Perennial Semantic Data Terms of Use for Decentralized Web

Rui Zhao
University of Oxford
Oxford, UK
rui.zhao@cs.ox.ac.uk

Jun Zhao
University of Oxford
Oxford, UK
jun.zhao@cs.ox.ac.uk

ABSTRACT

In today's digital landscape, the Web has become increasingly centralized, raising concerns about user privacy violations. Decentralized Web architectures, such as Solid, offer a promising solution by empowering users with better control over their data in their personal 'Pods'. However, a significant challenge remains: users must navigate numerous applications to decide which application can be trusted with access to their data Pods. This often involves reading lengthy and complex Terms of Use agreements, a process that users often find daunting or simply ignore. This compromises user autonomy and impedes detection of data misuse. We propose a novel *formal* description of Data Terms of Use (DToU), along with a DToU reasoner. Users and applications specify their own parts of the DToU policy with local knowledge, covering permissions, requirements, prohibitions and obligations. Automated reasoning verifies compliance, and also derives policies for output data. This constitutes a "perennial" DToU language, where the policy authoring only occurs once, and we can conduct ongoing automated checks across users, applications and activity cycles. Our solution is built on Turtle, Notation 3 and RDF Surfaces, for the language and the reasoning engine. It ensures seamless integration with other semantic tools for enhanced interoperability. We have successfully integrated this language into the Solid framework, and conducted performance benchmark. We believe this work demonstrates a practicality of a perennial DToU language and the potential of a paradigm shift to how users interact with data and applications in a decentralized Web, offering both improved privacy and usability.

CCS CONCEPTS

• **Computing methodologies** → **Knowledge representation and reasoning**; • **Security and privacy** → *Usability in security and privacy*; • **Information systems** → *Semantic web description languages.*

KEYWORDS

Decentralized Web; Data Terms of Use; Usage Control; Formal Modelling; Automated Reasoning; Notation 3

ACM Reference Format:
Rui Zhao and Jun Zhao. 2024. Perennial Semantic Data Terms of Use for Decentralized Web. In *Proceedings of the ACM Web Conference 2024 (WWW '24), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589334.3645631

This work is licensed under a Creative Commons Attribution International 4.0 License.

WWW '24, May 13–17, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0171-9/24/05.
https://doi.org/10.1145/3589334.3645631

1 INTRODUCTION

After years of development, the Web has become an indispensable part of people's life. However, the centralization of the Web has risen as a pressing challenge, leading to various issues like pervasive user behavioural manipulation, privacy breaches and an imbalance of power [34, 40]. Decentralization is viewed as a potential solution to address these issues [15, 21], and initiatives like Solid (Social Linked Data) [30] have gained attention for their aim to return *data and control* to the user, while respecting the openesss and fairness of Web standards and infrastructures.

In a decentralized Web, users store their data in their own storage (such as Solid Pods), and applications must request permission to use and store data there. This shift in data control has the potential to reduce 'vendor-lock-in', as it limits the privileged ownership of data currently prevalently observed in large platforms. Furthermore, it is also crucial to fostering competition among different applications. However, one aspect that has received less attention in the decentralized setting is *how users can sensibly decide* which application should be granted permission to access their data.

Assume in the decentralized setting, Alice wants to use a shopping app to buy shoes, which may require access to several types of data in her Pod, including shoe size, delivery address, and billing information. She is concerned about how such an app may handle her data ethically. In the meantime, Bob, the developer of a shopping app, HappyShop, wants to build trust with users and is willing to provide descriptions on how HappyShop handles users data through its 'Terms of Use'. However, users like Alice typically do not read these terms (aka. "the biggest lie on the Internet") because of information overload and their length [20, 24]. The problem is exacerbated in the decentralized setting with its numerous apps that may require users' decision regarding access to their data Pods, especially if, e.g. an accounting app, TotalAcc, reuses the data produced by HappyShop (e.g. order details). As a result, users autonomy may still be compromised in the decentralized Web, and can lead to many issues related to data misuse [8, 35, 39] or loss of trust [13].

Facing these challenges, we propose the concept of "perennial" Data Terms of Use (DToU), characterizing a formal language model that addresses the following **challenges** in a decentralized Web context: **C1**) expressing data provider's DToU for their data; **C2**) imparting application's (developer's) DToU on how they handle data; **C3**) performing compliance checking over data usage requests; **C4**) supporting DToU policy reusing across applications and data providers; **C5**) facilitating apt DToU-compliant cross-application data sharing. With such a language model, automated reasoning can be performed thus only exposing distilled important information to users, reducing the amount of information and numbers of decisions exposed to the user, thus incentivizing responsible handling of Terms of Use. It is called "perennial" because a stakeholder only

needs to specify the DToU once in the beginning, and it can be reused across activity cycles and across stakeholders, just like the plants being implanted once and kept growing in the future.

The *perennial* concept is in contrast to *sticky policy* [22, 26] which proposed principles for supporting distributed DToU compliance, but, in the core definition, only explicitly discussed requirements 1 and 3. *Sticky policy* supports DToU-compliant cross-application data sharing, but suggests the policy staying the same regardless of data processing history, thus is not *apt*, leading to the potential of frequent user disruption. Existing research on policy languages, often coined as access control [27] and usage control [19, 32], also proposed formal models for expressing certain parts of DToU. There are different flavours of them, targeting at different scenarios, with diverse properties and capacities. As we will introduce in Sec 2, they provide many design principles and concepts that are useful across contexts, but impose their individual limitations.

In this paper, we propose a *perennial* policy language that addresses challenges from decentralization while maintaining expressiveness, by using heterogeneous yet interoperable data policy and application policy. The language and reasoning mechanism are built on semantic technologies, namely Turtle [5], Notation 3 [6] and RDF Surfaces [12]. This facilitates the creation of a common vocabulary, and enables integration with other semantic tools, particularly ontologies and ontological reasoning. Furthermore, our approach is integrated with Solid, a decentralized user-focused Web architecture, and we evaluate its performance across various workloads[1]. To the best of our knowledge, we are the first work proposing a (semantic) perennial policy language that supports stakeholders expressing their DToU in the context of the decentralized Web. We believe this work provides a good starting of the paradigm shift for enhancing user autonomy and control.

2 RELATED RESEARCH

Several explorations have delved into the utilization of computer-interpretable formal encoding of policies to enable (semi-)automated decision-making of data usage authorization. These range from the classical access control to more advanced dynamic usage descriptions. In this section, we examine this body of research and discusses their relevance to a decentralized Web context. The main features of several closely-related policy languages are summarized in Table 1 (see Appendix A for further details of term explanation), and this section discusses their general properties. Where relevant, we will refer to the challenges or concepts shown in the table.

The most well-known line of research involves various access control models, each based on different principles. For example, models such as Mandatory Access Control (MAC) [33], Access Control List (ACL), Role-Based Access Control (RBAC) [31] or Attribute-Based Access Control (ABAC), can be based on information ranging from narrower details like user identity to broader categories such as user groups, and contextual information. Several languages uses or implements them, such as E-P3P [18], WAC [4] and P2U [16]. There is also research that falls in the middle ground, such as Label-Based Access Control (LaBAC) [7], a simplified variant of ABAC while more expressive than MAC and RBAC.

Other policy languages may also use some concepts from access control models, with additional features, such as eXtensible Access Control Markup Language (XACML) [1], a widely-known XML-based standard for ABAC with additional constructs like obligations for cloud services, and Open Digital Rights Language (ODRL) [2], an expressive policy language serializable to JSON-LD (also to be discussed later). Thoth [9], on the other hand, uses its own logic-like policy language to express policies not only for read action, but also for write, update and declassification actions. It permits intricate evaluations of formulae that involve rich operators and the incorporation of external sources of information during policy evaluation.

These approaches vary in terms of expressiveness and usage contexts. However, they tackle the problem either purely relying on the data providers, or from a holistic view requiring a 'supervisor'. In both settings, they require the policy author with abundant knowledge to actively compile and maintain policies during personnel changes (e.g. adding a new application, or assigning a user/operator with a new role). While this assumption is reasonable for managing data usage at an institutional level, it presents challenges in contexts with multiple independent stakeholders that engage and disengage dynamically (esp. **C2**, **C4** and **C5**), as is the case in a decentralized Web. For example, they can only define the policy *after* a user like Alice has determined whether a shopping app like HappyShop should be granted permission or not, rather than supporting proactive decisions.

In contrast, some research recognizes the dynamic nature of data and applications, and offers different solutions. LoNet [14], for example, employs Information Flow Control (IFC) for RBAC, along with conditional predefined policy evolution (**C5**). It tries to address the expressiveness through the so-called meta-code (an external script) for checking additional policy information like time. However, the meta-code is arbitrary code and difficult to statically verify. CamFlow [25], on the other hand, borrows concepts from Decentralized IFC (DIFC) [23], and employs homogeneous policy constructs (tags and labels) to encode both the data policy and application capability, and checks their compatibility (**C2** & **C4**). It expresses policy evolution (**C5**) by adding or removing tags for output (policy) based on input (policy). Smart object [29] uses complex constructs to define permitted and prohibited use of data, and combines this with the application information (using relational calculus) (**C2**) to derive policies for output data (**C5**), assuming and leveraging a tabular structure of data. Dr.Aid [28] focuses on the context of data-intensive scientific workflows, employing different structures to express data rules and process rules (**C2**). It supports policy reasoning and derivation for workflow graphs composed of multi-input-multi-output processes (**C5**). A common feature among these approaches is the separation of data policy from application policy. They have demonstrated that this separation allows the reasoner to verify if an application complies with the data policy, aligning with our intended goal. However, they often have relatively limited expressiveness compared to the earlier research.

There are also policy languages that utilize semantic technologies or are tailored for specific use cases within the decentralized Web.

[1]See our repo https://github.com/OxfordHCC/solid-dtou for the source code. Experimental logs and other information are in the supplementary artifact at https://doi.org/10.5281/zenodo.10685603.

Table 1: Summarization of features under different categories of related policy languages.

C1 - C5 refers to the Challenges 1 - 5 identified for perennial language. For column C1, A means authorization, O means obligation, and + means additional features. For columns C3 & C4, ✓ means true, ☑ means true if using same environmental information schema. For columns C2, C5 and Condition, A means application, C means capacity, D means data type / category, E means environmental information, ***E*** means entire environmental information exposed in knowledge base, I means multi-input, M means use mode, O means multi-output, P means purpose, T means transformation, U means user, X means external information.

Language	C1	C2	C3	C4	C5	Condition	Policy Author	Format
E-P3P[18]	AO		✓	☑		EPU	Supervisor	Custom
XACML[1]/ODRL[2]	AO+		✓	☑		EPU	Owner	XML/JSON-LD
WAC[4]/ACP[3]	A		✓			AU/AUX	Owner	Turtle
P2U[16]	A		✓			AU	Owner	XML
LaBAC[7]	A		✓			AU	Supervisor	Custom
AIR[17]	A		✓	☑		***E***U	Owner	N3
Thoth[9]	A		✓			EUX	Owner	Logic-like
Eddy[8]	AO	MP		☑	T	MPU	Supervisor	OWL-DL
LoNet[14]	A		✓	✓	T	EU	Supervisor	Custom
CamFlow[25]	A	C	✓	✓	T	C	Owner	Custom
Smart object[29]	A	DIP	✓	✓	T	DP	Owner	Custom
Dr.Aid[28]	AO	IP	✓	✓	OT	EPU	Owner	Custom
This paper	AO	CDIP	✓	✓	OT	CEPU	Owner	Turtle

One notable example is ODRL [2], which offers a comprehensive set of concepts, including permissions, prohibitions, obligations, remedies, conditions, purposes and agents. However, originated as a data right expression language, ODRL primarily focuses on expressing what is (not) permitted for the data, and lacks a corresponding mechanism for expressing application information. While efforts are being made to address this issue [10], there is still uncertainty regarding the eventual resolution. The AIR [17] policy language, based on Notation 3 (N3) [6], specifies permitted and prohibited actions for data processing. It allows for the expression of custom rules directly on the contextual knowledge base of data processing. However, in the absence of general agreements across applications, this requires the policy author to have the knowledge of information and structure of the knowledge base, resulting in a strong coupling with the application. Eddy [8] examined real-life Terms of Use from online services and proposed a policy language based on OWL-DL to express key data requirements, including permissions, obligations and prohibitions. The language distinguishes between different use modes (collect, use, retain and transfer) and demonstrated compliance checking between two policy sets of two services. However, there is no clear demonstration of how data providers can utilize this language, as it necessitates highly detailed descriptions. Solid, a decentralized Web architecture based on Linked Data, has its own policy languages, mainly for access control purposes. This includes WAC [4], a policy language for expressing ACL, and ACP [3], an advanced policy language that supports a broad range of conditions, such as Verifiable Credentials. WAC and ACP leverage contextual information exposed by Solid but are less expressive compared to ODRL and AIR.

In summary, the various policy languages exhibit different features tailored for their specific use cases. Among them, we find the second category of research, which separates data policy from application policy, to be the most suitable for our intended design context. However, they are not directly applicable to our context, primarily due to limitations in their expressiveness. Our research takes inspiration from them, addresses such limitations, and provides better expressiveness in an extensible way.

3 A PERENNIAL POLICY LANGUAGE

3.1 Language design

Broadly speaking, our language model consists two parts: the *data policy* and the *app(lication) policy*. This design is crucial to enable automated data access negotiation between a data owner and a data consumer (e.g. applications). The *data* policy enables data providers to define policy-related metadata and expectations for the data consumers. The *app* policy allows app developers to encode the promises and expectations for accessing the data by the application.

The reasoner performs three types of tasks: a) conformance check: deciding whether the application can use the data; b) obligation check, assessing what obligations are activated by the application; and c) policy derivation: determining the policy for output data and saving a data owner to define data usage policy for derived data. Figure 1 gives an overview of the language and the relation between different concepts.

In the following, we provide more detailed descriptions about the language, with all examples expressed in Turtle[2] [5].

3.1.1 *Data Policy.*

Conceptually, the data policy consists of two layers: attributes and semantics. Attributes form the base layer, expressing *information* about the data and/or the policy. On top of that is the semantic layer, providing semantic concepts like tags, prohibitions and obligations, while referencing information expressed in the attribute layer. In the Turtle syntax of the policy encoding, each statement/tuple is expressed as a node with corresponding type and properties.

More precisely, each *attribute* is a simple tuple: `(name, class, value)`. While they alone do not carry semantics for the policy, they

[2] For simplicity, we omit the prefixes, and only use `:` for named nodes.

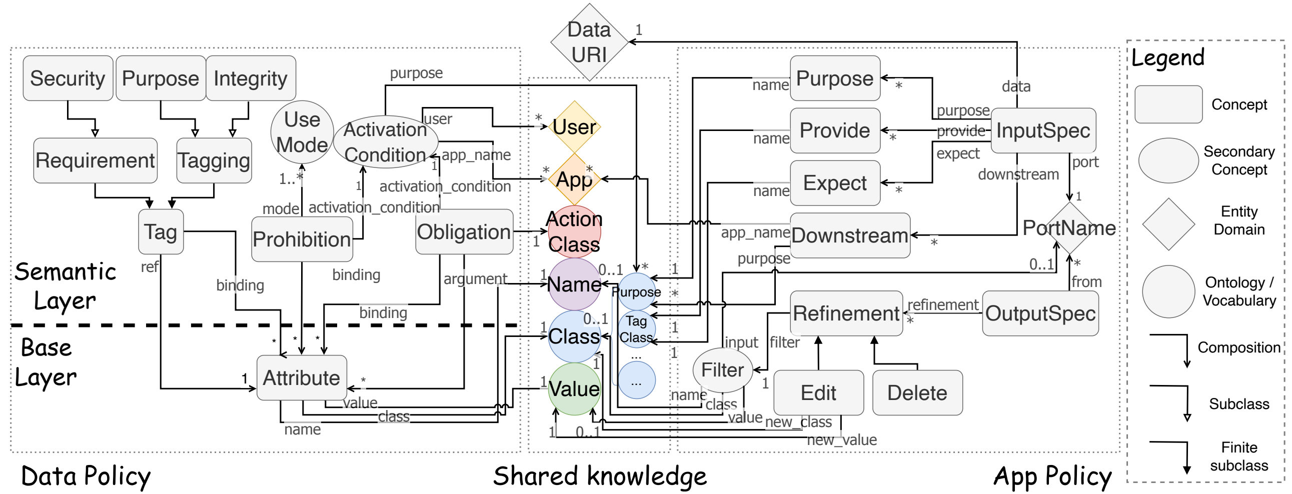

Figure 1: Language design and relation between concepts

are designed with a flexible structure to describe information about data and policy. For example, attributes can be used to define Alice as an author, e.g. `(:author, :string, "Alice")`, or encode a textual description (e.g. `(:ack-text, :string, "This dataset is by Alice")`), or specify the data fields (e.g. `(:col-2, :field, :column-2)`). This concept of *attributes* is borrowed from Dr.Aid [28], which has been demonstrated as a powerful structure for supporting policy derivation. The following example shows how Alice defines her email address information as an attribute – the attribute is a kind of 'string' and has the value of "alice@a.b".

```
:attr1 a :Attribute;
  :name :alice-email;
  :class :string;
  :value "alice@a.b".
```

Tags, from the semantic layer of our model, can be used to specify the requirements or available resources. Two typical categories of tags are *security* and *integrity*: *security* specifies which security clearance an application needs to possess to use the data; and *integrity* identifies the integrity level(s) this dataset has, to be requested by the application (policy). Inspired by CamFlow [25] (see Sec 2), tags can be used to facilitate *capacity* of conformance check, see later Sec 3.2.2.

The descriptor of a tag (i.e. which tag it is) is associated with the `class` of an *attribute* using `attribute_ref`; other information in the *attribute* is not used by a tag. Our policy language also allows expressing additional categories of tags, such as *Purposes*.

For example, the following excerpt (from Appendix B.1) shows how *tags* can be used to define that Alice requires any application to respect `banking` security level (to use payment info):

```
:tag2 a :SecurityTag;
    :attribute_ref :attr-tag2;
    :validity_binding :attr2.

:attr-tag2 a :Attribute;
    :name :tag-2;
    :class :banking;
    :value :nil.
```

```
:attr2 a :Attribute;
    :name :det;
    :class :data-content;
    :value :payment-details.
```

This example also contains a *validity binding* for that tag, meaning that the validity of the tag is dependent on the existence of the referenced attribute(s), i.e. `:attr2`, denoting the exact content of `:payment-details`.

Prohibitions specify additional restrictions on the data consumer, independent of the capacity (i.e. *tags*) of the application. A prohibition contains an *activation condition*, which is a matcher against the context, e.g. user, application and purpose. If the condition matches the usage context, the usage is deemed prohibited. *Validity bindings* can also be specified. Additionally, our model also offers an extension point, allowing users to specify the `:mode` when a match is considered. Currently, we only support the `:Use` mode, denoting the reading or processing of the data. We plan to explore other types of use modes to be integrated.

For example, Alice dislikes a payment processor, `<http://duckpay.com/>`, and does not want it to use her payment info, either directly or indirectly. This can be encoded as:

```
:pr1 a :Prohibition;
    :mode :Use;
    :activation_condition
        [ :app_name <http://duckpay.com/> ];
    :validity_binding :attr2.
```

An *obligation* denotes a potential obligation that will be triggered under certain conditions. Its core is an *obligation definition*, specifying what the triggered obligation will be, and an *activation condition*, specifying the condition to trigger this obligation. An *obligation definition* contains an obligation class, and a list of arguments – references to attributes. Upon activation/instantiation of the obligation, the actual value of these attributes will be used, rather than the reference. The *activation condition* and *validity bindings* are the same as those for *prohibitions*.

The following example for shoe size shows that Alice expects any application to email her when used for research purposes:

```
:ob1 a :Obligation;
    :obligation_class :send-email;
    :args (:attr1);
    :activation_condition [ :purpose :research ].
```

Policy set. Finally, the policy terms explained above should be put together as a policy set. A policy set contains a `Policy` node which contains the relevant policy terms, and a `Data` node which pairs the policy and the data IRI that this policy applies to. For example, the policy set for Alice's payment information looks like:

```
:data-payment a :Data;
    :uri <http://a.b/payment-info>;
    :policy :policy-1.

:policy-1 a :Policy;
    :attribute :attr-tag2, :attr-tag3,:attr-tag4, :attr2;
    :security :tag2;
    :purpose :tag3, :tag4;
    :prohibition :pr1.
```

3.1.2 *Application Policy.*

An *app policy* contains basic information about the application, and the policy specification for the inputs and outputs. In addition, it also specifies relevant *downstream* data consumers (e.g. third-party APIs to send data to) that this application will use.

An application may take multiple inputs, and thus multiple *input specifications* (`:InputSpec`). Normally, each input specification describes basic information about that input (*name* and *data*) and its *capacity*: the *tags* it conforms to, including the *security* levels, the *integrity* it expects, and the *purpose* it will use the (input) data for. If the application sends the input data to an external location for processing (e.g. an API call), a *downstream* should be specified as a simplified app policy for that downstream stakeholder, specifying the (app) *name*, *user* and *purpose* of that *downstream*[3].

For example, HappyShop states this input specification for reading the payment information:

```
:input1 a :InputSpec;
    :data <http://a.b/payment-info>;
    :port [:name "payment-info-in"];
    :security :banking;
    :purpose :making-payment;
    :downstream [
        :app_name <http://goodpay.com/>;
        :purpose :making-payment ].
```

It says an input port named `"payment-info-in"` reads data from `<http://a.b/payment-info>`, and promises to comply with security level `:banking`, and will use the data only for purpose of `:making-payment`; it will send the data to a downstream, named `<http://goodpay.com/>`, for purpose of `:making-payment`. It is compatible with Alice's data policy for payment info.

Similarly, an application may wish to store data into user's Pods, so it may need to specify multiple *output specifications*. Apart from the *name*, each *output specification* describes the related input that this output data is derived *from*, and the *refinements* that the data policies are subject to.

The *from* statement, (during reasoning) associates the output with a set of data policies, each pertinent to the input used to derive the output. In a higher level, this reflects the general information flow in the application. The removal or change of information is captured by a *refinement*, which expresses how (the *attributes* of) the data policies should be modified to reflect the processing that has been applied to the data. Two types of *refinements* are supported: *delete* and *edit*. A *delete* has a *filter*, meaning that all attributes matching the *filter* should be deleted, and the *filter* is used to specify the matching *attribute* information: *name*, *class* and *value*. Similarly, an *edit* means that any *attribute* matching the filter will be assigned a new class and value.

The following example output specification shows how HappyShop may record purchase histories in user's Pod, which contains derived data (the copy) of the delivery address, and a declassified version of payment details:

```
:out1 a :OutputSpec;
    :from [ :name "address-in" ],
        [ :name "payment-info-in" ];
    :refinement :refine-no-payment-details.

:refine-no-payment-details a :Delete;
    :filter [
        :class :data-content;
        :value :payment-details ].
```

This policy snippet means this output is related to the data from the inputs named `"address-in"` and `"payment-info-in"`, and has one refinement, which will delete all attributes of class `:data-content` and value `:payment-details`, reflecting a fact that the output data will not contain payment details even though it uses payment data.

The *refinements* directly operate on *attributes*, but references to attributes (*bindings*) in the semantic layer ensure that all operations will be inferred to related semantic concepts too. For instance, if an *attribute* is deleted, any *tags*, *prohibitions* and *obligations* that have a *binding* to this *attribute* will also be deleted.

Similar to data policy, the app policy statements should be put together as a policy set. Please refer to Appendix B.3 for example.

3.1.3 *Shared vocabulary.*

It is worth noting that shared vocabularies are assumed in all related research, with different levels of difficulties to achieve. In our work, this is achieved by making concepts explicit and using URIs/IRIs. This allows easy and decentralized provision of them, such as using OWL ontologies [38] as vocabularies. There are five sorts of vocabularies to be shared, as seen in the middle of Figure 1. They are centred around the data provider's wills, and thus is most natural to be provided by them, or some intermediaries. We mainly identify this mechanism, and leave this to the practitioners to consolidate the vocabularies. Relatedly, OWL reasoning may be integrated and performed before policy reasoning to maximize interoperability.

3.2 Reasoning

Our language accommodates three types of reasoning tasks: conformance check, obligation check, and policy derivation. This section explains the reasoning mechanism in more details.

In general, the reasoning rules can be expressed using first-order logic, encoded using RDF Surfaces [12] and Notation 3 (N3) [6] in our implementation. N3 is a language supporting the expression of both semantic data and reasoning rules with a rich syntax; RDF Surfaces, building on N3, provides an (easy) translation of first-order logic (FOL), including representing negations. Some of these rules

[3]The app developer should verify that the downstream capacity is in line with that of the input specification, so its tags do not need to be explicitly expressed.

contain explicit negations, which are implemented using N3 built-ins like `log:collectAllIn`, enabling scoped negation-as-failure (SNAF) [6]. For the sake of brevity, we only introduce the foundational principles here, and refer the reader to Appendix C for the axioms.

3.2.1 *Context preparation.*

Before embarking on the three reasoning tasks, it is essential to inject the contextual information and link the application policy with data policy.

The contextual information (see e.g. Appendix B.4) specifies the relevant app policy *app_pol*, the *user* and the *time* of data usage. These parameters are only known during actual application requests and should be provided to the reasoner dynamically each time. Each reasoning task involves a distinct *UsageContext*.

Furthermore, all relevant policy content is added to the same knowledge base. This leverages the fact that Turtle is a sub-language of N3, making all policy specifications in Turtle valid N3 statements. The linkage between data policy and app policy is established by identifying the data URIs as specified in their respective fields. This is achieved through our reasoning rules, removing the need for additional injection of data policy into the corresponding inputs.

3.2.2 *Conformance check.*

The fundamental purpose of our policy language is to determine whether a data usage should be permitted, i.e. conformance checking. In our language, there are three types of conflicts to be checked: unsatisfied requirements, unmatched expectations, and prohibited uses. (See Appendix C.2.)

An *unsatisfied requirement* occurs when a *requirement tag* (e.g. security) in the data policy is absent in the app policy. Therefore, the reasoning process involves determining all corresponding inputs and data, and verifying their requirement tags.

Conversely, an *unmatched expectation* arises when a *tag expectation* (e.g. integrity) in the app policy is missing in the data policy. This task also covers the verification of whether all purposes are permitted. Both the reasoning about *unsatisfied requirement* and *unmatched expectation* require a 'closed-world assumption,' as it is essential to determine if a tag does not exist. Because the policy documents are as the sole reliable source of information, we utilize SNAF provided by N3 built-ins.

A *prohibited use* is identified when a prohibition is triggered. Prohibitions have their own semantics, including activation conditions, and should be checked in accordance with these conditions.

3.2.3 *Obligation check.*

Our policy language can also reason about the obligations triggered during data use. This process is similar to checking prohibitions and involves verifying the *activation conditions.* But because obligations contain arguments that are references to attributes, values of these attributes also need to be returned from the query for further assessment in application logic. (See Appendix C.3.)

3.2.4 *Policy derivation.*

When the application produces output data (i.e. storing data to users' Pods), policy derivation becomes crucial to produce derived data policy for the output, based on the output specification. This aspect is central to (addressing challenge 5 of) the *perennial* nature of our language. Policy derivation involves merging the data policies from all corresponding `from` inputs and performing *refinements*, for each corresponding output port. (See Appendix C.4.)

The derivation of *output attributes* is a primary focus because these attributes are vital for handling output policies for the semantics layer, particularly for bindings. Because of *refinements*, each input attribute will either have a copy, be edited, or cease to exist in the output. This process creates new nodes for the output attributes in the RDF graph, corresponding to the existential quantifier in the conclusions. The linkage between the input and output attributes is also recorded, which will be used for policy derivation of the semantic layer. SNAF plays a crucial role here as it determines what 'happens' to the rest of the input attributes that do not have a matching *refinement* – there should be an output attribute with identical name, class and value.

The *tags* in an output are based on the collection of all tags of related inputs, while removing those tags with deleted attribute bindings. Because tags have categories, they can be treated uniformly when reasoning about their existence, and the reasoner can make use of the categories afterwards.

The *output obligations* and *output prohibitions* are derived similarly. We first check if any binding is deleted, like for *output tags.* If not, a new node for obligation (or prohibition) is created, replicating all fields of the original obligation (or prohibition) in input. For obligations, that covers the obligation definition (obligated action class and arguments), validity binding and activation condition; for prohibitions, that covers the use mode, validity binding and activation condition.

Intuitively, the attribute references (of statements in the semantics layer) for the output policies will be the corresponding output attributes, instead of the input attributes.

4 SOLID INTEGRATION

To test and demonstrate the language, we integrated the language into Solid, a decentralized Web architecture based on Linked Data that emphasizes on user autonomy. This allowed us to express *data policies* and *application policies* and perform reasoning, in a realistic context, and also serving as the foundation for our benchmark.

To achieve this, we extended the Community Solid Server [11] v6.0, a modular and extensible Solid server implementation written in TypeScript. For policy reasoning, we use an off-the-shelf reasoner, EYE [36], which supports RDF Surfaces and N3 reasoning. In particular, we used the `eyereasoner` package available on npm, which is a WebAssembly distribution of EYE with a JavaScript interface.

In the sequence diagram depicted in Figure 2, we illustrate the key components and actions related to policy reasoning for applications, assuming that the data already have DToU policies associated.

Before reasoning, the application needs to register its application policy. The DToU handler, located behind the API endpoint on the server, processes the request and creates a temporary policy record.

Subsequently, the application requests a **conformance check** before utilizing the data. The DToU handler retrieves the corresponding policy, establishes the `UsageContext`, identifies the input data from the application policy, retrieves the relevant data policy, and calls the policy engine to perform the conformance check. The results are then provided to the application for further action, such as user display. With a complete transformation to DToU (from current access control), the DToU handler shall deny usage of data without policies or in the presence of conflicts.

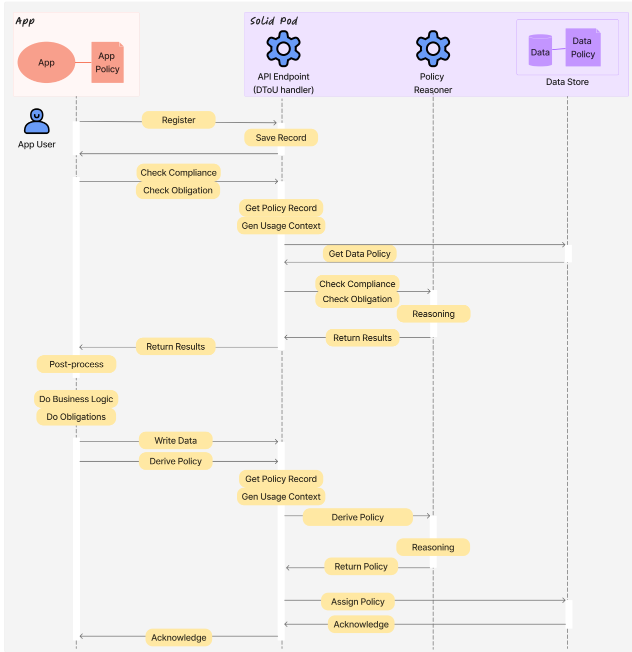

Figure 2: Sequence diagram for the Solid integration of the DToU language

Similarly, the application can request an **obligation check** and then processes the information accordingly, including displaying it to the user at the appropriate time or automatically fulfilling obligations when applicable. Our language distinguishes between `UserObligation` and `ProcessObligation` classes to facilitate this distinction, leaving the specific obligations up to policy authors.

Finally, when the application intends to write output data to the Pod, it should send a **policy derivation** request along with the data. The DToU handler will then perform policy derivation and store the policy with the data. This allows subsequent applications to automatically carry out DToU reasoning when they request this data as input. As mentioned, the DToU handler may deny usage of data without policies, thus effectively enforcing policy derivation – if it does not perform it, the stored data will not be usable.

DToU reasoning is performed by the policy engine and DToU handler in the modified Solid service, reducing the burden on the application developers and Pod owners. From the Pod owner's perspective, supporting DToU requires expressing relevant data policies for input data. For the applications, DToU support involves preparing app policies and sending policy-related requests to the (modified) Solid service while also handling responses. These application policies can be statically attached to the project, or dynamically generated from a template, offering flexibility for different users. The optimal method for authoring app and data policies remains an open question, outside the scope of this paper.

5 PERFORMANCE BENCHMARK

We conducted benchmark tests to evaluate the performance of our integration and to gain insights into its scalability across different reasoning tasks and workloads. This section presents the results and our discussion.

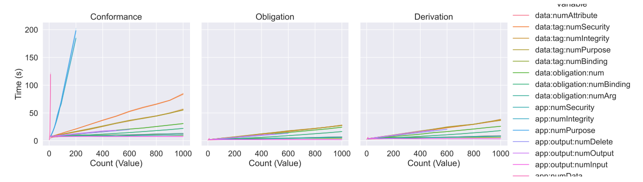

Figure 3: General benchmark results of different workloads

5.1 Benchmark settings

The benchmark encompasses a wide range of workloads of incorporating key variables: the number of different terms in the data policy and app policy. We varied these numbers, ranging from 10 to 1000, while keeping other variables at a fixed value of 10. There are two exceptions that are not fixed to 10: the number of attributes, which is set to 100, and the number of inputs, which is limited to 4. We chose these values because attributes require references to them, and handling only 10 would not suffice for distribution among tags, prohibitions and obligations; the number of inputs significantly impacts performance, so we opted for a smaller yet reasonable number. Policies are generated through a random process.

It is worth noting that the range we benchmarked should be demonstrable to what usually exists in current real-world policies. For example, [8] reviewed Facebook, Zynga and AOL policies and identified a total of 131, 190 and 75 statements. Although not equitable, each statement roughly corresponds to one term in the semantic layer and several attributes, or one term in the app policy.

For the benchmark, we utilized the WebAssembly distribution of the eyereasoner v6.9.5, which is available on NPM. We started the server using the provided file-storage configuration, without special parameters. The benchmark tests were conducted on a consumer-level laptop, with an Intel Core i5-1135G7 (2.4 GHz) and 16GB of RAM, running on Linux kernel 6.1.55 (x86_64). Each workload was repeated 10 times, and the time taken from sending requests to receiving results was recorded.

5.2 Results and discussions

5.2.1 *General results.*

Figure 3 shows the primary results of our benchmark. Most variables exhibit a linear or sublinear scaling trend. However, some variables do not reach 1000 because they encountered time-out or connection resets with further increase.

Names of variables correspond to their specific workload. For example, `app:output:numDelete` means the variable being changed is the number of delete refinements in each output specification of the app policy.

The three specific variables, namely `app:numData`, `app:numPurpose` and `app:numIntegrity`, displayed exceptional growing trends in conformance checking, resulting in time-outs, which we will discuss separately later. After removing these exceptional cases, and subtracting the base time for policy loading and general reasoning (e.g. axioms for `rdfs:subClassOf`), we obtain Figure 4. It demonstrates a clearer linear growth trend in the remaining variables.

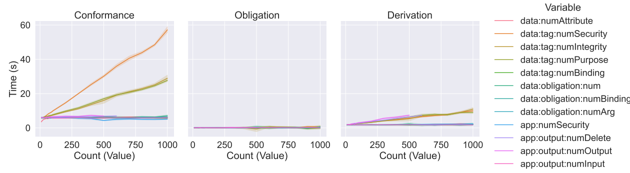

Figure 4: Selective benchmark results for most variables, subtracting the base reasoning time for general axioms

Figure 5: Experiment results for different conformance checking tasks for exceptional variables, `app:numData`, `app:numPurpose` and `app:numIntegrity`

Among them, most variables reached 1000, except for `app:output:numDelete` and `app:output:numOutput`, which stopped at 600 and 500. This limitation was due to connection resets, caused by the server encountering JavaScript out-of-heap errors during the 10 repetitions. This indicates a potential area for optimization of the memory consumption. However, it is worth noting that having 500 output ports or 600 delete refinements per output port (thus 6000 in total) for an application is exceptionally large and likely not an issue in realistic scenarios.

5.2.2 *Unmatched expectations.* Now, regarding the exceptional growths, as shown in Figure 3, two variables, `app:numPurpose` and `app:numIntegrity`, correspond to the checking of unmatched expectations. We further performed additional experiments to understand the time spent on different conformance checking (sub-)tasks, as shown in Figure 5. It verified the intuition that the time was mainly consumed by checking the unmatched expectations.

However, it is interesting to note that the time for its symmetric task, unsatisfied requirements, did not exhibit the same exceptional growth, as seen in Figure 3 for variable `data:tag:numSecurity` and `app:numSecurity`. This suggests that the issue is likely not due to a mis-implementation of our axioms, but rather a performance bottleneck for RDF Surfaces or the EYE reasoner related to specific rule combinations and orders. Ideally, future work should explore and address this issue, with the potential to further optimize by reducing the coefficient for tag matching.

5.2.3 *Number of data / inputs.* Changing the number of data inputs led to the most significant growth in reasoning time, displaying a growth pattern that appears higher than linear, as shown in the sub-figure for unmatched expectations in Figure 5. This could be explained because all reasoning tasks depend on the pairing of data policies and inputs, making the complexity increase faster with the number of data inputs. However, it is interesting to note that obligation checking and policy derivation did not pose the same exceptional growth trend as conformance checking in Figure 3. A plausible explanation is that conformance check involves a heavier workload, making this effect more pronounced. This is supported by the fact that the time spent on a more complex subtask, such as unmatched expectation, also grew faster than that of unsatisfied requirements and prohibited use, as reflected in Figure 5. This suggests that our usage of RDF Surfaces may be suboptimal and should be addressed and optimized in future work.

5.3 Conclusion

From our benchmark, it is clear that the reasoning cost for all tasks shows a linear growth concerning all but one factor. It is important to note that DToU policy reasoning is not performed frequently (typically three possibilities in an application's lifecycle), so real-time performance is not a strict requirement. However, our result highlight the significant potential for deploying our DToU language on a large scale with a substantial volume of policies. Additionally, we delved into the specifics of the suboptimal results and proposed potential reasons for their behaviour. In a production system, it is essential to focus on optimizing the reasoning tasks and the underlying reasoner to ensure efficient and reliable performance.

6 SUMMARY AND FUTURE WORK

In this work, we have undertaken a comprehensive exploration of the challenges and advantages associated with introducing DToU into a decentralized Web context, exemplified by Solid, with the overarching aim of enhancing user autonomy. Our efforts have included identifying the pertinent challenges and benefits, delineating the specific requirements for a policy language within this context, and introduced our own DToU policy language, which is based on semantic technologies. We have also detailed the design of data policies, application policies and the underlying reasoning mechanism. Furthermore, we showed how our solution integrates with Solid, along with benchmark tests that assessed the scalability of our implementation, highlighting its potential for wider adoption with a substantial volume of policies. We discussed areas where further optimization is required.

The next step of our work involves evaluating the language expressiveness and its understandability for users. We are also interested in exploring simpler methods for policy authorization, which could potentially involve leveraging NLP technologies. In general, our work underscores a paradigm shift in how application may be selected, permission granted, and interoperability achieved. This shift is driven by the automated reasoning of *perennial* DToU policies. It points out a wide spectrum of challenges and opportunities, including but not limited to enhancing the language expressiveness, improving usability, effectively maintaining policies, and optimizing performance. Effectively addressing these social-technical challenges will require interdisciplinary collaboration, and we are committed to contributing to this ongoing effort.

ACKNOWLEDGMENTS

This research was supported by the Oxford Martin School Programme for Ethical Web and Data Architectures in the Age of AI at the University of Oxford. Special thanks to Nigel Shadbolt, Tim Berners-Lee and Ruben Verborgh for their engagement and comments in early discussions of this work.

REFERENCES

[1] 2013. eXtensible Access Control Markup Language (XACML) Version 3.0. https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[2] 2018. ODRL Information Model 2.2. https://www.w3.org/TR/odrl-model/

[3] 2022. Access Control Policy (ACP). https://solid.github.io/authorization-panel/acp-specification/

[4] 2022. Web Access Control. https://solid.github.io/web-access-control-spec/

[5] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. 2014. RDF 1.1 Turtle. https://www.w3.org/TR/turtle/

[6] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. 2008. N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming* 8, 3 (May 2008), 249–269. https://doi.org/10.1017/S1471068407003213 Publisher: Cambridge University Press.

[7] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. 2016. Label-Based Access Control: An ABAC Model with Enumerated Authorization Policy. In *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control (ABAC '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2875491.2875498

[8] Travis D. Breaux, Hanan Hibshi, and Ashwini Rao. 2014. Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements. *Requirements Engineering* 19, 3 (Sept. 2014), 281–307. https://doi.org/10.1007/s00766-013-0190-7

[9] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. 2016. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, Berkeley, CA, USA, 637–654. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/elnikety

[10] Beatriz Esteves, Harshvardhan J. Pandit, and Víctor Rodríguez-Doncel. 2021. ODRL Profile for Expressing Consent through Granular Access Control Policies in Solid. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 298–306. https://doi.org/10.1109/EuroSPW54576.2021.00038 ISSN: 2768-0657.

[11] Joachim Van Herwegen, Ruben Verborgh, Ruben Taelman, Thomas Dupont, Matthieu Bosquet, Mend Renovate, Jasper Vaneessen, smessie, Arthur Joppart, wkerckho, Simone Persiani, Wouter Termont, Michiel de Jong, Noel De Martin, Stijn Taelemans, Wout Slabbinck, Jesse Wright, jaxoncreed, surilindur, zg009, Aaron Coburn, Adler Faulkner, Brandon Aaron, Charlie Blevins, Dylan Van Assche, Emelia Smith, Freya, and Gertjan De Mulder. 2023. CommunitySolid-Server/CommunitySolidServer. https://doi.org/10.5281/zenodo.8410285

[12] Patrick Hochstenbach, Jos De Roo, and Ruben Verborgh. 2023. RDF Surfaces: Computer Says No. http://arxiv.org/abs/2305.08476 arXiv:2305.08476 [cs].

[13] Jina Huh-Yoo and Emilee Rader. 2020. It's the Wild, Wild West: Lessons Learned From IRB Members' Risk Perceptions Toward Digital Research Data. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (May 2020), 059:1–059:22. https://doi.org/10.1145/3392868

[14] Håvard D. Johansen, Eleanor Birrell, Robbert van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. 2015. Enforcing Privacy Policies with Meta-Code. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys '15)*. ACM Press, Tokyo, Japan, 1–7. https://doi.org/10.1145/2797022.2797040

[15] Iulia Ion, Niharika Sachdeva, Ponnurangam Kumaraguru, and Srdjan Čapkun. 2011. Home is safer than the cloud! privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security (SOUPS '11)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/2078827.2078845

[16] Johnson Iyilade and Julita Vassileva. 2014. P2U: A Privacy Policy Specification Language for Secondary Data Sharing and Usage. In *2014 IEEE Security and Privacy Workshops*. 18–22. https://doi.org/10.1109/SPW.2014.12

[17] Lalana Kagal, Chris Hanson, and Daniel Weitzner. 2008. Using Dependency Tracking to Provide Explanations for Policy Management. In *2008 IEEE Workshop on Policies for Distributed Systems and Networks*. 54–61. https://doi.org/10.1109/POLICY.2008.51

[18] Günter Karjoth, Matthias Schunter, and Michael Waidner. 2002. Platform for Enterprise Privacy Practices: Privacy-Enabled Management of Customer Data. In *Privacy Enhancing Technologies (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 69–84. https://doi.org/10.1007/3-540-36467-6_6

[19] Aliaksandr Lazouski, Fabio Martinelli, and Paolo Mori. 2010. Usage control in computer security: A survey. *Computer Science Review* 4, 2 (May 2010), 81–99. https://doi.org/10.1016/j.cosrev.2010.02.002

[20] Aleecia M McDonald and Lorrie Faith Cranor. 2008. The cost of reading privacy policies. *Isjlp* 4 (2008), 543. Publisher: HeinOnline.

[21] Christian Meurisch, Bekir Bayrak, and Max Mühlhäuser. 2020. Privacy-preserving AI Services Through Data Decentralization. In *Proceedings of The Web Conference 2020*. Association for Computing Machinery, New York, NY, USA, 190–200. http://doi.org/10.1145/3366423.3380106

[22] M. C. Mont, S. Pearson, and P. Bramhall. 2003. Towards accountable management of identity and privacy: sticky policies and enforceable tracing services. In *14th International Workshop on Database and Expert Systems Applications, 2003. Proceedings.* 377–382. https://doi.org/10.1109/DEXA.2003.1232051

[23] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 129–142. https://doi.org/10.1145/268998.266669

[24] Jonathan A. Obar and Anne Oeldorf-Hirsch. 2020. The biggest lie on the Internet: ignoring the privacy policies and terms of service policies of social networking services. *Information, Communication & Society* 23, 1 (Jan. 2020), 128–147. https://doi.org/10.1080/1369118X.2018.1486870

[25] Thomas F. J.-M. Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. 2017. CamFlow: Managed Data-sharing for Cloud Services. *IEEE Transactions on Cloud Computing* 5, 3 (July 2017), 472–484. https://doi.org/10.1109/TCC.2015.2489211 arXiv: 1506.04391.

[26] S. Pearson and M. Casassa-Mont. 2011. Sticky Policies: An Approach for Managing Privacy across Multiple Parties. *Computer* 44, 9 (Sept. 2011), 60–68. https://doi.org/10.1109/MC.2011.225

[27] Jing Qiu, Zhihong Tian, Chunlai Du, Qi Zuo, Shen Su, and Binxing Fang. 2020. A Survey on Access Control in the Age of Internet of Things. *IEEE Internet of Things Journal* 7, 6 (June 2020), 4682–4696. https://doi.org/10.1109/JIOT.2020.2969326 Conference Name: IEEE Internet of Things Journal.

[28] Rui Zhao, Malcolm Atkinson, Petros Papapanagiotou, Federica Magnoni, and Jacques Fleuriot. 2021. Dr.Aid: Supporting Data-governance Rule Compliance for Decentralized Collaboration in an Automated Way. In *The 24th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW)*. https://doi.org/10.1145/3479604

[29] Gokhan Sagirlar, Barbara Carminati, and Elena Ferrari. 2018. Decentralizing privacy enforcement for Internet of Things smart objects. *Computer Networks* 143 (Oct. 2018), 112–125. https://doi.org/10.1016/j.comnet.2018.07.019

[30] A. Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, D. Zagidulin, Ashraf Aboulnaga, and T. Berners-Lee. 2016. Solid: A Platform for Decentralized Social Applications Based on Linked Data. *MIT CSAIL & Qatar Computing Research Institute, Tech. Rep.* (2016). https://www.semanticscholar.org/paper/Solid-%3A-A-Platform-for-Decentralized-Social-Based-Sambra-Mansour/5ac93548fd0628f7ff8ff65b5878d04c79c513c4

[31] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. 1996. Role-based access control models. *Computer* 29, 2 (Feb. 1996), 38–47. https://doi.org/10.1109/2.485845 Conference Name: Computer.

[32] Ravi Sandhu and Jaehong Park. 2003. Usage Control: A Vision for Next Generation Access Control. In *Computer Network Security (Lecture Notes in Computer Science)*, Vladimir Gorodetsky, Leonard Popyack, and Victor Skormin (Eds.). Springer, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-540-45215-7_2

[33] R.S. Sandhu and P. Samarati. 1994. Access control: principle and practice. *IEEE Communications Magazine* 32, 9 (Sept. 1994), 40–48. https://doi.org/10.1109/35.312842 Conference Name: IEEE Communications Magazine.

[34] Jake M L Stein, Vidminas Vizgirda, Max Van Kleek, Reuben Binns, Jun Zhao, Rui Zhao, Naman Goel, George Chalhoub, Wael S Albayaydh, and Nigel Shadbolt. 2023. 'You are you and the app. There's nobody else.': Building Worker-Designed Data Institutions within Platform Hegemony. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–26. https://doi.org/10.1145/3544548.3581114

[35] Welderufael B. Tesfay, Peter Hofmann, Toru Nakamura, Shinsaku Kiyomoto, and Jetzabel Serna. 2018. I Read but Don't Agree: Privacy Policy Benchmarking using Machine Learning and the EU GDPR. In *Companion Proceedings of the The Web Conference 2018 (WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 163–166. https://doi.org/10.1145/3184558.3186969

[36] Ruben Verborgh and Jos De Roo. 2015. Drawing Conclusions from Linked Data on the Web: The EYE Reasoner. *IEEE Software* 32, 3 (May 2015), 23–27. https://doi.org/10.1109/MS.2015.63 Conference Name: IEEE Software.

[37] W3C. 2014. RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts/

[38] W3C OWL Working Group. 2012. OWL 2 Web Ontology Language Document Overview (Second Edition). https://www.w3.org/TR/owl2-overview/

[39] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. 2016. Automated Analysis of Privacy Requirements for Mobile Apps. In *2016 AAAI Fall Symposium Series*. https://www.aaai.org/ocs/index.php/FSS/FSS16/paper/view/14113

[40] Shoshana Zuboff. 2019. *The age of surveillance capitalism: The fight for a human future at the new frontier of power: Barack Obama's books of 2019.* Profile books.

A FURTHER ON RELATED-WORK TABLE

This section provides further explanations of the terms used in Table 1.

For column C1, authorization (A) means a reasoning outcome that can decide whether the data usage is permitted (authorized) or not (prohibited); obligation (O) means the reasoning outcome can contain obligations (pending actions) that needs to be performed due to the specified data usage; + means the language supports more types of features (that are not usually covered in other languages), such as *remedy* in ODRL.

For column C3 and C4, ✓ means the challenge is addressed by the design or feature of the language; ☑ means the challenge is addressed, but only if assuming a shared custom schema (that is not or cannot be a part of the language specification) for describing environmental information across all policies and the environment.

For columns C2, C5 and Condition, many terms should be intuitive. Specifically, capacity (C) means the properties that the data and/or the application/process need to possess to allow the data usage (e.g. security levels); E and ***E*** differs because E only refers to the *finite* (types of) environmental information covered by the language specification or its foreseeable extension, while ***E*** allows arbitrary environmental information; use mode (M) denotes the potential data usage modes (aka. codes or action verbs in [8]); multi-input (I) and multi-output (O) means the policy language possesses the mechanism to handle policies from multiple inputs and multiple outputs respectively; transformation (T) means the output policy is subject to a transformation based on the input policy and the actual processing.

B POLICY LANGUAGE EXAMPLES

This section presents the full form of the example policy snippets used in the paper.

B.1 For payment info

Alice requires any application to respect `banking` security level, and permit `make-payment` and `verify-ownership` purposes (for payment info).

```
:attr-tag2 a :Attribute;
    :name :tag-2;
    :class :banking;
    :value :nil.
:attr-tag3 a :Attribute;
    :name :tag-3;
    :class :make-payment;
    :value :nil.
:attr-tag4 a :Attribute;
    :name :tag-4;
    :class :verify-ownership;
    :value :nil.

:attr2 a :Attribute;
    :name :det;
    :class :data-content;
    :value :payment-details.

:tag2 a :SecurityTag;
    :attribute_ref :attr-tag2;
    :validity_binding :attr2.
:tag3 a :PurposeTag;
    :attribute_ref :attr-tag3;
    :validity_binding :attr2.
:tag4 a :PurposeTag;
    :attribute_ref :attr-tag4;
    :validity_binding :attr2.
```

B.2 For shoe size

This policy set specifies the policy for Alice's shoe size data:

```
:shoe-size a :Data;
    :uri <http://a.b/shoe-size>;
    :policy :policy-2.

:policy-2 a :Policy;
    :attribute :attr1;
    :obligation :ob1.
```

B.3 App policy set

```
:app-policy a :AppPolicy;
        :name <http://happy.shop>;
        :input_spec :input1, :input2;
        :output_spec :out1.

:input2 a :InputSpec;
    :data <http://a.b/address>;
    :port [ ;name "address-in" ];
    :integrity :full-address.
```

B.4 Sample usage context

An example usage context for Alice to use HappyShop may be like this:

```
:usageContext1 a :UsageContext;
        :user <http://a.b/alice#card>;
        :app [a :AppInfo; :policy :app-policy];
        :time "20230823".

:rui a :User.
```

C AXIOMS FOR REASONING

The axioms for reasoning are encoded as RDF Surfaces in our reasoner. For simplicity, we present the equivalent first-order logic axioms for performing reasoning for our language here. Because our policy language is based on Turtle thus RDF [37], knowledge is represented as triples in the ABox. We employ some conventions here for the expression in first-order logic:

(1) $A(x, SomeType)$ denotes that the (RDF) type of entity x is $SomeType$, i.e. the triple $(x, \texttt{rdf:type}, SomeType)$ exists in the ABox;
(2) $hasPredicate(x, y)$ denotes that there is a relation $predicate$ between entity x and y, i.e. the triple $(x, predicate, y)$ exists in the ABox;
(3) Other predicates starting with a capital letter refer to a shorthand, which will be explained individually;
(4) Variables are universally quantified over the scope of the entire formulae, unless explicitly quantified or stated otherwise;
(5) For clarify, we write the conclusion in the beginning, similar to a Horn clause;
(6) Existential quantified entities in the conclusions denote a new node in the ABox, which is handled nicely by RDF Surfaces;
(7) Negations denote scoped negation-as-failure, implemented using N3's built-ins;

(8) We use double-line arrows to visually distinguish between the main premise and conclusions; however, they still represent material implications, same as the single-line arrows, and are encoded in RDF Surfaces as such;
(9) Namespaces are dropped from the formulae for clarity.

C.1 Helper axioms

To simplify the axioms for actual reasoning tasks, we extract two common parts as helper axioms: $RelatedDataAppInput$ and $InputPolicyForOutput$.

The $RelatedDataAppInput(usage, data, app, input)$ identifies and groups data policy (*data*) and corresponding input specification (*input*) together, as well as the usage context *usage* and application policy *app*. Typically, in a reasoning, there is only one usage context (*usage*) and one application (*app*). Conformance checking often uses this predicate. It is defined as:

$$
\begin{aligned}
&RelatedDataAppInput(usage, data, app, input) \Leftarrow \\
&A(usage, UsageContext) \\
&\wedge hasApp(usage, apps_s) \wedge hasPolicy(app_s, app) \\
&\wedge hasInputSpec(app, input) \wedge A(data, Data) \wedge hasUri(data, uri) \\
&\wedge hasData(input, uri)
\end{aligned}
$$

The $InputPolicyForOutput(input, policy, output)$ identifies which input (specification) and its corresponding data policy is related to an output (specification). One input has only one data policy, while one output may have multiple inputs, leading to several $InputPolicyForOutput$s. It is defined as:

$$
\begin{aligned}
&InputPolicyForOutput(input, policy, output) \Leftarrow \\
&RelatedDataAppInput(usage, data, app, input) \wedge \\
&hasOutputSpec(app, output) \wedge hasFrom(output, inputPort) \wedge \\
&hasPort(input, inputPort) \wedge hasPolicy(data, policy)
\end{aligned}
$$

C.2 Conformance check

We have explained them in the main text of the document. Therefore, this part only presents the formulae, and their explanations where necessary.

C.2.1 Unsatisfied requirement.

$$
\begin{aligned}
&UnsatisfiedRequirement(t, n, input) \Leftarrow \\
&\quad RelatedDataAppInput(usage, data, app, input) \wedge \\
&\quad hasPolicy(data, pol) \wedge hasRequirement(pol, req) \wedge \\
&\quad hasCategory(req, t) \wedge hasDescriptor(req, n) \wedge \\
&\quad \neg\exists prov.(hasProvide(input, prov) \wedge \\
&\qquad\qquad hasCategory(prov, t) \wedge hasDescriptor(prov, n))
\end{aligned}
$$

The definition of $UnsatisfiedRequirement(t, n, input)$ is:

$$
\begin{aligned}
&UnsatisfiedRequirement(t, n, input) \equiv \\
&\quad \exists x.A(x, UnsatisfiedRequirement) \wedge hasCategory(x, t) \wedge \\
&\quad hasDescriptor(x, n) \wedge hasInput(x, input)
\end{aligned}
$$

C.2.2 Unmatched expectation.

$$
\begin{aligned}
&UnmatchedExpectation(t, n, input) \Leftarrow \\
&\quad RelatedDataAppInput(usage, data, app, input) \\
&\quad hasPolicy(data, pol) \wedge hasExpect(input, exp) \wedge \\
&\quad hasCategory(exp, t) \wedge hasDescriptor(exp, n) \wedge \\
&\quad \neg\exists tag.(hasTagging(pol, tag) \wedge hasCategory(tag, t) \wedge \\
&\qquad hasDescriptor(tag, n))
\end{aligned}
$$

The definition of $UnmatchedExpectation(t, n, input)$ is:

$$
\begin{aligned}
&UnmatchedExpectation(t, n, input) \equiv \\
&\quad \exists x.A(x, UnmatchedExpectation) \wedge hasCategory(x, t) \wedge \\
&\quad hasDescriptor(x, n) \wedge hasInput(x, input)
\end{aligned}
$$

C.2.3 Prohibited use.

$$
\begin{aligned}
&ProhibitedUse(u, n, p, input) \Leftarrow \\
&\quad RelatedDataAppInput(usage, data, app, input) \wedge \\
&\quad hasPolicy(data, pol) \wedge hasProhibition(pol, pro) \wedge \\
&\quad hasMode(pro, Use) \wedge hasActivationCondition(pro, ac) \wedge \\
&\quad hasUser(ac, u) \wedge hasApp(ac, n) \wedge hasPurpose(ac, p) \wedge (\\
&\quad (hasUser(usage, u) \wedge hasName(app, n) \wedge hasPurpose(input, p)) \\
&\quad \vee \\
&\quad (hasDownstream(input, ds) \wedge hasAppName(ds, n) \wedge \\
&\quad hasPurpose(ds, p)) \\
&\quad)
\end{aligned}
$$

The definition of $ProhibitedUse(u, n, p, input)$ is:

$$
\begin{aligned}
&ProhibitedUse(m, n, p, input) \equiv \\
&\quad \exists x.A(x, ProhibitedUse) \wedge hasMode(x, Use) \wedge hasUser(x, u) \\
&\quad \wedge hasName(x, n) \wedge hasPurpose(x, p) \wedge hasInput(x, input)
\end{aligned}
$$

C.3 Obligation check

$$
\begin{aligned}
&ActivatedObligation(ob, args, input) \Leftarrow \\
&\quad RelatedDataAppInput(usage, data, app, input) \wedge \\
&\quad hasPolicy(data, pol) \wedge hasObligation(pol, obl) \wedge \\
&\quad hasObligationClass(obl, ob) \wedge hasArgument(obl, args) \wedge \\
&\quad hasActivationCondition(obl, ac) \wedge hasUser(ac, u) \wedge \\
&\quad hasApp(ac, n) \wedge hasPurpose(ac, p) \wedge \\
&\quad hasUser(usage, u) \wedge hasName(app, n) \wedge hasPurpose(input, p)
\end{aligned}
$$

The definition of $ActivatedObligation(ob, args, input)$ is:

$$
\begin{aligned}
&ActivatedObligation(ob, args, input) \equiv \\
&\quad \exists x.A(x, ActivatedObligation) \wedge hasObligationClass(x, ob) \wedge \\
&\quad hasArgument(x, args) \wedge hasInput(x, input)
\end{aligned}
$$

Slightly different from the conflicts, when querying activated obligations, the corresponding attributes should be returned as well.

C.4 Policy derivation

C.4.1 Output attribute.

$$
\begin{aligned}
&OutputAttribute(n, t, v, p, attr) \Leftarrow \\
&\quad InputPolicyForOutput(input, policy, output) \wedge \\
&\quad hasPort(output, p) \wedge hasPort(input, port) \wedge (\\
&\quad (hasAttribute(policy, attr) \wedge hasName(attr, n) \wedge \\
&\quad\; hasClass(attr, t) \wedge hasValue(attr, v) \wedge \neg\exists refi, filter. \\
&\quad\; hasRefinement(output, refi) \wedge hasFilter(refi, filter) \wedge \\
&\quad\; hasInput(filter, port) \wedge hasName(filter, n) \wedge \\
&\quad\; hasClass(filter, t) \wedge hasValue(filter, v)) \vee \\
&\quad (hasAttirbute(policy, attr) \wedge hasName(attr, n) \wedge \\
&\quad\; hasClass(attr, t') \wedge hasValue(attr, v') \wedge \\
&\quad\; hasRefinement(attr, refi) \wedge A(refi, Edit) \wedge \\
&\quad\; hasFilter(refi, filter) \wedge hasInput(filter, port) \wedge \\
&\quad\; hasName(filter, n) \wedge hasClass(filter, t') \wedge \\
&\quad\; hasValue(filter, v') \wedge hasNewClass(refi, t) \wedge \\
&\quad\; hasNewValue(refi, v)) \\
&\quad)
\end{aligned}
$$

Same as above, $OutputAttribute(n, t, v, p, attr)$ is a shorthand. However, different from them, it involves multiple nodes:

$$
\begin{aligned}
&OutputAttribute(n, t, v, p, attr) \equiv \\
&\quad \exists attr'.A(attr', Attribute) \wedge hasName(attr', n) \wedge \wedge \\
&\quad hasClass(attr', t) \wedge hasValue(attr', v) \wedge \\
&\quad \exists fl.A(fl, ForwardLink) \wedge hasOrigin(fl, attr) \wedge \\
&\quad hasPort(fl, p) \wedge hasRef(fl, attr')
\end{aligned}
$$

This means that we create a node $attr'$ which is of type $Attribute$, and assign its corresponding fields; we also create a linking relation (the node fl) between the original attribute $attr$ and the created attribute node $attr'$ at output P.

In addition to that, because we will often refer to the output attribute, we define this shorthand:

$$
\begin{aligned}
&IOPair(in, out, p) \equiv \\
&\quad A(fl, ForwardLink) \wedge hasOrigin(fl, in) \wedge \\
&\quad hasPort(fl, p) \wedge hasRef(fl, out)
\end{aligned}
$$

C.4.2 Output tag.

$$
\begin{aligned}
&OutputTag(t, ar, p, tag) \Leftarrow \\
&\quad InputPolicyForOutput(input, policy, output) \wedge \\
&\quad hasPort(output, p) \wedge hasTag(policy, tag) \wedge hasCategory(tag, t) \wedge \\
&\quad hasAttributeRef(tag, ar_0) \wedge IOPair(ar_0, ar, p) \\
&\quad (\forall vb.hasValidityBinding(tag, vb) \rightarrow \\
&\qquad \exists attr.IOPair(vb, attr, p))
\end{aligned}
$$

where the full form of $OutputTag$ is:

$$
\begin{aligned}
&OutputTag(t, ar, p, tag) \equiv \\
&\quad \exists ot.A(ot, Tag) \wedge hasCategory(ot, t) \wedge \\
&\quad hasAttributeRef(ot, ar) \wedge hasPort(ot, p) \wedge \\
&\quad (\forall vb, attr.hasValidityBinding(tag, vb) \wedge IOPair(vb, attr, p) \\
&\qquad \rightarrow hasValidityBinding(ot, attr))
\end{aligned}
$$

C.4.3 Output prohibition.

$$
\begin{aligned}
&OutputProhibition(m, ac, p, pr) \Leftarrow \\
&\quad InputPolicyForOutput(input, policy, output) \wedge \\
&\quad hasPort(output, p) \wedge hasProhibition(policy, pr) \wedge \\
&\quad hasMode(pr, m) \wedge hasActivationCondition(pr, ac) \wedge \\
&\quad \forall vb.(hasValidityBinding(pr, vb) \rightarrow \exists attr.IOPair(vb, attr, p))
\end{aligned}
$$

where the full form of $OutputProhibition$ is:

$$
\begin{aligned}
&OutputProhibition(m, ac, p, pr) \equiv \\
&\quad \exists op.A(op, Prohibition) \wedge hasMode(op, m) \wedge \\
&\quad hasActivationCondition(op, ac) \wedge hasPort(op, p) \wedge \\
&\quad \forall vb, attr.(hasValidityBinding(pr, vb) \wedge IOPair(vb, attr, p) \\
&\qquad \rightarrow hasValidityBinding(op, attr))
\end{aligned}
$$

C.4.4 Output obligation.

$$
\begin{aligned}
&OutputObligation(oc, args, ac, p, ob) \Leftarrow \\
&\quad InputPolicyForOutput(input, policy, output) \wedge \\
&\quad hasPort(output, p) \wedge hasObligation(policy, ob) \wedge \\
&\quad hasObligationClass(ob, oc) \wedge hasArgument(ob, args) \wedge \\
&\quad \forall x.(member(args, x) \rightarrow \exists x'.IOPair(x, x', p)) \wedge \\
&\quad hasActivationCondition(ob, ac) \wedge \\
&\quad \forall vb.(hasValidityBinding(ob, vb) \rightarrow \exists attr.IOPair(vb, attr, p))
\end{aligned}
$$

where the full form of $OutputObligation$ is:

$$
\begin{aligned}
&OutputObligation(oc, args, ac, p, ob) \equiv \\
&\quad \exists oo.A(oo, Obligation) \wedge hasObligationClass(oo, oc) \wedge \\
&\quad \exists args'.hasArgument(oo, args') \wedge \\
&\quad\; (\forall x.member(args, x) \rightarrow member(args', x)) \wedge \\
&\quad (\forall vb, attr.hasValidityBinding(ob, vb) \wedge IOPair(vb, attr, p) \\
&\qquad \rightarrow hasValidityBinding(oo, attr))
\end{aligned}
$$

where $member(args, x)$ means x is a member of the list (RDF Collection) $args$.