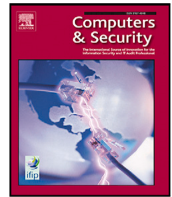
Computers & Security 150 (2025) 104282

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose



Full Length Article

Open Digital Rights Enforcement framework (ODRE): From descriptive to enforceable policies

Andrea Cimmino *, Juan Cano-Benito, Raúl García-Castro

Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain

ARTICLE INFO

Keywords:
Open digital rights language
Privacy policies
ODRL enforcement

ABSTRACT

From centralised platforms to decentralised ecosystems, like Data Spaces, sharing data has become a paramount challenge. For this reason, the definition of data usage policies has become crucial in these domains, highlighting the necessity of effective policy enforcement mechanisms. The Open Digital Rights Language (ODRL) is a W3C standard ontology designed to describe data usage policies, however, it lacks built-in enforcement capabilities, limiting its practical application. This paper introduces the Open Digital Rights Enforcement (ODRE) framework, whose goal is to provide ODRL with enforcement capabilities. The ODRE framework proposes a novel approach to express ODRL policies that integrates the descriptive ontology terms of ODRL with other languages that allow behaviour specification, such as dynamic data handling or function evaluation. The framework includes an enforcement algorithm for ODRL policies and two open-source implementations in Python and Java. The ODRE framework is also designed to support future extensions of ODRL to specific domain scenarios. In addition, current limitations of ODRE, ODRL, and current challenges are reported. Finally, to demonstrate the enforcement capabilities of the implementations, their performance, and their extensibility features, several experiments have been carried out with positive results.

1. Introduction

In recent decades, proposals for sharing and consuming data from a set of wide-ranging domains among different actors have evolved; from large centralised data platforms to decentralised ones (Kelbert and Pretschner, 2012). Many of these proposals follow a policy-based approach (Henze et al., 2014); such as European Data Spaces (Akaichi et al., 2024). On the one hand, they rely on a vocabulary that is used to unequivocally specify conditions and terms set by the data owner under which such data can be used by a third party, and, on the other hand, they rely on a software mechanism that must verify whether these conditions are met, that is, an enforcement procedure implementation. Some of these proposals are standards, such as XACML (Standard, 2013) or ODRL (Monegraph and Villata, 2018). However, not all of them provide enforcement, and some are only tailored to describe usage policies without the enforcement capabilities, like ODRL.

The Open Digital Rights Language (ODRL) is a W3C standard ontology that provides the vocabulary to describe policies in decentralised ecosystems, such as the Web, promoting data sovereignty. The ODRL vocabulary is widespread in numerous data sharing contexts as a mechanism to describe data licences (Rodríguez-Doncel et al., 2013), for example, extending ODRL to support GDPR privacy policies (De Vos et al., 2019); which entails specifying certain abstract conditions to reuse data that cannot be enforced computationally. However, this standard goes beyond and allows describing more fine-grained enforceable conditions to use data (for example, data can be accessed only during the night), and actions are allowed to be performed on such data if conditions are met (Steyskal and Polleres, 2014). Due to this reason, this standard is a candidate in many data-centric initiatives, such as European Data Spaces (Eitel et al., 2021).

However, the ODRL standard has only promoted a vocabulary to define its policies, but it lacks an enforcement specification. Therefore, policies cannot be evaluated to allow or deny data usage or invoke actions that must be taken if the constraints of these policies are met whether a third party actor requests the usage of a data resource (Kebede et al., 2021). Due to this reason, from the practical point of view, ODRL cannot be used to ensure the usage of data by third parties in the terms described by a data owner or to take the actions set by them (Akaichi and Kirrane, 2022).

In addition, as ODRL policies are now endowed with this standard, they lack privacy in certain scenarios since they require stone-written conditions in its policies leading to privacy leak information, and, also, they lack a dynamic data handling mechanism which can also lead to privacy leaks. For instance, a policy based on geofencing that allows

* Corresponding author.
E-mail addresses: andreajesus.cimmino@upm.es (A. Cimmino), juan.cano@upm.es (J. Cano-Benito), r.garcia@upm.es (R. García-Castro).

https://doi.org/10.1016/j.cose.2024.104282
Received 16 September 2024; Received in revised form 28 November 2024; Accepted 12 December 2024
Available online 19 December 2024
0167-4048/© 2024 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

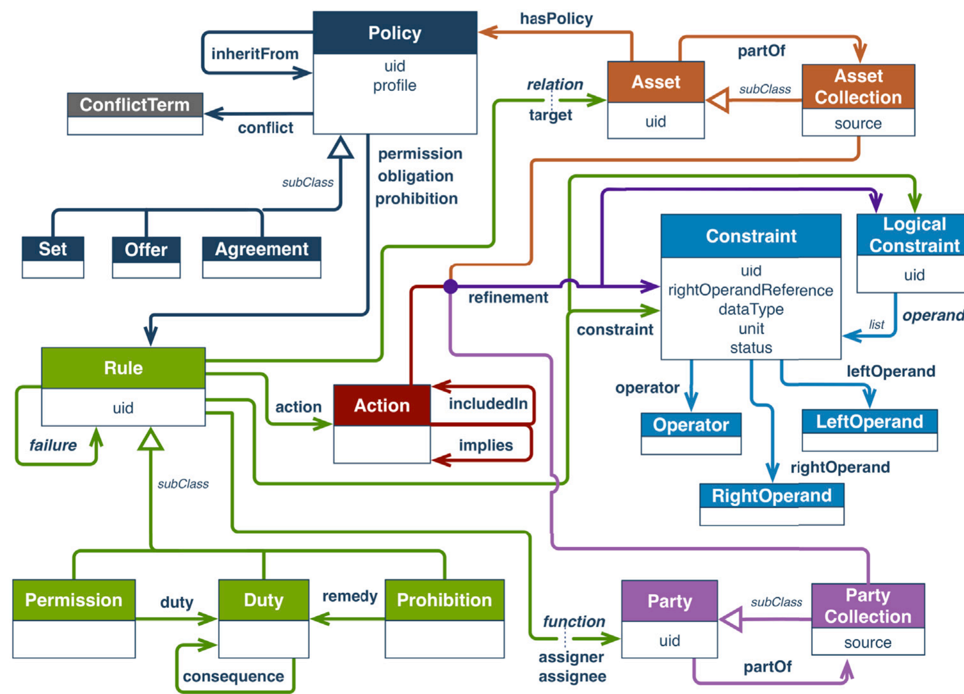


Fig. 1. ODRL vocabulary.
Source: Extracted from (Monegraph and Villata, 2018)

reading certain data to an actor if he/she is inside a place would show explicitly the geometry of such place (privacy leak) and will have no mechanism to handle the ever-changing GPS position of the actor (dynamic data) and, therefore, the GPS data will need to be continuously written also explicitly in the policy (privacy leak).

In this article, the Open Digital Rights Enforcement (ODRE) framework is presented; which aims to extend the ODRL standard encompassing its ontology with enforcing features so policies described the ODRL vocabulary can be enforced. First, the article presents a novel approach on how these policies should be written so that they are not only descriptive, but also enforceable. This approach finds its roots in the way that behaviour and representation are addressed by the different types of grammar depending on the Chomsky classification (Chomsky, 2014) and how Model (behaviour) and View (representation) are separated in the template engines (Parr, 2004). The bottom line idea is to consider that ODRL policies are written with a descriptive language, and rely on adding one or more abstract languages to allow express behaviour, such as dynamic data values injection (data interpolation) or functions evaluation. To this end, a comprehensive analysis of the different types of policies that can be written and using different languages is presented, as well as some implementation languages that could be used.

Following this approach, the article then presents an implementation-agnostic algorithm, or procedure, that aligns with the ODRL standard vocabulary to enforce ODRL policies. This algorithm checks that the data usage restrictions described in the policies are met and then, if necessary, invokes related actions described in such policies. In addition, the algorithm is tailored with a mechanism for handling dynamic data (preventing privacy leaks) and counts with the necessary extension points to adapt ODRL policies to domain-specific scenarios. As part of the framework, a small extension of the ODRL has been developed and published[1] to showcase the extensibility feature of the enforcing algorithm.

For instance, the aforementioned geofencing policy requires firstly to extend the ODRL vocabulary to express the actor GPS position as a point and, also, to express the place as a geometry. Secondly, it requires a new functionality for the enforcement that is able to check whether the point is within the geometry. Finally, the policy must be written in a way that does not reveal the geometry or the GPS point, but it can be enforced with these values. Note that the enforcing algorithm and the ODRL vocabulary, or any extension, must be coupled.

Finally, to prove the implementation-agnostic nature of the algorithm, the article reports two implementations published as open source; one based on Python[2] and the other on Java.[3] These implementations have been tested with the same 24 unit tests each, and the time they required to enforce 24 different policies is reported. These tests and their related policies check different features of the implementations, such as their extensibility or implementation of the ODRL vocabulary.

The rest of this article is structured as follows. Section 2 introduces the ODRE framework approach to express ODRL policies so that they can become enforceable; Section 3 presents the ODRE framework algorithm to enforce policies with a running example and reports limitations of ODRE, ODRL and future challenges; Section 4 presents two implementations of the ODRE enforce algorithm along with some experimental results; Section 5 presents a survey of related proposals from the literature; finally, Section 6 presents a discussion about ODRL and its usage in practical scenarios and the conclusions of the article.

2. Classification of ODRL enforceable policies

ODRL is an ontology that provides the vocabulary to describe or express usage policies following the model depicted in Fig. 1. From

[1] https://w3id.org/def/odre-time

[2] https://pypi.org/project/pyodre/
[3] https://github.com/ODRE-Framework/odre-java

now on, the prefix *odrl*: for the ODRL ontology namespace[4] is assumed. These policies may have a set of rules; which may denote permission, duty, or prohibition. Regardless of their type, rules consist of an action and a set of constraints that encode the conditions under which the action must be performed. The constraints have three main descriptive elements: an operator, a left operand, and a right operand. The operands must be either concepts defined in the ontology that represent a certain information that is not explicitly written in the policy (for example, *odrl*:*dateTime*[5] represents the current date with time), or data constants that have a value and a *xsd* datatype[6] (for example, {"@*value*":"2018−01−01","@*type*":"*xsd*:*dateTime*"}). Operators are always concepts defined in the ontology and represent predicates, i.e., functions that always output a Boolean result.

In order to enforce an ODRL policy, first its operands, which are ontology concepts, must be replaced in the policy by a data constant with the information they stand for. In the case of *odrl*:*dateTime*, a constant representing the actual date and time; which will look similar to the one shown above. Then, the operator must be computed taking the left and right operands as input. In the event the operator provides a positive result (true), the action of the policy must be taken; this may involve running some code, like for the operator, or notifying a third party (either a person or a software) who will perform such action.

Note that ODRL policies are expressed in the W3C standard format RDF (Manola et al., 2004) which allows expressing concepts as triplets without any functionality or behaviour, and the ODRL ontology only narrows down the terms that a policy expressed in RDF can contain. As a result, it is necessary to wrap or transform the descriptive terms of the ODRL ontology into another language that can be interpreted and, therefore, that can replace operands for respective constants, evaluate operators, or run actions. In order to achieve this goal, the authors classify the ODRL policies into the following types depending on the ancillary language used.

The policies expressed in RDF with the ODRL ontology will be classified as *∅-Level* and the language used to write them as *descriptive language* (in this case RDF with the terms from the ODRL ontology), and those expressed with an *interpreted language* classified as *A-Level*. The result of interpreting an *A-Level* policy is a *Usage Decision*; that specifies a set of actions that should be performed either as part of the enforcement procedure or by notifying a third party who must perform them. These actions are present in the *Usage Decision* if their related constraints were enforced positively (true).

Note that ∅-Level and A-Level policies must show no differences in terms of lexicon, they both contain terms from the ODRL ontology, i.e., the descriptive language, as shown by Listing 8. However, the interpreted language couples certain terms (highlighted in Listing 8) from the descriptive language (i.e., ontology terms) with certain functionalities and, therefore, is able to interpret them and produce an enforcement result, i.e., Usage Decision. It is worth to mention that the terms that the interpreted language recognises must exist in the descriptive language, that is, the ontology.

```
{"@context": "...",
"@type": "Policy",
"uid": "https://upm.es/policy/1",
"permission": [{
   "target": "https://jsonplaceholder.typicode.
      com/users/1",
   "action": "read",
   "constraint": [{
      "leftOperand": "dateTime",
      "operator": "gt",
      "rightOperand": { "@value": "2018-01-01T00
          :40:30", "@type": "xsd:dateTime" }
   },{
      "leftOperand": "media",
      "operator": "eq",
      "rightOperand": { "@value": "online", "
          @type": "xsd:string" }
   }]
}]}
```

Listing 1: A sample, *∅-Level* or *A-Level*, policy that grants usage if datetime and media are greater or equals, respectively, to a certain value

Under certain circumstances, policies may need to contain certain terms that do not belong to the ODRL ontology and, instead, refer to data variables whose values shall be given by whoever is enforcing the policy; e.g., a third party an actor. These policies, classified as *B1-Level*, require an additional level of expressiveness provided by an additional language that wraps the *A-Level* policies, that is, the *interpolated language*. This language only allows to use terms that refer to data variables.

Note that B1-Level policies, and following levels, introduce differences in terms of lexicon since they need to express terms outside the ODRL ontology and, therefore, these policies are written mixing RDF with the interpolated language. An example of a B1-Level policy is shown by Listing 2. The interpolated language used in the policy, which is highlighted, is Freemarker.[7]

```
{"@context": "...",
"@type": "Policy",
"uid": "https://upm.es/policy/1",
"permission": [{
   "target": "https://jsonplaceholder.typicode.
      com/users/1",
   "action": "read",
   "constraint": [{
      "leftOperand": { "@value": "[=requestToken]
         ", "@type": "xsd:string" },
      "operator": "eq",
      "rightOperand": { "@value": "eyJhbGciO...",
          "@type": "xsd:string" }
    }]
 }]
}
```

Listing 2: A sample *B1-Level* policy that grants usage if data variable (*requesterToken*) is equals to a certain value

Most of the time, the interpolated language counts with more expressiveness than just terms to refer to data variables; providing additional ones for data flow control, iterative statements, or custom procedures that are defined relying only on this language; in this case, the policy is classified as *B2-Level* and the language is known as *templated language*, which is a richer or enhanced *interpolated language*.

Listing 3 shows a B2-Level policy expressed with an extended version of the templated language from Listing 2. Note that this policy contains several procedures concatenated after the procedure called now, which returns current time, and an variable creation using assign. Note that, in this language, control terms are preceded by the token [# whereas data variables are preceded by the token [=.

```
{"@context": "...",
"@type": "Policy",
"uid": "https://upm.es/policy/1",
```

[4] http://www.w3.org/ns/odrl/2/

[5] https://www.w3.org/TR/odrl-vocab/#term-dateTime

[6] https://www.w3.org/TR/xmlschema11-2/

[7] https://freemarker.apache.org/docs/dgui_misc_alternativesyntax.html

```
"permission": [{
   "target": "https://jsonplaceholder.typicode.
       com/users/1",
   "action": "read",
   "constraint": [{
 [#assign timeVar=.now?time?iso("Europe/Rome")?
     replace('\\+.*','', 'r')]
      "leftOperand": { "@value": "[=timeVar]", "
          @type": "xsd:time" },
      "operator": "gt",
      "rightOperand": { "@value": "08:00:00", "
          @type": "xsd:time" }
    }]
 }]}
```

Listing 3: A sample C-Level policy that grants usage if the current time is greater than 08:00 AM

Finally, policies may be wrapped into a higher abstract language that is able to code complex behaviour, these are known as $C\text{-}Level$ policies. This complex behaviour can be written using a *coded language* and passed as a term of the *templated language* or the *coded language* can literally wrap the policy; which will be a data variable in the *coded language*. In the former case, no new languages besides the *templated language* will appear explicitly written in the policy. In the latter case, no reference to this complex behaviour will appear in the policy. It is recommended to avoid this last case since it may evolve in adding usage restrictions (i.e., constraints) in the code without appearing in the policy and, therefore, not using the ODRL ontology to express them. Conditions as constraints for the usage of a certain resource must always be expressed with the terms from the ODRL ontology.

Listing 4 shows a sample $C\text{-}Level$ policy that relies on the complex function request (highlighted in blue) that performs a GET request to a remote API. This function is not implemented, and is not possible to be implemented, with a templated language as procedure. Instead it has to be written using a coded language, such as Python or Java, and passed as term to the templated language. After getting the data from the API, the policy filters its results and injects a fragment of data as a left operand of the policy.

```
[#-- Data access --]
[#assign weather=request("GET", https://api.open-
    meteo.com/v1/forecast?latitude=40.4050099&
    longitude=-3.839519&hourly=temperature_2m)]
[#-- Data handling --]
[#assign tmp="ERROR"]
[#list weather.hourly.time as elem]
  [#if elem?contains("T"+currentTime) || elem?
      contains("T0"+currentTime)]
    [#assign tmp=weather.hourly.temperature_2m[
        elem?index]]
    [#break]
  [/#if]
[/#list]

{"@context": "...",
"@type": "Set",
"@type": "Policy",
"uid": "https://upm.es/policy/1",
"permission": [{
   "target": "https://jsonplaceholder.typicode.
       com/users/1",
   "action": "read",
   "constraint": [{
      "leftOperand": { "@value": "[=tmp]", "
          @type": "xsd:float"},
      "operator": "gt",
      "rightOperand": { "@value": "35", "@type":
          "xsd:float" }
   } ]
}]}
```

Listing 4: A sample $C\text{-}Level$ policy that grants usage if temperature from external API is above 35º

Fig. 2 shows the hierarchy between these languages and possible choices to implement them. Note that the enforcement occurs with the *interpreted language*, which is the only one that cannot be omitted. Instead, the rest of the language layers on top of it, which are optional, are used to handle input dynamic data (*interpolated and templated languages*) or to handle events outside the policy and use their data (*coded languages*). In fact, when a $C\text{-}Policy$ is enforced, it must produce a $B2\text{-}Level$ or $B1\text{-}Level$ policy. Similarly, when $B2\text{-}Level$ is enforced, a $B1\text{-}Level$ policy is produced, and when $B1\text{-}Level$ is enforced a $A\text{-}Level$ policy. The only policies which result after the enforcement is a *Usage Decision* are the $A\text{-}Level$ policies.

On the other hand, Fig. 2 shows several languages that can be used to implement these language layers. Note that although $A\text{-}Level$ policies do not introduce new terms outside the ODRL ontology lexicon, when they need to be enforced, its interpretable terms (operands, operators, and actions) must be extracted and expressed with the *interpreted language* so that they can be evaluated and produce a result. In order to extract these terms, a JSON Path or a SPARQL query can be used. As *interpreted language* many choices exists, for instance the SPARQL query language (Harris and Seaborne, 2013) or even Python, as long as they implement functions and these can be coupled with the terms that appear in the policies. The next section delve into the details of the necessary algorithm to enforce these policies regardless of the implementation language, and two implementations are reported and tested. One relying on SPARQL as *interpreted language*, Freemarker as *templated language* and Java as *coded language*. The other relies on Python as *interpreted language*, Jinja as *templated language* and Python as *coded language*.

Note that the approach described in this section also provides different extension points, as shown in Fig. 2. This feature is paramount since ODRL is general purpose-based, and specific domains will need to extend the ODRL policies to adapt to new usage conditions. To this end, first the *descriptive language* must be extended, that is, the ODRL ontology, by adding new terms following the good and well-established ontological practices for it (Suárez-Figueroa et al., 2015). Then, it is necessary to implement new functions in the *interpreted language* and couple them to the new terms created in the ontology. These two extension points allow practitioners to adapt ODRL to new domains.

The *templated language* can be extended with new procedures, which are functions defined with the same *templated language*. These are helpful for encoding repetitive and complex data handling processes into a single procedure that can be called several times; note that such procedures are defined within the policy. Finally, the *coded language* can extend the functions that the *templated language* can use. However, practitioners must be careful using the *templated language* and *coded language* as well as their extensions, as they can define usage restrictions using these languages instead of relying on ODRL ontology terms. The usage conditions must always be express uniquely with terms from the ODRL ontology, or extensions of it, and always handled with the *interpreted language*. The *interpolated, templated, and coded languages* are only meant, and must only be used, to handle dynamic data or hide stone-written data values to avoid leakage of private information.

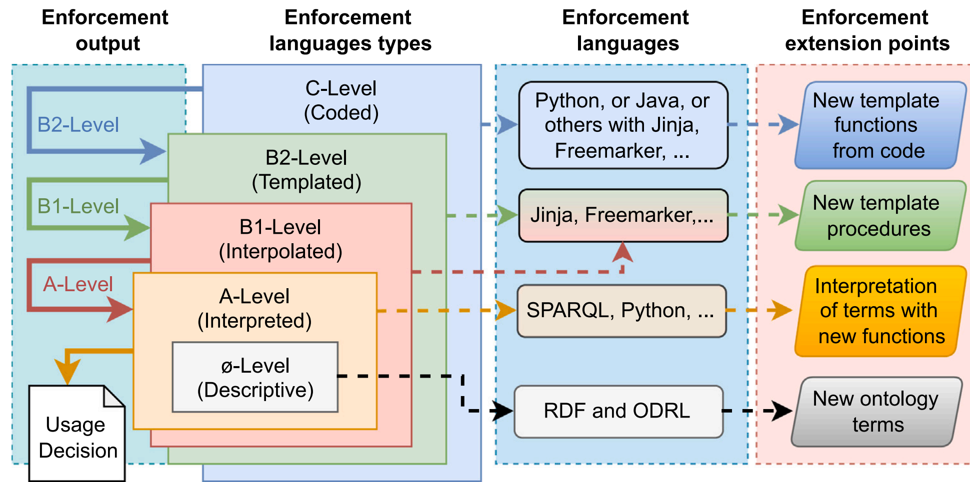


Fig. 2. ODRL enforcement hierarchy, languages, and extension points.

3. Enforcement algorithm

This section presents the ODRE framework algorithm endowed for enforcing ODRL policies expressed the approach explained in the previous section. However, before delve into the details, several concepts used by the algorithm must be defined.

Policy. P_i is defined as a policy written with the terms from the ODRL ontology (or any ontological extension of it) and which level is denoted by i that takes as value the letter of the level it represents, for instance, an $A\text{-}Level$ policy is denoted as P_A. In the case the policy level is unknown, or does not matter (it could be either A, $B1$, $B2$, or C), the policy is denoted as P_*.

Policy reduction. Reducing a policy is the process of enforcing any policy which level is above $A\text{-}Level$ ($B1$, $B2$, or C levels) and producing as output an $A\text{-}Level$ policy. To this end, the procedure $reduce$ is defined, which receives as input a policy P_*, a set M that contains data variables and their values, and a set F that contains function variables and their implementations with a *coded language*. Both sets can be empty in the case no dynamic data is handled or no functions are implemented with the *coded language*.

Policy filtering. Three filtering operations are needed to enforce a given policy. To this end, the procedures $rules$, $constrais$, and $action$ are defined. The first, $rules$, takes as argument an $A\text{-}Level$ policy and returns a set R_{id} containing all the ids of every rule written in the policy; each of one denoted as r_{id}. The second, $constrais$, takes as argument an $A\text{-}Level$ policy and the id of a rule it contains (r_{id}); produces as result a set C_R containing tuples of the form (o, o_L, o_R) where o is an operator, o_L and o_R are its left and right operand respectively. The third, $action$, takes as argument an $A\text{-}Level$ policy and the id of a rule it contains (r_{id}); provides as result the action associated to that rule denoted by a.

Policy transformation. In order to enforce the constrains of a policy these have to be expressed into an *interpreted language*. To this end, the $transformConstraints$ procedure is defined; which takes as argument the set C_R that contains tuples of the form (o, o_L, o_R) being o an operator, and being o_L and o_R its left and right operand respectively. As a result, $transformConstraints$ produces a policy expressed using an *interpreted language* denoted by P_I.

In order to enforce actions, these must be translated into an *interpreted language* as long as such language supports that action, i.e., it knows how to interpret it. To this end, the $supported$ procedure is defined; which takes as argument an action a and returns a Boolean value indicating whether such action is supported by the *interpreted language*. Also, the procedure $transformAction$ is defined that takes as input the action a and returns such action expressed with the *interpreted language*, A_I so it can be evaluated.

Policy Evaluation. A policy expressed using an *interpreted language*, denoted by P_I, can be evaluated producing a Boolean value as a result. The procedure $evaluate$ is defined so that it receives as argument a policy P_I written with the *interpreted language*, and outputs a value d that can be true or false.

Perform Action. An action expressed using an *interpreted language*, denoted by A_I, can be evaluated (a.k.a performed) producing a result. Note that only actions coupled with procedures in the *interpreted language* can be evaluated. The procedure $perform$ is defined so that it receives as argument an action A_I written with the *interpreted language*, and outputs any value as a result of performing such action denoted by V. Note that the result may be an empty value.

Usage Decision. After an $A\text{-}Level$ policy is enforced, a *Usage Decision* is provided. This concept, denoted by D, is a set of tuples filled during the evaluation of the different constraints written in the policy and denoted by C_R. The tuples of the *Usage Decision* D have as the first element the action a, the value of the second element may differ depending on one of the following scenarios. If the action a is not supported by the $interpretedlanguage$, i.e., $supports(a)$ is false, the second element of the tuple is filled with the same action a so a third party is informed about the fact that the action is not performed by the algorithm. When the action a is coupled with a procedure in the $interpretedlanguage$, i.e., $supports(a)$ is true, then the tuple contains as second argument the result of performing such action.

Taking the previously defined concepts as granted, Algorithm 1 presents the ODRE framework enforcement algorithm. The algorithm takes as input a policy P_*, a set of data variables and their values (M), and a set of function variables and their implementations (F). As a result, the algorithm produces a $Usage\ Decision$ set (D).

The algorithm first reduces the input policy P_* to an $A\text{-}Level$ policy (line 2). To this end, it relies on the $reduce$ procedure which, on the one hand, replaces the data and function variables written in the policy with either the *interpolated* or *templated* language with their values encoded in the sets M or F, respectively. On the other hand, $reduce$ enforces the policy that, as a result, obtains a $A\text{-}Level$ one. Note that if the input is already a $A\text{-}Level$ policy, $reduce$ has no effect since it only works on the *interpolated* or *templated* languages.

Then, the algorithm extracts and iterates over the rules ids (R_{id}) written in the reduced policy P_A (lines 3–4). For each rule (r_{id}) the algorithm fills the set of usage decisions D (lines 4–16). To this end, first, it extracts the constraints written in the policy that are associated with each of the specific rules extracted based on their id (r_{id}) (line 5). The algorithm then translates these constraints into a policy written in an *interpreted language* and evaluates it (lines 6–7). In the event that the evaluation returns a value d that is true, the algorithm extracts the

Algorithm 1: ODRE Enforcement algorithm

```
   Data: P_*, M, F
   Result: D
1  D ← ∅;
2  P_A ← reduce(P_*, M, F);
3  R_id ← rules(P_A);
4  for r_id ∈ R_id do
5  |  C_R ← constraints(P_A, r_id);
6  |  P_I ← transformConstraints(C_R);
7  |  d ← evaluate(P_I);
8  |  if d then
9  |  |  a ← action(P_A, r_id);
10 |  |  if supported(a) then
11 |  |  |  A_I ← transformAction(a);
12 |  |  |  V ← perform(A_I);
13 |  |  |  D ← D ∪ {(a, V)};
14 |  |  end
15 |  |  else
16 |  |  |  D ← D ∪ {(a, a)};
17 |  |  end
18 |  end
19 end
```

action associated with it; otherwise it outputs an empty $Usage\ Decision$ D (lines 8).

In the case where d is true and once the action a is extracted, the algorithm checks if this action is supported by the *interpreted language* by means of the $supported$ procedure (line 10). If the action is eligible for enforcement, it is translated and expressed in the *interpreted language* by means of the procedure $transformAction$ and then enforced with the procedure $perform$. The result of performing the action V is included in the $Usage\ Decision$ as the tuple (a, V) (lines 11–13). On the contrary, if the action is not eligible for enforcement, the $Usage\ Decision$ as the tuple (a, a) (line 16), so a third party is informed that the action was not performed by the algorithm either because it cannot be performed as a specific action because it is too abstract or because the third party is responsible for performing it.

3.1. Running sample

In order to provide a running sample using Algorithm 1, let us assume two implementations that use Python and SPARQL as *interpreted language*, respectively, and Freemarker as *templated language*. Table 1 shows a glance of how Algorithm 1 would enforce the policies of Listing 8, Listing 2, Listing 3, and Listing 4. For this running sample, it is also assumed as input of Algorithm 1 the values {"$requestToken$" : "$eyJhbGciO...$"} for the set M and the set F containing the function $request$ implemented with a Java method (since Listing 4 is expressed with Freemarker that supports Java-coded functions as *coded language*).

Table 1 presents three columns. The first, common to both implementations, is named C_R : $reduced\ constraints$ and displays the result running the filtering procedure $constraints$ for each policy; i.e., the line 5 from Algorithm 1. It is important to mention that before running the $constraints$ procedure the algorithm has also run procedures $reduce$ and $rules$. The result of the $constraints$ procedure is the tuple (o, o_L, o_R) where o is an operator, o_L and o_R are its left and right operands, respectively. It is worth mentioning that the left operand of Listing 2 had its [= $requestToken$] Freemarker variable replaced by the value from M under the same name, the left operand of Listing 3 had its [= $timeVar$] Freemarker variable replaced with Listing the current time injected by the Freemarker engine, and, finally, Listing 4 had its [= *tmp*] variable replaced with the value provided by the API which was retrieved with the $request$ function implemented with Java and passed through the set F to the Algorithm 1.

The second and third columns of Table 1 present the result of running the $transformConstraints$ procedure on line 6 of Algorithm 1 for each policy shown in Listings 1 to 4. The result is an equivalent policy P_I expressed according to an *interpreted language*; in the case of the second column is Python and in the case of the third column is SPARQL.

For Python, it can be observed that a possible implementation of the procedures $transformConstraints$ and $transformAction$ may rewrite each of the reduced constraints C_R as a set of binary functions where the name of the function is the operator and as an argument the operands; then these constraints are concatenated with the Python Logical Operator *and*. Note that each operand, left or right, is casted into a Python type that is equivalent to their *xsd* type. This step is crucial since certain operands act differently depending on the types; e.g., comparing two dates is not the same as comparing dates with time.

For implementing the $evaluate$ procedure, the *eval* Python function.[8] may be used. Note that, for enforcing policies from Listings 1 to 4, it is also crucial to have implemented in Python the functions that appear in the *Python Interpreted Policy* column, such as *odrl_datetime* or *cast_datetime* The *eval* native Python function will try to invoke these functions when evaluating the policies from *Python Interpreted Policies*.

For SPARQL, it can be observed that a possible implementation of the $transformConstraints$ procedure may wrap each of the reduced constraints C_R as a SELECT query with one BIND statement; which would concatenate every constraint with the AND operator from SPARQL. The same approach can be taken for the implementation of $transformAction$. Note that the constraints are rewritten as custom binary SPARQL functions, which is why their prefix changes to *fnc*. Also, note that the *xsd* types are preserved for the same casting reasons as mentioned above. For implementing the $evalue$ procedure, any SPARQL library can be used insofar as it supports extension of the language with native functions; which is needed to implement non-SPARQL native functions *fnc:now* and *fnc:media*. Note that SPARQL needs less custom functions, as it already supports most of the operations needed. For example, the native operand >, or ==, already support dates or numbers as arguments.

3.2. Practical scenario in AURORAL: Smart lab

In the context of the European project AURORAL, the ODRE framework is integrated as part of the AURORAL middleware. This project interconnects a set of distributed nodes through the AURORAL platform that follows a decentralised architecture where each node relies on a middleware that is responsible for Gómez-Carmona et al. (2023): discover nodes, exchange data, ensure policy-based access, and validate the data that have been exchanged. The features related to the policy-based access are implemented in the AURORAL middleware with the Java implementation of ODRE.

In this scenario, the Universidad Politécnica de Madrid (UPM), which is a data provider on the AURORAL platform, wishes to share the data of one of their laboratories with other AURORAL nodes. However, UPM has some sensitive data and needs to share the data with other nodes through the platform only under certain conditions: (A) Data can be shared only if the occupancy sensors within the laboratory report that there is someone inside the laboratory; and (B) Data can be shared only during working hours, from 08:00 am to 18:00 pm. Fig. 3 depicts the aforementioned scenario. In order to ensure access to its infrastructures, UPM must define a policy that encodes the aforementioned conditions, i.e. the lab policy depicted in Fig. 3.

To this end, the policy must be written referring to the ODRL ontology using the ODRL ontology context[9] and must define several mandatory elements: the id of the policy, the type of policy (permission,

[8] https://www.w3schools.com/python/ref_func_eval.asp
[9] https://www.w3.org/ns/odrl.jsonld

Table 1
An enforcement running sample using Python and SPARQL as *interpreted languages*.

Policy	C_R: Reduced constraints	P_I : Python Interpreted Policy	P_I : SPARQL Interpreted Policy
Policy in Listing 8	{(odrl:gt, odrl:dateTime, “2018-01-01T00:40:30”^^xsd:dateTime), (odrl:eq, odrl:media, “online”^^xsd:string) }	odrl_gt(odrl_datetime(), cast_dateTime(“2018-01-01T00:40:30“)) and odrl_eq(odrl_media(), cast_string(“online”))	SELECT ?ruleId { BIND (fnc:now() >xsd:dateTime(“2018-01-01T00:40:30”) AND fnc:media() == xsd:string(“online”) AS ?ruleId) }
Policy in Listing 2	{ (eq, “eyJhbGciO...”^^xsd:string, “eyJhbGciO...”^^xsd:string)}	odrl_eq(cast_string(“eyJhbGciO...”), cast_string(“eyJhbGciO...”) }	SELECT ?ruleId { BIND (xsd:string(“eyJhbGciO”) == xsd:string(“eyJhbGciO...”) AS ?ruleId) }
Policy in Listing 3	{ (gt,“13:30”^^xsd:time, “08:00:00”^^xsd:time) }	odrl_gt(cast_time(“13:30”), cast_time(“08:00:00”))	SELECT ?ruleId { BIND (xsd:time(“13:30”) > xsd:time(“08:00:00”) AS ?ruleId) }
Policy in Listing 4	{ (gt,“32”^^xsd:float, “35”^^xsd:float) }	odrl_gt(cast_float(“32”), cast_float(“35”))	SELECT ?ruleId { BIND (xsd:float(“32”) > xsd:float(“35”) AS ?ruleId) }

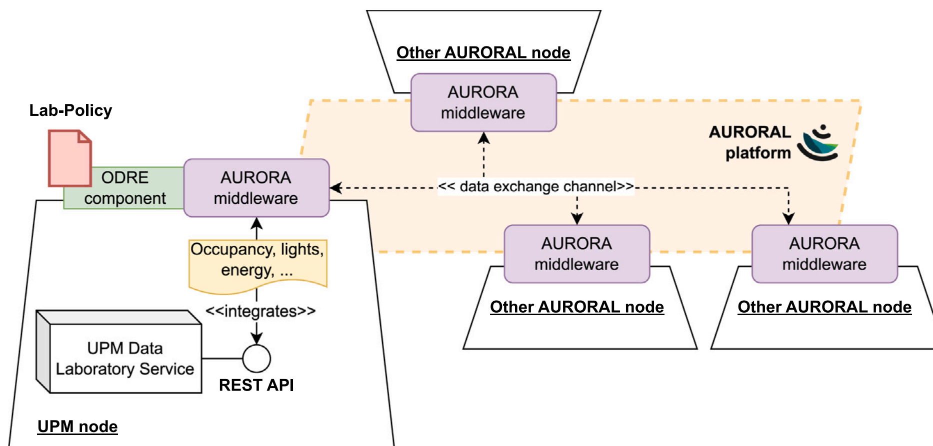


Fig. 3. Policy-based access from the AURORAL use case.

prohibition, or duty), the resource to be protected, and the action to be performed in the case the access is granted. Listing 5 shows the lab policy that contains these elements for the case of UPM as data provider.

```
{"@context": "...",
"@type": "Policy",
"uid": "uuid:359a4066-d08d-4176-8dd2-7dadf7ea88f5
    ",
"permission": [{
   "target": "https://auroral-nodes.com/node/
       eeaa5959-54c1-4e55-ad7d-a1e60cb1508f/
       properties/laboratory",
   "action": "read",
   [...]
}]}
```

Listing 5: Fragment of the Lab-Policy defined by UPM as data provider in AURORAL

The ODRL ontology, and its context, provides the terms to be used to define the aforementioned elements: *“uid”* denotes the identifier of the policy which should be universal, *“@type”* and *“permission”* allow to define the type of the policy and, in particular, the fact that it is a permission. Then, the term *“target"* is used to specify the resource to be protected, in this case, the endpoint of the UPM laboratory data in AURORAL. Finally, the term *“action”* defines the action that is allowed to perform in the case that the policy is enforced positively. Note that the value of the JSON keys *“@type”* and *“action"* must match one of the available defined as part of the ODRL ontology context.[10] Note that the policy in Listing 5 still does not encode the data restrictions required by UPM. To this end, it is necessary to add one or more constraints per restriction to the policy. ODRL ontology provides the term *“constraints”* to codify these restrictions.

The first restriction, *(A) Data can be shared only if occupancy sensors within the laboratory report that there is someone inside the laboratory* is encoded, as shown in the Listing 6, by injecting the occupancy value directly as a left operand. To achieve this behaviour, when the policy is evaluated, ODRE accesses the UPM data infrastructure, extracts, and then injects the occupancy value using the template language and a coded function; highlighted in green and blue, respectively, in Listing 6.

```
{ ... [#assign headers= "{ 'Authorization' : '
    Bearer ...' }" ]
  [#assign data=request("GET", headers, "http://
      smartlab.auroral-factory.linkeddata.es/api
      /.../entity_id=occupancy_sensor")?eval]
  [#assign occupancy=data[-1]]
  {
      "leftOperand": "[=occupancy]",
      "operator": "gt",
      "rightOperand": { "@value": "0", "@type": "
          xsd:integer" }
   }  ... }
```

[10] https://w3c.github.io/odrl/bp/

Listing 6: Constraint that checks if there is someone at UPM laboratory.

The second restriction *(B) Data can be shared only during working hours, from 08:00 am to 18:00 pm.* is encoded, as shown in Listing 7, using the ODRL time extension[11] and, in particular, the *between* operator that checks whether the current time is allocated after the value of the left operand and before the value of the right operand as long as both values in the left and right operand are xsd:time.

```
{ ... {
      "leftOperand": { "@value": "08:00", "@type
          ": "xsd:time" }
      "operator": "otime:between",
      "rightOperand": { "@value": "18:00", "@type
          ": "xsd:time" }
   }  ... }
```

Listing 7: Constraint that checks it the current time is within a range.

Putting the previous listings together, the Listing 8 shows a complete version of the lab policy. Note that this policy is a *C-Level* and relies on Freemarker as *templated language* and Java as *coded language*.

```
{"@context": "...",
"@type": "Policy",
"uid": "uuid:359a4066-d08d-4176-8dd2-7dadf7ea88f5
    ",
"permission": [{
   "target": "https://auroral-nodes.com/node/
       eeaa5959-54c1-4e55-ad7d-a1e60cb1508f/
       properties/laboratory",
   "action": "read",
   "constraint": [
   [#assign headers= "{ 'Authorization' : 'Bearer
        ...' }" ]
    [#assign data=request("GET", headers, "http
        ://smartlab.auroral-factory.linkeddata.es
        /api/.../entity_id=occupancy_sensor")?
        eval]
    [#assign occupancy=data[-1]]
  {
      "leftOperand": "[=occupancy]",
      "operator": "gt",
      "rightOperand": { "@value": "0", "@type": "
          xsd:integer" }
   },{
      "leftOperand": { "@value": "08:00", "@type
          ": "xsd:time" }
      "operator": "otime:between",
      "rightOperand": { "@value": "18:00", "@type
          ": "xsd:time" }
   }]
}]}
```

Listing 8: Complete Lab-Policy defined by UPM as data provider in AURORAL

3.3. Discussion: limitations and challenges ahead

Once the approach on how to express the ODRL policies to enforce them and handle dynamic data is presented (Section 2) and Algorithm 1, it is important to analyse and highlight several limitations: about the ODRL vocabulary and how it affects Algorithm 1 and about our proposal. In addition, future challenges are discussed.

ODRE bad practices: The approach presented in the article presents a drawback that may imply having different ODRL policies that may seem different but are the same. In addition, these policies would not clearly codify the constraints under which a resources can be used. This is due to the fact that certain policies may mix RDF and an *interpolated* or *templated language* which may also inject some behaviour using a *coded language*. Specifying these constraints without the semantics provided by the ODRL ontology is possible but is a bad practice since, on the one hand, looses the goal of using an ontology that provides a unequivocally definition of a concept, and, on the other hand, makes policies harder to compare, understand by a person, or to write since they will enclose a large amount of behaviour with languages different from the descriptive.

Synchronous vs asynchronous enforcement: the enforcement of a policy may happen synchronously; whether someone desires to access a certain resource and then the enforcement procedure is invoked. Alternatively, enforcement may occur asynchronously, the constraints of a policy are continuously evaluated and when they enforce positively their related action is performed. Note that these two methods are not exclusive and, instead, there should be policies that must be enforced synchronously and others asynchronously. To this end, ODRL lacks of the semantics to label this feature in a policy and it will be a future challenge to tackle. Also, the Algorithm 1 from the ODRE framework has been endowed and tested only to perform synchronous enforcement.

Finally, regardless the synchronous or asynchronous approach for enforcement it is important to mention the potential problems derived from concurrency during the enforcement process. In the synchronous evaluation concurrency should not entail any issue since when the evaluation is triggered the policy and related data are evaluated and, thus, in the case of concurrency each thread would have a copy of the policy and related data (that may be even different). Note that between two, or more, threads there is no shared data besides the policy that can be a copy of the original one, solving any derived issue. Nevertheless, in the asynchronous scenario authors forecast a potential issue since asynchronous data will be required for the evaluation of the policy and different actors may override the asynchronous data needed for the enforcement of a policy. This will be a future challenge to be tackled.

ODRL abstract definitions: the ODRL vocabulary presents some important limitations in terms of expressiveness for its operands, operators, and actions. Some of them have abstract definition and, therefore, it is almost impossible to implement them or, several implementations may drastically differ one from the other depending on the particular use case. For instance, the action *read* can be interpreted as gaining the access to a certain API or as if the result of the enforcement is the output of reading such API; other interpretations are also possible, as if someone can read in the physically world a document. This issue can be handled using ODRE by relying on the *interpolated/templated language* however, this also entails losing certain semantics as already explained before. This can be view as a benefit that ODRE offers, but also, as a bad practice.

The only approach to solve this issue correctly, without losing semantics or incurring in bad practices, is to extend the ODRL ontology to such specific use cases, providing operands, operators, and actions that are unequivocally described and, therefore, can only be implemented and interpreted in a unique way.

ODRL binary predicates: the ODRL ontology defines the constraints as an element that relates to two operands and an operator. In other words, the operators are terms designed to consider only two inputs, i.e., the left and right operands. This may be a limitation in the case other operators are needed and, these, require more operators than two. For instance, a policy that grants usage if two WGS84 coordinates are located in a proximity threshold of less than 1 km. To express such

[11] https://w3id.org/def/odre-time

policy it will be necessary to define an operator that considers the two coordinates and the threshold of proximity, i.e., three input arguments.

To tackle this issue, it would seem as a good idea to define a new operator that supports a third operand; however, how to name this new operand is a challenge itself, since the current semantics for operands are left and right. If we scale the input of operators to have N-ary predicates, this problem becomes even more complex. In addition, operands hold no order for the operator explicitly; which is also a potential problem particularly for N-ary predicates. As before, this issue can be addressed using ODRE and relying on the *interpolated/templated language*, however, although it may partially solve this problem, it will also entail the bad practice explained in the first place.

ODRL 0-ary operands and actions: the ODRL standard presents operands as constant values or as ontology terms that must be interpreted and replaced during the enforcement with a value. Nevertheless, the current representation of these operands is limited since they assume that such operands require no input to produce an output value. For instance, *odrl:dateTime* requires no input to provide the current date with time. Far from practical scenarios, some operands may require input values to operate correctly and this, as it is now endowed the ODRL ontology, cannot be specified. For instance, *odrl:dateTime* may require as input the timezone to provide the correct date and time. Similar to operands, ODRL standard presents actions as ontology terms that must be interpreted. However, these actions may also require some input information to be performed. These kind of scenarios are currently out of the scope of ODRL but, as it has been adopted in wide spread scenarios like Data Spaces, is a challenge ahead. In addition to this, a challenge ahead is to consider having operands that have as input nested operands, or actions that take as input some operand. As before, this issue can be addressed using ODRE and relying on the *interpolated/templated language*, however, although it may partially solve this problem, it will also entail the bad practice explained in the first place.

ODRL Stone-written conditions: the ODRL ontology relies on terms that are defined as left or right operands to specify data that is not specifically written in the policy but somehow must be used or considered when enforcing. However, the information output as result of interpreting these terms is always compared to data conditions, usually right operands, that are constant. For instance, the policy from Listing 8 compares the $dateTime$ left operand with the constant *2018-01-01*. Note that the latter value is stone written in the policy and readable if the policy is accessed, leading to a potential privacy leak. Although this issue may seem not important, if we have a look at the policy from Listing 2 one can soon realise how leaking a token API is a severe security breach.

In addition, in the ODRL ontology, there is an unclear specification of where the data come from. For example, with the $dateTime$ left operand it can be assumed that is the date and time of the system that runs the enforcement algorithm. However, the policy shown in the Listing 2 clearly needs the input of a person, that is, the token, and should rely on a third-party API to check if the token exists instead of having the stone-written value as the right operand. In other words, policies may require data from the system that runs the enforcement algorithm, data from the person trying to access a resource, and data coming from a third party entity (service or person). However, this issue is not addressed by the ODRL standard or their ontology, which is an important challenge. The ODRE approach provides technical support for this problem by relying on the *interpolated, templated* or $coded$ $languages$. Note that in these cases, the bad practice explained in the first place is not happening since no semantic is bypassed with terms outside the ontology.

ODRL Responsibilities: since the ODRL standard has only promoted the ODRL ontology and the enforcement is currently out of the scope of this standard, there is lack expressiveness about the responsibilities related to a policy. For instance, who must perform an action written in the policy? is the enforcement algorithm or an external actor?; who is the responsible of checking the constraints of a policy?, a third-party actor (that may be an enforcement algorithm), a local enforcement algorithm, or a shared task?; which actors are eligible to try to use a resource and thus trigger the enforcement algorithm?. Considering the operand $odrl : spatial$ that stands for geospatial named area who is the responsible of providing such data? There is an unclear answer to all these questions and how these responsibilities should be specified.

4. Evaluation

In order to evaluate Algorithm 1 and prove that the ODRE approach is language agnostic, which means that different languages can be used as *interpreted, interpolated, templated,* or *coded*, two implementations are provided: $pyodre$ and $odre-java$. Both implementations are published on GitHub and distributed as open-source under the APACHE 2.0 licence. $Pyodre$,[12] relies on Python as *interpreted language* Jinja2 as *interpolated* and *templated language*, and Python as *coded language*. This implementation is also distributed through the pip package installer.[13] The $odre-java$,[14] relies on SPARQL as *interpreted language* Freemarker as *interpolated and templated language* and Java as *coded language*. This implementation is distributed through maven central.[15]

Both implementations cover most of the operands from ODRL, namely: *odrl:lt, odrl:lteq, odrl:eq, odrl:neq, odrl:gt, odrl:gteq*. Also, they implement the left operand *odrl:dateTime*. In addition, to prove the extensibility of our proposal, a small extension of the ODRL vocabulary about time has been created. The ontology of this extension is available at https://w3id.org/def/odre-time#, let us assume $otime$ as its prefix. The ontology extension includes a new operand named $otime{:}time$ that provides the current time, a new operator $otime{:}between$ that returns true if the current time is between the ones specified as the left and right operands. Finally, as another extension, the action $dummy{:}read$ has been provided to test the enforcement of the actions. This action reads a dummy API and provides as output its result. Note that this action has no ontology related, since it has been created only to show how actions can be enforced but is not meant to be used in real scenarios.

In order to test these implementations, 24 policies have been defined using the following criterion: 6 policies defined as the left and right operands a constant and each one tested one of the operators lt, $lteq$, eq, neq, gt, $gteq$; 6 policies defined with the left operand $odrl{:}dateTime$ and each tested one of the previous operators; 6 policies defined with the left operand $otime{:}time$ and each tested one of the previous operators; 1 policy defined two constants and tested the $otime{:}between$ operand; 1 policy defined two constants and tested the $dummy{:}read$; and, finally, 4 policies used the their *templated language* to inject values or functions (implemented with Python or Java depending on the implementation). Then, these policies were used to define 24 unit tests in both Python[16] and Java[17] to test their respective implementations.

In addition to running the unit tests successfully, the time it takes to carry out the enforcement has been analysed. To this end, each unitary test was run 15 times and the seconds they took recorded. The first five measurements were discarded as warm-up, and the rest were averaged and compared between implementations. Fig. 4 displays these results, showing the results obtained by the Python and Java implementations to run the same policy in seconds.

12 https://github.com/ODRE-Framework/odre-python
13 https://pypi.org/project/pyodre/
14 https://github.com/ODRE-Framework/odre-java
15 https://central.sonatype.com/artifact/io.github.odre-framework/odre-core
16 https://github.com/ODRE-Framework/odre-python/tree/main/test
17 https://github.com/ODRE-Framework/odre-java/blob/main/src/test/java/tests/odre

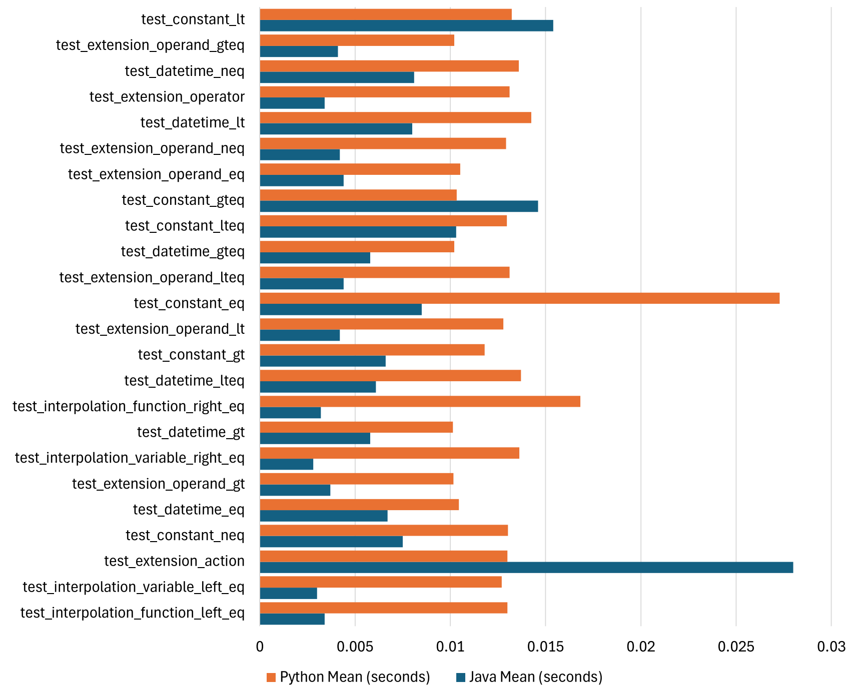


Fig. 4. ODRL enforcement duration in seconds for Python and Java implementations.

It can be observed that, in general, both implementations enforce all the policing in less than 0.03 s. It is worth mentioning that in general, the Java implementation behaves better than python with the exception of *test_extension_action*, the difference observed can be due to the fact that such policy performed a request to an API and Java handles this functionality with streams, increasing the running time. For the sake of reproducibility, and following the FAIR principles, a frozen version of the implementations have been uploaded to the Zenodo repository,[18] along with two reports about the implementations and instructions for each on how reproduce these experiments.

5. Related work

The definition of policies for the usage of data resources has been extensively analysed by researchers (del Álamo et al., 2022). However, it is important to differentiate between two types of research proposals; those focussing on the description of these policies and related challenges, and those focussing on the enforcement of these policies (Leicht and Heisel, 2019). In addition, another feature to take into account when analysing proposals similar to ODRE is the fact that they rely, or not, on standards vocabularies to express the policies. In this article, the proposals are narrowed down to those that rely on policies expressed with the W3C ODRL standard and which scope is the enforcement of ODRL policies, discarding the rest as out of scope. The proposals related to enforcement are numerous (Akaichi and Kirrane, 2022; Kirrane et al., 2018), in order to ease their review, the authors have classified them into the following types:

Theoretical enforcement - these are proposals that promote theoretical ideas to perform the enforcement of ODRL policies, or challenges, but without implementations to support their claims. The proposal of Munoz-Arcentales et al. (2020) presents the need for flexible usage control solutions that can be adapted and used in different scenarios, but enforcing ODRL is out of the scope. The proposal of Cirillo et al. (2020) rely on a subset of ODRL and does not tackle how to enforce the ODRL policies, but instead, policies expressed with an extended version of ODRL for specific use cases. The proposal of Cirillo et al. (2020) is not considered to enforce ODRL since it does not support policies expressed only with ODRL terms. The proposal of Akaichi et al. (2024) presents an architecture to handle ODRL policies in scenarios that require dynamic constraints (addressing the limitation *ODRL Stone-written conditions*). Their proposal focuses on a specific client–server scenario and do not delve in the details of how to express the ODRL policies from a generic point of view of the enforcement. In addition, their proposal extends the ODRL ontology in such a way that their solution works only with it. Finally, their proposal is theoretical and has no implementation related.

Discussion: some of these proposals (Munoz-Arcentales et al., 2020; Cirillo et al., 2020) do not actually aim to support ODRL enforcement but, instead, use a subset of the ODRL for certain limited scenarios not promoting a general purpose enforcement solution. Furthermore, these two proposals do not provide a generic algorithm for enforcing ODRL. The proposal closer to ODRE is the one of Akaichi et al. (2024) since they addressed one of the limitations that also ODRE tackles. However their proposal is not generic for the ODRL ontology or any extensions of it. In addition, they provide no generic algorithm for enforcing ODRL.

Practical enforcement - these are proposals available on the Web. ODRL-PAP[19] promotes an enforcement engine in the context of the DOME-Marketplace project.[20] The MOSAICroWN Policy Engine[21] is built in the context of the MOSAICrOWN project.[22] Both of the previous proposals are a specific solutions for projects and have their own vocabulary based on ODRL, but different from the ODRL standard vocabulary.

Discussion: Although these proposals perform the enforcement, the policies they support is limited and are not ODRL policies expressed with the standard ontology. Furthermore, as major drawback these proposals are not related to research papers and do not provide any procedure pseudo-code to reproduce their results.

[18] https://zenodo.org/records/13735971

[19] https://github.com/wistefan/odrl-pap

[20] https://dome-marketplace.eu

[21] https://github.com/mosaicrown/policy-engine

[22] https://mosaicrown.eu

Traverse enforcing - these are proposals focussing on the enforcement of ODRL policies by traversing the policies into another vocabulary to express policies that support enforcement (Akaichi and Kirrane, 2022). Some proposals focus on aligning ODRL with the XACML standard (Standard, 2013) and its architecture (Dam et al., 2023). However, although these proposals may benefit from the enforcement of XACML, it is not possible to fully align ODRL with XACML and, therefore, this approach can be suitable for certain scenarios but not as a general solution to enforce ODRL; which is the goal of our article. Another similar approach was proposed by Hosseinzadeh et al. (2020); which consists of translating ODRL policies into MYDATA Policy Language.[23] Another approach was proposed in a position article that opened the possibility of enforcing ODRL policies as mappings (Cano-Benito et al., 2023). These are commonly used to translate heterogeneous data sources into RDF (Manola et al., 2004), i.e., the format of the ODRL policies. Again, this proposal is suitable for certain scenarios but has major drawbacks, the most important one is that the policies must always be expressed mixing the terms from ODRL and the mapping language.

Discussion: These are the proposals closer to the one presented in this article, however, they do not provide a general algorithm to enforce ODRL using the standard vocabulary, or extensions of it, as presented in Section 2. These proposals rely on translating ODRL policies into others expressed with vocabularies that have enforcing capabilities. In other words, these proposals rely on translating $\emptyset$-$Level$ policies expressed with the ODRL ontology into a different, yet somehow equivalent, $\emptyset$-$Level$ policy expressed with another vocabulary. This approach has to main drawbacks, on the one hand, the ODRL vocabulary is not fully equivalent to others (and vice versa) and thus not any ODRL policy can be translated, on the other hand, in the case of extending ODRL it will be necessary to extend also the vocabulary to which the policies will be translated and their enforcement algorithms. As a result, these approaches are suitable for certain scenarios but do not provide a genuine enforcement procedure for ODRL and, instead, rely on the enforcement of other proposals.

The approach of using a metalanguage to enforce ODRL policies presented in our previous article (Cano-Benito et al., 2023) was the basis to build the ODRE framework. However, the approach presented by Cano-Benito et al. (2023) only considered the policies $B1$-$Level$ or above policies and did not provide the analysis presented in Section 2. In addition, no algorithm was provided. Thinks kind of approach can be classified as policy as code.

Policy as code - this approach consist on expressing the policies as code that can be run, and thus, the policies evaluated. Two sub-approaches exists: (A) policy-compilation (Guarnieri and Livshits, 2009; Yang et al., 2012; Johansen et al., 2015; Lowry et al., 2001); it relies on two languages for policies, one to describe the policy restrictions and the other that can be enforced, a policy expressed in the former language can be translated into the second, i.e., compiled; (B) policy as automata; rely on machine states to represent the policy enforcement (Basin et al., 2013; Walker, 2000; Erlingsson and Schneider, 1999).

In the context of the policy-compilation, some proposals are described: Guarnieri et al. proposes a formal grammar to describe policies and that, then, is evaluated using JavaScript (Guarnieri and Livshits, 2009). Yang et al. defines policies using Jeeves language and then translates the policies into a constraint language called λ_j that is implemented in Scala (Yang et al., 2012). Johansen et al. proposes to describe policies with meta-code that can be later compiled and run (Johansen et al., 2015). Lowry et al. besides using domain-specific language for describing policies and a programming language for enforcement, they introduce an abstraction hierarchy among the different languages that can be used to express and evaluate the policies (Lowry et al., 2001).

Discussion: None of these proposals address ODRL directly, however, the proposals are aligned with the policy hierarchy presented in Fig. 2. In general, the articles do not revolve around a language hierarchy formalisation, with the exception of Lowry et al. (2001). However, authors have not found a common hierarchy that could be used, or that applies, to all these proposals transversally. Bear in mind that, although all the proposals are about policies, each of them applies these policies in very particular and different scenarios. In this sense, the work proposed by Liang et al. (2023) is particularly interesting: they propose a hierarchy among languages to define policies describing robot's behaviour. To this end, they propose to rely on natural language as the more abstract level in the hierarchy and use LLMs and prompting to translate it into the lower level in the hierarchy; that could be a descriptive or coded language. In future, this idea will be explored for ODRL and the natural language added to the hierarchy of Fig. 2.

To the best of the author's knowledge, this article is the first presenting a generic enforcement procedure with an algorithm for ODRL policies that supports the standard and provides several extension mechanisms. In addition, in the articles reviewed in this section, the different key points discussed in Section 6 have not been tackled by the authors. In conclusion, the ODRE framework presents a novel approach to support the ODRL standard and allow ODRL enforcement; which is a feature demanded for this standard (Akaichi and Kirrane, 2022).

6. Conclusions

In this article, the ODRE framework has been presented including: a novel approach to write and classify ODRL policies enabling their enforcement, a generic enforcement algorithm, and two implementations. In addition, an analysis on the current limitations of ODRE and ODRL has been presented along with future challenges. Previous proposals have delved into how to enforce ODRL: some exclusively from the theoretical point of view, others providing only implementations without research articles, and others traversing the enforcement to other initiatives. These latest proposals have mainly relied on translating ODRL policies into other vocabularies that already supported enforcement (such as XACML). However, these proposals fall short when used to enforce any ODRL policy, since the expressiveness (operands, operators, actions, and more) present in the ODRL ontology does not always exist in that target vocabulary. To the best of our knowledge, ODRE is the first proposal that provides a solution to enforce ODRL and is flexible enough to support future extensions of ODRL for specific domain challenges. In addition, the experiments carried out prove that the implementations perform enforcement efficiently for real-world scenarios.

In the future, the authors will address the limitations of ODRL and try to tackle challenges identified; such as defining N-ary operators or operands or addressing how to express synchronous and asynchronous enforcement in policies and extend ODRE to support the second. In addition, specific extensions will be developed for the time and geospatial domains. Regarding ODRE limitations, the authors plan to publish good-practices and guidelines documentation to prevent bad practices. Finally, the authors will analyse the impact on the enforcement that may produce using ODRL with other related vocabularies, preferably standard, such as the Data Privacy Vocabulary[24] (DPV) to enrich the expressiveness of usage and privacy use cases.

CRediT authorship contribution statement

Andrea Cimmino: Writing – review & editing, Writing – original draft, Software, Resources, Methodology, Investigation, Conceptualization. **Juan Cano-Benito:** Writing – review & editing, Resources, Conceptualization. **Raúl García-Castro:** Writing – review & editing, Funding acquisition, Conceptualization.

[23] https://developer.mydata-control.de/language/

[24] https://w3c.github.io/dpv/dpv/

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Raul Garcia Castro reports financial support was provided by European Commission. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through the AURORAL project, Grant Agreement No. 101016854.

Data availability

All data or code used in the article has been shared through GitHub or Zenodo, links are located in the article.

References

Akaichi, I., Kirrane, S., 2022. Usage control specification, enforcement, and robustness: A survey. arXiv:2203.04800.

Akaichi, I., Slabbinck, W., Rojas, J.A., Van Gheluwe, C., Bozzi, G., Colpaert, P., Verborgh, R., Kirrane, S., 2024. Interoperable and continuous usage control enforcement in dataspaces. In: The Second International Workshop on Semantics in Dataspaces, Co-Located with the Extended Semantic Web Conference.

del Álamo, J.M., Guamán, D.S., García, B., Diez, A., 2022. A systematic mapping study on automated analysis of privacy policies. Computing 104 (9), 2053–2076. http://dx.doi.org/10.1007/s00607-022-01076-3.

Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E., 2013. Enforceable security policies revisited. ACM Trans. Inf. Syst. Secur. 16 (1), http://dx.doi.org/10.1145/2487222.2487225.

Cano-Benito, J., Cimmino, A., García-Castro, R., 2023. Injecting data into ODRL privacy policies dynamically with RDF mappings. In: Companion Proceedings of the ACM Web Conference 2023. In: WWW '23 Companion, Association for Computing Machinery, New York, NY, USA, pp. 246–249. http://dx.doi.org/10.1145/3543873.3587358.

Chomsky, N., 2014. Aspects of the Theory of Syntax. (11), MIT Press.

Cirillo, F., Cheng, B., Porcellana, R., Russo, M., Solmaz, G., Sakamoto, H., Romano, S.P., 2020. IntentKeeper: Intent-oriented data usage control for federated data analytics. In: 2020 IEEE 45th Conference on Local Computer Networks. LCN, pp. 204–215. http://dx.doi.org/10.1109/LCN48667.2020.9314823.

Dam, T., Krimbacher, A., Neumaier, S., 2023. Policy patterns for usage control in data spaces. arXiv:2309.11289.

De Vos, M., Kirrane, S., Padget, J., Satoh, K., 2019. ODRL policy modelling and compliance checking. In: Rules and Reasoning: Third International Joint Conference, RuleML+ RR 2019, Bolzano, Italy, September 16–19, 2019, Proceedings 3. Springer, pp. 36–51.

Eitel, A., Jung, C., Brandstädter, R., Hosseinzadeh, A., Bader, S., Kühnle, C., Birnstill, P., Brost, G., Gall, M., Bruckner, F., et al., 2021. Usage Control in the International Data Spaces. Aufl. IDS Association, Berlin.

Erlingsson, U., Schneider, F.B., 1999. SASI enforcement of security policies: A retrospective. In: Proceedings of the 1999 Workshop on New Security Paradigms. pp. 87–95.

Gómez-Carmona, O., Buján-Carballal, D., Casado-Mansilla, D., de Ipiña, D.L., Cano-Benito, J., Cimmino, A., Poveda-Villalón, M., García-Castro, R., Almela-Miralles, J., Apostolidis, D., Drosou, A., Tzovaras, D., Wagner, M., Guadalupe-Rodriguez, M., Salinas, D., Esteller, D., Riera-Rovira, M., González, A., Clavijo-Ágreda, J., Díez-Frias, A., del Carmen Bocanegra-Yáñez, M., Pedro-Henriques, R., Ferreira-Nunes, E., Lux, M., Bujalkova, N., 2023. Mind the gap: The AURORAL ecosystem for the digital transformation of smart communities and rural areas. Technol. Soc. 74, 102304. http://dx.doi.org/10.1016/j.techsoc.2023.102304, URL https://www.sciencedirect.com/science/article/pii/S0160791X23001094.

Guarnieri, S., Livshits, V.B., 2009. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In: USENIX Security Symposium, Vol. 10. pp. 78–85.

Harris, S., Seaborne, A., 2013. SPARQL 1.1 Query Language. In: W3C Recommendation.

Henze, M., Hermerschmidt, L., Kerpen, D., Häußling, R., Rumpe, B., Wehrle, K., 2014. User-driven privacy enforcement for cloud-based services in the internet of things. In: 2014 International Conference on Future Internet of Things and Cloud. pp. 191–196. http://dx.doi.org/10.1109/FiCloud.2014.38.

Hosseinzadeh, A., Eitel, A., Jung, C., 2020. A systematic approach toward extracting technically enforceable policies from data usage control requirements. In: ICISSP. pp. 397–405.

Johansen, H.D., Birrell, E., Van Renesse, R., Schneider, F.B., Stenhaug, M., Johansen, D., 2015. Enforcing privacy policies with meta-code. In: Proceedings of the 6th Asia-Pacific Workshop on Systems. pp. 1–7.

Kebede, M.G., Sileno, G., Van Engers, T., 2021. A critical reflection on ODRL. In: Rodríguez-Doncel, V., Palmirani, M., Araszkiewicz, M., Casanovas, P., Pagallo, U., Sartor, G. (Eds.), AI Approaches To the Complexity of Legal Systems XI-XII. Springer International Publishing, Cham, pp. 48–61.

Kelbert, F., Pretschner, A., 2012. Towards a policy enforcement infrastructure for distributed usage control. In: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies. SACMAT '12, Association for Computing Machinery, New York, NY, USA, pp. 119–122. http://dx.doi.org/10.1145/2295136.2295159.

Kirrane, S., Villata, S., d'Aquin, M., 2018. Privacy, security and policies: A review of problems and solutions with semantic web technologies. Semant. Web 9 (2), 153–161. http://dx.doi.org/10.3233/SW-180289.

Leicht, J., Heisel, M., 2019. A survey on privacy policy languages: Expressiveness concerning data protection regulations. In: 2019 12th CMI Conference on Cybersecurity and Privacy. CMI, pp. 1–6. http://dx.doi.org/10.1109/CMI48017.2019.8962144.

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., Zeng, A., 2023. Code as policies: Language model programs for embodied control. In: 2023 IEEE International Conference on Robotics and Automation. ICRA, IEEE, pp. 9493–9500.

Lowry, M., Pressburger, T., Rosu, G., 2001. Certifying domain-specific policies. In: Proceedings 16th Annual International Conference on Automated Software Engineering. ASE 2001, IEEE, pp. 81–90.

Manola, F., Miller, E., McBride, B., et al., 2004. RDF primer. W3C Recomm. 10 (1–107), 6.

Monegraph, R.I., Villata, S., 2018. ODRL Information Model 2.2. In: W3C Recommendation.

Munoz-Arcentales, A., López-Pernas, S., Pozo, A., Alonso, A., Salvachúa, J., Huecas, G., 2020. Data usage and access control in industrial data spaces: Implementation using FIWARE. Sustainability 12 (9), http://dx.doi.org/10.3390/su12093885, URL https://www.mdpi.com/2071-1050/12/9/3885.

Parr, T.J., 2004. Enforcing strict model-view separation in template engines. In: Proceedings of the 13th International Conference on World Wide Web. pp. 224–233.

Rodríguez-Doncel, V., Gómez-Pérez, A., Mihindukulasooriya, N., 2013. Rights declaration in linked data. In: Proceedings of the Fourth International Conference on Consuming Linked Data - Volume 1034. COLD '13, CEUR-WS.org, Aachen, DEU, pp. 158–169.

Standard, O., 2013. Extensible access control markup language (xacml) version 3.0. A:(22 January 2013). URl: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

Steyskal, S., Polleres, A., 2014. Defining expressive access policies for linked data using the ODRL ontology 2.0. In: Proceedings of the 10th International Conference on Semantic Systems. SEM '14, Association for Computing Machinery, New York, NY, USA, pp. 20–23. http://dx.doi.org/10.1145/2660517.2660530.

Suárez-Figueroa, M.C., Gómez-Pérez, A., Fernandez-Lopez, M., 2015. The NeOn methodology framework: A scenario-based methodology for ontology development. Appl. Ontology 10 (2), 107–145.

Walker, D., 2000. A type system for expressive security policies. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '00, Association for Computing Machinery, New York, NY, USA, pp. 254–267. http://dx.doi.org/10.1145/325694.325728.

Yang, J., Yessenov, K., Solar-Lezama, A., 2012. A language for automatically enforcing privacy policies. ACM SIGPLAN Not. 47 (1), 85–96.