Syntactic Translation of Message Payloads Between At Least Partially Equivalent Encodings

Emanuel Palm, Cristina Paniagua, Ulf Bodin, Olov Schelén
Luleå University of Technology,
Luleå, Sweden
E-mail: {$firstname.lastname$}@ltu.se

***Abstract*—Recent years have seen a surge of interest in using IoT systems for an increasingly diverse set of applications, with use cases ranging from medicine to mining. Due to the disparate needs of these applications, vendors are adopting a growing number of messaging protocols, encodings and semantics, which result in poor interoperability unless systems are explicitly designed to work together. Key efforts, such as Industry 4.0, put heavy emphasis on being able to compose arbitrary IoT systems to create emergent applications, which makes mitigating this barrier to interoperability a significant objective. In this paper, we present a theoretical method for translating message payloads in transit between endpoints, complementing previous work on protocol translation. The method involves representing and analyzing encoding syntaxes with the aim of identifying the concrete translations that can be performed without risk of syntactic data loss. While the method does not facilitate translation between all possible encodings or semantics, we believe that it could be extended to enable such translation.**

I. Introduction

Improved IoT device interoperability has become an increasingly important ambition during the last few decades, motivating research within both industry and academia. For instance, the upcoming *Industrial IoT* (IIoT) paradigm, including efforts such as *Industry 4.0* [1], put heavy emphasis on making IoT devices work together to create emergent applications [2] [3]. Factory plant owners are expected to be able to buy sensors, actuators, vehicles and other machinery that can work together with little integration effort. Consequently, finding a means to dynamically facilitate device interoperability becomes a paramount objective. We contribute to this effort by presenting a theoretical method for translating message payloads in transit between interacting endpoints, fitting into systems such as the one depicted in Figure 1.

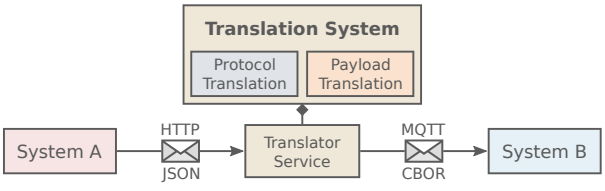

Fig. 1: A conceptual translation system producing a transient service for protocol and payload translation at runtime, which allows two otherwise incompatible systems to communicate.

A significant ambition of this work is to provide message payload translation capabilities to systems such as the Arrowhead Framework [4], a SOA-based IoT framework that supports the creation of scalable cloud-based automation systems. This kind of framework facilitates device service discovery and orchestration at runtime,without any need for human intervention. A would-be Arrowhead translator service could be thought of as an intermediary situated between two communicating systems, as is shown in Figure 1. As work has already been done on creating a multiprotocol translation system [5] for Arrowhead, the translation method we describe in this paper could be thought of as a complement to existing systems only supporting protocol translation.

At the time of writing, the list of message encodings in common use includes XML [6], ASN.1 [7], JSON [8], CBOR [9], Protocol Buffers [10] and many others. The length of this list could be attributed to the diversification of connected computing devices observed during the last few decades. As dimensions and costs of computers have been decreasing, new paradigms—such as the *Internet of Things* (IoT), *Big Data* and *Machine Learning*—have spurred the development of a growing variety of computing hardware, ranging from low-power wearable gadgets to quantum computers. Even though the use of many encodings may be important for optimizing the utility of these devices, it may become a stumbling block when the need to interoperate arises.

Previous interoperability efforts of relevance include (1) *protocol translation* [5], [11], [12], (2) *encoding libraries* supporting multiple concrete encodings [13], [14], [15] and (3) *ontological translation* [16], [17], [18]). For instance, Derhamy *et al.* present a multiprotocol translator useful within the Arrowhead IoT framework in [5]. Other protocol-related solutions include translation agents, protocol gateways [19] and adapters [20]. These efforts are not, however, concerned with message payloads, only with message protocols. Additionally, while there are several software libraries that can translate message payloads between encodings, none of these formally prove their translations to be lossless, which is an important focus of this work. Libraries of this kind include Jackson [14], Serde [13], Json.NET [15] and many others. Finally, ontological translation is concerned exclusively with message encodings describing effective arrays of triples, while our model can work with any kind of encoding.

In this paper, we present a theoretical method for translating encoding syntaxes. The method is useful for formulating *intersection encodings* that allow for encoded messages to be translated between multiple encodings with intersecting syntaxes without risk of syntactic information loss. In particular, the method is described in terms of representations and validation functions, differing from traditional encoding specifications in that they are concerned with abstract structures and elements instead of strings of bits.

II. Problem Description

The purpose of this work is to provide a rigorous approach to reasoning about and preventing information loss during syntactic encoding-to-encoding translation. However, before presenting such an approach, we first provide our definitions of encoding, translation and lossless translation and describe when a translation is not lossless.

A. Message Encodings

We define *encoding* as a set of rules followed to convert *interpreted messages* to and from binary strings. To convert such a message into a string, or to *encode* it, one is required to (1) construct a syntax tree representing the original message, (2) convert the syntax tree into a string of lexemes, and then, finally, (3) turn the lexemes into a binary string.[1] The result can then be *decoded* back into the original message by following the same steps in reverse order as depicted in Figure 2.

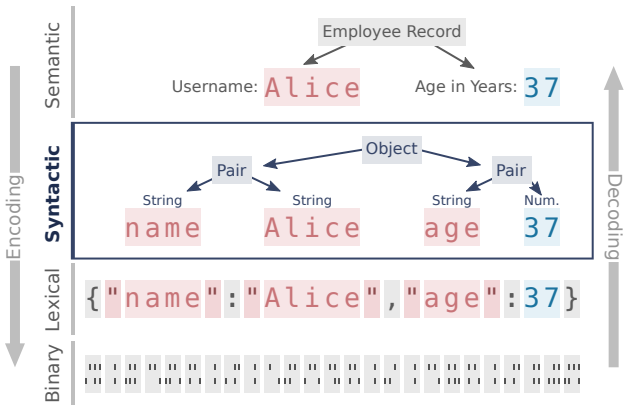

Fig. 2: Interpretational levels of a JSON message. At the *syntactic* level, the types and values of the individual parts of the message can be identified, but their context is unknown.

B. Message Translation

Message translation is the process of transforming one encoded message into another with a different encoding, preserving some level of meaning associated with the original message. A translator could be described as a function accepting a string c_a adhering to encoding a and returning either string c_b encoded with b or an error ϵ, such as

$$f_{a,b} : c_a \mapsto \{c_b, \epsilon\} \tag{1}$$

A translator $f_{a,b}$ can operate at any of the levels depicted in Figure 2, each providing different knowledge about the string being translated. A *syntactic translator* $f_{\Sigma a,b}$ knows only of a syntax tree V_a constructed from its input string c_a, as well as the specifications of its source and target encodings a and b. Facts not recorded in V_a, such as knowledge of certain structures being equivalent, canceling each other out or having insignificant ordering, cannot be acted upon. Specifically, $f_{\Sigma a,b}$ converts V_a it constructed from c_a into an equivalent form V_b in the syntax of the target encoding b, and then encodes V_b into string c_b as shown in Figure 3.

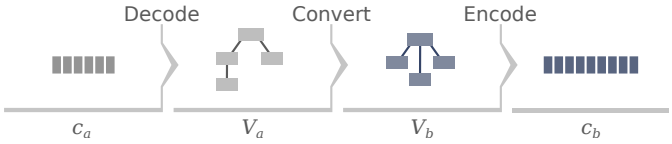

Fig. 3: Syntactic translation is the process of decoding a string, converting its syntax tree, and encoding the converted tree.

In the rest of this paper, we assume that syntactic translation from string c_a to string c_b can be performed if a syntax tree V_a can be converted into another tree V_b expressed with the syntax of the desired target encoding. As we are only concerned with syntactic translation, none of the other kinds are given any further treatment.

C. Syntactically Lossless Message Translation

For translation from syntax tree V_a into syntax tree V_b to be considered syntactically lossless, V_a and V_b must express the same structural information. This requirement means that if V_a holds an array of three integers, then V_b must also contain an array of the same three integers, even though V_a and V_b being formulated with different encoding type systems. Specifically, a syntactic translation from V_a to V_b using $f_{\Sigma a,b}$ is syntactically lossless only if there exists a translator $f_{\Sigma b,a}$ such that

$$f_{\Sigma b,a}(f_{\Sigma a,b}(V_a)) = V_a \tag{2}$$

In other words, if the original syntax tree can be recovered from the translated tree, all original data exist in the translated tree, and translation is lossless.

Note that syntax trees rather than strings are being compared. The input c_a and output c'_a of a lossless translation could differ even if their syntax trees are identical. Many encodings allow for the same syntactic structures to be encoded in multiple ways, such as numbers being allowed to have multiple bases (decimal, hexadecimal, etc.) or text strings being allowed to have particular characters escaped in several ways. Comparisons must be considered as being made between abstract objects, such as numbers, lists or texts, rather than concrete binary strings.

[1] We use the term *syntax tree* exclusively for referring to abstract syntax trees. Concrete syntax trees, or parse trees, are not strictly necessary for either encoding or decoding; hence we do not consider them here.

D. Syntactically Lossy Message Translation

What would cause a translation to be syntactically lossy? An encoding could be thought of a set of data structures useful for expressing arbitrary messages. When translating between two encodings, a translator is required to express the data of the original message using the data structures of the target encoding. During translation, information may be omitted or changed, so that the original message can no longer be reconstructed. Consider the example in Figure 4, in which a binary string is converted from XML to JSON and back.

Fig. 4: XML [6] string translated to JSON [8] and back to XML, resulting in syntactic information loss.

The XML messages in Figure 4 are syntactically different and, therefore, do not satisfy the lossless property defined in Section II-C. The first message uses attributes, while the second uses child nodes. Additionally, the second message no longer has the original name of its root element. XML provides no type equivalent to a JSON object, which in this case resulted in lossy syntax transformations. A lossless translation between XML and JSON would have required a rigorous syntax transformation scheme, or *syntax simulation*, described further in Section IV.

III. Representation, Validation and Migration

Having established the notion of a syntactic encoding-to-encoding translation, we now proceed to define syntax trees, syntaxes and intersection syntaxes and discuss how the latter can guarantee lossless translation. In particular, we are interested in the representations of these entities and in knowing when a given representation is valid. Our definitions are presented using common constructs of *first-order logic*.[2]

A. Syntax Trees

We consider a *syntax tree* to be a directed acyclic graph constructed from *nodes*. Given a *syntax type name* t and a *syntax value* x, we define a node V as a tuple:

$$V = \langle t : x \rangle \tag{3}$$

The type name t is a reference to a certain *syntax type definition* T_i that describes the set of objects that are allowed to occupy the syntax value $x \in \mathbb{X}$. There are two varieties of syntax values: *sequences* and *elements*. If a syntax value is a sequence of child nodes, its node is considered a Branch, while if it is an element, its node is considered a Leaf.

[2]See [21] for an introduction. Note that we allow for any lowercase letter to denote a variable, use one capital letter to denote a set or another collection, use $P(x)$ to denote a *predicate* P with a single term x, use $P(x, y)$ to denote a predicate with two terms x and y, use $\wedge$ instead of & to signify *conjunction* (AND), use $\oplus$ as *exclusive disjunction* (XOR) connective, use $\exists!$ as the uniqueness quantifier and, finally, consider the implication $\theta \rightarrow \psi$ to be equivalent to $\psi \iff \theta$.

1) Branch Nodes: A Branch node V holds a type name t and a sequence of child nodes $S \in \mathbb{S}$, as follows:

$$V = \langle t : S = [V_0, V_1, ..., V_{|S|}] \rangle \tag{4}$$

$\mathbb{S}$ is the set of all possible child node sequences, while $|S|$ is the number of child nodes in S. Above, t names a syntax type definition T_i that identifies a relevant subset of $\mathbb{S} \subset \mathbb{X}$. Various data structures serving to group values together, such as arrays, tuples, sets, dictionaries, classes, etc., are suitably represented by Branches. We consider all sequences to be fundamentally ordered and view the property of being unordered as superimposed at the level of semantics.

2) Leaf Nodes: Every Leaf node V contains a type name t and an element $e \in \mathbb{E}$ as follows:

$$V = \langle t : e \rangle \tag{5}$$

$\mathbb{E}$ is the set of all objects not considered to be collections of other objects. Above, t names a syntax type definition T_i that identifies a relevant subset of $\mathbb{E} \subset \mathbb{X}$. Such a subset could include all numbers within a specific range or all strings of bytes conforming to a certain text encoding. What concrete members $\mathbb{E}$ contains is subject to interpretation, but useful definitions could include `null`, `true`, `false`, all other enumerators, all numbers and all binary strings.

3) Example: The syntactic level of the JSON [8] object in Figure 2 could be written with our tuple notation as

$$\begin{aligned} &\langle \text{Object: } [\\ &\quad \langle \text{Pair: } [\langle \text{String: } \text{"}name\text{"} \rangle, \langle \text{String: } \text{"}Alice\text{"} \rangle] \rangle, \\ &\quad \langle \text{Pair: } [\langle \text{String: } \text{"}age\text{"} \rangle, \langle \text{Number: } 37 \rangle] \rangle \ ] \rangle \end{aligned} \tag{6}$$

The message is also shown in Figure 5 that makes the difference between Branch and Leaf nodes more apparent.

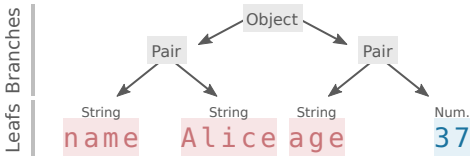

Fig. 5: The Branches and Leaves of a JSON [8] syntax tree. Branches refer to other nodes while Leaves hold elements.

B. Syntaxes

For a syntax tree to be *valid*, it must conform to the structure imposed by an encoding *syntax*. Such validity is typically guaranteed during source string decoding via a set of *parse rules* that effectively limit the set of acceptable lexemes to only those resulting in valid tree nodes. However, syntactic translation entails converting a syntax tree into another such tree, and the new tree may not be guaranteed to be valid. This possibility necessitates the formulation of rules able to verify syntax trees directly. Consequently, we define a *syntax* Σ_j as

a collection of such rules, here named *syntax type definitions* $T = \{T_0, T_1, ..., T_{|T|}\}$, and a *root set* R as follows:

$$\Sigma_j = \langle T, R \rangle \tag{7}$$

Each syntax type definition $T_i \in T$ describes one type of syntax tree node permitted by one particular encoding, while the root set R identifies the types of nodes that are allowed to be at the root of a complete syntax tree.

1) Syntax Type Definitions: A syntax-type definition $T_i \in T$ is a predicate, accepting a syntax tree node V as its only term. T_i *must not* be satisfied unless the type name $t \in V$ equals a type name t_i associated only with T_i. Consequently, every T_i must be defined in general as

$$T_i(V) \iff \underbrace{(V = \langle t : x \rangle)}_{1} \wedge \underbrace{(t = t_i)}_{2} \wedge \underbrace{p(x)}_{3} \tag{8}$$

$T_i(V)$ is satisfied only if three conditions are fulfilled: (1) the tested term is a syntax value $V = \langle t : x \rangle$, (2) the syntax-type name $t \in V$ is equal to t_i, which is the name of T_i, and (3) an arbitrary function $p(x)$ yields $true$ when provided with the syntax value $x \in V$. To make syntax-type definitions less verbose, the following form is also used:

$$T_i(\langle t : x \rangle) \iff (t = t_i) \wedge p(x) \tag{9}$$

For instance, the syntax-type definition for a Boolean LEAF type could be specified as follows:

$$Boolean(\langle t : e \rangle) \iff (t = \text{Boolean}) \wedge (e \in \{1, 0\}) \tag{10}$$

The syntax node $V = \langle \text{Boolean} : 1 \rangle$ would satisfy the $Boolean(V)$ predicate, while the nodes $\langle \text{Boolean} : true \rangle$ and $\langle \text{Binary} : 0 \rangle$ would not.

2) Syntax Tree Validation: We are now able to represent a syntax as a tuple $\Sigma_j = \langle T, R \rangle$ and are able to determine if any individual syntax tree node is valid. To determine if the entire syntax tree is valid, however, we must examine both the validity of its root node and whether its syntax type is allowed at the root of the tree. For this reason, we define the predicate $Valid_{\Sigma_j}(V, \Sigma_j)$, satisfied as follows:

$$\begin{aligned} Valid_{\Sigma_j}(V, \Sigma_j) \iff & \overbrace{(V = \langle t : x \rangle)}^{1} \wedge \overbrace{(\Sigma_j = \langle T, R \rangle)}^{2} \\ & \wedge \underbrace{(t \in R)}_{3} \wedge \underbrace{(\exists! T_i \in T)(T_i(V))}_{4} \end{aligned} \tag{11}$$

In other words, if (1) V is a syntax tree node, (2) Σ_j is a syntax, (3) the syntax type name $t \in V$ exists in the syntax root set R, and (4) there exists exactly one syntax type definition $T_i \in T \in \Sigma_j$ satisfied by V, then V describes a valid syntax tree according to Σ_j.

3) Example: We have now provided a sufficient number of definitions to formulate a complete syntax. An example of a naive syntax is given for the JSON [8] encoding in Table I. A sample JSON syntax tree has already been presented in Equation 6.

TABLE I: Naive JSON syntax.

Syntax Type Definitions (T)

BRANCHES

$Object(\langle t : S \rangle) \iff (t = \text{Object}) \wedge (\forall V_i \in S)(Pair(V_i))$
$Pair(\langle t : S \rangle) \iff (t = \text{Pair}) \wedge (S = [V_0, V_1]) \wedge String(V_0) \wedge v(V_1)$
$Array(\langle t : S \rangle) \iff (t = \text{Array}) \wedge (\forall V_i \in S)(v(V_i))$

LEAFS

$Number(\langle t : e \rangle) \iff (t = \text{Number}) \wedge (e \in \mathbb{F} \wedge\ e \notin \{+\infty, -\infty, \text{NaN}\})$
$String(\langle t : e \rangle) \iff (t = \text{String}) \wedge (e \in \text{UTF-8})$
$True(\langle t : e \rangle) \iff (t = \text{True}) \wedge (e = \text{true})$
$False(\langle t : e \rangle) \iff (t = \text{False}) \wedge (e = \text{false})$
$Null(\langle t : e \rangle) \iff (t = \text{Null}) \wedge (e = \text{null})$

Auxiliary Function

$v(V) = Object(V) \oplus Array(V) \oplus Number(V) \oplus String(V) \oplus True(V) \oplus False(V) \oplus Null(V)$

Root Set (R)

{Object, Array}

$\mathbb{F}$ is the set of all IEEE 754-2008 [22] binary64 floating-point numbers, while UTF-8 is the set of all UTF-8 compliant byte strings. Note that $u(V)$ does not mention $Pair$ since nodes of that type may only occur inside $Objects$.

C. Intersection Syntaxes and Syntax Tree Migration

When comparing any two syntaxes, one may discover that some of their types are similar. Both may define syntax types for numbers, text strings, arrays, maps, etc. Deciding on a list of associations between the types of encoding syntaxes, one might be able to identify syntax trees that would be considered valid by either syntax, i.e., the type name of every node in a tree could be replaced with the name of its corresponding type. We regard such a set of syntaxes with explicit type associations as an *intersection syntax* and refer to the process of changing the type names of a syntax tree as *syntax migration*.

If $C = \{C_0, C_1, ..., C_{|\Sigma|}\}$ is a set of *syntax type intersections* and $\Sigma = \{\Sigma_0, \Sigma_1, ..., \Sigma_{|\Sigma|}\}$ is a set of *concrete syntaxes*, we formally define an intersection syntax $\hat{\Sigma}$ as follows:

$$\hat{\Sigma} = \langle C, \Sigma \rangle \tag{12}$$

1) Syntax Type Intersections: Each $C_i = \{T_0, T_1, ..., T_{|\Sigma|}\}$ is a set of associated syntax type definitions, where exactly one T_k is taken from each associated syntax $\Sigma_j \in \Sigma$.[3] We refer to every such association C_i as a *syntax type intersection* and consider each an effective syntax-type definition, describing the intersection of the sets of syntax values every $T_k \in C_i$ deems valid. Consequently, each C_i can be used to validate a syntax node V as follows:

$$\begin{aligned} Valid_{C_i}(V, C_i) \iff & \overbrace{(V = \langle t : x \rangle)}^{1} \wedge \\ & \underbrace{(\exists! T_k \in C_i)(t = t_k)}_{2} \wedge \underbrace{(\forall T_k \in C_i)(T_k(\langle t_k, x \rangle))}_{3} \end{aligned} \tag{13}$$

[3]We do not provide any algorithmic means of determining correct or optimal sets of C_i in this paper, even though it could be a relevant topic for future research.

$Valid_{C_i}(V, C_i)$ is satisfied by three conditions: (1) the tested term is a syntax value $V = \langle t : x \rangle$, (2) the syntax type name $t \in V$ is identical to the name of exactly one associated type definition $T_k \in C_i$, and (3) $x \in V$ together with the type name t_k of each $T_k \in C_i$ satisfies every associated predicate $T_k(V)$. In other words, if a syntax node V names one syntax-type definition in C_i and satisfies all such predicates in C_i, $Valid_{C_i}(V, C_i)$ is also satisfied.

2) Syntax Tree Validation: Ensuring that the entire syntax tree V is valid according to a certain intersection syntax $\hat{\Sigma}$ requires both that every syntax tree node be valid as asserted by $Valid_{C_i}(V, C_i)$ and that the root node of that tree be valid in every concrete syntax $\Sigma_j \in \Sigma \in \hat{\Sigma}$, as follows:

$$\begin{aligned} Valid_{\hat{\Sigma}}(V, \hat{\Sigma}) \iff & \overbrace{(\hat{\Sigma} = \langle C, \Sigma \rangle)}^{1} \wedge \\ & \underbrace{(\exists! C_i \in C)(ValidRoot(C_i, \Sigma) \wedge Valid_{C_i}(V, C_i))}_{2} \\ ValidRoot(C_i, \Sigma) \iff & \\ & (\forall T_k \in C_i)((\exists! \langle T, R \rangle \in \Sigma)(t_k \in R)) \end{aligned} \quad (14)$$

In other words, if (1) $\hat{\Sigma}$ is an intersection syntax $\langle C, \Sigma \rangle$ and (2) there exists exactly one syntax-type intersection $C_i \in C$ where (a) every concrete type definition $T_k \in C_i$ is a valid root type and (b) $Valid_{C_i}(V, C_i)$ is satisfied, then V describes a valid syntax tree according to $\hat{\Sigma}$. If an intersection syntax is able to successfully validate at least one syntax tree V, we refer to its encodings as being *at least partially equivalent*.

3) Syntax Migration: In Section II-B, we claimed that if a syntax tree could be converted into another such with another encoding syntax, syntactic translation could be performed. We have just defined $Valid_{\hat{\Sigma}}$ that can be used to determine if a syntax tree would be considered valid by a different syntax if only its type names were changed, or *migrated*, to those of a related syntax. This definition means that if an intersection syntax $\hat{\Sigma}$ can be formulated and a syntax tree V satisfies $Valid_{\hat{\Sigma}}$, V can be translated to any other encoding in $\hat{\Sigma}$.

More formally, given two syntaxes $\{\Sigma_a, \Sigma_b\} \subset \Sigma \in \hat{\Sigma}$ and a syntax tree V_a of Σ_a satisfying $Valid_{\hat{\Sigma}}(V_a, \hat{\Sigma})$, syntax migration is the process of replacing every type name t of every node in V_a with its corresponding type of Σ_b, resulting in V_b. The correspondence between types in Σ_a and Σ_b is established by the syntax-type intersections $C \in \hat{\Sigma}$.

4) Example: We have again reached the point where we can formulate a concrete example based on the presented theory. As an intersection syntax requires at least two concrete syntaxes, we provide a naive CBOR [9] subset (CBORS). CBOR is used due to being similar to JSON.[4] Disregarding the names of CBOR's and JSON's syntax types, they differ only in JSON requiring the first $Pair$ element to be a $String$, in not being able to express the same numbers, and in only CBOR having a dedicated type for arbitrary byte strings. Table II outlines the CBORS syntax, while Table III shows a JSON/CBORS intersection syntax.

[4]This is no coincidence, as CBOR was designed to be able to encode everything expressible with JSON. For illustrative purposes, however, we do not include enough of the CBOR specification for this to be true here.

TABLE II: Naive CBOR Subset (CBORS) syntax.

Syntax Type Definitions (T)

BRANCHES

$Map(\langle t : S \rangle)$	$\Longleftarrow (t = \text{Map}) \wedge (\forall V_i \in S)(Pair(V_i))$
$Pair(\langle t : S \rangle)$	$\Longleftarrow (t = \text{Pair}) \wedge (S = [V_0, V_1]) \wedge d(V_0) \wedge d(V_1)$
$Array(\langle t : S \rangle)$	$\Longleftarrow (t = \text{Array}) \wedge (\forall V_i \in S)(d(V_i))$

LEAFS

$Integer(\langle t : e \rangle)$	$\Longleftarrow (t = \text{Integer}) \wedge (e \in \mathbb{Z} \wedge -2^{64} < e < 2^{64})$
$ByteString(\langle t : e \rangle)$	$\Longleftarrow (t = \text{ByteString}) \wedge (e \in \text{STRING})$
$TextString(\langle t : e \rangle)$	$\Longleftarrow (t = \text{TextString}) \wedge (e \in \text{UTF-8})$
$True(\langle t : e \rangle)$	$\Longleftarrow (t = \text{True}) \wedge (e = \text{true})$
$False(\langle t : e \rangle)$	$\Longleftarrow (t = \text{False}) \wedge (e = \text{false})$
$Null(\langle t : e \rangle)$	$\Longleftarrow (t = \text{Null}) \wedge (e = \text{null})$

Auxiliary Function

$d(V) = Map(V) \oplus Array(V) \oplus Integer(V) \oplus ByteString(V) \oplus TextString(V) \oplus True(V) \oplus False(V) \oplus Null(V)$

Root Set (R)

{Map, Array, Integer, ByteString, TextString, True, False, Null}

STRING is the set of all possible byte strings. Note that $d(V)$ does not mention $Pair$ since nodes of that type may only occur inside $Maps$.

TABLE III: JSON/CBORS intersection syntax.

Syntax Type Intersections (C)

Σ_{JSON}		Σ_{CBORS}
$Object$	$\leftrightarrow$	Map
$Pair$	$\leftrightarrow$	$Pair$
$Array$	$\leftrightarrow$	$Array$
$Number$	$\leftrightarrow$	$Integer$
	$\not\leftrightarrow$	$ByteString$
$String$	$\leftrightarrow$	$TextString$
$True$	$\leftrightarrow$	$True$
$False$	$\leftrightarrow$	$False$
$Null$	$\leftrightarrow$	$Null$

Associated Syntaxes (Σ)

$\{\Sigma_{\text{JSON}}, \Sigma_{\text{CBOR}}\}$

$\leftrightarrow$ denotes syntax type correspondence, while $\not\leftrightarrow$ signifies a syntax type not having a corresponding type.

Consider the JSON syntax tree in Equation 6. Assuming that we desire to convert this syntax tree to CBORS, we first ensure that it is valid according to our intersection syntax $\hat{\Sigma}$ by testing if it satisfies $Valid_{\hat{\Sigma}}(V, \hat{\Sigma})$. After ensuring this, we proceed to migrate it to the desired syntax, resulting in

$$\begin{aligned} &\langle \text{Map: } [\\ &\quad \langle \text{Pair: } [\langle \text{TextString: "}name\text{"} \rangle, \langle \text{TextString: "}Alice\text{"} \rangle] \rangle, \\ &\quad \langle \text{Pair: } [\langle \text{TextString: "}age\text{"} \rangle, \langle \text{Integer: } 37 \rangle] \rangle \ ] \rangle \end{aligned} \quad (15)$$

A JSON syntax tree V not satisfying $Valid_{\hat{\Sigma}}(V, \hat{\Sigma})$ would have to refer to a number that is not an integer in the range $(-2^{64}, 2^{64})$. If we, on the other hand, were translating from CBORS to JSON, an unsatisfactory CBORS syntax tree V would have to use a $Pair$ with a node that is not a $TextString$

as the first element, an integer not expressible as a binary 64-bit IEEE float [22], or contain any $ByteString$.

IV. Conclusions and Directions for Future Work

The problem of heterogeneous system interoperability has received significant attention in recent years. However, while efforts have made to translate message protocols, the problem of rigorous message payload translation has been largely ignored. In this paper, we presented a theoretical method for message payload translation that facilitates preventing information loss during syntactic encoding-to-encoding translation. The method is described in terms of representations, validation, intersections and migration of syntax trees.

The formalism of representation is a key aspect of the proposed method. If there is any ambiguity, the result may be erroneous. Therefore, the method needs to be used correctly and rigorously. Despite possible disadvantages, we believe that a method of defining various encodings strictly is necessary for being able to correctly translate message payloads between heterogeneous systems.

As the interpretation of a given encoding specification may leave room for ambiguity, multiple incompatible syntaxes could be formulated for that encoding. To prevent such a case, vendors and developers would be responsible for the implementation and provision of syntaxes, avoiding the mismatch between the used encoding and syntax.

Future work includes the extension and refinement of the method, which involves investigating various aspects of syntax simulation and translator implementation.

The syntactic translation solution we presented in this paper only allows for conversions between encodings with intersecting syntaxes. To be able to translate *any* syntax tree to *any other* encoding, there must be a way to simulate syntactic structures that cannot be expressed in the native type system of the target encoding. Such simulation would require one to reason about the types of simulated data structures the receiver of a translated message would be able to interpret correctly, i.e., the activity of simulation could be regarded as constructing new encodings out of existing encodings.

Lastly, we believe that the implementation and evaluation of a multiencoding translator using our translation method would be a significant complement to this work.

Acknowledgements

This work was funded via the *Productive 4.0* project (EU ARTEMIS JU grant agreement no. 737459).

References

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, 2014. [Online]. Available: http://dx.doi.org/10.1007/s12599-014-0334-4

[2] P. Varga, F. Blomstedt, L. L. Ferreira *et al.*, "Making system of systems interoperable — the core components of the arrowhead framework," *Journal of Network and Computer Applications*, vol. 81, pp. 85–95, 2017. [Online]. Available: https://doi.org/10.1016/j.jnca.2016.08.028

[3] M. Weyrich and C. Ebert, "Reference architectures for the internet of things," *IEEE Software*, vol. 33, no. 1, pp. 112–116, 2016. [Online]. Available: https://doi.org/10.1109/MS.2016.20

[4] J. Delsing, *Iot automation: Arrowhead framework*. CRC Press, 2017.

[5] H. Derhamy, J. Eliasson, and J. Delsing, "Iot interoperability—on-demand and low latency transparent multiprotocol translator," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1754–1763, October 2017. [Online]. Available: https://doi.org/10.1109/JIOT.2017.2697718

[6] T. Bray *et al.*, "Extensible markup language (XML) 1.1 (second edition)," W3C, W3C Recommendation, Aug 2006, accessed 2018-06-12. [Online]. Available: http://www.w3.org/TR/2006/REC-xml11-20060816

[7] International Telecommunication Union, "Information Technology – ASN.1 Encoding Rules – Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)," ITU-T Recommendation X.690, July 2002, accessed 2018-06-12. [Online]. Available: http://handle.itu.int/11.1002/1000/12483

[8] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159, Mars 2014. [Online]. Available: https://doi.org/10.17487/RFC7159

[9] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 7049, October 2013. [Online]. Available: https://doi.org/10.17487/RFC7049

[10] Google. (2018) Protocol buffers. Accessed 2018-01-30. [Online]. Available: https://developers.google.com/protocol-buffers

[11] C. Lerche, N. Laum, F. Golatowski *et al.*, "Connecting the web with the web of things: lessons learned from implementing a coap-http proxy," in *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*, October 2012. [Online]. Available: https://doi.org/10.1109/MASS.2012.6708525

[12] K. L. Calvert and S. S. Lam, "Adaptors for protocol conversion," in *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies*, vol. 2. IEEE, June 1990, pp. 552–560. [Online]. Available: https://doi.org/10.1109/INFCOM.1990.91294

[13] D. Tolnay *et al.* (2018) Serde. Accessed 2018-06-13. [Online]. Available: https://serde.rs

[14] T. Saloranta *et al.* (2018) Jackson project home @ GitHub. Accessed 2018-06-13. [Online]. Available: https://github.com/FasterXML/jackson

[15] Newtonsoft. (2018) Json.NET. Accessed 2018-06-13. [Online]. Available: https://www.newtonsoft.com/json

[16] M. Ganzha, M. Paprzycki, W. Pawłowski *et al.*, "Streaming semantic translations," in *21st International Conference on System Theory, Control and Computing (ICSTCC)*, 2017, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ICSTCC.2017.8107003

[17] D. Dou, D. McDermott, and P. Qi, "Ontology translation on the semantic web," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, R. Meersman, Z. Tari, and D. C. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 952–969. [Online]. Available: https://doi.org/10.1007/978-3-540-39964-3_60

[18] H. Chalupsky, "Ontomorph: A translation system for symbolic knowledge," in *17th International Conference on Knowledge Representation and Reasoning (KR-2000*, April 2000, pp. 471–482, accessed 2018-07-06. [Online]. Available: https://isi.edu/~hans/publications/KR2000.pdf

[19] M. Jung, J. Weidinger, C. Reinisch *et al.*, "A transparent ipv6 multiprotocol gateway to integrate building automation systems in the internet of things," in *2012 IEEE International Conference on Green Computing and Communications (GreenCom)*. IEEE, 2012, pp. 225–233. [Online]. Available: https://doi.org/10.1109/GreenCom.2012.42

[20] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 2, pp. 292–333, 1997. [Online]. Available: https://doi.org/10.1145/244795.244801

[21] S. Shapiro and T. Kouri Kissel, "Classical logic," in *The Stanford Encyclopedia of Philosophy*, spring 2018 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2018, accessed 2018-08-29. [Online]. Available: https://plato.stanford.edu/archives/spr2018/entries/logic-classical

[22] IEEE Computer Society, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, August 2008. [Online]. Available: https://doi.org/10.1109/IEEESTD.2008.4610935