Generic Execution Traces Specification

MODMED

This document specifies a model of generic execution traces data allowing trace providers to further define their own event data while ensuring interoperability with a variety of generic analysis tools such as those developed by the MODMED project.

Contributors

	Function	Name	Partner
Written By	Coordinator	Arnaud Clère	MinMaxMedical
Reviewed By	Researcher	Yoann Blein	LIG
Reviewed By	Researcher	Yves Ledru	LIG

Revisions

What	Who	When
DRAFT	Arnaud Clère	21/09/2017
V1 Many clarifications and references, plus: - Use EBNF, ERE formalisms - Added _Integer/_Decimal/_Timestamp/_Bytes/_Types - Use XSD to reflect primitive types in XML Physical Model - CBOR Physical Model	Arnaud Clère	13/11/2017
V1.1 Many minor edits after 1st review, plus: - Separated _Null from _Text to eliminate ambiguities - Renamed _source_path, _source_line to just _path, _line - Defined _message (_format + _args), _severity_id - Simplified TSV+JSON and fixed a problem with JSON string "null" - Changed some physical models requirements to recommendations - Recommended ways to convey metadata about _Traces - Reworked redundancy elimination rules and encodings - Added examples	Arnaud Clère	28/11/2017
V1.2 Minor edits and corrections, plus: - Renamed _Identifier _Name to make clear unicity is not required - Recommended way to handle duplicate _Names - Added <n/> for XML _Null	Arnaud Clère	25/01/2018

Consortium

MinMaxMedical
Minimally Invasive Interventions for Maximum Efficiency

L I G

BlueOrtho®
AUGMENTED ORTHOPAEDICS

Partly funded by:

ANR
PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Contents

Examples

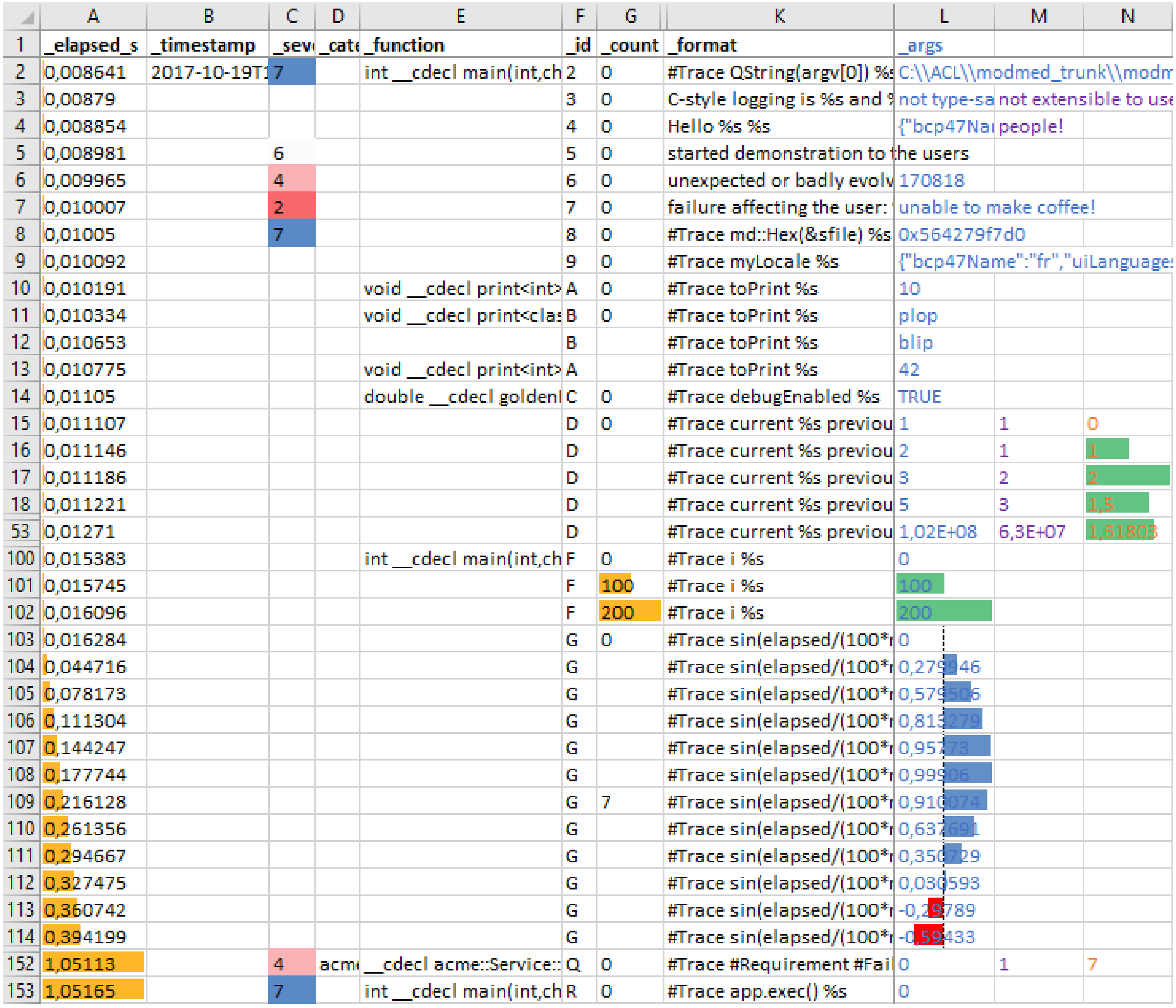
1. Goals

Our goal is to define a generic data model for execution traces that can:

1. exploit existing tracepoints to the best;
2. facilitate human exploration;
3. allow automatic analysis by various tools; and
4. allow simple and efficient trace provider implementations.

To illustrate these goals, let us look at an example of trace that satisfies them using the TSV+JSON physical data model and the modmedLog C++ trace library:

- All the structure of this trace comes from usual printf-like and stream-like C++ tracepoints
- Human exploration is facilitated by emphasizing changes in metadata, providing a constant _format for all events issued by the same tracepoint
- A lot of trace analysis can be done without parsing using common worksheet data processing



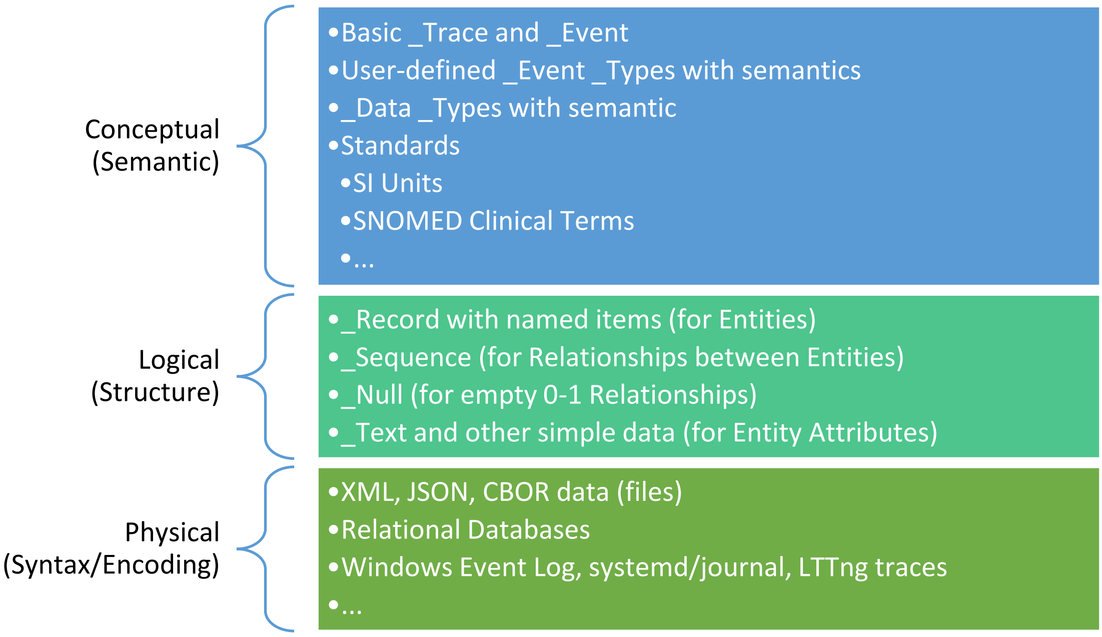
	A	B	C	D	E	F	G	K	L	M	N
1	_elapsed_s	_timestamp	_sev	_cate	_function	_id	_count	_format	_args		
2	0,008641	2017-10-19T1	7		int __cdecl main(int,ch	2	0	#Trace QString(argv[0]) %s	C:\\ACL\\modmed_trunk\\modm		
3	0,00879					3	0	C-style logging is %s and %	not type-sa	not extensible to use	
4	0,008854					4	0	Hello %s %s	{"bcp47Na	people!	
5	0,008981		6			5	0	started demonstration to the users			
6	0,009965		4			6	0	unexpected or badly evolv	170818		
7	0,010007		2			7	0	failure affecting the user: '	unable to make coffee!		
8	0,01005		7			8	0	#Trace md::Hex(&sfile) %s	0x564279f7d0		
9	0,010092					9	0	#Trace myLocale %s	{"bcp47Name":"fr","uiLanguage:		
10	0,010191				void __cdecl print<int>	A	0	#Trace toPrint %s	10		
11	0,010334				void __cdecl print<clas	B	0	#Trace toPrint %s	plop		
12	0,010653					B		#Trace toPrint %s	blip		
13	0,010775				void __cdecl print<int>	A		#Trace toPrint %s	42		
14	0,01105				double __cdecl goldenl	C	0	#Trace debugEnabled %s	TRUE		
15	0,011107					D	0	#Trace current %s previou	1	1	0
16	0,011146					D		#Trace current %s previou	2	1	1
17	0,011186					D		#Trace current %s previou	3	2	2
18	0,011221					D		#Trace current %s previou	5	3	1,5
53	0,01271					D		#Trace current %s previou	1,02E+08	6,3E+07	1,61803
100	0,015383				int __cdecl main(int,ch	F	0	#Trace i %s	0		
101	0,015745					F	100	#Trace i %s	100		
102	0,016096					F	200	#Trace i %s	200		
103	0,016284					G	0	#Trace sin(elapsed/(100*r	0		
104	0,044716					G		#Trace sin(elapsed/(100*r	0,279946		
105	0,078173					G		#Trace sin(elapsed/(100*r	0,579506		
106	0,111304					G		#Trace sin(elapsed/(100*r	0,813279		
107	0,144247					G		#Trace sin(elapsed/(100*r	0,95773		
108	0,177744					G		#Trace sin(elapsed/(100*r	0,99906		
109	0,216128					G	7	#Trace sin(elapsed/(100*r	0,910074		
110	0,261356					G		#Trace sin(elapsed/(100*r	0,637691		
111	0,294667					G		#Trace sin(elapsed/(100*r	0,350729		
112	0,327475					G		#Trace sin(elapsed/(100*r	0,030593		
113	0,360742					G		#Trace sin(elapsed/(100*r	-0,29789		
114	0,394199					G		#Trace sin(elapsed/(100*r	-0,59433		
152	1,05113		4	acme	__cdecl acme::Service::	Q	0	#Trace #Requirement #Fail	0	1	7
153	1,05165		7		int __cdecl main(int,ch	R	0	#Trace app.exec() %s	0		

Example 1: TSV+JSON _Trace with data highlighted in a worksheet processing software

The specification section (page 6) contains the minimum requirements deemed necessary to allow powerful automatic analysis (goal #3) while keeping simple implementations possible (goal #4). In particular, it leaves a lot of freedom to trace providers (goal #1). However, contrary to simpler specifications like JSON, it also contains many recommendations to facilitate human exploration (goal #2) or guide implementers.

Tracing libraries that put restrictions on tracepoints and the types of associated arguments to reach the maximum level of performance (such as LTTng or WPP) provide traces that can usually be translated to one of the defined physical formats to pursue goals #2 and #3.

In order to allow many different implementations (such as XML, or Concise Binary Object Representation) targeting various needs and tradeoffs between trace performance and completeness, this specification uses the classical Conceptual, Logical, and Physical layers of data to separately model various aspects of traces as depicted below:



Conceptual (Semantic)

- Basic _Trace and _Event
- User-defined _Event _Types with semantics
- _Data _Types with semantic
- Standards
  - SI Units
  - SNOMED Clinical Terms
  - ...

Logical (Structure)

- _Record with named items (for Entities)
- _Sequence (for Relationships between Entities)
- _Null (for empty 0-1 Relationships)
- _Text and other simple data (for Entity Attributes)

Physical (Syntax/Encoding)

- XML, JSON, CBOR data (files)
- Relational Databases
- Windows Event Log, systemd/journal, LTTng traces
- ...

Defining traces data using these 3 layers enables interoperability between trace producers and consumers (monitors, analyzers but also stores, transmission channels, etc.). For instance, transmission channels and stores may only need to know about the Physical model, while filtering tools may ignore the Conceptual model. On the other hand, trace providers and analyzers can use the Logical model to remain independent from the Physical model (such as a wire or file format) and know the minimum about the Conceptual model required for the task at hand.

Separating the Conceptual and Logical models also allows to delay or limit the arduous classification and standardization work to the data one wants to use in a particular application. Indeed, this specification just defines what are a generic _Trace and _Event. Further conceptual definitions such as additional data and _Event _Types are left to trace providers.

2. Specification

This section of the document is normative. The keywords "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this section are to be interpreted as described in RFC 2119.

All `definitions using this typography` use the Extended Backus-Naur Form (EBNF) formalism with the extension that uppercase and lowercase characters in literals are considered equals and will not be explicitly mentioned.

Summary of EBNF notations used:

=	start of definition	"…"	string literal (case insensitive)
*	repetition (zero or more)	(…)	group
,	concatenation	(*…*)	comment
|	alternative	;	end of definition

Where convenient, definitions may restrict the EBNF with case-insensitive POSIX Extended Regular Expressions (ERE) specified in EBNF comments like: *(* matching … *)*.

As usual in EBNF the order of items in concatenation and repetition is meaningful and must be preserved during transfer and processing. This is obviously the case for the order of _Events in a _Trace. The only exception is the order of a _Record’s items which is NOT meaningful and may be altered during transfer or processing.

NB: UML class diagrams are not normative but facilitate understanding definitions.

a) Preliminary definitions

TRACEPOINT

A location in executable code that is tracing event occurrences by adding _Events to a _Trace. A single location in template source code may result in several TRACEPOINTs in executable code.

EVENTDATA

Any _Data part within an _Event, including its _args.
It should be reachable either by name or position or any sequence thereof. For instance, in JavaScript: ".identifier", "[‘identifier’]" to access _Record items ; "[0]" to access _Sequence items.

b) Conceptual model

Let us start with the root of the Conceptual model of trace data:

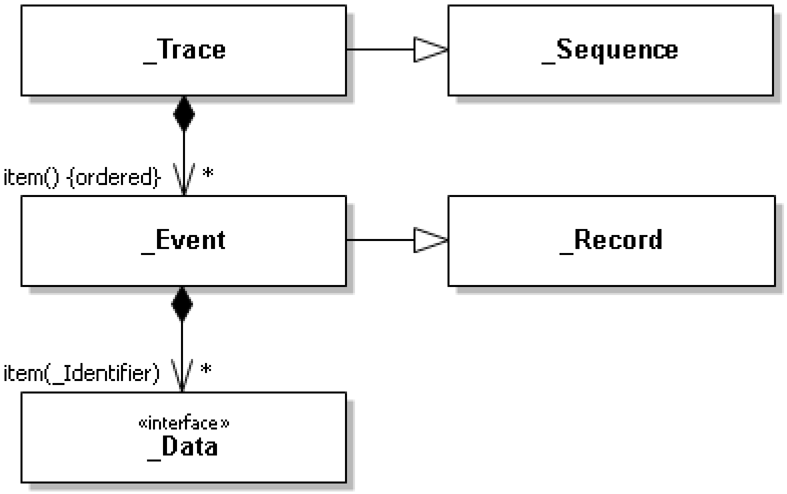
```
_Trace = _Sequence (* of _Event *) ;
```

It must be a flat, ordered sequence of non-overlapping _Events. In particular, groups of related _Events must be flattened using, for instance, dedicated "start" and "stop" _Events.
Particular analyses may have to restore the grouping in nested _Event trees which are outside the scope of this specification.
_Events order in a _Trace may only be partial. For instance:

- _Events issued by different processes may only be ordered up to their timestamp resolution.
- _Events issued by different threads may only be ordered up to thread interleaving after _Event’s occurrence and before actual insertion into the _Trace.

A _Trace may be the union of several _Traces provided a (partial) ordering procedure is given.

NB: In case of a crash, the most recent (and important) _Events may be absent from the _Trace. Requiring that all _Events be flushed immediately would prevent many performance optimizations such as buffering and queuing. As a result, memory dumps are necessary to diagnose those problems. Some implementations may give access to unflushed _Events from memory dumps.



UML class diagram 1: Conceptual _Trace data model

Examples

```
[]
```

Example 2: Empty JSON _Trace

```
[{"_elapsed_s": 0.01458 ,"_timestamp":"2013-11-12T00:12:56+00:00"
 ,"_format"    :"1st empty event"
 ,"_args"      :[]}
,{"_elapsed_s": 0.0152
 ,"_format"    :"2nd empty event"
 ,"_args"      :[]}
]
```

Example 3: Simplistic JSON _Trace

```
_Event = _Record (* satisfying the requirements below *) ;
```

It must represent a single occurrence of a TRACEPOINT.

The 1st _Event in a _Trace must contain the following required (_Name , _Data) item:

```
( "_timestamp" , _Timestamp )
```

Its value must unambiguously represent the point in Coordinated Universal Time (UTC) at which the _Event occurred. For consistency, _timestamp value of the 1st _Event in a _Trace must be measured less than 0,1 second before its _elapsed_s value below.

The 1st _Event may be delayed until both measures can be taken within this range. Subsequent _Events _timestamp may be dismissed.

All _Events must contain at least the 3 following required (_Name , _Data) items:

```
( "_elapsed_s" , _Decimal )
```

Its value must be a monotonically increasing _Decimal representing elapsed seconds between a single point in time and the current _Event. Its precision must be greater than or equal to the precision of _timestamp.

The value of the 1st _Event in a _Trace should be in the [0-1[range to facilitate human exploration. It may not be exactly 0.0 for implementation reasons (see for instance Example 1).

NB: These requirements provide a simple common time scale for all _Events in a _Trace to facilitate human exploration and tools analysis without having to deal with the complexity of _Timestamp values (parsing, UTC offset, etc.).

```
( "_format" , _Text )
```

It must be identical for all occurrences of a particular TRACEPOINT. Different TRACEPOINTs may have identical _format though, in which case it will be necessary to use other EVENTDATA to select _Events issued by the desired TRACEPOINT.
It should informally give meaning to the _Event. Moreover, trace providers should put as much constant information as possible from TRACEPOINTs into _format, to satisfy goal #1 while allowing gradual TRACEPOINT improvements.
For instance, TRACEPOINTs may follow encoding rules for _format value such as the C++ printf function to give formal meaning to _args below. They may additionally use _Tags to give well-defined meaning to _Events.

```
( "_args" , _Sequence )
```

It must contain the values of TRACEPOINT arguments. When _format gives formal meaning to _args, the _Sequence values must appear in the same order as in _format.
Decoders may provide direct access to _args items by position, though they should not count as _Event items, so, one may write code like:

```
var event = {_args:['a',1]};
for (var i=0; i<event._args.length; i++) writeln(event[i]);
```

The following optional _Names have reserved meaning:

```
( "_arg_names" , _Sequence (* of _Name | _Null *) )
```

It must have the same items count and ordering as _args and must only contain _Names of TRACEPOINT arguments, or _Null for _args with no known name.
Decoders may provide direct access to _args items by _arg_names, though they should not count as _Event items, so, one may write code like:

```
var event = {_args:['a',1],_arg_names:['first','last']};
for (var i=0; i<event._arg_names.length; i++) writeln(event[event._arg_names[i]]);
writeln(event.first);
writeln(event['last']);
```

If, and only if, _arg_names contains duplicate _Names, decoders should concatenate values in a _Sequence since _Names denote a relationship between the value and its enclosing _args.

```
( "_arg_types" , _Sequence (* of _Type | _Null *) )
```

It must have the same items count and ordering as _args and must only contain _Types of TRACEPOINT arguments, or _Null for _args with unknown type.

NB: When user-defined _Types in a _Trace are unknown, it is still possible to analyze the _Trace based on its logical structure and _Base_Types.

```
( "_message" , _Text )
```

It must contain _Text formatted according to _format with corresponding _args.

It must not replace _format and _args since it may be impossible for _Trace users to understand its structure.

```
( "_severity" , _Integer )
```

The meaning of values must correspond to RFC5424 (Syslog) "PRI" severities where 0 is the most severe and 7 is the least one.

Values 0-1 should not be used by libraries since these libraries may be used by unimportant applications.

_Trace providers may define other notions of "priority" associated to their own _Name.

```
( "_severity_id" , _Name )
```

It must be the RFC5424 (Syslog) "PRI" name corresponding to _severity value: 0="EMERGENCY" ; 1="ALERT" ; 2="CRITICAL" ; 3="ERROR" ; 4="WARNING" ; 5="NOTICE" ; 6="INFORMATIONAL" ; 7="DEBUG"

```
( "_category" , _Text )
```

It must be identical for related TRACEPOINTs.

All TRACEPOINTs written by a development individual or team or corporation should contain a common part. TRACEPOINTs of library code should set a non-empty value.

Application-level TRACEPOINTs may not set a value.

```
( "_function" , _Text )
```

It must be identical for TRACEPOINTs belonging to the same "function" of the source code language when such notion exists.

The function name may be simplified to remove information redundant with other items such as _category (for instance, if C++ namespace is used as category and duplicated in _function).

```
( "_path" , _Text )
```

It must be a path to the source code file that generated the TRACEPOINT.

```
( "_line" , _Integer )
```

It must be the line in _path that generated the TRACEPOINT.

```
( "_id" , _Text )
```

It must be identical for all _Events issued by the same TRACEPOINT, although _Trace providers must not be obliged to manually assign _ids.

It should be as stable as possible to facilitate analysis of multiple _Traces, though automatic and stable _ids usually do not exist (executable code addresses are relocatable at run-time, static data addresses are relocatable at compile-time, etc.). It should also be different for different kind of _Event.

Beware though that, in practice, different _Events from different sources may use the same _id.

Non-empty values may be used to check the homogeneity of filtered _Event _Sequences.

```
( "_count" , _Integer )
```

It must be the number of times a TRACEPOINT was hit before it issued the current _Event during an execution (this is zero-based as most programming languages indices).

One may use this to detect _Event occurrences missing from a _Trace.

```
( "_computer_id" , _Text )
```

It must be identical for all _Events issued by the same computer.

The representation should be one used by the Operating System.

```
( "_process_id" , _Text )
```

It must be identical for all _Events issued by the same Operating System process.

The representation should be one used by the Operating System.

```
( "_thread_id" , _Text )
```

It must be identical for all _Events issued by the same Operating System thread.

The representation should be one used by the Operating System.

```
( "_user_id" , _Text )
```

It must be identical for all _Events issued by the same Operating System user.

The representation should be one used by the Operating System.

```
( "_group_id" , _Text )
```

It must be identical for all _Events issued by the same Operating System user group.

The representation should be one used by the Operating System.

```
( "_object_id" , _Text )
```

It must be identical for all _Events issued by the same source code language object.

The representation should be one used by the source code language.

Examples

```
{"_elapsed_s": 0.0152
,"_format"   :""
,"_args"     :[]}
```

Example 4: Minimal _Event in JSON

```
{"_elapsed_s": 0.0152
,"_severity" : 7
,"_function" :"int main(int,char*[])"
,"_path"     :"test.c"
,"_line"     : 57
,"_format"   :"C-style logging is %s and %s"
,"_args"     :["not type-safe (may crash!)","not extensible to user types"]
,"_message"    :"C-style logging is not type-safe (may crash!) and not extensible to user
types"}
```

Example 5: Realistic printf-like _Event in JSON

```
{"_elapsed_s": 1.01458
,"_timestamp":"2017-10-19T18:37:26+02:00"
,"_severity" : 4
,"_category" :"acme"
,"_function" :"Service::~Service(void)"
,"_path"     :"test.c"
,"_line"     : 57
,"_thread_id":"1664"
,"_id"       :"Q"
,"_count"    : 0
,"_format"   :"#Trace #Requirement #Failure m_submitted == m_processed + m_rejected"
,"_args"     :[ 0           , 1          , 7            ]
,"_arg_names":["m_processed","m_rejected","m_submitted"]
,"_arg_types":["_Integer"   ,"_Integer"   ,"_Integer"   ]}
```

Example 6: Hypothetic fully structured _Event in JSON

c) Logical model

This model relates the Conceptual and Physical Models using as few as required data structures and base types to support a broad range of Conceptual and Physical Models.

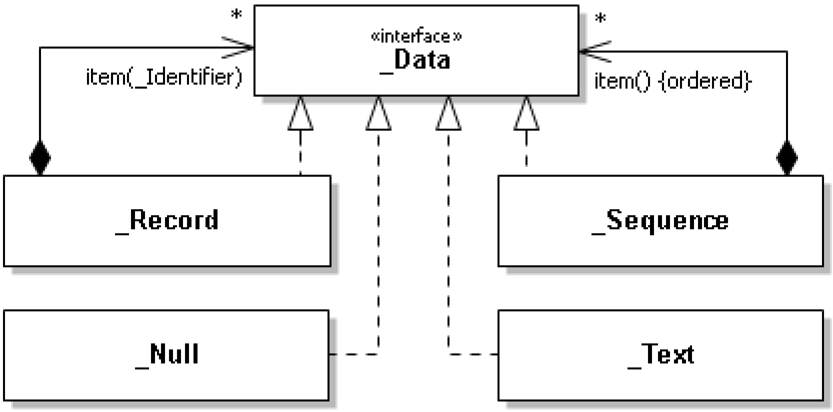
It also aims to be:

1. Open to a wide range of platforms, languages and formats
2. Usable without external data schemas such as event catalogs or hard-coded Run-Time Type Information
3. Translatable to a text format readable by non-programmers
4. Amenable to time- and space- efficient implementations

This model is not expected to be able to encode complex object graphs automatically (it will neither prevent loops in a graph, nor automatically assign references to avoid duplication). We encourage users to use existing file formats like STL, OBJ to encode large or complex data sets like meshes, and standard ways to point to these data from the trace data such as URIs (including relative file paths).

```
_Data = _Record | _Sequence | _Null | _Text ;
```

Physical models must provide some way to distinguish between the 4 alternatives. They can use any internal representation suitable for their purpose such as text, binary, contiguous memory, trees, etc. Moreover, all _Data that is not represented as a _Record or _Sequence or _Null must have a well-defined _Text representation. Physical models can use specialized binary representations of, say, numerical data, provided they also support a translation to a well-defined _Text representation.



UML class diagram 2: Minimal Logical model (required)

User-defined data models should follow the following guidelines to favor interoperability with tools:

- ER Entities should be defined as a _Record except for the simplest ones (see below).
- Simple ER entities like, for instance, a "KeyValue" entity with "key" and "value" items should also be defined as a _Record with explicit _Names rather than as a fixed-size _Sequence or _Text with implicit semantic.
- ER attribute values and entities so simple that decomposition in separately identified items seems useless should be represented as _Text.
- 0-n and 1-n ER relationships should be defined as a _Sequence
- 0-1 ER relationships between entities should directly use the related entity or the special _Text value _Null to denote empty relationships.

- n-n ER relationships may be expressed by _Sequences storing foreign keys of related _Records.

```
_Record = ( _Name , _Data )* (* ordering MAY NOT be preserved *) ;
```

Decoders must provide some way to iterate through _Record items that relates the corresponding _Name and _Data. When iterating items with duplicate _Name, the respective order of these items must be preserved (in case it is meaningful). They can use any internal representation suitable for their purpose such as list of (_Name,_Data) pairs, including non-order-preserving hash maps, etc. An empty _Record which can correspond to an existing, empty entity must not be interpreted as _Null.

Decoders may provide direct access to _Record items by _Name. If, and only if, _Record contains duplicate _Names, decoders should concatenate values in a _Sequence since _Names denote a relationship between the value and its enclosing _Record.

Examples

```
{ "name":"John", "children" : []}
```

Example 7: Simple JSON _Record

```
_Sequence = _Data * ;
```

Physical models must provide some way to access each _Data item in the order it was defined. They can use any internal representation suitable for their purpose such as lists, vectors, etc. An empty _Sequence which can correspond to an existing, empty relationship must not be interpreted as _Null.

Examples

```
[ null,"foo" , {}]
```

Example 8: Simple JSON _Sequence

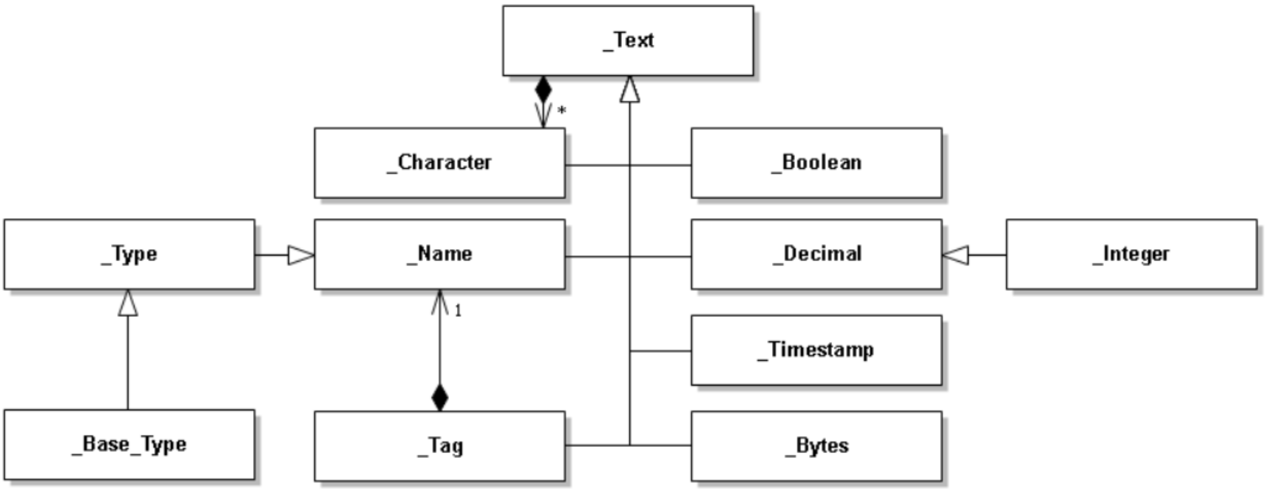
```
_Null = (* absence of information *) ;
```

It must be interpreted as absence of information. In particular, a _Record with a _Name associated with _Null must be considered equal to the same _Record without the _Name. On the contrary, a _Sequence with a _Null item must not be considered equal to the same _Sequence with no item.

NB: An empty _Record or _Sequence or _Text must not be interpreted as _Null.

```
_Text = _Character * ;
```

An empty _Text which can correspond to blanked out information must not be interpreted as _Null. **Unless otherwise specified by the Conceptual or Physical models used**, _Text values matching one of the _Base_Types below should be interpreted as a value of the corresponding _Base_Type. This implies that different _Text representations of the same _Type value (say, `1.2` and `1.20`) should be considered equal. On the contrary, _Text values matching some Physical textual model like `"null"` _Text matching a JSON null value representation should not be automatically interpreted as such. It should always be possible for trace users to reinterpret some _Type value as _Text if necessary to, say, sort values alphabetically.



UML class diagram 3: Specified _Text representations (optional)

Decoders with specialized _Type representations may be able to distinguish between values like the _Boolean value `true` and the _Text value `"true"` based on physical representation or context.

Examples

```
[[{"foo":null},{}]
,[true,"tRue","TRUE"]
,[false,"falSE","FALSE"]
,[123,123.0000,"123."]
,["2013-11-12T03:12:56+00:00","2013-11-11T21:12:56-06:00"]]
```

Example 9: _Sequence of _Sequences of equal _Data encoded in JSON Physical model

User-defined data

_Text representations of user-defined data should follow these guidelines to favor interoperability with tools:

- They should start with a sequence different from representations defined in this specification.
- They should use well-established standards like iso8601 for date and time values, even if it is not explicit in the representation.

With knowledge of such user-defined _Types, _Text values may be further decomposed or interpreted. For instance, given the definition: `Point2D="(",_Decimal,"_",_Decimal,")";` the _Text "(1.2_0.4)" may be interpreted as a point in a 2D coordinate system.

```
_Boolean = "TRUE" | "FALSE" ;
```

It should be interpreted as the corresponding truth value.
Physical models may use canonical representations.

```
_Integer = _Text (* matching [+-]?[0-9]+ *) ;
```

It should be interpreted as the corresponding integer in decimal notation.
Physical models may limit the range of integer numbers.

```
_Decimal = "NaN" | (""|"+"|"-") , "INFINITY"
         | _Text (* matching [+-]?[0-9]*(.[0-9]*)?([eE][+-]?[0-9]+)? *) ;
```

Unless otherwise specified by a Physical or Conceptual model, a _Decimal without a decimal or fractional part should be processed as an _Integer. Otherwise, it should be interpreted as the corresponding number in decimal exponent notation as specified for XSD precisionDecimal (with "INF" being replaced with the more explicit "INFINITY").

Physical models may limit the range or precision of decimal numbers and may use canonical representations.

```
_Timestamp = _Text (* matching ISO8601 format YYYY-MM-DDThh:mm:ss±hh:mm *) ;
```

It should be interpreted as the corresponding ISO8601 point in time as specified for XSD dateTimeStamp.

Physical models may use canonical representations for UTC offset.

```
_Bytes = "0x" , _Text (* matching ([0-9a-f][0-9a-f])+ *) ;
```

It should be interpreted as the corresponding hex encoding of the sequence of bytes in network order.

Binary data requires Conceptual knowledge of its internal structure to be used, so, in general, its representation should be defined for each user data type using other constructions of the Logical model. When it is necessary to store binary data for better time or space performance, Physical models handling binary data should be used. The convention above may only be used as a last resort.

```
_Tag = "#" , _Name ;
```

This should be used to emphasize user-defined terms in _format.

```
_Name = _Text (* matching [_A-Za-z][_A-Za-z0-9]* *) ;
```

Identifiers are the principal way for a Conceptual model to convey meaning. As such, they should be carefully chosen and must respect the following rules:

- A "_" at the beginning is not prohibited but reserved for future standardization (as is the case in many languages)
- Although upper case may be used for readability, case may be altered depending on source language and operating systems and must not be considered significant for comparisons. "_" should be used to separate words in a complex _Name.
- They should be unique in a _Record or in _arg_names because the handling of duplicate _Names is undefined (for the same reason as explained in JSON RFC7159 section 4).
- They should convey the Conceptual data type of its value representation using appropriate standards and taxonomies including:
  - SI units or derived units: s, kg, mm, min, N_m, …
    ("per_" may be used for negative exponent quantities like: per_s, kg_per_m3)
  - RFC terms: ip_v4 …
  - SNOMED terms: varus, distal …
  - Pharmacological IU (International Unit)
  - User-defined taxonomies
- They should convey the role of its value in relation with an enclosing _Record:

  ```
  { "persons": [ {"first_name":"John", "last_name":"Doe", "birth_date":"08/05/1945"} ] }
  ```

  Example 10: Meaningful _Record _Names

- They should not convey implementation details such as:
  - C++ object member beginning with "m_": ~~m_~~birth_date
  - C++ pointer to implementation: ~~m_impl~~

- They may be equal to a _Type when there is no additional useful meaning:

```
{ "novels": [ {"author_name":"John", "_text":"Once upon a time …"} ] }
```

Example 11: _Types as _Record _Names

```
_Base_Type = "_Trace" | "_Event" | "_Record" | "_Sequence" | "_Null" | "_Text"
           | "_Boolean" | "_Integer" | "_Decimal" | "_Timestamp" | "_Bytes"
           | "_Tag" | "_Name" | "_Base_Type" | "_Type" | "_Character" ;
```

```
_Type = _Base_Type | _Name (* for user-defined subset of _Text with defined semantic *) ;
```

Every _Type _Name must define a _Text representation and semantic for all its values. Equal _Text representations must always denote equal values.

Different _Text may represent equal values, though (for instance: "123" and "123.0").

New _Type _Names may define operations on values for further analysis.

```
_Character = (* a single Unicode character *) ;
```

Encoding is left to Physical Models.

d) Physical models

This specification defines JSON, TSV+JSON, XML and CBOR physical models of the same conceptual _Trace. JSON is arguably the most universal and readable physical model and probably the first one to read to make sense of the specification. It is very close to Common Event Expression JSON encoding but with more explicit and fewer standard _Names to leave more space to domain-specific _Names and an open set of user _Data values.

Choosing another physical model may better suit particular needs:

- One advantage of TSV+JSON is readability
- One advantage of XML is its toolset
- One advantage of CBOR is performance (less encoding, more memory copies)

JSON

As stated at http://json.org/ :

> *JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is **easy for humans to read and write**. It is **easy for machines to parse and generate**. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is **completely language independent** but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an **ideal data-interchange language**. JSON is built on two structures:*
>
> - *A collection of name/value pairs: a JSON object begins with { and ends with }.*
> - *An ordered list of values: a JSON array begins with [and ends with].*

Example

```
[{"_elapsed_s": 0.01458
 ,"_timestamp":"2013-11-12T00:12:56+00:00"
 ,"_severity" : 7
 ,"_format"   :"#Trace QString(argv[0]) %s"
 ,"_args"     :[""               ]
 ,"_arg_names":["QString_argv_0"]
 ,"_arg_types":["_Text"          ]
 }
,{"_elapsed_s": 0.0152
 ,"_timestamp":"2013-11-12T00:12:56+00:00"
 ,"_severity" : 7
 ,"_format"   :"C-style logging is %s and %s"
 ,"_args"     :["not type-safe (may crash!)","not extensible to user types"]
 }
]
```

Example 12: Simple JSON _Trace

_Trace

When a JSON _Trace is contained in a JSON object, it should have a "_events" field containing the _Sequence of _Events.

The JSON object may be used to convey other metadata such as a reference to a specific Conceptual model to use to understand the _Trace.

The Logical model is encoded as follows:

_Record

It must be a JSON object.

_Sequence

It must be a JSON array.

_Null

It must be a JSON null.

_Text

It must be a JSON string unless the provider knows for sure it is one of the values below.

_Boolean

It should be a JSON true or false.

It may be a JSON string though for interoperability reasons.

_Integer

It should be a JSON number.

It may be a JSON string though for interoperability reasons.

_Decimal

It should be a JSON number when possible, or a JSON string (for "NaN" and "Infinity" values).

It may be a JSON string though for interoperability reasons.

TSV+JSON

This physical model uses the aforementioned JSON encoding of the logical model inside a TSV format. It aims to facilitate human exploration without sacrificing tools analysis. It places metadata common to all _Events in columns that can be used for filtering or simply eliminated when irrelevant to the task at end. It emphasizes changes in the _Trace by eliminating redundant values between 2 subsequent _Events.

NB: This format can be read following W3C best practices for parsing tabular data with the following non-default parameters: comment prefix: "#" ; delimiter: "\t" ; escape character "\", although eliminated redundancy between subsequent _Events must be restored specifically.

Example

_elapsed_s	_timestamp	_severity	_format	_other_data	_args	
0.00864119	2017-10-19 T18:37:26 +02:00	7	#Trace QString(argv[0]) %s	{"_path": "main.cpp"}	my.exe	
0.00879013			C-style logging is %s and %s	{}	not type-safe	not extensible
0.00898055		6	started demonstration to users			
0.0100073		2	failure affecting the user: %s		null	
0.0100504		7	#Trace md::Hex(&sfile) %s	{"_path": "main.cpp"}	0x79f7d0	
0.0101914			#Trace toPrint %s		10	
0.0103344			#Trace toPrint %s		plop	
0.0106528			#Trace toPrint %s		blip	
0.0107753			#Trace toPrint %s		42	
0.0110503			#Trace debugEnabled %s		TRUE	
0.0111072			#Trace current %s previous %s		1	1
0.0111459			#Trace current %s previous %s		2	1

Example 13: Simple TSV+JSON _Trace with hidden "\n" and "\t" between rows and cells

_Trace

The _Trace is split in lines that must end with "\n" (LF, U+000A) and/or "\r" (CR, U+000D) characters (as specified by the platform).

Lines starting with "#" are comment lines with unspecified meaning that may be used to convey _Trace metadata (such as a reference to a specific Conceptual model to use to understand the _Trace) or filter out _Event lines (as defined below).

The 1st non-comment line is a TSV nameline that must contain a sequence of _Names separated by "\t" (HT, U+0009) characters. This sequence:

- must include: "_elapsed_s", "_timestamp", "_format"
- must end with a required "_args" (further columns are implicitly interpreted as corresponding to the remaining _args items)
- should include: "_severity", "_category", "_function", and, when available: "_id", "_count", "_arg_names", "_ arg_types", "_other_data"

The subsequent non-comment TSV lines must represent the _Sequence of _Events from the _Trace in the same order.

_Event

The _Event items' values must be written into TSV fields separated by "\t" characters as follows:

- Each TSV field must contain the _Event item's value corresponding to the 1st line _Name, except _args
- When present, the TSV field corresponding to "_other_data" in the 1st line must contain a JSON _Record of all remaining _Event items
- Each _args items must be added as separate TSV fields in the same order

All values must be represented in JSON and all "\t", "\n", "\r" characters in JSON whitespace must be removed (JSON encodes them everywhere else).

Redundancy elimination

In each TSV column, an empty TSV field ("\t\t" without any character in between) denotes a value equal to the one in the previous TSV line (equality may be up to some arbitrary precision). This is the only use of "\t\t" (empty JSON string is: "\t""\t" and JSON null is "\tnull\t"). Encoders should use this value to eliminate redundancy as follows:

- _elapsed_s, _format, _id values should not be eliminated to facilitate human exploration (_format and _id give meaning to the _Event)
- _severity values different from 7 should not be eliminated to lower the risk of ignoring non-debug _Events during human exploration (space savings are not interesting anyway)
- By default, values in the following TSV columns equal to the one in the previous TSV line should be eliminated: _timestamp, _severity, _function, _path, _line, _count, _computer_id, _process_id, _thread_id, _user_id, _group_id, _object_id
- _timestamp values in the following range should not be eliminated to facilitate human exploration
  (previous._timestamp) +/- 1min
  (it allows synchronizing _Event with anything happening in the environment)
- _timestamp values in the following range may be eliminated:
  (previous._timestamp - previous._elapsed_s + _elapsed_s) +/- 0,1s

Decoders must replace empty TSV fields with the previous _Event's value.

XML

As stated at https://www.w3.org/XML/ :

> ***Extensible Markup Language*** *(XML) is a simple, very* ***flexible text format*** *derived from SGML [on which HTML is defined. It is] playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.*

It features an impressive amount of associated technologies that help validate, transform and process XML.

Example

```
<?xml version="1.0" encoding="utf-8" ?>
<trace>
<s name="_events">
   <r><t name="_elapsed_s" type="precisionDecimal">0.01458</t>
      <t name="_timestamp" type="dateTimeStamp">2013-11-12T00:12:56+00:00</t>
      <t name="_severity"  type="integer">7</t>
      <t name="_format">#Trace QString(argv[0]) %s</t>
      <s name="_args">
      </s>
   </r>
   <r><t name="_elapsed_s">0.0152</t>
      <t name="_timestamp">2013-11-12T00:12:56+00:00</t>
      <t name="_severity">7</t>
      <t name="_format">C-style logging is %s and %s</t>
      <s name="_args">
         <t>not type-safe (may crash!)</t>
         <t>not extensible to user types</t>
      </s>
   </r>
</s>
</trace>
```

Example 14: Simple XML _Trace

_Trace

An XML _Trace document should have a "trace" root element. Its _Sequence of _Events "s" element should have a "_events" name attribute.

The document root may be used to convey user-defined schemas for _Trace requirements and others.

The Logical model is encoded as follows:

_Record

It must be a XML element "r" with XML attribute "name" added to each child element and containing the corresponding _Identifier.

Although not required by the Logical Model, the order of child elements should be preserved as usual in XML documents.

_Sequence

It must be a XML element "s".

_Null

It must be a XML empty element "n" (that is to say <n/>, not a "n" element with empty text node <n></n>).

_Text

It must be a XML element "t".

The tag may contain a "type" attribute with the name of a XSD built-in data type containing the value.

_Boolean

It must be a XML element "t".

It should have type="boolean" attribute and the corresponding lexical representation.

_Integer

It must be a XML element "t".

It should have type="integer" attribute and the corresponding lexical representation.

_Decimal

It must be a XML element "t".

It should have type="precisionDecimal" attribute and the corresponding lexical representation.

_Timestamp

It must be a XML element "t".

It should have type="dateTimeStamp" attribute and the corresponding lexical representation with a preference for the default _Timestamp format.

_Bytes

It must be a XML element "t".

It should have type="hexBinary" or "base64Binary" attribute and the corresponding lexical representation.

CBOR

As stated at http://cbor.io/ :

> *The Concise Binary Object Representation (CBOR) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation.*
>
> - ***JSON data model**: CBOR is based on the wildly successful JSON data model: numbers, strings, arrays, maps (called objects in JSON), and a few values such as false, true, and null.*
> - ***No Schema needed**: One of the major practical wins of JSON is that successful data interchange is possible without casting a schema in concrete. This works much better in a world where both ends of a communication relationship may be evolving at high speed.*
> - ***Embracing binary**: Some applications that would like to use JSON need to transport binary data, such as encryption keys, graphic data, or sensor values. In JSON, these data need to be encoded (usually in base64 format), adding complexity and bulk.*
> - ***Concise encoding**: Some applications also benefit from CBOR itself being encoded in binary. This saves bulk and allows faster processing. One of the major motivators for the development of CBOR was the Internet of Things, which will include very simple, inexpensive nodes where this counts.*
> - ***Stable format**: CBOR is defined in an Internet Standards Document, RFC 7049. The format has been designed to be stable for decades.*
> - ***Extensible**: To be able to grow with its applications and to incorporate future developments, a format specification needs to be extensible. CBOR defines tags as a mechanism to identify data that warrants additional information beyond the basic data model. Both future RFCs and third parties can define tags, so innovation is "permissionless" but can still be coordinated.*

This physical model uses CBOR to allow storing and transferring data on constrained memory and processing hardware (IoT, embedded). It is not designed for efficient access like SQLite (which uses pages for the purpose).

Example

In CBOR diagnostic notation (inspired by JSON):

```
55799(
[_{_"_elapsed_s":0.01458_3
  , "_timestamp":0("2013-11-12T00:12:56+00:00")
  , "_severity" :7
  , "_format"   :"#Trace QString(argv[0]) %s"
  , "_args"     :[_]
  }
, {_"_elapsed_s":0.0152_3
  , "_format"   :"C-style logging is %s and %s"
  , "_args"     :
    [_           (_"not type-safe (may crash!)")
    ,            (_"not extensible to user types")
    ]
  }
])
```

Example 15: Simple CBOR diagnostic notation _Trace

The corresponding CBOR 251 bytes in hex encoding (as given by http://cbor.me):

```
D9 D9F7                                                                 # tag(55799)
  9F                                                                    # array(*)
    BF                                                                  # map(*)
      6A                                                                # text(10)
        5F656C61707365645F73                                            # "_elapsed_s"
      FB 3F8DDC1E7967CAEA                                               # primitive(4579558419549637354)
      6A                                                                # text(10)
        5F74696D657374616D70                                            # "_timestamp"
      C0                                                                # tag(0)
        78 19                                                           # text(25)
          323031332D31312D31325430303A31323A35362B30303A3030            # "2013-11-12T00:12:56+00:00"
      69                                                                # text(9)
        5F7365766572697479                                              # "_severity"
      07                                                                # unsigned(7)
      67                                                                # text(7)
        5F666F726D6174                                                  # "_format"
      78 1A                                                             # text(26)
        2354726163652051537472696E6728617267765B305D29202573            # "#Trace QString(argv[0]) %s"
      65                                                                # text(5)
        5F61726773                                                      # "_args"
      9F                                                                # array(*)
        FF                                                              # primitive(*)
      FF                                                                # primitive(*)
    BF                                                                  # map(*)
      6A                                                                # text(10)
        5F656C61707365645F73                                            # "_elapsed_s"
      FB 3F8F212D77318FC5                                               # primitive(4579915825216065477)
      67                                                                # text(7)
        5F666F726D6174                                                  # "_format"
      78 1C                                                             # text(28)
        432D7374796C65206C6F6767696E6720697320257320616E64202573        # "C-style logging is %s and %s"
      65                                                                # text(5)
        5F61726773                                                      # "_args"
      9F                                                                # array(*)
        7F                                                              # text(*)
          78 1A                                                         # text(26)
            6E6F7420747970652D7361666520286D61792063726173682129        # "not type-safe (may crash!)"
          FF                                                            # primitive(*)
        7F                                                              # text(*)
          78 1C                                                         # text(28)
            6E6F7420657874656E7369626C6520746F2075736572207479706573    # "not extensible to user types"
          FF                                                            # primitive(*)
        FF                                                              # primitive(*)
      FF                                                                # primitive(*)
    FF                                                                  # primitive(*)
```

Example 16: Simple CBOR hex binary _Trace

Legend: Logical structure, Conceptual _Names assigning meaning, Data, Comments

_Trace

When a CBOR _Trace is contained in a CBOR map, it should have a "_events" field containing the _Sequence of _Events.

The CBOR map may be used to convey other metadata such as a reference to a specific Conceptual model to use to understand the _Trace. The CBOR file may start with CBOR tag 55799 to distinguish its content from frequently used file types and particularly from any Unicode file.

Redundancy elimination

To save space and CPU time, encoders must eliminate redundancy between 2 subsequent _Events of a _Trace as follows:

- An _Event item present in the previous _Event and missing from the current one must be present with its _Name and set to _Null (this does not happen for items common to all _Events)
- An _Event item value equal to the previous _Event one should be eliminated along with its _Name
- _timestamp values in the following range may be eliminated:
  (previous._timestamp - previous._elapsed_s + _elapsed_s) +/- 0,1s

Decoders must replace missing items with the previous _Event's ones.

To save even more space, CBOR stringref tags may be used, especially for _Names and common _Event items such as _path.

The Logical model is encoded in CBOR as follows:

_Record

It must be a CBOR indefinite-length map (major type 5).

_Sequence

It must be a CBOR indefinite-length array (major type 4).

_Null

It must be the CBOR value 22 (Null) (major type 7).
The CBOR value 23 (Undefined) should be interpreted as _Null too.

_Text

It must be a CBOR text string (major type 3).
It should have a definite-length unless it costs too much performance.

_Boolean

It must be a CBOR value 20 (False) or 21 (True) (major type 7).

_Integer

It must be a CBOR integer (major type 0 or 1 depending on sign with appropriate 5-bit value followed by appropriate integer type).

_Decimal

It must be a CBOR double precision float (major type 7 with 5-bit value 27 followed by double)
Other CBOR precision types may be used.

_Timestamp

It must be a CBOR tag 0 (major type 6) followed by definite-length text string (major type 3)

NB: Redundancy elimination rules eliminate the need for complex binary encodings capturing time zone offsets and increased precision.

_Bytes

It must be a CBOR byte string (major type 2).
It should have a definite-length unless it costs too much performance.

3. Related Work

eXtensible Event Stream (XES) http://www.xes-standard.org/

MXML and XES define standard Trace formats used in Business Process Engineering. XES data structures are a mix of map (unique keys) and lists of key-values. We think the addition of simple _Sequences of values are required, especially to represent 1-n ER relationships. Moreover, XES mandates typing of all attributes. We propose a kind of structural typing, mostly explicit (_Record and _Sequence), partly implicit (if some _Text looks like a _Timestamp, we should use it accordingly), which is more convenient for intermediate transformations and sufficient for analysis (who needs to know more about event data anyway). All in all, XES looks like a big step to climb for developers logging raw text and this specification proposes a smoother path to structure logs with existing TRACEPOINTs.

Common Event Expression (CEE) https://cee.mitre.org/language/1.0-beta1/overview.html

CEE is a discontinued effort to standardize "network" event streams which is arguably the most advanced standardization work on structured logs. CEE Log Syntax describes both a JSON and XML encodings. We extend the approach by proposing a generic conceptual and logical trace model that can be implemented by many Physical models including binary formats. CEE Taxonomies are a very flexible way to add user-defined meaning to events. Since the object, action and status terms can almost never collide, this specification proposes to add all of them as _Tags into TRACEPOINT _format as a more informal but even more flexible way to classify events.

Syslog https://tools.ietf.org/html/rfc5424

We recognize syslog is a de facto standard for traces and use its definition of _severity because of its prevalence and operational background. But we argue its encoding of EVENTDATA is too complicated for simple analysis tools and propose a more general Logical model with a simple JSON encoding.

Windows Event Logs

We argue that the need to identify all events and describe them externally can only be done for the most lasting software, i.e. Operating Systems and core services, not for most applications. Thus, we propose to use _format along with other EVENTDATA to filter _Events and use their _args.