Semantic Hashing to Remedy Uncertainties in Ontology-Driven Edge Computing*

Konstantin Ryabinin[1,2][0000-0002-8353-7641] and Svetlana Chuprina[2][0000-0002-2103-3771]

[1] Saint Petersburg State University, 7/9 Universitetskaya Emb., Saint Petersburg 199034, Russia
[2] Perm State University, 15 Bukireva Str., Perm 614068, Russia
`kostya.ryabinin@gmail.com, chuprinas@inbox.ru`

Abstract. This paper discusses the specific kind of uncertainties, which appear in ontology-driven software development. We focus on the development of IoT applications whose source code is generated automatically by an ontology-driven framework. So-called "compatibility uncertainties" pop up when the ontology is being changed while the corresponding generated application is in operation. This specific kind of uncertainties can be treated as a variant of implementation uncertainties. The algorithm of its automated handling is presented. The proposed algorithm is implemented within the SciVi platform and tested in the real-world project devoted to the development of custom IoT-based hardware user interfaces for virtual reality. We use the SciVi platform as a toolset for the automatic generation of IoT devices firmware for ontology-driven Edge Computing but the problem discussed is common for any tools which are used for the generation of ontology-driven software.

Keywords: Ontology-Driven Edge Computing · IoT · Firmware Generation · Semantic Hashing · Implementation Uncertainties.

1 Introduction

The spread of Ubiquitous Computing faces the challenges of configurability, interoperability, and context awareness of programmable microelectronic devices, which build up the Internet of Things (IoT). These devices are expected to interconnect into sustainable computing networks, maintain the data flow, and support user-friendly human-machine interaction to fulfill the Ubiquitous Computing paradigm [3,1,11]. One of the strategies to improve networking stability and performance within the Ubiquitous Computing environment is to push the capabilities of artificial intelligence to the networks' edge. This is all about providing the end-point lightweight devices (so-called Edge Nodes) with the ability to track the context of their work and make decisions on their own without

* This study is supported by the research grant No. ID92566385 from Saint Petersburg State University.

relying on more powerful hub devices (so-called Fog Nodes) or remote services (so-called Cloud Nodes).

The promising approach to enhance the intelligent features of smart devices is the bridging of the IoT with the Semantic Web technologies that is called the Semantic Web of Things (SWoT) [9]. SWoT assumes leveraging ontologies to build formal models of knowledge about the devices, interfaces, networks, and processed data to overcome the interoperability, configurability, and accessibility limitations of the traditional IoT.

The logical step of SWoT development is the ontology-driven Edge Computing (ODEC) vision [25,1,27,28] that enables ontologies to act not just as descriptive models of IoT artifacts, but also as full-fledged drivers of computation and communication processes on the network's edge.

In our previous work [25], we contributed to the development of the ODEC by the following. First, we created the semantic compression algorithm to fit the task ontologies [31] to the memory of resource-constrained edge devices. Second, we implemented a tiny reasoner that is capable of running on edge devices, interpreting the compressed task ontologies, and performing the computation and communication tasks according to ontological specifications. The reasoner consists of an immutable core and a dynamic set of functions (so-called operators), which can be referenced from within task ontologies. Actual firmware for a particular edge device is generated automatically by means of services provided by our platform SciVi[3] [24], and namely, its special toolset called EdgeSciVi Workbench. The firmware generation is driven by an application ontology that combines domain and task ontologies and offers a specified system of concepts for a particular application [31]. In our case, the application ontology describes a set of supported hardware components, data processing mechanisms, communication protocols, and interoperability techniques (examples of such kind of application ontologies can be found in [26]). Once generated, the firmware can be installed on the desired device. After that, the actual programs this device should execute can be encoded as task ontologies and pushed to the device without reflashing. To compose the task ontologies, we have a special high-level visual programming tool based on the data flow paradigm. The building blocks for the data flow diagrams are operators, which are derived from the application ontology and correspond to the functions incorporated in the embedded reasoner. The entire workflow is based on the so-called low-code concept [16] allowing users with no programming hard skills to declaratively describe the data processing pipelines by means of high-level building blocks. All the low-level details of computation, communication, and configuration are automatically covered by the SciVi platform, which extracts all the relevant knowledge from the underlying ontological knowledge base.

Although this approach provides a fast and easy way to organize the Edge Computing process, it still has limitations. The one we address in the present work is the uncertainty that can be called “compatibility uncertainty” of a random edge device discovered in the network.

[3] `https://scivi.tools/`

The problem is as follows. Let us have an edge device, whose firmware is an embedded reasoner that has been generated by SciVi according to the application ontology $\mathcal{D}$. This reasoner incorporates a set of functions (operators) Φ that is a subset of all the operators described by $\mathcal{D}$ (the actual subset of operators for a particular firmware is being chosen by the user). After the firmware installation, let us turn off this device for a while. Let us change $\mathcal{D}$ while the device is offline. After this device reappears in the network, it will be discovered as capable of ODEC. However, as $\mathcal{D}$ has been changed since the firmware generation, attempts to use $\mathcal{D}$ as a source for task ontologies for this device will lead to undefined behavior because the operators described by $\mathcal{D}$ can be incompatible anymore with the functions in the device firmware.

The obvious solution for this problem would be version numbering control. A version number can be assigned to ontology $\mathcal{D}$ and stored in the embedded reasoner by its generation. With each change, the version number of $\mathcal{D}$ should be increased. When used in ODEC context, the device should indicate the version of $\mathcal{D}$ it contains and this version should be matched with the actual version of $\mathcal{D}$. If these versions differ, the firmware has to be regenerated and reinstalled. However, this naive solution has two major drawbacks. First, ontology $\mathcal{D}$ is in fact a merge of small ontologies, which describe individual data processing operators, data types, communication protocols, hardware elements, etc. The merging of these ontologies is performed by SciVi automatically at runtime [4], while the knowledge engineers work with the small initial ontologies, which are much more readable and handy. So, version numbering of $\mathcal{D}$ cannot be implemented directly. The second drawback is the uncertainty of whether the changed $\mathcal{D}$ is really incompatible with the current version of the device firmware or not. If any single change of any part of $\mathcal{D}$ requires a firmware update, all the ODEC profit in device reconfiguration speed and easiness is nullified.

In fact, only those changes of $\mathcal{D}$ matter, which affect the description of operators incorporated in the particular reasoner. To track these changes, we propose using special semantic hashing of the application ontology. This hashing allows us to build unique fingerprints of individual operators, store them in the generated firmware, and use them to check the compatibility of firmware with the current version of $\mathcal{D}$.

The proposed hashing algorithm is implemented in SciVi along with the new improved version of embedded task ontologies representation format (so-called EON, stands for Embedded ONtology [25]) to drive the Edge Computing process. The new version of EON (EON 2.0) allows efficient and straightforward encoding of operators' instances with their fingerprints. We tested the proposed improvements of our ODEC implementation by creating a custom configurable hardware control panel to automate the debugging of experimental scenes in virtual reality (VR).

In this paper, we present an improvement of our ontology-driven Edge Computing implementation. The following key results can be highlighted:

1. The semantic hashing algorithm for application ontologies to build unique 16-bit fingerprints of data processing operators described by these ontologies.

2. The improved representation format to store task ontologies in the memory of resource-constrained edge devices.
3. The approach to tackling the problem of edge devices' "compatibility uncertainty" within the ontology-driven Edge Computing paradigm.

2 Related Work

Reducing the different types of uncertainties plays a pivotal role not only in optimization and decision-making processes but also in solving a variety of real-world problems of software development/validation, including the context of programmable microelectronic devices within the IoT ecosystem [14]. In this paper, we focus on some kind of uncertainties that occurs when the ontology is being changed while the corresponding generated ontology-driven application is in operation. We consider this specific kind of uncertainties as a variation of implementation uncertainties [21].

F. Scioscia and M. Ruta are among the pioneers, who proposed blending Semantic Web and IoT technologies to build so-called SWoT [29,22]. This vision was first introduced to tackle the issues of "data management in pervasive environments" [29] and further extended to address the "challenges associated with standardization, interoperability, discovery, security, and description of IoT resources and their corresponding data" [18]. The most valuable contributions to SWoT are systematically reviewed by A. Rhayem et al. [18]. Along with this review, F. Qaswar et al. fulfilled an analysis of the most interesting cases when ontologies were been applied in IoT [17]. This analysis shows up, that ontologies are nowadays intensively used in IoT design and development to tackle interoperability, integration, and privacy issues.

One of the possible directions of SWoT evolution is ODEC [25], the vision of using ontologies to describe the entire functioning process of IoT devices and corresponding data flows, which allows declaratively specified management of Edge Computing. The pioneering works in ODEC are related to describing edge device capabilities by ontologies [27]; ontology-driven edge device virtualization [28]; using ontologies to retrieve devices, infer their interoperability, and select their operation mode [5]; development of embedded ontology-based expert systems to detect anomalies within IoT and robotics [30]; model-driven middleware generation for semantic integration of devices within Ubiquitous Computing environment [1]. Our contribution to ODEC is the development of a full-fledged ontology reasoner that is embedded into edge devices and capable of executing functions described by task ontologies [25].

One of the vast problems of ODEC is adequate ontology representation suitable for edge devices [32]. The most popular standard formats for storing ontologies are OWL[4] and RDF[5], but both of them are too verbose and thereby unacceptable for most resource-constrained edge devices. To tackle this problem,

[4] https://www.w3.org/OWL/
[5] https://www.w3.org/RDF/

several compression algorithms are proposed, which aim to reduce syntactic, semantic, and structural redundancies in the representation of ontologies [12,8]. As shown by M. A. Hernández-Illera et al., exploiting structural redundancies is the most promising direction in terms of compression ratio [8]. The results of different state-of-the-art approaches to reducing structural redundancies are reported independently by M. Röder et al. and T. Sultana et al. M. Röder et al. state that the best compression can be achieved by so-called k^2-trees [20], while T. Sultana et al. propose a grammar-based knowledge graph compression algorithm that outperforms all the most popular ontology compression techniques [33]. However, all the above-mentioned approaches deal with regular desktop-based ontology reasoners and cannot be efficiently ported to edge devices due to limitations of RAM capacity and CPU frequency.

X. Su et al. [32] and K. Sahlmann et al. [27] propose promising approaches to bridge the gap between knowledge graph representation requirements and edge devices capabilities, but none of them ensures the fitting of complete task ontologies into the edge device RAM for the full-fledged reasoning. Our previous contribution was a mix of reducing structural and semantic redundancies to achieve an extreme compression ratio of task ontologies specifically for ODEC use cases [25]. We proposed a special format called EON to represent embedded ontologies. But applying this format to practical ODEC use cases revealed some limitations, which we overcome in its next version, EON 2.0, described in more detail in Section 4.

To make ODEC accessible not only for programmers and knowledge engineers but also for casual users without special IT hard skills, high-level management tools should be provided. In this regard, an emerging trend is the spread of so-called low-code development platforms, which allow composing software in visual editors without writing source code [16]. This approach can be efficiently combined with ontology-driven software development [13]. The toolset for the visual editor can be automatically generated according to the application ontology that describes available software building blocks. Then, the visual software model composed by the user can be converted to task ontology. Finally, the result software product can be automatically generated with a help of a semantic reasoner that processes this task ontology [4].

To remedy the "compatibility uncertainty" between the application ontology and the task ontologies, so-called semantic hashing can be utilized to efficiently track the relevant changes in the corresponding ontologies. Semantic hashing is an approach of encoding specific documents (for example, ontologies) in compact binary vectors (hash codes) to allow efficient and effective similarity search [7]. According to the specification of EON, codes to operate with should be only 16 bits long to fit in the RAM of target edge devices [25]. So, we decided to use Pearson hashing [15] that can be easily adapted to the hash values of any length starting with 8 bits. With a predefined lookup table [15], this hashing algorithm gives good results in terms of collision avoidance. To fully prevent collisions, additional MD5 hashing [19] is proposed (see Section 3).

3 Semantic Hashing to Reduce Uncertainties

As mentioned above, in this paper we discuss the so-called "compatibility uncertainty".

3.1 Formal Model of Operator

The SciVi platform is based on a microservice architecture, where each microservice corresponds to a data processing operator. We formalize the operator as

$$\mathit{\Delta} : \{I, S\} \to O, \tag{1}$$

where $I = \{I_k | k = \overline{1, |I|}\}$ is a set of typed inputs, $S = \{S_l | l = \overline{1, |S|}\}$ is a set of typed parameters (also denoted as settings), $O = \{O_t | t = \overline{1, |O|}\}$ is a set of typed outputs of the $\mathit{\Delta}$ operator [4]. Let the set of available types be denoted as Q.

Theorem 1. *Any operator that adheres to (1) can be described by the $\mathcal{D}_\Delta$ lightweight application ontology.*

Proof. According to [6], $\mathcal{D}_\Delta$ ontology can be expressed as $\mathcal{D}_\Delta = \{T, R, A\}$, where T is a thesaurus of concepts, R is a set of relationships between concepts from T, and A is a set of axioms related to the elements of T and R.

By the assumption, $\mathcal{D}_\Delta$ is a lightweight ontology, which means $A = \emptyset$. Let us identify the basic categories of elements of T:

1. *Input* (element of I) – independent variable within an operator.
2. *Setting* (element of S) – constant within an operator.
3. *Output* (element of O) – dependent variable within an operator.
4. *Operator* (element $\mathit{\Delta}$ of formula (1)) – transformation that maps the values of independent variables (inputs) and constants (settings) to the values of dependent variables (outputs).
5. *Type* (element of Q) – a concept that defines the set of values that an operator's input, setting, and output can take, as well as the allowed operations on these values.

For any particular $\mathit{\Delta}$ operator, the T set can be composed as

$$\begin{aligned} T = \{&Operator, Input, Setting, Output, Type, \mathit{\Delta}, \\ &I_1, I_2, \ldots, I_{|I|}, S_1, S_2, \ldots, S_{|S|}, \\ &O_1, O_2, \ldots, O_{|O|}, Q_1, Q_2, \ldots, Q_{|I|+|S|+|O|}\}. \end{aligned}$$

Let us use two types of relationships to build R:

1. *is_a* – paradigmatic relationship "subclass-class".
2. *has* – paradigmatic relationship "class-property".

Let us use description logic attributive language with complement, so-called $\mathcal{ALC}$ [2], to formulate a terminology component (so-called TBox):

$$
\begin{aligned}
&I_k \sqsubseteq Input, k = \overline{1, |I|},\\
&S_l \sqsubseteq Setting, l = \overline{1, |S|},\\
&O_t \sqsubseteq Output, t = \overline{1, |O|},\\
&Q_p \sqsubseteq Type, p = \overline{1, |I| + |S| + |O|},\\
&I_k \sqsubseteq Q_{p(k)},\\
&S_l \sqsubseteq Q_{p(|I|+l)},\\
&O_t \sqsubseteq Q_{p(|I|+|S|+t)},\\
&\Delta \equiv Operator \sqcap \exists has.I_1 \sqcap \ldots \sqcap \exists has.I_{|I|}\\
&\qquad \sqcap \exists has.S_1 \sqcap \ldots \sqcap \exists has.S_{|S|}\\
&\qquad \sqcap \exists has.O_1 \sqcap \ldots \sqcap \exists has.O_{|O|}.
\end{aligned}
$$

Here $p(x)$ is a function that maps indices of elements of the I, S, and O sets to the indices of corresponding types from the Q set. This TBox corresponds to the knowledge graph shown in Fig. 1.

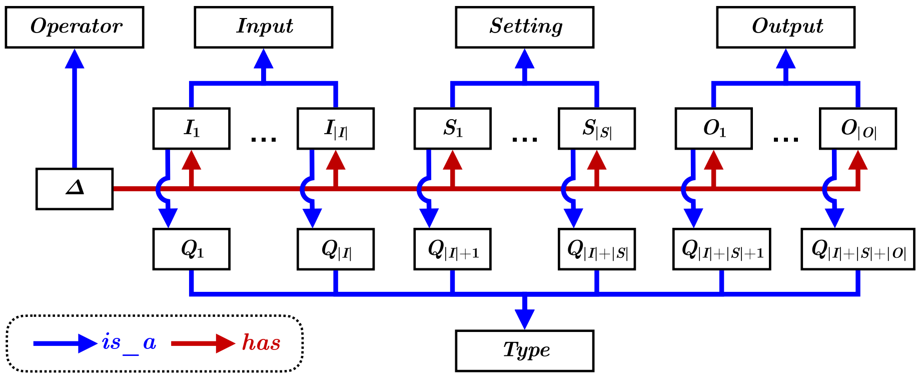

Fig. 1: Generalized description of Δ operator in the form of knowledge graph

From the above, $\mathcal{D}_\Delta = \{T, R\}$ is the lightweight ontology that precisely describes the Δ operator. This is an application ontology because it was been engineered for specific use within an ontology-driven data processing platform. □

To describe complex types, for example, arrays, the *base_type* relationship is added to R to represent type specifications.

Let us assume that we have n operators which relate to some specific class of data processing problems. Then, according to Theorem 1, this set of operators can be described by the following application ontology:

$$\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_{\Delta_i}. \qquad (2)$$

3.2 Unique Identifier of Operator

According to the EON format specification, operators should have unique 16-bit identifiers to be encoded into the concise semantically compressed ontology representation [25]. Previously, we used just a regular numbering of operators according to the order of their appearance in the $\mathcal{D}$ ontology (i.e. the i index in Equation 2). However, this approach leads to "compatibility uncertainty" as mentioned above because having a particular identifier of the operator we cannot check whether it corresponds to the current version of $\mathcal{D}$. To remedy this uncertainty, we propose calculating the operator's identifier as a hash of the operator's structure to be able to detect possible ontology changes. This hash should preserve the operator's semantics within the context of the operator's execution.

Only those changes of $\mathcal{D}$ break the operator's compatibility, which affect the operator's execution process, so the hash function should be invariant to any irrelevant changes. In the compilers theory, the function is primarily identified by its name and type signature [10]. Similarly, an operator adhering to (1) can be identified by its name (denoted as name($\varDelta$)) and the names of types of its inputs, settings, and outputs (denoted as name(Q_p), $p = \overline{1, |I| + |S| + |O|}$). It must be noted, that in case of complex types (for example, enumerations or structures) or hierarchical types, name(Q_p) should produce a concatenation of all the names of types in hierarchical order, with the ">" sign as a delimiter. Assuming that operator and all the types have string names, the following equation can be used to build the operator's string identifier:

$$\begin{aligned}\sigma(\varDelta) = \mathrm{name}(\varDelta) + \text{“@}\mathtt{I}\text{”} + \sum_{i=1}^{|I|} \mathrm{name}(Q_i) + \text{“@}\mathtt{S}\text{”} + \sum_{i=|I|+1}^{|I|+|S|} \mathrm{name}(Q_i) \\ + \text{“@}\mathtt{O}\text{”} + \sum_{i=|I|+|S|+1}^{|I|+|S|+|O|} \mathrm{name}(Q_i),\end{aligned} \tag{3}$$

where sums denote string concatenation of operands sorted in lexicographical order with the ":" sign as a delimiter.

To get the operator's unique identifier, we propose calculating a 16-bit Pearson hash [15] of $\sigma(\varDelta)$:

$$\pi(\varDelta) = \mathrm{Pearson}(\sigma(\varDelta)). \tag{4}$$

Usually, the operators are described as a taxonomy. In this case, Q_p is assembled by the reasoner taking into account the operators' inheritance. Fig. 2 demonstrates an example of application ontology that describes two operators handling general input-output pins of a microcontroller within an edge device.

Operators *Input Pin* and *Output Pin* inherit from their parent operator *GPIO* (General-Purpose Input-Output) the *Pin Number* that is the setting of complex type (enumeration of numbers). In addition, the *Input Pin* operator introduces a Boolean output *Pin In* and the *Output Pin* operator introduces a

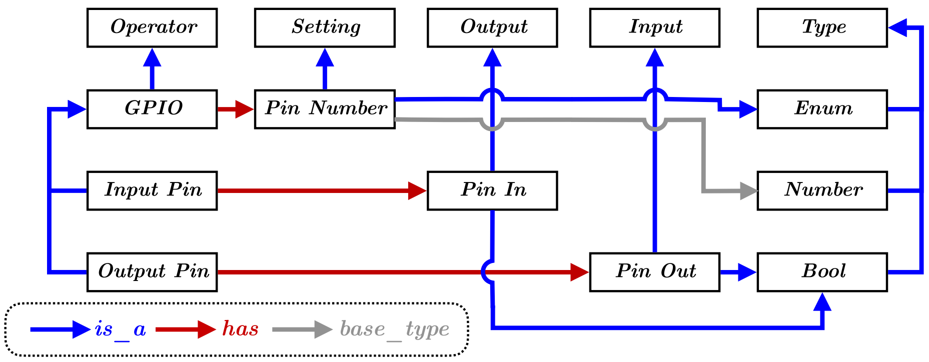

Fig. 2: Fragment of application ontology describing ODEC operators

Boolean input *Pin Out*. The instances of these operators provide the ability to read and write logical values to the microcontroller's pins and thereby control different peripheral electronic components of custom edge devices. For these operators, the following signatures are calculated according to the described algorithm:

$$\sigma(\mathit{Input\ Pin}) = \text{``}\texttt{Input Pin@SEnum>Number@OBool}\text{''};$$
$$\sigma(\mathit{Output\ Pin}) = \text{``}\texttt{Output Pin@IBool@SEnum>Number}\text{''}.$$

The actual hash values depend on the particular Pearson's lookup table. In our case:

$$\pi(\mathit{Input\ Pin}) = 19218;$$
$$\pi(\mathit{Output\ Pin}) = 57372.$$

The described hashing algorithm is invariant to changes of the names and the order of inputs, settings, and outputs, as well as possible taxonomy changes of the operator as long as the taxonomy changes do not lead to changing the set of operators' inputs, outputs, or settings due to inheritance. Of course, the changes of other operators' descriptions do not affect the hash of the current one. Only the change of the current operator's signature that directly affects this operator's execution process, will alternate the hash result.

The obvious problem of short Pearson hashing is the relatively high probability of collisions, but the total number of operators used in a single class of data processing problems is pretty low. On average, the SciVi knowledge base currently contains 40 operators per problem class (although this number may grow with further SciVi development), which is ca. 1600 times less than the capacity of a 16-bit Pearson hash. In this case, the collision probability is about 1.2%. To get the collision probability higher than 50%, more than 300 operators are needed. However, to detect possible collisions, we use an additional check that is described in Section 3.3.

3.3 Embedded Reasoner Compatibility Check

As described in [25], the ODEC implies the following working principle. Let us assume, $\mathcal{D}$ is an ontology that describes a set of n operators as in Equation

(2). First, the user chooses a subset $\varPhi = \{\varDelta_1, \varDelta_2, \ldots, \varDelta_m\}$ of operators from $\mathcal{D}$ to be included in the embedded reasoner, $m \leq n$. Usually, $m \ll n$ due to the program memory restrictions of edge devices. Next, the firmware for the edge device is generated. This firmware contains the embedded reasoner with a special module of functions (FM) that incorporates implementations of operators from $\varPhi$. After this firmware is installed on the target edge device, the user can compose task ontologies, describing the particular data collection, data processing, and communication tasks the edge device should execute. For this, SciVi provides high-level visual programming tools based on data flow diagrams. These data flow diagrams are automatically converted to task ontologies referring to the operators from $\varPhi$ [25,4,24]. These task ontologies are uploaded to the edge device and processed by the reasoner that triggers the necessary functions of FM, which correspond to the operators. The triggering order, input parameters, and related settings of functions are inferred from the task ontology.

To call the appropriate function for the corresponding operator, a special lookup table is generated that maps the operators' identifiers to the addresses of corresponding functions in program memory. To check if the particular reasoner installed on the particular edge device is compatible with the current ontology $\mathcal{D}$, the set of lookup table identifiers $\varPi = \{\pi(\varDelta_1), \pi(\varDelta_2), \ldots, \pi(\varDelta_m)\}$ (see Equations (3) and (4)) is sent from the reasoner to the SciVi server and matched with the identifiers of all the operators described by $\mathcal{D}$. This requires $m \cdot n$ comparisons of 16-bit integer values. The total number of comparisons is fairly small (as mentioned above, the average value for n is 40, and $m \leq n$), moreover, the comparisons are performed on the server side, not on the edge device, so this operation is very fast despite its quadratic complexity.

To defeat potential collisions, an additional check is performed. Along with 16-bit Pearson hashes of operators' signatures $\sigma(\varDelta_1), \sigma(\varDelta_2), \ldots, \sigma(\varDelta_m)$ (see Equation (3)), a 128-bit MD5 hash [19] of the following value is calculated:

$$\xi = \sum_{i=1}^{m} \sigma(\varDelta_i), \quad \mu = \mathrm{MD5}(\xi), \tag{5}$$

where sum denotes string concatenation of operands sorted in lexicographical order with the ";" sign as a delimiter.

This MD5 value is stored in the reasoner's source code along with the lookup table and sent to the SciVi server together with the lookup table identifiers.

SciVi server first searches the elements of $\varPi$ among the operators' identifiers from $\mathcal{D}$. If at least one $\pi(\varDelta_i), i = \overline{1,m}$ has no corresponding operator described by $\mathcal{D}$, the reasoner is treated as incompatible. Otherwise, the $\varPhi$ set is reconstructed containing operators with the identifiers from $\varPi$. Then, to make sure there were no collisions, the received MD5 value μ is compared with the MD5 calculated for the signatures of operators from $\varPhi$. If these hashes are not equal, the reasoner is treated as incompatible (collision of Pearson hashes took place). Otherwise, the reasoner is compatible with $\mathcal{D}$.

The incompatible reasoner should be updated, while the compatible one can be used as is for subsequent ODEC process.

4 EON 2.0 Ontology Representation Format

In the EON 1.0 format [25], it was not efficient enough to represent different instances of the same operator. However, the edge devices like custom control panels, which are supposed to handle many similar buttons, often require executing the same operator many times during a single data processing iteration. Herewith, each call of the operator has its own settings and each operator's execution result is transmitted to its own branch of a data processing pipeline.

To improve the handling of operators' instances, we have upgraded EON to version 2.0 by slightly altering the memory layout of compressed task ontology. The EON 2.0 blob structure is as follows:

`| DFChunkLen | DFChunk | SChunkLen | SChunk | IChunk |`

`DFChunkLen` (stands for Data Flow Chunk Length, 1 byte) is a number of 3-byte elements in `DFChunk`. `DFChunk` (stands for Data Flow Chunk, 3 times `DFChunkLen` bytes) is a chunk containing the sequence of data transmission links of the task ontology formed like this:

`| OpInstA | Output | Input | OpInstB |`

`OpInstA` (stands for Operator Instance A, 1 byte) and `OpInstB` (stands for Operator Instance B, 1 byte) are task ontology identifiers of the operators' instances, whereby the output of `OpInstA` is linked to the input of `OpInstB`. `Output` (4 bits) is an index of output of `OpInstA`. `Input` (4 bits) is an index of input of `OpInstB`.

`SChunkLen` (stands for Settings Chunk Length, 2 bytes) is a length in bytes of `SChunk`. `SChunk` (stands for Settings Chunk, `SChunkLen` bytes) is a chunk containing the sequence of settings formed like this:

`| OpInst | Setting | Type | Value |`

`OpInst` (stands for Operator Instance, 1 byte) is a task ontology identifier of the operator's instance that has the corresponding setting. `Setting` (4 bits) is an index of setting of `OpInst`. `Type` (4 bits) is an internal type identifier of setting (currently, the following types are supported: signed and unsigned 8-bit, 16-bit, 32-bit integers, 32-bit float, and string). `Value` is an encoded setting's value (length depends on the type, strings are null-terminated).

`IChunk` (stands for Instances Chunk, length is not stored) is a chunk containing the sequence of operators' instances formed like this:

`| UIDOp | OpInst1 | OpInst2 | ... | 0x0 |`

`UIDOp` (stands for Unique Identifier of Operator, 2 bytes) is a unique identifier of the operator (calculated as described in Section 3.2), whose prototype is described in the application ontology and instances are described in the task ontology. `OpInst1, OpInst2, ...` (stand for Operator Instance, each is 1 byte) are task ontology identifiers of corresponding operator's instances. The `0x0` (1 byte) is a zero-byte terminating the list of instances.

The described format allows a very concise representation of task ontologies, including the instancing of operators. As evaluated in [25], EON-formatted ontologies require about 800 times less storage space than OWL-formatted ones. The new EON memory layout introduced in this paper does not decrease the

EON efficiency and allows for explicit encoding of operators' instances. Thereby, it increases the area of EON applications within ODEC.

5 Discussion and Conclusion

WWe introduce the above-mentioned approach to remedy ODEC uncertainty and improvements of EON format within the research project "Text processing in L1 and L2: Experimental study with eye-tracking, visual analytics and virtual reality technologies"[6] (supported by the research grant No. ID92566385 from Saint Petersburg State University). One of the goals of this project is to study the peculiarities of the reading process of humans within a VR environment using eye tracking to estimate the differences in information perception in virtual and physical reality. For this, VR scenes with different texts are sequentially shown to informants. Their eye gaze tracks are sampled with a tracker embedded into the VR head-mounted display (HMD) and transmitted to SciVi for subsequent analysis and storage [23].

Under normal circumstances, the experiment requires a director who uses the SciVi Web interface to switch the scenes, start/stop eye gaze recording, and tune the analytics pipeline settings if needed. However, during the debugging of the analytics pipeline, it is more efficient when the pipeline developer plays both the role of an informant and a director simultaneously, without engaging additional people and wasting their time. However, the problem is that the informant has their eyes covered with the HMD, so it is nearly impossible to use a SciVi Web interface at that time. Taking the HMD on and off is not an option, because for the correct eye tracking, calibration is needed after taking the HMD on, and continuous re-calibrations would be very tedious. To solve this problem, custom reconfigurable edge-device-based controllers can be used, whose hardware control elements (buttons, potentiometers, etc.) are mapped to the particular pipeline settings, which are debugged. In this use case, ODEC is very efficient, because it allows both hardware and software reconfiguration and remapping without reprogramming and reflashing. Utilizing the built-in VR controllers instead of custom edge devices will not be as efficient, because it would require rebuilding the VR scene each time when the controllers' role should change. Moreover, the limited set of buttons on the built-in controller does not provide as much reconfiguration freedom as a custom device.

We successfully adopted ODEC in our VR-based research project. The proposed method to remedy the ODEC "compatibility uncertainty" through the semantic hashing of application ontologies allowed the efficient reuse of edge devices powered by ODEC in a situation of continuous enrichment of the SciVi repository with new operators and ontologies. The implementation of the proposed compatibility checking reduces the amount of firmware generation and device reflashing cycles. It works fast even though formally it has quadratic complexity. Calculating the semantic hash of a particular operator described by

[6] L1 and L2 stand for the native and foreign languages respectively.

an application ontology with 328 nodes and 845 relationships takes 2.15 ms on average (on a MacBook Pro 2.3 GHz 8-Core Intel Core i9 CPU, 16 Gb RAM). This enables the real-time processing of compatibility requests. The memory footprint of the semantic hash is fairly small: it requires appending just 16 bytes of MD5 hash sum to the device firmware, while the hashes of individual operators are stored in the identifiers of a functions module lookup table and do not require extra storage space.

The average time of updating our ODEC-powered edge device (based on the ESP8266 microcontroller) in different development cases is shown in Table 1. The operators used in the ODEC device are denoted as "related operators", the unused ones are denoted as "unrelated operators". In each development case, the behavior of the ODEC device is changed by uploading a new task ontology. If the task ontology is treated as compatible with the reasoner installed on this device, the only action needed to update the device behavior is transmitting this ontology from the computer, which takes 16 ms on average (just a single tick of ODEC device processing loop, which is set up to the rate of 60 Hz in our implementation). Otherwise, if the task ontology is treated as incompatible, its uploading should be preceded by a firmware regeneration, device reflashing, reboot, and WiFi reconnection, which takes about 30 s on average in total.

Table 1: Performance comparison of ODEC device updating

Development case: type of changes in $\mathcal{D}$	No changes	Changes of related operators' structure	Changes of related operators' parameters naming	Changes of unrelated operators	Average
Conventional versioning	16 ms	30000 ms	30000 ms	30000 ms	22504 ms
Semantic hashing	16 ms	30000 ms	16 ms	16 ms	7512 ms

As seen in the table, the introduced semantic hashing increases the performance of updating the ODEC device 3 times on average. The software implementation of operators' descriptions semantic hasher and EON 2.0 format encoder are available in SciVi open source repository: `https://github.com/scivi-tools/scivi.web/blob/master/onto/hasher.py`.

For the next step of ODEC development, we plan to design an ontology-driven bus for joining hardware components of edge devices on plug-and-play principles. This will further increase the efficiency of the device reconfiguration process, which is crucial in the cases of creating custom hardware user interfaces within the IoT ecosystem.

References

1. Abdulrab, H., Babkin, E., Kozyrev, O.: Semantically Enriched Integration Framework for Ubiquitous Computing Environment. In: Babkin, E. (ed.) Ubiquitous Computing, chap. 9, pp. 177–196. IntechOpen (2011). https://doi.org/10.5772/15262

2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
3. Calderon, M., Delgadillo, S., Garcia-Macias, A.: A More Human-Centric Internet of Things with Temporal and Spatial Context. Procedia Computer Science **83**, 553–559 (2016). https://doi.org/10.1016/j.procs.2016.04.263
4. Chuprina, S., Ryabinin, K., Koznov, D., Matkin, K.: Ontology-Driven Visual Analytics Software Development. Programming and Computer Software **48**(3), 208–214 (2022). https://doi.org/10.1134/S0361768822030033
5. Dibowski, H., Kabitzsch, K.: Ontology-Based Device Descriptions and Device Repository for Building Automation Devices. EURASIP Journal on Embedded Systems (2011). https://doi.org/10.1155/2011/623461
6. Golitsyna, O.L., Maksimov, N.V., Okropishina, O.V., Strogonov, V.I.: The Ontological Approach to the Identification of Information in Tasks of Document Retrieval. Automatic Documentation and Mathematical Linguistics **46**, 125–132 (2012). https://doi.org/10.3103/S0005105512030028
7. Hansen, C., Hansen, C., Simonsen, J.G., Alstrup, S., Lioma, C.: Unsupervised Multi-Index Semantic Hashing. In: Proceedings of the Web Conference 2021. pp. 2879–2889 (2021). https://doi.org/10.1145/3442381.3450014
8. Hernández-Illera, A., Martínez-Prieto, M.A., Fernández, J.D.: RDF-TR: Exploiting Structural Redundancies to Boost RDF Compression. Information Sciences **508**, 234–259 (2020). https://doi.org/10.1016/j.ins.2019.08.081
9. Jara, A.J., Olivieri, A.C., Bocchi, Y., Jung, M., Kastner, W., Skarmeta, A.F.: Semantic Web of Things: An Analysis of the Application Semantics for the IoT Moving towards the IoT Convergence. International Journal of Web and Grid Services **10**(2/3), 244–272 (2014). https://doi.org/10.1504/IJWGS.2014.060260
10. Kernighan, B.W., Ritchie, D.M.: C Programming Language. Prentice-Hall (1988)
11. Mao, S., Khalifa, Y., Zhang, Z., Shu, K., Suri, A., Bouzid, Z., Sejdic, E.: Chapter 14 - Ubiquitous Computing. In: Godfrey, A., Stuart, S. (eds.) Digital Health, pp. 211–230. Academic Press (2021). https://doi.org/10.1016/B978-0-12-818914-6.00002-8
12. Martínez-Prieto, M.A., Fernández, J.D., Hernández-Illera, A., Gutiérrez, C.: RDF Compression, pp. 1–11. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-63962-8_62-1
13. Pan, J.Z., Staab, S., Aßmann, U., Ebert, J., Zhao, Y. (eds.): Ontology-Driven Software Development. Springer (2013). https://doi.org/10.1007/978-3-642-31226-7
14. Patel, A., Debnath, N.C., Bhushan, B. (eds.): Semantic Web Technologies: Research and Applications (1st ed.). CRC Press (2022). https://doi.org/10.1201/9781003309420
15. Pearson, P.K.: Fast Hashing of Variable-Length Text Strings. Communications of the ACM **33**(6), 677–680 (1990). https://doi.org/10.1145/78973.78978
16. Pinho, D., Aguiar, A., Amaral, V.: What about the usability in low-code platforms? A systematic literature review. Journal of Computer Languages **74** (2023). https://doi.org/10.1016/j.cola.2022.101185
17. Qaswar, F., Rahmah, M., Raza, M.A., Noraziah, A., Alkazemi, B., Fauziah, Z., Hassan, M.K.A., Sharaf, A.: Applications of Ontology in the Internet of Things: A Systematic Analysis. Electronics **12**(1) (2023). https://doi.org/10.3390/electronics12010111

18. Rhayem, A., Mhiri, M.B.A., Gargouri, F.: Semantic Web Technologies for the Internet of Things: Systematic Literature Review. Internet of Things **11** (2020). https://doi.org/10.1016/j.iot.2020.100206
19. Rivest, R.: The MD5 Message-Digest Algorithm. RFC 1321, RFC Editor (1992). https://doi.org/10.17487/RFC1321
20. Röder, M., Frerk, P., Conrads, F., Ngomo, A.C.N.: Applying Grammar-Based Compression to RDF. Lecture Notes in Computer Science **12731**, 93–108 (2021). https://doi.org/10.1007/978-3-030-77385-4_6
21. Roza, M.: Verification, Validation and Uncertainty Quantification Methods and Techniques (An Overview and their Application within the GM-VV Technical Framework). Science and Technology Organization, NATO (2014)
22. Ruta, M., Scioscia, F., Di Sciascio, E.: Enabling the Semantic Web of Things: Framework and Architecture. In: 2012 IEEE Sixth International Conference on Semantic Computing. pp. 345–347 (2012). https://doi.org/10.1109/ICSC.2012.42
23. Ryabinin, K., Belousov, K.: Visual Analytics of Gaze Tracks in Virtual Reality Environment. Scientific Visualization **13**(2), 50–66 (2021). https://doi.org/10.26583/sv.13.2.04
24. Ryabinin, K., Chumakov, R., Belousov, K., Kolesnik, M.: Ontology-Driven Visual Analytics Platform for Semantic Data Mining and Fuzzy Classification. Frontiers in Artificial Intelligence and Applications **358**, 1–7 (2022). https://doi.org/10.3233/FAIA220363
25. Ryabinin, K., Chuprina, S.: Ontology-Driven Edge Computing. Lecture Notes in Computer Science **12143**, 312–325 (2020). https://doi.org/10.1007/978-3-030-50436-6_23
26. Ryabinin, K., Chuprina, S., Labutin, I.: Tackling IoT Interoperability Problems with Ontology-Driven Smart Approach. Lecture Notes in Networks and Systems **342**, 77–91 (2021). https://doi.org/10.1007/978-3-030-89477-1_9
27. Sahlmann, K., Scheffler, T., Schnor, B.: Ontology-driven Device Descriptions for IoT Network Management. In: 2018 Global Internet of Things Summit (GIoTS) (2018). https://doi.org/10.1109/GIOTS.2018.8534569
28. Sahlmann, K., Schwotzer, T.: Ontology-Based Virtual IoT Devices for Edge Computing. In: Proceedings of the 8th International Conference on the Internet of Things (2018). https://doi.org/10.1145/3277593.3277597
29. Scioscia, F., Ruta, M.: Building a Semantic Web of Things: Issues and Perspectives in Information Compression. In: 2009 IEEE International Conference on Semantic Computing. pp. 589–594 (2009). https://doi.org/10.1109/ICSC.2009.75
30. Seitz, C., Schönfelder, R.: Rule-based OWL Reasoning for specific Embedded Devices. Lecture Notes in Computer Science **7032**, 237–252 (2011). https://doi.org/{10.1007/978-3-642-25093-4_16}
31. Slimani, T.: Ontology Development: A Comparing Study on Tools, Languages and Formalisms. Indian Journal of Science and Technology **8**(24), 1–12 (2015). https://doi.org/10.17485/ijst/2015/v8i34/54249
32. Su, X., Riekki, J., Haverinen, J.: Entity Notation: Enabling Knowledge Representations for Resource-Constrained Sensors. Personal and Ubiquitous Computing **16**, 819–834 (2012). https://doi.org/10.1007/s00779-011-0453-6
33. Sultana, T., Lee, Y.K.: gRDF: An Efficient Compressor with Reduced Structural Regularities That Utilizes gRePair. Sensors **22**(7) (2022). https://doi.org/10.3390/s22072545