Latitude := 51.523704,
        Longitude := -0.158553,
        Address_1 := t_address
    } with { optional "implicit omit" }

    template Address t_address := {
        order := {"house_no_","subno","street","city"},
        city := "London",
        street := "Baker",
        house_no_ := 221,
        memberList := {{"subno",{str:="B"}}}
    }
```

will be encoded in JSON as:

```
{"Latitude":51.523704,"Longitude":-0.158553,"Address":
{"house no.":221,"subno":"B","street":"Baker","city":"London"}}
```

There is no type-specific encoding instruction defined for **`JSON.Object`**, in addition to the generic ones like "normalize" and "name as …".

6.4.5 JSON Literals

JSON specifies three literal values: ***`true`***, ***`false`*** and ***`null`***. These are mapped to the TTCN-3 types:

```
//When only the true and false literals are allowed
type boolean Bool with { variant "JSON:literal" }

//When only the null literal is allowed
type enumerated Null { null_ } with { variant "JSON:literal" }
```

NOTE: In the case if a JSON value could contain all three defined JSON literals, the user can define a union type of the above types.

There is no type-specific encoding instruction defined for the above types mapping JSON literals, in addition to the generic ones like "normalize" and "name as …".

EXAMPLE:

The TTCN-3 value:

```
const JSON.Bool c_true := true;
```

will be transformed to JSON e.g. as:

```
true
```

(its UTF-8 serialization is 74727565).

7 Using JSON to exchange data between TTCN-3 and other systems

7.1 General rules

Clause 7 of the present document specifies converting abstract TTCN-3 values into their JSON representation (see IETF RFC 7159 [2]) and converting JSON data into values of abstract TTCN-3 types. Clause 7 of the present document covers the conversion rules for the different TTCN-3 data types, while the encoding instructions, influencing the conversion are detailed in Annex B.

By default all instances of **top-level** TTCN-3 types are represented by a "wrapper" JSON object with a single object member, where the name of the object member shall be:

- the name of a **built-in** TTCN-3 type, except for **`anytype`**-s;
- the qualified name of the built-in TTCN-3 type **`anytype;`** or
- the qualified name of a user-defined TTCN-3 **type** (in case of a type alias, the name of the alias type shall be used);
- and the value of the object member is the JSON representation of the value of the given instance. The name of the top-level TTCN-3 type shall be transformed to a **`JSON.String`**.

In the TTCN-3 to JSON direction, generating the type-name wrapper object for the top-level type can be disabled by attaching the "noType" encoding instruction (see clause B.3.11). In the JSON to TTCN-3 direction, the presence of the TTCN-3 type's name in the JSON "wrapper object" makes the conversion unambiguous, but tools shall transform JSON values without the type-name wrapper object to TTCN-3 values, if the type of the TTCN-3 instance can be determined (e.g. known from an external function declaration or from a port type definition).

As the JSON value's syntax is unambiguous for float and boolean values only, tools should not rely on the JSON value's syntax only when determining the TTCN-3 type.

Conversion to and from JSON is allowed for types with the **`encode`** attributes specified in clause B.2.

EXAMPLE 1: Default conversion using the type wrapper

The TTCN-3 constant MyChar:

```
module Mymodule {
    type charstring MyChar with { encode "JSON" };
    const   MyChar c_char := "abc";
}
```

Will be represented in JSON as:

```
{ "Mymodule.MyChar" : "abc" }
```

EXAMPLE 2: Conversion without the type wrapper

```
type charstring MyChar with { encode "JSON"; variant "noType" };

const   MyChar c_char := "abc";
```

Will be represented in JSON as:

```
"abc"
```

7.2 JSON representations of TTCN-3 values

7.2.1 Character strings

TTCN-3 **`charstring`**, **`universal charstring`** values shall be encoded as **`JSON.Strings`** (see clause 6.4.2).

Charstrings shall appear exactly like in TTCN-3, with the exception that in the JSON representation the quotation mark (**`char`**`(U22)`, reverse solidus (**`char`**`(U5C)` and all C0 control characters (**`char`**`(U0)` through **`char`**`(U1F)` shall be escaped. Both forms of escaping, i.e. the USI-like \uHHHH and the short format can be used, unless otherwise regulated by the "escaped as…" encoding instruction (see clause B.3.7).

Universal charstrings shall be represented in JSON strings with UTF-8 encoding. JSON strings can contain the escaped character `\u` followed by 4 hex digit characters, the decoder shall convert this into the character represented by the hex digits.

EXAMPLE:

The TTCN-3 value:

```
const universal charstring c_uchar := char(U9) & "my string";
```

Will be represented in JSON as:

```
{ "universal charstring" : "\u0009my string" }
```

7.2.2 Binary Strings

TTCN-3 **bitstring**, **`hexstring`** and **`octetstring`** values shall be encoded in JSON as **`JSON`**.**`String`**-s (see clause 6.4.2) containing the bits or hex digits as capital characters. At JSON to TTCN-3 conversion all characters allowed for binary TTCN-3 string types as specified in clause 6.1.1 of ETSI ES 201 873-1 [1] shall be accepted and converted to TTCN-3, the characters Space (char(U20)), Horizontal tab (char(U9)), Line feed (char(UA)) and Carriage return (char(UD)) shall be ignored, while any other character shall cause an error, unless specified differently by the error behaviour encoding instruction (see clause B.3.13).

EXAMPLE:

The TTCN-3 value:

```
const hexstring c_hex1 := '00ABC'H;
const hexstring c_hex2 := '00abc'H;
```

Both values will be transformed to the JSON value:

```
{ "hexstring" : "00ABC" }
```

The JSON value:

```
{ "hexstring" : "00 abc" }
```

will be transformed to the TTCN-3 value:

```
'00ABC'H
```

7.2.3 Integer

All TTCN-3 **`integer`** values shall be encoded as the JSON numbers (see clause 6.4.1), without the optional fraction and exponent parts.

At decoding the JSON -0 value, by default shall be converted to the TTCN-3 value 0.

NOTE: Detection of the minus sign in the JSON -0 value during decoding is possible only if the decoded field is of a float type and the "useMinus" encoding instruction is attached to it (see clause B.3.6).

EXAMPLE:

The TTCN-3 value:

```
const integer c_int := 42;
```

Will be represented by the JSON value:

```
{ "integer" : 42 }
```

7.2.4 Float

Numeric TTCN-3 **`float`** values shall be encoded as JSON numbers (see clause 6.4.1).

At encoding the minus sign of the TTCN-3 –0.0 value shall be preserved. At decoding the JSON negative zero values, by default shall be converted to the TTCN-3 value 0.0, unless the "useMinus" encoding instruction is applied to the TTCN-3 type (see clause B.3.6).

The special float values "`infinity`", "`-infinity`" and "`not_a_number`" are encoded as JSON strings.

EXAMPLE:

The TTCN-3 value:

```
const float c_float := -42.5;
```

Will be represented by the JSON value:

```
{ "float" : -4.25E1 }
```

7.2.5 Boolean

TTCN-3 **`boolean`** values shall be encoded in JSON as the literals **`true`** and **`false`**.

EXAMPLE:

The TTCN-3 value:

```
const boolean c_bool := true;
```

Will be represented by the JSON value:

```
{ "boolean" : true }
```

7.2.6 Enumerated

TTCN-3 **`enumerated`** values shall be encoded as **`JSON.String`**-s.

For enumerated values with a single implicit or explicit associated integer value the string shall contain the name of the enumerated value.

For enumerated values with an associated integer value list or range, the string shall contain the name of the enumerated value and a single integer value, following the enumeration name in a pair of parenthesis, without any space.

NOTE: Enumerated values not defined by the relevant TTCN-3 type can be received from an external system, if the error handling behaviour for the top-level type is set to EB_WARNING or EB_IGNORE (see clause B.3.13), however, in this case the whole received JSON value is handled in TTCN-3 as a universal charstring value.

EXAMPLE:

The TTCN-3 values:

```
type enumerated MyEnumType {
    blue(0),
    yellow(1),
    green(3),
    other(2, 4..255)
} with { encode "JSON" };

const MyEnumType c_enum1 := blue;
const MyEnumType c_enum2 := other(4);
```

Will be represented by the JSON values respectively:

```
{ "MyEnumType": "blue" }
{ "MyEnumType": "other(4)" }
```

7.2.7 Verdicttype

TTCN-3 **verdicttype** values shall be encoded in JSON as ***JSON.String***-s. The string shall contain one of the values: "pass", "fail", "inconc" or "none". Any other value shall cause an error unless an error behaviour encoding instruction specifies otherwise (see clause B.3.13).

NOTE: Other verdict values can be received from an external system, if the error handling behaviour for the top-level type is set to EB_WARNING or EB_IGNORE (see clause B.3.13) and no type wrapper is present, the whole received JSON value is handled in TTCN-3 as a universal charstring value.

EXAMPLE:

The TTCN-3 value:

```
const verdicttype c_verdict := pass;
```

Will be represented e.g. by the JSON value:

```
{"verdicttype":"pass"}
```

7.2.8 Record and set

The body of TTCN-3 **record** and **set** values shall be encoded to JSON objects.

Each object member shall represent a field, where the object member's name shall be the name of the field and its value shall be the value of the field.

Omitted optional TTCN-3 fields can be handled in different ways in JSON:

- By default (i.e. when none of the below encoding instruction is applied to the field in its corresponding type definition), it shall be omitted at TTCN-3 to JSON conversion, i.e. no object member is generated for the field. At JSON to TTCN-3 conversion, **omit** shall be assigned to optional fields that do not appear in the JSON value.

- If the "omit as null" encoding instruction (see clause B.3.8) is applied to the corresponding field of its type definition, the omitted TTCN-3 field is represented in JSON as an object member, where the object member's name shall be the name of the field and its value shall be the ***null*** JSON literal value.
  At JSON to TTCN-3 conversion, JSON object members with the literal value ***null*** shall be converted to **omit** for TTCN-3 data-type fields, and to the TTCN-3 value **null** for default and component type fields.

- If the "default" encoding instruction (see clause B.3.9) is applied to the corresponding field of its type, no JSON object member shall be generated for the TTCN-3 field, while at JSON to TTCN-3 conversion, if no object member corresponds to the field, the default value defined in the instruction shall be assigned to the given TTCN-3 field.

At TTCN-3 to JSON conversion the order of the object members shall be the same as the order of the fields in the TTCN-3 value, for both **record**s and **set**s.

At JSON to TTCN-3 conversionJSON object members shall be accepted in any order, and in case of **record** types the TTCN-3 fields shall be ordered according to their textual order in the type definition, while in case of **set** types no re-ordering shall apply, i.e. the field order in TTCN-3 shall correspond to the object member's order in the JSON value.

EXAMPLE 1: Converting a record value

The TTCN-3 value c_myRecord*:*

```
module MyRecExample1 {

type record MyRecord {
    integer int,
    Myset myset
}

type set Myset {
    float value_,
    boolean case_
}

type record of integer MyRecordOfInt;

const MyRecord c_myRecord := { 5, { 5.5, true } }

} with { encode "JSON" }
```

Will be represented in JSON e.g. as:

```
{ "MyRecExample1.MyRecord" : { "int":5 , "myset" : { "value_":5.5, "case_":true }}}
```

EXAMPLE 2: Effect of the noType encoding instruction

When the TTCN-3 definitions in example 1 above are used, but adding the "noType" instruction:
(adding the instruction to MyRecord directly, would have the same effect)

```
module MyRecExample1 {
...
} with { encode "JSON"; variant "noType" }
```

In which case the above value c_myRecord is represented in JSON without the wrapper type-name object:

```
{ "int":5 , "myset" : { "value_":5.5, "case_":true }}
```

EXAMPLE 3: Effect of the "omit as null" encoding instruction

```
module MyRecExample2 {

type record PhoneNumber {
    integer countryPrefix optional,
    integer networkPrefix,
    integer localNumber
} with {        variant(countryPrefix) "omit as null"   }
const PhoneNumber c_pn := { omit, 20, 1234567 }
} with { encode "JSON"}
```

Will be represented in JSON e.g. as:

```
{ "MyRecExample2.PhoneNumber" :
    {"countryPrefix":null, "networkPrefix":20, "localNumber":1234567}}

// For comparison, the JSON representation without the attribute would be:
// { "MyRecExample2.PhoneNumber" : {"networkPrefix":20, "localNumber":1234567 }}
```

7.2.9 Record of, set of and arrays

TTCN-3 **record of**, **set of** and array values shall be encoded in JSON as arrays. The elements of the array shall be the JSON representations of the corresponding TTCN-3 elements.

JSON array elements shall appear in the same order as in the TTCN-3 value.

EXAMPLE:

The TTCN-3 value c_myRecOf*:*

```
module MyRecOfExample {

type record of integer MyRecordOfInt;

const MyRecordOfInt c_myRecOf := {1,2,3}

} with { encode "JSON" }
```

Will be represented in JSON e.g. as:

```
{ "MyRecOfExample.MyRecordOfInt" : [1,2,3] }
```

7.2.10 Union and anytype

By default TTCN-3 **union**s and **anytype** fields shall be encoded as JSON objects. The object shall contain one object member, the name of which shall be the name of the selected TTCN-3 alternative (name of the field for unions and name of the type for anytypes) and its value shall be the value of the selected TTCN-3 field.

EXAMPLE 1: Union representation

The TTCN-3 value c_myUnion*:*

```
module MyUnionExample {
    type union U1 { // proposed order of fields
        integer i,
        float f,
        octetstring os,
        charstring cs
    } with { encode "JSON" }

    const U1 c_myUnion := { f := 42.5 }
} with { encode "JSON" }
```

Will be represented in JSON e.g. as:

```
{ "MyUnionExample.U1" : { "f" : 4.25E1 }}
```

To TTCN-3 **union**s the "asValue" encoding instruction (see clause B.3.10) can be applied, in which case the JSON representation shall only contain the value of the chosen alternative, i.e. the TTCN-3 value is represented as the corresponding JSON value, without the name of the selected field. At JSON to TTCN-3 conversion the first alternative of the TTCN-3 **union** type shall be selected, allowed by the JSON value's syntax.

NOTE 1: The "as Value" is not allowed for **anytype**-s, as it is an "implicit union" without defined order of alternatives.

NOTE 2: The "asValue" instruction should be used with due caution. It is a good idea to declare more restrictive fields before less restrictive ones; e.g.: hexstring is more restrictive than universal charstring, because hexstring can only decode hex digits, whereas universal charstring can decode any character.

EXAMPLE 2: Union with the "asValue" encoding instruction

Considering the TTCN-3 module:

```
module Mymodule
    // proposed order of fields
    type union U1 {
        integer i,
        float f,
        octetstring os,
        charstring cs
    }
```

```
    // unhealthy order of fields
    type union U2 {
        float f,
        integer i,
        charstring cs,
        octetstring os
    }

    type record of U1 RoU1;
    type record of U2 RoU2;

    const RoU1 c_rou1 := { { i := 10 }, { f := 6.4 }, { os := '1ED5'O }, { cs := "hello" } };
    const RoU2 c_rou2 := { { i := 10 }, { f := 6.4 }, { os := '1ED5'O }, { cs := "hello" } };
} with { encode "JSON"; variant "noType"; variant "asValue" }
```

Both c_rou1 and c_rou2 will be represented in JSON e.g. as:

```
[10,6.4,"1ED5","hello"]
```

The above JSON array will be converted into the same value as c_rou1, when processed as type RoU1, however it will be converted into a value, different from c_rou2, when processed as RoU2: the float field will absorb both numbers and the charstring field will absorb both strings and the resulted TTCN-3 value will be:

```
{ { f := 10.0 }, { f := 6.4 }, { cs := "1ED5" },  { cs := "hello" } };
```

7.2.11 Object Identifiers

This clause shall be supported only if using ASN.1 with TTCN-3, i.e. ETSI ES 201 873-7 [5] is supported.

TTCN-3 **objid** values shall be encoded in JSON as **`JSON.String`**-s (see clause 6.4.2) containing the number forms of the object identifier components, separated by dot (char(U2E)) characters without whitespace character between the digits and the dots.

EXAMPLE:

The TTCN-3 values:

```
const objid c_objid := objid{ joint_iso_itu_t remote_operations(4) informationObjects(5)
                                          version1(0) };
```

Will be transformed to the JSON value:

```
{ "objid" : "2.4.5.0" }
```

8 JSON representations of TTCN-3 values based on ASN.1 types

8.1 General rules

Clause 8 shall only be supported by implementations supporting both ETSI ES 201 873-7 [5] and the present document.

Types and values imported from ASN.1 modules automatically shall have JSON encoding allowed and cannot have JSON encoding instructions (variant attributes) attached.

TTCN-3 values based on ASN.1 types can have encoding instructions attached, therefore the effect of the "noType" instruction shall be the same as in case of TTCN-3 values based on TTCN-3 types. In the type identification, the name of the equivalent TTCN-3 type shall be used (i.e. dashes in ASN.1 names are replaced by underscores). See clauses 7.1 and B.3.11 for further details. During the conversion of TTCN-3 values to and from JSON, always the TTCN-3 equivalent type of the values' (ASN.1) type shall be considered, as specified in clause 8 "ASN.1 and TTCN-3 type equivalents" and clause 9 "ASN.1 data types and values" of ETSI ES 201 873-7 [5].

8.2 Character strings

Values based on ASN.1 character string types shall be encoded to and from JSON the same way as values based on the TTCN-3 universal charstring type. See details in clause 7.2.1 of the present document.

Character string values defined in ASN.1 modules shall be converted by using the long (USI-like) escaping. When converting from JSON, both the long (USI-like) and the short escaping shall be accepted.

8.3 Binary strings

Values based on ASN.1 binary string types shall be encoded to and from JSON the same way as values based on the TTCN-3 bitstring, hexstring or octetstring types, respectively. See details in clause 7.2.2 of the present document.

NOTE: Please note TTCN-3 constants generated for ASN.1 named bits (see clause 9.1 of ETSI ES 201 873-9 [i.5]), which are processed the same way as TTCN-3 constants declared explicitly.

8.4 BOOLEAN

Values based on ASN.1 BOOLEAN types shall be encoded to and from JSON the same way as values based on the TTCN-3 boolean type. See details in clause 7.2.5 of the present document.

8.5 ENUMERATED

Values based on ASN.1 ENUMERATED types shall be encoded to and from JSON the same way as values based on the TTCN-3 enumerated type. See details in clause 7.2.6 of the present document.

NOTE: ASN.1 does not allow assigning a list or range of associated integers to ENUMERATED type members, therefore always the case with a single associated integer value will apply.

8.6 REAL

Values based on ASN.1 REAL types shall be encoded to and from JSON the same way as values based on the TTCN-3 float type. See details in clause 7.2.4 of the present document.

8.7 INTEGER

Values based on ASN.1 INTEGER types shall be encoded to and from JSON the same way as values based on the TTCN-3 integer type. See details in clause 7.2.3 of the present document.

NOTE: TTCN-3 constants generated for ASN.1 named numbers (see clause 9.1 of ETSI ES 201 873-9 [i.5]), which are processed the same way as TTCN-3 constants declared explicitly.

8.8 OBJID

Object identifier values based on ASN.1 types shall be encoded to and from JSON the same way as values based on the TTCN-3 objid type. See details in clause 7.2.11 of the present document.

8.9 NULL

Values based on the ASN.1 NULL type shall be encoded by the JSON literal *null*. The "omit as null" encoding instruction (see clause B.3.8) shall not be used for TTCN-3 structured value fields of ASN.1 NULL type.

NOTE: The value space of the ASN.1 NULL type consists of the single value NULL.

8.10 SEQUENCE and SET

Values based on ASN.1 SEQUENCE and SET types shall be encoded to and from JSON the same way as values based on the TTCN-3 record and set types respectively. See details in clause 7.2.8 of the present document.

8.11 SEQUENCE OF and SET OF

Values based on ASN.1 SEQUENCE OF and SET OF types shall be encoded to and from JSON the same way as values based on the TTCN-3 record of and set of types respectively. See details in clause 7.2.9 of the present document.

8.12 CHOICE and Open Types

Values based on ASN.1 CHOICE types shall be encoded to and from JSON the same way as values based on the TTCN-3 union type. See details in clause 7.2.10 of the present document.

In ASN.1 open types can be created by information object class definitions, which are not directly imported to TTCN-3. However, these open type fields of IO classes can be referenced by importable ASN.1 definitions (e.g. be the field type of ASN.1 SEQUENCE types). In which case values based on ASN.1 open types shall be converted to and from JSON the same way as values based on the TTCN-3 anytype type. See details in clause 7.2.10 of the present document.

Annex A (normative): TTCN-3 module JSON

This annex defines a TTCN-3 module containing type definitions equivalent to JSON built-in types and literal values.

```
module JSON {

//====================Types to define JSON Schemas ================================

    // JSON Number type (generic)
    type float Number (!-infinity .. !infinity) with {
        variant "JSON:number"
    }

    // Integer type
    type integer Integer (-infinity .. infinity) with {
        variant "JSON:integer"
    }

    // String type
    type universal charstring String with {
        variant "JSON:string"
    }


    // Array type
    type record of JSON.Values Array with {
        variant "JSON:array"
    }

    // Subsidiary array types
    type record of JSON.String StrArray with {
        variant "JSON:array"
    }

    type record of JSON.Number NumArray with {
        variant "JSON:array"
    }

    type record of JSON.Integer IntArray with {
        variant "JSON:array"
        }

    type record of JSON.Bool BoolArray with {
        variant "JSON:array"
    }

    type record of JSON.Object ObjArray with {
        variant "JSON:array"
    }

    // Object member
    type record ObjectMember {
        JSON.String name,
        JSON.Values value_
    } with {
        variant "JSON:objectMember"
    }

    // Generic JSON object type
    type record Object {
        record length (1..infinity) of JSON.ObjectMember memberList optional
    } with {
        variant "JSON:object"
    }

    type union Values {
        JSON.String str,
        JSON.Integer int,
        JSON.Number num,
        JSON.Object obj,
        JSON.StrArray strArray,
        JSON.IntArray intArray,
        JSON.NumArray numArray,
        JSON.BoolArray boolArray,
```

```
        JSON.ObjArray objArray,
        JSON.Array array,
        JSON.Bool bool,
        JSON.Null null_
    } with {
        variant "asValue"
    }

    //synonym with singular naming convention
    type Values Value with { variant "asValue" }

    //JSON literals
    //When only the true and false literals are allowed
    type boolean Bool with { variant "JSON:literal" }

    //When only the null literal is allowed
    type enumerated Null { null_ } with { variant "JSON:literal" }


//================================ Useful values ================================

    type JSON.String String_short with {variant "escape as short" };

    type JSON.String String_usi with {variant "escape as usi" };

    type JSON.String String_tr with {variant "escape as transparent" };

    const JSON.String_short cs_bs := char(U8); // encoded as "\b" (Backspace)
    const JSON.String_short cs_ht := char(U9); // encoded as "\t" (Horizontal tab)
    const JSON.String_short cs_lf := char(UA); // encoded as "\n" (Line feed)
    const JSON.String_short cs_ff := char(UC); // encoded as "\f" (Form feed)
    const JSON.String_short cs_cr := char(UD); // encoded as "\r" (Carriage return)
    const JSON.String_short cs_quot := """"; // encoded as "\"" (Quotation mark)
    const JSON.String_short cs_sol := "/"; // encoded as "\/" (Solidus or Slash)
    const JSON.String_short cs_rs := "\"; // encoded as "\\" (Reverse solidus or Backslash)

    const JSON.String_usi cu_nul := char(U0); // encoded as "\u0000" (Null character)
    const JSON.String_usi cu_soh := char(U1); // encoded as "\u0001", (Start of Heading
    const JSON.String_usi cu_stx := char(U2); // encoded as "\u0002" (Start of Text)
    const JSON.String_usi cu_etx := char(U3); // encoded as "\u0003" (End-of-text character)
    const JSON.String_usi cu_eot := char(U4); // encoded as "\u0004" (End-of-transmission character)
    const JSON.String_usi cu_enq := char(U5); // encoded as "\u0005" (Enquiry character)
    const JSON.String_usi cu_ack := char(U6); // encoded as "\u0006" (Acknowledge character)
    const JSON.String_usi cu_bel := char(U7); // encoded as "\u0007" (Bell character)
    const JSON.String_usi cu_bs := char(U8); // encoded as "\u0008" (Backspace)
    const JSON.String_usi cu_ht := char(U9); // encoded as "\u0009" (Horizontal tab)
    const JSON.String_usi cu_lf := char(UA); // encoded as "\u000A" (Line feed)
    const JSON.String_usi cu_vt := char(UB); // encoded as "\u000B" (Vertical tab)
    const JSON.String_usi cu_ff := char(UC); // encoded as "\u000C" (Form feed)
    const JSON.String_usi cu_cr := char(UD); // encoded as "\u000D" (Carriage return)
    const JSON.String_usi cu_so := char(UE); // encoded as "\u000E" (Shift Out)
    const JSON.String_usi cu_si := char(UF); // encoded as "\u000F" (Shift In)
    const JSON.String_usi cu_dle := char(U10); // encoded as "\u0010" (Data Link Escape)
    const JSON.String_usi cu_dc1 := char(U11); // encoded as "\u0011" (Device Control 1)
    const JSON.String_usi cu_dc2 := char(U12); // encoded as "\u0012" (Device Control 2)
    const JSON.String_usi cu_dc3 := char(U13); // encoded as "\u0013" (Device Control 3)
    const JSON.String_usi cu_dc4 := char(U14); // encoded as "\u0014" (Device Control 4)
    const JSON.String_usi cu_nak := char(U15); // encoded as "\u0015" (Negative-acknowledge charac.)
    const JSON.String_usi cu_syn := char(U16); // encoded as "\u0016" (Synchronous Idle)
    const JSON.String_usi cu_etb:= char(U17); // encoded as "\u0017" (End of Transmission Block)
    const JSON.String_usi cu_can := char(U18); // encoded as "\u0018" (Cancel character)
    const JSON.String_usi cu_em := char(U19); // encoded as "\u0019" (End of Medium)
    const JSON.String_usi cu_sub := char(U1A); // encoded as "\u001A" (Substitute character)
    const JSON.String_usi cu_esc := char(U1B); // encoded as "\u001B" (Escape character)
    const JSON.String_usi cu_fs := char(U1C); // encoded as "\u001C" (File Separator)
    const JSON.String_usi cu_gs := char(U1D); // encoded as "\u001D" (Group Separator)
    const JSON.String_usi cu_rs := char(U1E); // encoded as "\u001E" (Record Separator)
    const JSON.String_usi cu_us := char(U1F); // encoded as "\u001F" (Unit Separator)
    const JSON.String_usi cu_sp := " "; // encoded as "\u0020" (Space)
    const JSON.String_usi cu_quot := """"; // encoded as "\u0022" (Quotation mark)
    const JSON.String_usi cu_sol := "/"; // encoded as "\u002F" (Solidus or Slash)
    const JSON.String_usi cu_revs := "\"; // encoded as "\u005C" (Reverse solidus or Backslash)
    const JSON.String_usi cu_del := char(U7F); // encoded as "\u007F" (Delete)

    //NOTE: see ISO/IEC 10646 [3] and https://en.wikipedia.org/wiki/List_of_Unicode_characters

} with { encode "JSON" } //end module
```

Annex B (normative): Encoding instructions

B.1 General

This annex defines the encoding instructions for the JSON to TTCN-3 mapping. Encoding instructions are contained in TTCN-3 **`encode`** and **`variant`** attributes associated with the TTCN-3 definition, field or value of a definition.

A single attribute shall contain one encoding instruction only. Therefore, if several encoding instructions shall be attached to a TTCN-3 language element, several TTCN-3 attributes shall be used.

The "syntactical structure" paragraphs of each clause below identify the syntactical elements of the attribute (i.e. inside the **`with { }`** statement). The syntactical elements shall be separated by whitespaces, which shall contain one or more spaces (**`char`**`(U20))` and horizontal tab (**`char`**`(U9))` characters. No whitespace character is required between the opening and closing quotation marks (**`char`**`(U22)`) and the first and last syntactical elements of the instruction, respectively, though whitespace characters are allowed at those places as well. All characters (including whitespaces) between a pair of apostrophe (or single quote characters, **`char`**`(U27))` shall be part of the encoding instruction.

Typographical conventions: **bold** font identify TTCN-3 keywords. The syntactical elements *freetext* and *name* are identified by *italic* font; they shall contain one or more characters and their contents are specified by the textual description of the encoding instruction. Normal font identifies syntactical elements that shall occur within the TTCN-3 attribute as appear in the syntactical structure. The following character sequences identify syntactical rules and shall not appear in the encoding instruction itself:

- (|) - identify alternatives.
- [ ] - identify that the part of the encoding instruction within the square brackets is optional.
- { } - identify zero or more occurrences of the part between the curly brackets.
- """ - identify the opening or the enclosing quotation marks (**`char`**`(U22)`) of the encoding instruction.

B.2 The JSON encode attribute

The TTCN-3 encode attribute shall be used to identify that the definitions in the scope unit to which this attribute is attached shall be encoded in the following formats:

- "JSON" or "JSON RFC7159"

Syntactical structure

```
encode """ ( JSON | JSON RFC7159) """
```

Applicable to (TTCN-3)

Module, group, definition.

B.3 Encoding instructions

B.3.1 General rules

Faults in the JSON encoding/decoding process, by default shall cause errors. This can be modified with the error behaviour encoding instruction (see clause B.3.13).

Any number of white spaces (spaces and tabs only) can be added between each word or identifier in the attribute syntax, but at least one is necessary if the syntax does not specify a separator (a comma or a colon). The attribute can also start and end with white spaces.

EXAMPLE:

```
variant(field1) "omit as null";         // ok
variant(field2) " omit as null ";           // ok (extra spaces)
variant(field3) "   omit    as  null";  // ok (with tabs)
variant(field4) "omitasnull";           // not ok
```

B.3.2 JSON type identification

Syntactical structure(s)

variant """ (JSON:array | JSON:integer | JSON:literal | JSON:number | JSON:string | JSON:object | JSON: objectMember """

Applicable to (TTCN-3)

TTCN-3 type definitions.

Description

These encoding instructions are typically should not appear in TTCN-3 module describing a JSON Schema. They are attached to the TTCN-3 type definitions of the module named JSON in Annex A of the present document, corresponding to JSON "types" and literal values, and normally should be imported from this module.

The encoder and decoder shall handle instances of a type according to the corresponding JSON definition in IETF RFC 7159 [2].

B.3.3 Normalizing JSON Values

Syntactical structure(s)

variant """ normalize """

Applicable to (TTCN-3)

All TTCN-3 types or values

Description

JSON allows arbitrary number of whitespaces, composed of space(**char**(U20)), horizontal tabulator(**char**(U9)), line feed (**char**(UA)) or carriage return (**char**(UD)) characters between JSON syntactical elements.

In the encoding process this instruction shall result that in the encoded JSON value only a single space (**char**(U32)) character is used between any two JSON syntactical elements.

B.3.4 Name as

Syntactical structure(s)

variant """ name (as ('*freetext*' | changeCase) | all as <changeCase>) """,

where <changeCase> := (capitalized | uncapitalized | lowercased | uppercased)

Applicable to (TTCN-3)

Fields of records, sets and unions.

Description

Gives the specified field a different name in the JSON representation.

When the "name as '*freetext*'" form is used, *freetext* shall be used as the name of the JSON object member, instead of the name of the related TTCN-3 field or definition.

The "name as capitalized" and "name as uncapitalized" forms identify that only the first character of the related TTCN-3 name shall be changed to lower case or upper case respectively.

The "name as lowercased" and "name as uppercased" forms identify that each character of the related TTCN-3 name shall be changed to lower case or upper case respectively.

EXAMPLE:

```
type union PersionID {
  integer numericID,
  charstring email,
  charstring name
} with {
  variant(numericID) "name as 'ID'";
  variant(email) "name as 'Email'";
  variant(name) "name as 'Name'";
}
type record of PersionID PersionIDs;
var PersionIDs pids := { { numericID := 189249214 }, { email := "jdoe@mail.com" }, { name := "John
Doe" } };
// JSON code:
// [{"ID":189249214},{"Email":"jdoe@mail.com"},{"Name":"John Doe"}]
```

B.3.5 Number of fraction digits

Syntactical structure(s)

variant """ fractionDigits <an integer value>"""

Applicable to (TTCN-3)

Types and fields of *JSON.Number* type.

Description

By default, the number of fraction digits, and/or the use of the exponent part is a tool implementation option. The "fractionDigits" encoding instruction, at encoding constraints the maximum number of fractional digits following the decimal point in the encoded JSON value. TTCN-3 allows using either the decimal point notation or the E-notation for float values (see clause 6.1.0 of ETSI ES 201 873-1 [1]). In the encoding process, at most the number of fraction digits specified in the instruction shall be used. If representing the actual value does not require the number of fraction digits specified by this instruction, the encoder shall use the needed number of digits, if representing the actual (numeric) value would require more fraction digits than specified by this instruction, the encoder shall use a mixed fraction part + exponent part representation.

NOTE: The "fractionDigits 0" instruction will enforce the encoder to use the exponent part exclusively.

EXAMPLE: Number of fraction digits.

If:

```
const JSON.Number c_number1 := <actual value> with {variant "fractionDigits 3"};
```

then

<actual value>	encoded JSON value
0.0	0.0
3.14	3.14
3.142	3.142
3.1415	31.415E-1or 31.415e-1

If:

```
const JSON.Number c_number2 := <actual value> with {variant "fractionDigits 0"};
```

then

<actual value>	encoded JSON value
0.0	0E1 or 0e1
3.14	314E-2 or 314e-2
3.142	3142E-3 or 3142e-3
3.1415	31415E-4 or 31415e-4

This encoding instruction has no effect at decoding of JSON values.

B.3.6 Use the Minus sign

Syntactical structure(s)

variant """ useMinus """

Applicable to (TTCN-3)

Types and fields of *JSON.Number* and *JSON.Integer* types.

Description

By default, without the "useMinus" instruction, JSON numbers are decoded to TTCN-3 *JSON.Number* and IEEE 754 float useful types by their values; i.e. all the -0.0, 0.0, -0e<number>, 0e<number>, -0E<number>, 0E<number>, -0 and 0 JSON values are decoded in TTCN-3 as 0.0 for *JSON.Number* types and as 0 for *JSON.Integer* types (i.e. without the minus sign), where <number> is any positive or negative integer number.

The "useMinus" encoding instruction, when applied to a *JSON.Number* type, instructs the decoder to decode the JSON values -0.0, -0e<number>, -0E<number> and -0 in TTCN-3 as negative float numbers, i.e. together with their minus sign. The instruction has no effect on *JSON.Integer* types.

This encoding instruction has no effect at encoding and at decoding of any other JSON values than specified above.

B.3.7 Escape as

Syntactical structure(s)

variant """ escape as (short | usi | transparent) """,

Applicable to (TTCN-3)

Types and fields of *JSON.String* types.

Description

The "escape as short" encoding instruction tells the encoder that all characters in the TTCN-3 value, which has short escape sequences defined (see IETF RFC 7159 [2] and clause 6.4.2), shall be encoded using the short escape sequence.

The "escape as usi" encoding instruction tells the encoder that quotation mark ("""", **char**(U22)), reverse solidus ("\", **char**(U5C)), and the control characters (**char**(U0) through **char**(U1F)) in the TTCN-3 value shall be encoded using the USI-like escape sequence "\u<HHHH>" (see IETF RFC 7159 [2] and clause 6.4.2).

The "escape as transparent" encoding instruction tells the encoder that characters in the TTCN-3 value shall not be escaped in their JSON representation, except the C0 control characters (present in the TTCN-3 value in the **char**(…) representation).

NOTE: This instruction is useful, when a character string is copied from a JSON string, where the needed characters are already have been replaced by their escape sequences, into a TTCN-3 code.

This instruction has no effect at decoding, i.e. all escaped characters, using either the short or the USI-like escaping shall be decoded to and evaluated in its (abstract) character representation in TTCN-3 (e.g. at matching or in any other operations).

B.3.8 Omit as null

Syntactical structure(s)

variant """ omit as null """,

Applicable to (TTCN-3)

Optional fields of records and sets

Description

If set, the value of the specified optional field will be encoded with the JSON literal '`null`' if the value is omitted. By default omitted fields (both their name and value) are skipped entirely. The decoder ignores this attribute and accepts both versions.

This encoding instruction shall not be used for fields of ASN.1 NULL type.

B.3.9 Default

Syntactical structure(s)

variant """ default (<value>)"""

where <value> is a value of the type the instruction is applied to.

Applicable to (TTCN-3)

Fields of records and sets

Description

This encoding attribute has no effect at encoding.

The decoder shall set the given value to the field if it does not appear in the encoded JSON value.

The <value> shall contain a valid completely initialized TTCN-3 value of the field's type. The <value> can be a reference to a constant of a compatible type, the referenced value shall be known at compile time, i.e. it shall be statically bound. In case of structured values, sub-expressions are not allowed to be references.

Optional fields with a default value will be set to **`omit`** if the field is set to *`null`* in the JSON value being decoded, and shall use the default value if the field does not appear in the encoded JSON value.

EXAMPLE 1:

```
type record Product {
    charstring name,
    float price,
    octetstring id optional,
    charstring origin,
    universal charstring text
} with {
    variant(id) "default (FFFF)"
    variant(origin) "default(Hungary)"
    variant(text) "default (char(1,2,3,4) & char(5,6,7,8) & ""?"")"
}

// { "name" : "Shoe", "price" : 29.50, "text" : "available"  } will be decoded into:
// { name := "Shoe", price := 29.5, id := 'FFFF'O, origin := "Hungary",
//      text := char(1,2,3,4) & char(5,6,7,8) & "?" }

// { "name" : "Shirt", "price" : 12.99, "id" : null } will be decoded into:
// { name := "Shirt", price := 12.99, id := omit, from := "Hungary" }
```

```
type record Shopping_cart {
    charstring name,
    Product product
} with {
    variant(product) "default ({""Shirt"", 12.99, omit, ""Hungary"", ""available"" })"
}

// { "name" : "test shopper" } will be decoded into:
// { name := "test shopper",
//   product := { name := "Shirt", price := 12.99, id := omit, origin := "Hungary",
//                text := "available" }}
```

EXAMPLE 2:

```
type record Shopping_cart_erroneous {
    charstring name,
    Product product
} with {
    // error: syntactically invalid default value
    variant(product) "default ({""Shirt"", 12..99, omit, ""Hungary"", ""available"" })"
}
```

EXAMPLE 3: Demonstrating top level reference

```
const Product c_defaultProduct:=
{
    name := "Size ""M"" Shirt",
    price := 12.99,
    id := omit,
    origin := "Hungary",
    text := "available"
}

type record Shopping_cart_2 {
    charstring name,
    Product product
} with {
    variant(product) "default (c_defaultProduct)"
}

// { "name" : "test shopper" } will be decoded into:
// { name := "test shopper",
//   product := { name := "Size ""M"" Shirt", price := 12.99, id := omit,
//      origin := "Hungary", text := "available" }}
```

B.3.10 As value

Syntactical structure(s)

variant """ asValue """,

Applicable to (TTCN-3)

Unions types and fields.

Description

The TTCN-3 union value shall be encoded as a JSON value instead of as a JSON object with one object member (see clause 7.2.10). This allows the creation of heterogenous arrays in the JSON document (e.g. ["text",10,true,null]).

Since the field name no longer appears in the encoded JSON value, the decoder shall determine the selected field based on the syntax of the JSON value. The first union field (in the textual order of declaration) that can successfully decode the JSON value will be the selected one.

B.3.11 No Type

Syntactical structure(s)

variant """ noType """

Applicable to (TTCN-3)

TTCN-3 types.

Description

The "noType" encoding instruction shall cause that the "type wrapper" of the top-level TTCN-3 type of the instance being encoded, is omitted in the JSON representation, and at decoding a JSON value, it is not expected to be present. However, the presence of the type wrapper shall not cause a decoding error. See also clause 7.1 General rules.

B.3.12 Use order

Syntactical structure(s)

variant """ useOrder """

Applicable to (TTCN-3)

Record type definition, generated for JSON objects (see clause 6.4.4).

Description

The encoding instruction designates that the encoder shall encode the JSON object members according to the strings' order in the `order` field. The object member names are equally identified by the field names of the record type generated for the object (except the `order` and `memberList` fields) and the values of the `name` fields of the `memberList` field's record of elements. If the `order` field contains more or less elements than object members identified by the TTCN-3 value, or a member name non-existing in the TTCN-3 value, the encoder shall cause a test case error. If the `order` field is missing, the textual order of their fields in the corresponding type definition - and the elements' order in the `memberList` field of the value being encoded - shall be followed.

At decoding, if the `order` field exists in the type definition of the value being decoded, the received values of the JSON object members shall be placed in their corresponding record fields or into a subsequent element of the `memberList` field if no dedicated field exists for the member, and a new record of element containing the member's name shall be inserted into the `order` field for each JSON object member processed (i.e. the final order of the record of elements shall reflect the order of the members in the JSON object being decoded).

B.3.13 Error behaviour

Syntactical structure(s)

variant """ `errorbehavior` "(" <error_type> ":" <error_handling>
{ "," <error_type> ":" <error_handling> } ")""",

where the error types (<error_type>) and possible related error behaviours (<error_handling>) are defined in tables B.1 and B.2 correspondingly.

Table B.1: Conversion failure types

ET_UNDEF	Undefined/unknown error.
ET_DEC_ENUM	Decoding of an unknown enumerated value.
ET_INCOMPL_MSG	Decode error: incomplete message.
ET_INVAL MSG	Decode error: invalid message.
ET_CONSTRAINT	The value breaks some constraint.
ET_ALL	All error types.

Table B.2: Values of error behaviours

EB_ERROR	Causes a test case error if the selected error type occurs.
EB_WARNING	Logs a warning but tries to continue the operation. If the decoding fails, it returns the undecoded JSON value to TTCN-3 as a universal charstring value.
EB_IGNORE	Like EB_WARNING but without the logging a warning message. If the decoding fails, it returns the undecoded JSON value to TTCN-3 as a universal charstring value.

Applicable to (TTCN-3)

All types.

Description

TTCN-3 predefined decoding functions ensure that the decoding trial of the inout parameter encoded_value will not cause a test case error, when the decoding fails or is incomplete (see clause C.5 of ETSI ES 201 873-1 [1]). This encoding instruction provides a similar feature, i.e. avoid test case error at JSON to TTCN-3 conversion, when the TTCN-3 test system is communicating with other systems via an interface that allows alternatively receiving abstract TTCN-3 values or TTCN-3 **`universal charstring`** values, like TTCN-3 communication ports.

EB_ERROR is the default error behaviour for all conversion failure types (see tables B.1 and B.2), in which case a JSON to TTCN-3 conversion failure shall cause a test case error. When a JSON to TTCN-3 conversion failure occurs, and for the type of the failure AND for the target TTCN-3 language element a different error behaviour than EB_ERROR is specified, no test case error shall occur, but the relevant behaviour in table B.2 shall be followed.

Annex C (informative): Bibliography

ECMA International® Standard ECMA-404: "The JSON Data Interchange Format".

ISO/IEC 646: "Information technology -- ISO 7-bit coded character set for information interchange".

History

Document history		
V4.7.1	June 2017	Publication
V4.8.1	May 2018	Publication
V4.9.1	June 2021	Publication
V4.10.1	March 2023	Membership Approval Procedure MV 20230502: 2023-03-03 to 2023-05-02
V4.10.1	May 2023	Publication