Modeling a GDPR Compliant Data Wallet Application in Prova and AspectOWL

Ralph Schäfermeier[1], Theodoros Mitsikas[2,3] and Adrian Paschke[1,3]

[1]Fraunhofer FOKUS, Berlin, Germany
[2]National Technical University of Athens, Zografou, Greece
[3]Institut für Angewandte Informatik, Leipzig, Germany

Abstract
We present a GDPR-compliant data privacy and access use case of a distributed data wallet and we explore its modeling using two options, AspectOWL and Prova. This use case requires a representation capable of expressing the dynamicity and interaction between parties. While both approaches provide the expressiveness of non-monotonic states and fluent state transitions, their scope and semantics are vastly different. AspectOWL is a monotonic context ontology language, able to represent dynamic state transitions and knowledge retention by wrapping parts of the ontology in isolated contexts, while Prova can handle state transitions at runtime using non-monotonic state transition semantics. We present the two implementations and we discuss the similarities, advantages, and differences of the two approaches.

Keywords
GDPR, Knowledge Representation, Prova, AspectOWL

1. Introduction

In the wake of the introduction of the European GDPR (General Data Protection Regulation) in 2016 and its effective enforcement since 2018 businesses worldwide were obliged to review and adapt their data privacy policies if they desired to continue offering their online services to EU citizens. The complexity of regulatory works such as the GDPR and the large amount of parties affected by them has led to an increased interest in research on the problem of automatic legal and ethical compliance checking. The foundation of such systems is an adequate formalization of the body of rules under consideration.

A key concept in GDPR is the concept of consent, meaning that the user (data subject) must agree to any processing of personal data. Moreover, the user has the right to receive the collected personal data in a machine-readable format, as well as the right to transmit those data to another controller (right to data portability) [1].

Adhering to the above principles, efforts and projects such as Solid aim to give users more control over their personal data. Solid uses Semantic Web technologies to decouple user data

RuleML+RR'22: 16th International Rule Challenge and 6th Doctoral Consortium, September 26–28, 2022, Virtual
ralph.schafermeier@gmail.com (R. Schäfermeier); mitsikas@central.ntua.gr (T. Mitsikas); adrian.paschke@fokus.fraunhofer.de (A. Paschke)
https://www.data-analytics-center.org/team-0886abf92d9d2018 (R. Schäfermeier); https://www.mi.fu-berlin.de/inf/groups/ag-csw/Members/members/paschke.html (A. Paschke)
0000-0002-4349-6726 (R. Schäfermeier); 0000-0002-7570-3603 (T. Mitsikas); 0000-0003-3156-9040 (A. Paschke)

from the applications that use this data, which makes it easy for users to switch between applications that use the same data and to switch between storage providers that host the data, while having access control to their data [2].

A typical architecture of such an ecosystem has different components, each having a specific role: an Identity Provider (IDP) manages the user identity information and also provides authentication services, a Data Wallet Provider (DWP) stores user data, and Relaying Parties are applications that can access and process the data. Decoupling user data from the applications requires data to be stored in a structured way, compatible across the DWPs, and provides the user with control over their data, as users must demonstrate consent to allow applications to access and process the data [2].

To this end, we present a use case addressing a generic distributed data wallet scenario, with consent being a central concept both for access control (sharing a picture to other users) and for personal data processing (using personal data for a personalized Web search). We provide two implementations using two formalisms that are both sufficiently expressive: AspectOWL, a version of OWL extended by means for expressing context-sensitive knowledge, which allows the representation of dynamic and deontic aspects of the domain, and the rule engine Prova.

The remainder of this paper is organized as follows. Section 2 presents the languages Prova and AspectOWL. Section 3 describes the use case, while Section 4 discusses the implementation in AspectOWL and Prova. Section 5 compares and evaluates the two approaches and finally, Section 6 concludes the paper and proposes future work.

2. Background

2.1. AspectOWL

AspectOWL [3] is an extension of the W3C OWL 2 ontology language[1] that permits the representation of contextualized knowledge by adding formal context descriptions (here called *aspects*) to TBox, RBox, and ABox axioms of an OWL ontology.

AspectOWL is an instance of a general Knowledge Representation (KR) approach to the formalization of context, named *Aspect-Oriented Ontology Development (AOOD)* [4]. AOOD, in turn, is inspired by the Aspect-Oriented Programming (AOP) paradigm [5], from which it lends most of its basic concepts and accompanying terminology.

2.1.1. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) aims at improving software modularity. The principle idea is the separation of pieces of (normally entangled) software code by the *concerns* they address. This is accomplished by diverting from the strictly execution-flow-centered semantics of classical imperative programming languages and moving code that addresses different concerns into different, isolated modules (called *aspects*).

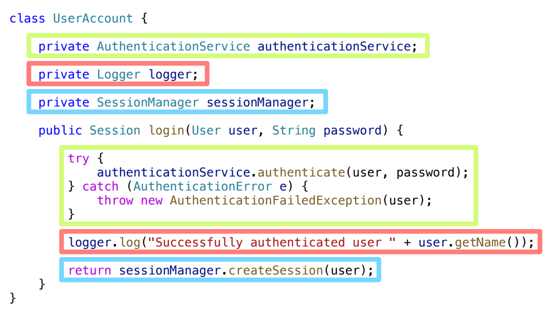
The information on *when* each aspect is executed resides in the aspect itself and is provided in a reifying manner, usually employing either an annotation system or queries in the form of formulae quantifying over the set of method signatures. Such a reference is called a *join point*

[1]https://www.w3.org/TR/owl2-overview/

and a set of join points defined by a reifying query is referred to as a *pointcut*. The code in the aspect along with its pointcut description is called *advice*.

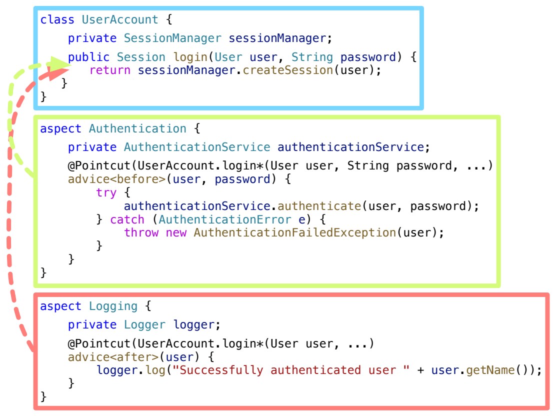
The modules affected by the code of the aspect remain oblivious about it; it is the responsibility of the runtime environment to intercept existing join points and divert the control flow accordingly. Due to this property and the quantification used for the definition of pointcuts, an often quoted statement about AOP is that its two fundamental principles are *quantification and obliviousness* [6]. Aspects are a special kind of class (in terms of object orientation) with executable methods. As a consequence, they carry potential join points themselves and may therefore be arbitrarily nested.

As an example, a login procedure might require authentication of the user, possibly logging for later auditing and then creating a user session, addressing the concerns "security", "auditing", and the actual "business logic", leading to a situation where all of these cross-cutting concerns are executed in the same method body (see Figure 1a). In AOP the code addressing the first two concerns would reside in their own aspects, respectively and only the code handling the actual business logic would remain in the original method (see Figure 1b).



```
class UserAccount {
    private AuthenticationService authenticationService;
    private Logger logger;
    private SessionManager sessionManager;
    public Session login(User user, String password) {
        try {
            authenticationService.authenticate(user, password);
        } catch (AuthenticationError e) {
            throw new AuthenticationFailedException(user);
        }
        logger.log("Successfully authenticated user " + user.getName());
        return sessionManager.createSession(user);
    }
}
```

(a) The three cross-cutting concerns "security" (green), "auditing" (red), and "business logic" (blue) are handled in the same method body, leading to entangled code.



```
class UserAccount {
    private SessionManager sessionManager;
    public Session login(User user, String password) {
        return sessionManager.createSession(user);
    }
}
aspect Authentication {
    private AuthenticationService authenticationService;
    @Pointcut(UserAccount.login*(User user, String password, ...)
    advice<before>(user, password) {
        try {
            authenticationService.authenticate(user, password);
        } catch (AuthenticationError e) {
            throw new AuthenticationFailedException(user);
        }
    }
}
aspect Logging {
    private Logger logger;
    @Pointcut(UserAccount.login*(User user, ...)
    advice<after>(user) {
        logger.log("Successfully authenticated user " + user.getName());
    }
}
```

(b) Separation of concerns using aspects. The main concern (business logic) remains in the method body, the two other concerns become encapsulated in their respective aspects. Pointcut queries define the locations at which the aspects intercept the main code (illustrated by the arrows).

2.1.2. Aspect-Oriented Ontologies: Modularization by Context-Level

AOOD lends its fundamental principles from AOP and applies them to knowledge representation formalisms, such as OWL ontologies. It uses a similar quantification/reification mechanism in order to connect aspects to primitives of the underlying formalism (in the case of OWL it connects them to OWL axioms), by also using primitives from the formalism in order to describe the aspects themselves (in case of OWL, for example, aspects are OWL class expressions). Applied to KR formalisms, aspects can be used to convey context to axioms (for example temporal information) that restricts the validity of the target primitive.

AspectOWL uses Modal Logics as a context language and makes use of the fact that Modal Logics are syntactic variants of Description Logics [7], i. e., (almost) every Modal Logic can be translated to a particular Description Logic (and thereby to OWL) so that modal contexts (such as temporal, spatial, or deontic) can be expressed using OWL.

For this purpose, Aspect OWL introduces a new axiom type called *aspect assertion axiom.* An aspect assertion is a binary relation between an OWL axiom and an advice class expression (the context of the axiom). Syntactically, aspect assertion axioms resemble annotation assertion axioms. They differ from the latter in that they have a defined model theoretic semantics, which makes use of combined interpretations, which we call a $\mathcal{SROIQ}_{Kripke}$ interpretation.

Definition 1. *A $\mathcal{SROIQ}_{Kripke}$ interpretation is a tuple $\mathcal{J} := (W, R, L, \cdot^{\mathcal{J}}, \Delta, (\cdot^{\mathcal{I}_w})_{w \in W})$ with W being a nonempty set, called* possible worlds*, and L a Kripke interpretation, assigning truth values to propositional symbols in each world $w \in W$ as described in the subsection about Modal Logics. For every $A \subseteq W$, $\mathcal{I}_A$ is a DL interpretation.*

The semantics of an aspect of an axiom is then defined as follows:

Definition 2. *Let $\mathcal{J} := (W, R, L, \cdot^{\mathcal{J}}, \Delta, (\cdot^{\mathcal{I}_w})_{w \in W})$ be a possible-world DL interpretation. We interpret an aspect under which an axiom α holds as follows:*

$(\mathsf{hasAspect}(\alpha, A))^{\mathcal{J}} \to A^{\mathcal{J}} \subseteq C^{\mathcal{J}} := \{w \in W \mid \mathcal{I}_w \models \alpha\}$*. Because of the correspondence as described in the subsection about Modal Logics we can set $W = C^{\mathcal{J}}$, such that on the semantic level each individual corresponds to a possible world. Furthermore, we set L such that $L(\alpha)^{\mathcal{J}} := A^{\mathcal{J}}$.*

AspectOWL currently implements three ways of axiom reification for pointcut selection: By DL-based modules, using SPARQL queries, and by a set of specialized SWRL built-ins[2].

For a full description of the features and semantics of AspectOWL 2, see [3][3].

2.2. Prova

Prova is both a (Semantic) Web rule language and a distributed (Semantic) Web rule engine. It supports reaction rule based workflows, event processing, and reactive agent programming. It integrates Java scripting with derivation and reaction rules, supporting message exchange with various communication frameworks [8, 9, 10].

Syntactically, Prova builds upon the ISO Prolog syntax and extends it, notably with the integration of Java objects, typed variables, F-Logic-style slots, and SPARQL and SQL queries. Prolog-like compound terms can be represented as generic Prova lists (e.g., a standard Prolog-like compound term $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{N})$ is a syntactic equivalent of the Prova list $[\mathtt{f}, \mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{N}]$). Slotted terms in Prova are implemented using the arrow expression syntax '`->`'as in RIF and RuleML, and can be used as sole arguments of predicates. They correspond to a Java HashMap, with the keys limited to Stings [11].

Semantically, Prova provides the expressiveness of serial Horn logic with a linear resolution for extended logic programs (SLE resolution) [12], extending the linear SLDNF resolution with goal memoization and loop prevention. Negation as failure support in the rule body can be added to a Knowledge Base (KB) by implementing it using the cut-fail test as follows:

[2]https://github.com/RalphBln/aspect-swrl-builtins

[3]For an overview over the complete abstract syntax of Aspect OWL 2, see http://www.aspectowl.xyz/syntax/.

```
not(A) :-
    derive(A),
    !,
    fail().
not(_).
```

Notice the Prova syntax for `fail` that requires parentheses, as well as the built-in meta-predicate `derive` that allows to define (sub) goals dynamically with the predicate symbol unknown until run-time [10].

Prova's reactive agents are instances of a running rulebases that include message passing primitives. These built-in primitives are the predicates `sendMsg/5`, `rcvMsg/5`, as well as their variants `sendMsgSync/5`, `rcvMult/5`. The position-based arguments for the above predicates are [11]:

1. *XID* - conversation id of the message
2. *Protocol* - name of the message passing protocol
3. *Destination* (on sending) or *Sender* (on receiving) - the agent name of the receiver/sender
4. *Performative* - the message type broadly characterizing the meaning of the message
5. *Payload* - a Prova list containing the actual content of the message

Prova defines the Java interface `ProvaService` and its default implementation `ProvaServiceImpl` that allows for a runner Java class – depending on the modularization (mapping each agent to a separate bundle vs. multiple agents in a bundle) – to embed one or more agents communicating with each other via messaging.

The fundamental method is the method `send` that takes the following arguments:

```
send(String xid, String destination, String sender,
     String performative, Object payload, EPService callback)
```

The arguments have a direct correspondence with the message passing primitives, while `EPService` is a superclass of the `ProvaService` interface. Also, the message passing protocol is selected automatically (in our use case, the osgi protocol is selected).

3. Use Cases

In this section, we describe two data wallet use cases in terms of interaction sequences and data exchange between the different parties involved. The use case revolves around fictional data wallet owners that share personal data using relaying parties that provide specialized applications (a search applications and a picture sharing application).

Use Case 1: Personalized Search (Figure 2)

- Alice opens an account with an Identity Provider IdP_{Alice}.
- She demonstrates consent for the IdP to store her *OpenID* and her *login credentials* for the purpose of *confirming her identity towards third parties.*
- Now Alice opens an account with a Data Wallet Provider *DWP_Alice.*

- She demonstrates consent for the DWP to store data she uploads to the DWP along with her WebID document, which represents her online identity and contains a link to her OpenID (managed by her IdP).
- Alice seeks some information at the third-party app *SearchApp* (relying party, *RP*)
- She launches SearchApp and enters a search query.
- The app requests personal data about her previous search history from Alice's personal data wallet for personalization of the current search.
- The app also asks if Alice's data might be used for data analytics by SearchApp.
- Alice expresses her consent for both purposes by giving read permission for the requested data to SearchApp (identified by its WebID)
- SearchApp reads and stores the data Alice gave them permission for and derives an anonymized dataset for data analytics purposes.
- Later Alices demonstrates the revocation of her consent to use the data for data analytics and personalisation purposes by SearchApp by revoking the read permission.
- SearchApp may continue to use the derived (anonymized) data.
- SearchApp must delete the personal data it has obtained from Alice's data wallet.
- SearchApp is now denied to get updates from Alice's search history data from her data wallet.

Use Case 2: Sharing Pictured via a Wallet-Enabled Sharing App (Figure 3)

- Alice decides to share a personal picture with her friends Bob and Cesar using PictureApp (relying party 2, *RP2*).
- She demonstrates consent for PictureApp to retrieve the picture from her data wallet.
- She demonstrates consent for PictureApp (identified by its WebID) to make the picture available to her friends Bob and Cesar, both identified by their WebIDs.
- Later, Alice demonstrates the revocation of her consent to share the picture with Cesar by revoking read permission for Cesar.
- PictureApp is still permitted to store a copy of Alice's image.
- PictureApp has the obligation to deny Cesar access to the image.

4. Implementation

4.1. AspectOWL

As OWL is a monotonic, declarative knowledge representation formalism it is suited for representing the static aspects of the domain under consideration. With AspectOWL, however, it is also possible to represent dynamic behavior: Different states of the universe may be represented by different contexts (in the form of OWL aspects) in which certain axioms hold respectively. The transition between states may be represented in terms of events that happen at a certain point in time with the contexts representing two subsequent states having a temporal extension either before or after the point in time.

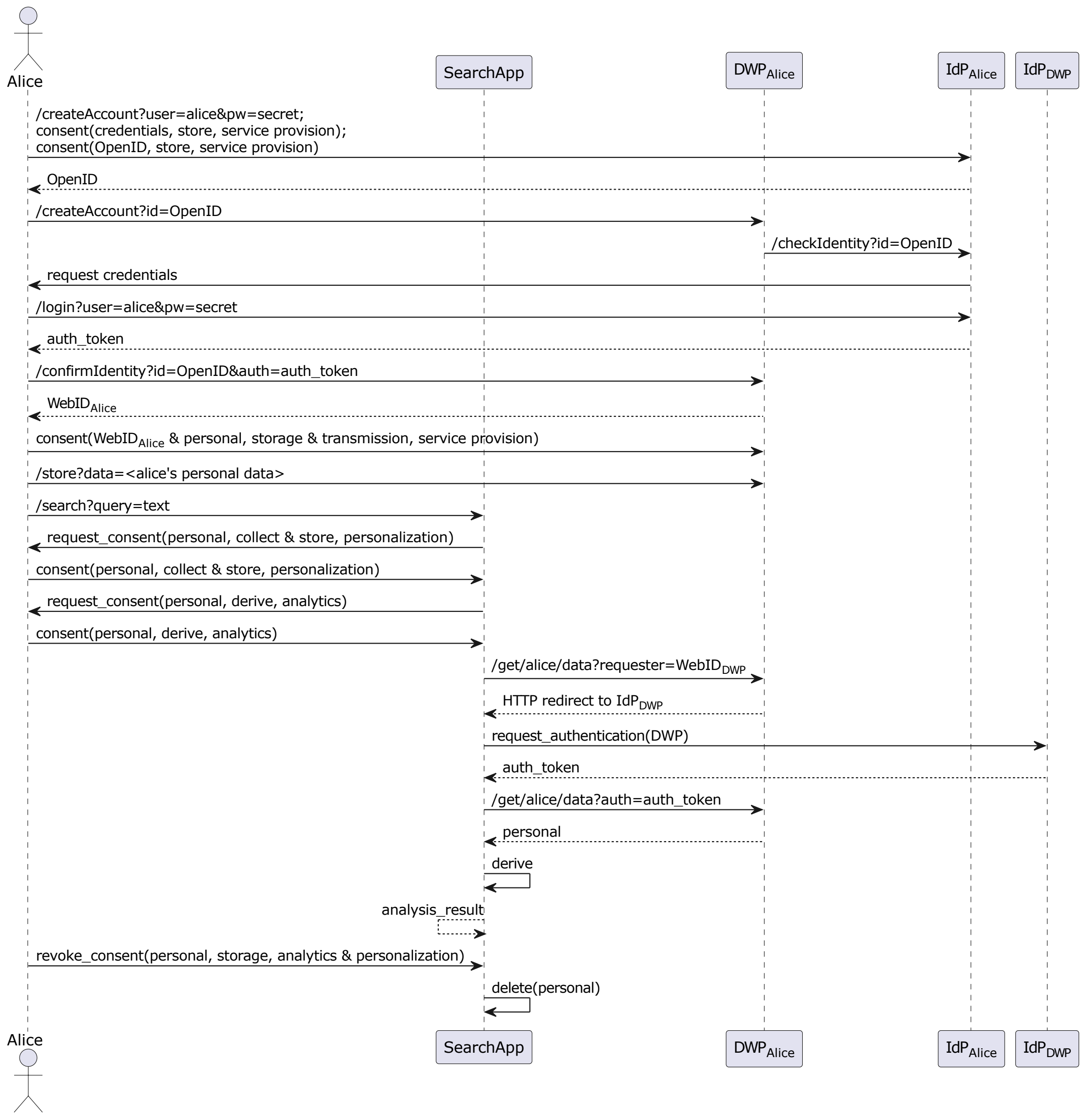

Alice
SearchApp
DWP_{Alice}
IdP_{Alice}
IdP_{DWP}
/createAccount?user=alice&pw=secret;
consent(credentials, store, service provision);
consent(OpenID, store, service provision)
OpenID
/createAccount?id=OpenID
/checkIdentity?id=OpenID
request credentials
/login?user=alice&pw=secret
auth_token
/confirmIdentity?id=OpenID&auth=auth_token
$WebID_{Alice}$
consent($WebID_{Alice}$ & personal, storage & transmission, service provision)
/store?data=<alice's personal data>
/search?query=text
request_consent(personal, collect & store, personalization)
consent(personal, collect & store, personalization)
request_consent(personal, derive, analytics)
consent(personal, derive, analytics)
/get/alice/data?requester=$WebID_{DWP}$
HTTP redirect to IdP_{DWP}
request_authentication(DWP)
auth_token
/get/alice/data?auth=auth_token
personal
derive
analysis_result
revoke_consent(personal, storage, analytics & personalization)
delete(personal)
Alice
SearchApp
DWP_{Alice}
IdP_{Alice}
IdP_{DWP}

Figure 2: First part of the use case: Alice creates an account with a DWP, demonstrates consent for sharing personal data for the purposes of sharing and analysis, later revokes her consent for the purpose of analysis.

Furthermore, AspectOWL permits the application of deontic modalities to OWL axioms. Since nesting of aspects is also allowed, it is possible to combine the two and represent dynamic change of deontic modalities.

Our development process was guided by the following principles: We aimed to reuse existing ontologies if available. A significant amount of concepts could be imported from the current publicly available version of the PrOnto ontology [13]. Furthermore, we use existing ontology design patterns (ODPs) whenever applicable. Links to ODPs used in this work are provided in

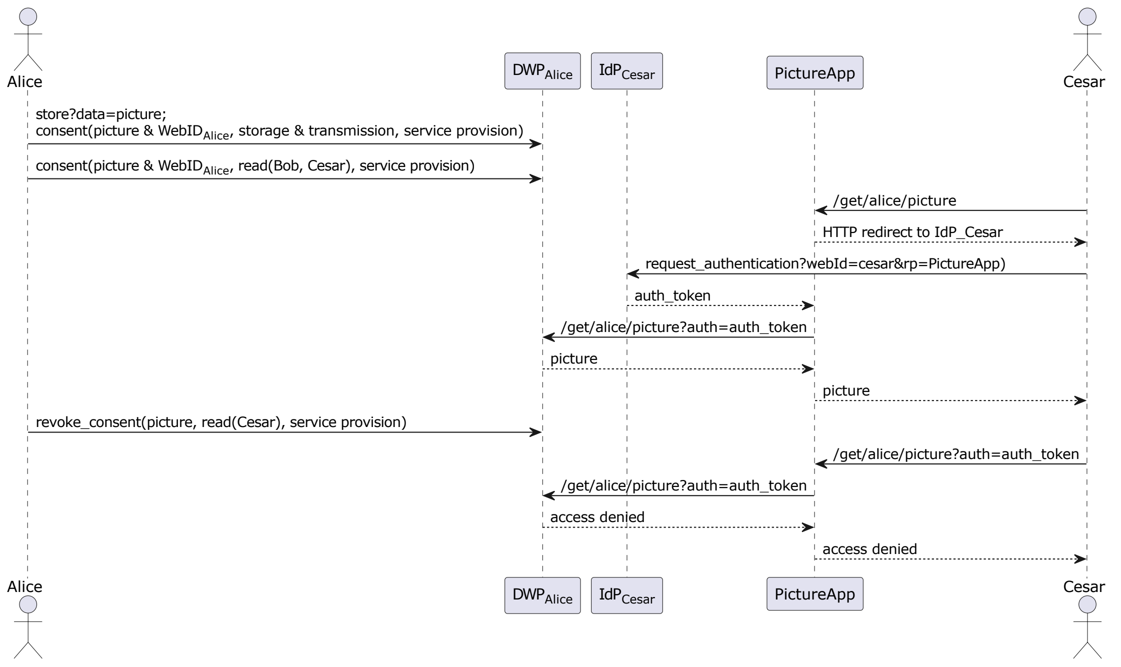

Alice
DWP$_{Alice}$
IdP$_{Cesar}$
PictureApp
Cesar
store?data=picture;
consent(picture & WebID$_{Alice}$, storage & transmission, service provision)
consent(picture & WebID$_{Alice}$, read(Bob, Cesar), service provision)
/get/alice/picture
HTTP redirect to IdP_Cesar
request_authentication?webId=cesar&rp=PictureApp)
auth_token
/get/alice/picture?auth=auth_token
picture
picture
revoke_consent(picture, read(Cesar), service provision)
/get/alice/picture?auth=auth_token
/get/alice/picture?auth=auth_token
access denied
access denied

Figure 3: Second part of the use case: Alice shares a picture from her data wallet with Bob and Cesar. Cesar accesses her picture using a wallet-enabled sharing app. Later, Alice decides to revoke permission from Cesar to access her picture. He and the sharing app can no longer access the picture.

the footnotes.

4.1.1. Static Part

Data

Data is the central concept of the GDPR domain around which everything else revolves.

We reuse the data concept hierarchy from the PrOnto ontology [13], which makes Data a subclass of InformationObject, which in turn is a subclass of the class FRBRWork from the Functional Requirements for Bibliographic Records (FRBR) vocabulary.

1. Data $\sqsubseteq$ InformationObject
2. InformationObject $\sqsubseteq$ akn:FRBRWork[4]
3. akn:FRBRWork $\sqsubseteq$ owl:Thing

Ownership of Data

The GDPR is concerned about usage of data by different agents. Agents may either be human persons or non-human organizations.

[4]akn: Akoma Ntoso XML for parliamentary, legislative & judiciary documents (OASIS). FRBR: Functional Requirements for Bibliographic Records (IFLA)

4. Person(Bob)
5. ownedBy(dataHabit, Bob)

Agent Roles

Agents, i. e., both persons and organizations, may assume roles as defined by the GDPR, namely the role of the data subject, the data controller, and the data processor. The same agent may assume several of these roles at the same time, e. g., a company may be both data controller and data processor.

6. $\exists$hasRole.DataSubject(Bob)
7. $\exists$hasRole.Controller(CompanyA)
8. $\exists$hasRole.Processor(CompanyB)

Demonstration of Consent

Demonstration of consent is modeled in terms of a contract.

6. ConsentAction $\sqsubseteq$ te:Action[5]
7. ConsentAction(ca)
8. pwo:happened(ca, t1)[6]
9. pwo:produces(ca, co), Consent(co), Consent $\sqsubseteq$ Contract
10. allowsAction(co, ac)
11. bpe:actionHasParticipant(ac, org)[7]
12. hasSubject(ac, data)

Data Processing Purpose

As mandated by the GDPR, user consent for the processing of personal data must be explicitly given for a specific purpose and is only valid for that particular purpose.

13. Purpose $\sqsubseteq$ owl:Thing
14. Advertising $\sqsubseteq$ Purpose
15. Marketing $\sqsubseteq$ Purpose
16. Optimisation $\sqsubseteq$ Purpose
17. allowedPurpose $\sqsubseteq$ owl:topObjectProperty
18. allowedPurpose(dataHabit, advertising)

Data Processing Action

For the conceptualization of data processing actions we re-use the existing concept Action from the PrOnto ontology along with a number of ontology design patterns. PrOnto defines actions as parts of workflows and relies on the Basic Plan Execution design pattern for doing so. The

[5]http://www.ontologydesignpatterns.org/cp/owl/taskexecution.owl
[6]http://purl.org/spar/pwo/
[7]http://www.ontologydesignpatterns.org/cp/owl/basicplanexecution.owl

latter distinguishes between abstract workflow descriptions (which are abstract plans) and their concrete executions.

Consequently, a workflow description may have arbitrarily many workflow execution instantiations, which in turn may involve arbitrarily many actions.

19. Analysis $\sqsubseteq$ DataProcessing
20. DataProcessing $\sqsubseteq$ Workflow
21. Workflow $\sqsubseteq$ Plan
22. Plan $\sqsubseteq$ Description
23. Workflow $\sqsubseteq$ $\forall$isSatisfiedBy.bpe:PlanExecution[8]

A concrete execution of an abstract workflow is represented as follows:

24. pwo:WorkflowExecution $\sqsubseteq$ bpe:PlanExecution
25. pwo:WorkflowExecution $\sqsubseteq$ tis:TimeIndexedSituation[9]
26. bpe:PlanExecution $\sqsubseteq$ sit:Situation
27. tis:TimeIndexedSituation $\sqsubseteq$ sit:Situation

And finally, an action is part of a workflow execution:

28. pwo:WorkflowExecution $\sqsubseteq$
$\exists$pwo:involvesAction.(pwo:Action $\sqcap$ $\exists$te:executesTask.pwo:Step)

4.1.2. Dynamic Part

In this section, we demonstrate how dynamic processes (in terms of transitions between different states of the universe over time, usually in succession of an event) can be represented using AspectOWL. Demonstrating consent for the processing of data by a third party leads to a transition between two states; from one in which the processing is not permitted to one where it is. It is possible to represent the two different states using two OWL aspects, each representing one state. As the transition is triggered by an event that happened at a certain point in time T_DC_1 the states may also be represented as temporal contexts, the boundaries of both of which coincide at T_DC_1.

29. StateAspect1 $\sqsubseteq$ aot:TemporalAspect $\sqcap$ time:before.{T_DC_1}[9]
30. StateAspect2 $\sqsubseteq$ aot:TemporalAspect $\sqcap$ ({T_DC_1} $\sqcup$ time:after.{T_DC_1})

As it is not possible for the same thing to be permitted and prohibited at the same time it must be made sure that the two aspects representing the states are disjoint.

31. StateAspect1 $\sqcap$ StateAspect2 $\sqsubseteq$ owl:Nothing

[7]http://www.ontologydesignpatterns.org/cp/owl/timeindexedsituation.owl
[8]http://www.ontologydesignpatterns.org/cp/owl/situation.owl
[9]https://ontology.aspectowl.xyz/temporal#

The action that is allowed in the state after the demonstration of consent is represented by a simple object property assertion:

32. processesData(o1, data), where
33. Organisation(o1) ⊓ Data(data)

However, the aspect cannot be directly applied to the above assertion axiom since this would mean that starting from time point T_DC_1 the organization o1 processes data, while what we want to represent is the fact that o1 *is permitted* to process data starting at T_DC_1.

Representing the permission involves the creation of a further aspect of the type deontic aspect.

34. PermissionAspect ⊑ aod:DeonticAspect ⊓ ∃aod:legallyAccepts.aod:Reality[10]

Application of the deontic aspect to the assertion and the nesting of the resulting aspect assertion into the temporal aspect StateAspect2 yield the representation of the statement that o1 is allowed to proccess data starting at T_DC_1.

35. Aspect(StateAspect2, Aspect(PermissionAspect, processesData(o1, data)))

In order to derive statements of this kind automatically when an assertion of demonstration of consent is encountered in the knowledge base, the following SWRL rule can be employed.

```
-:
DemonstrateConsent(?ca), pwo:happened(?ca, ?t1),
actedBy(?ca, ?ds), pwo:produces(?ca, ?co),
Consent(?co), allowsAction(?co, ?ac),
bpe:actionHasParticipant(?ac, ?org),
hasSubject(?ac, ?data),
aspectswrl:createOPA(collectsDataFrom, ?org, ?ds, ?a),
aspectswrl:temporal(?a, time:after, ?t1, true),
aspectswrl:deontic(?perm, legallyAccepts, aod:Reality),
aspectswrl:nest(?perm, ?a)
```

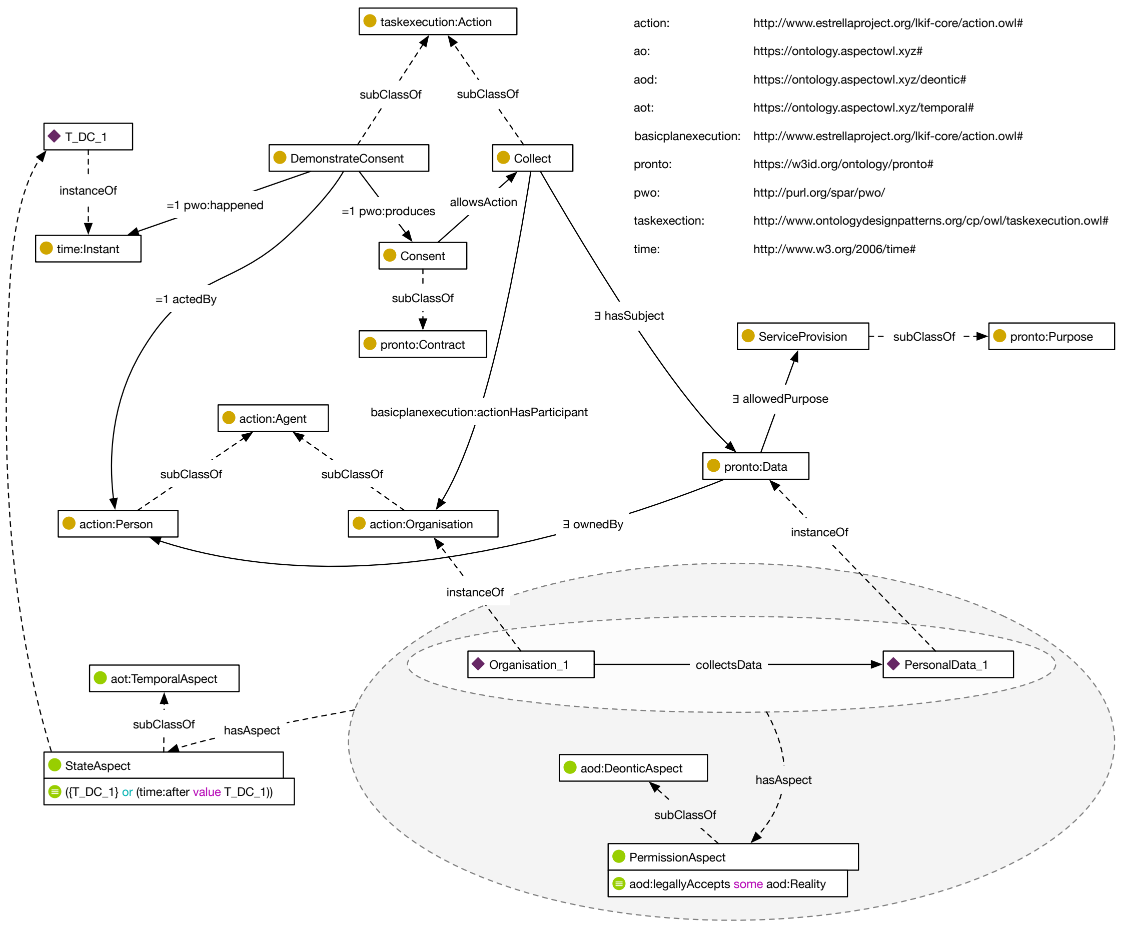
The rule makes use of multiple AspectSWRL built-ins. `aspectswrl:createOPA`, which creates the OWL object property assertion and wraps it in an aspect bound to variable `?a`. `aspectswrl:temporal` creates the temporal aspect a. The first parameter binds the resulting aspect to ?a. The second parameter determines the accessibility relation used, in this case time:after. The third parameter determines the time individual used in conjunction with the accessibility relation, which we set to the value of the variable ?t1 and which corresponds to the object in the pso:happened predicate. The fourth parameter is a boolean determining whether the individual should be included in the interval defining the aspect or not. In this case, we want ?t1 to be included in the interval. aspectswrl:deontic works similarly with the parameters being the variable to which the resulting aspect should be bound, the accessibility relation, and

[10] https://ontology.aspectowl.xyz/deontic#

the individual representing reality. aspectswrl:nest nests takes to aspects as parameters and results in a nesting of the first into the second.

When the user retracts their consent, an instance of the class RevokeConsent is created, which has the same properties as the DemonstrateConsent. A second SWRL rule, similar to the one above, with the exception that it contains RevokeConsent instead of DemonstrateConsent in the antecedent and legallyProhibits instead of legallyRejects then creates the new temporal context in which the former modality of permission of the data processing is replaced by a prohibition modality.

Figure 4 provides an overview of the ontology and the aspects created by the rules[11].



taskexecution:Action
subClassOf
subClassOf
T_DC_1
DemonstrateConsent
Collect
instanceOf
=1 pwo:happened
=1 pwo:produces
allowsAction
time:Instant
Consent
=1 actedBy
subClassOf
pronto:Contract
∃ hasSubject
ServiceProvision
subClassOf
pronto:Purpose
∃ allowedPurpose
action:Agent
basicplanexecution:actionHasParticipant
subClassOf
subClassOf
pronto:Data
action:Person
action:Organisation
∃ ownedBy
instanceOf
instanceOf
Organisation_1
collectsData
PersonalData_1
aot:TemporalAspect
subClassOf
hasAspect
StateAspect
({T_DC_1} or (time:after value T_DC_1))
aod:DeonticAspect
hasAspect
subClassOf
PermissionAspect
aod:legallyAccepts some aod:Reality
action: http://www.estrellaproject.org/lkif-core/action.owl#
ao: https://ontology.aspectowl.xyz#
aod: https://ontology.aspectowl.xyz/deontic#
aot: https://ontology.aspectowl.xyz/temporal#
basicplanexecution: http://www.estrellaproject.org/lkif-core/action.owl#
pronto: https://w3id.org/ontology/pronto#
pwo: http://purl.org/spar/pwo/
taskexection: http://www.ontologydesignpatterns.org/cp/owl/taskexecution.owl#
time: http://www.w3.org/2006/time#

Figure 4: An excerpt from the RECOMP AspectOWL ontology showing the part used for representing the act of demonstration of consent and its consequences. Aspects are represented by areas of different shades of gray. The light-gray area depicts a an OWL object property assertion axiom, which is target of a deontic aspect, representing the fact that the validity of the target axiom is permitted. It is nested in a temporal aspect, which temporally restricts the validity of the permission. The assertion and the nested aspects are the result of the execution of a SWRL rule, which uses the AspectOWL SWRL built-ins.

[11]The ontology files are available at https://github.com/RalphBln/recomp-use-cases

4.2. Prova

The Prova implementation[12] uses the message passing primitives mentioned in Section 2. All parties are represented as agents that communicate via message passing. All actions of the workflow are initiated by messages sent by the Java runner class to the appropriate agent.

To demonstrate the Prova implementation, we focus on the first part of the use case presented in Section 3, where Alice (represented by the agent `alice`) performs a web search. The SearchApp (represented by the agent `searchApp`) requests consent for personal data access and delivers a personalized search result if the consent was demonstrated, or a non-personalized result otherwise.

The actions of `alice` are controlled from Java, essentially creating a script with the messages that `alice` receives, which in turn are forwarded to other agents. Therefore, initially we pass the following message to `alice` from Java:

```
payload.put("agent","searchApp");
payload.put("operation","search");
payload.put("webID","alice.example.com");
payload.put("query","travel suggestions");
payload.put("dwp","dwp");
service.send("xid","alice","javaRunner","request",payload,this);
```

where `payload` is a Java HashMap, with its elements corresponding to the slot names and fillers. These are the agent that performs the search, the operation, `alice`'s WebID, the search query, and the agent that serves as a data wallet provider.

The above message is captured by the following inline reaction rule [11]:

```
alice() :-
    rcvMult(XID,P,From,request,
        {agent->A,operation->Op,webID->WebID,query->Q,dwp->DWP}),
    sendMsg(XID,P,A,request,
        {operation->Op,webID->WebID,query->Q}).
```

This rule instructs that upon receiving a message of this pattern, `alice` will send a message to the appropriate agent (e.g., to the agent `searchApp`) requesting a specific operation (e.g., `search`).

On the `searchApp` side, the inline reaction rule for the search operation is the following:

```
searchApp() :-
    rcvMult(XID,P,From,request,
        {operation->search,webID->WebID,query->Q,dwp->DWP}),
    search(WebID,Q,From,Result,P,DWP),
    sendMsg(XID,P,From,inform,{msg->Result}).
```

This rule evaluates when `searchApp` receives a message with the above pattern, evaluates `search(WebID,Q,From,Result,P,DWP)` and sends the message containing the search result back to the agent, in this case `alice`.

[12]The Prova implementation is available at https://github.com/tmitsi/recomp-usecases

The search/6 predicate is used to perform the search and always evaluates to true (as some search result is always returned regardless of the user consent), binding the search result in the variable Result. It has alternatives, covering the cases where the user is not yet identified through the idp, where the user was already asked for consent in the past, as well as the following alternative covering the case where the user is already logged in and performs the first search:

```
search(WebID,Q,From,Result,P,DWP) :-
    loggedIn(From,WebID,_),
    not(alreadyAsked(WebID,personal,personalization)),
    Msg = "The searchApp wants... ...personalized results"
    sendMsg(XID,P,From,input_request, {operation->consent,
        data->personal, purpose->personalization, msg->Msg}),
    rcvMsg(XID,Protocol,From,response,{answer->In}),
    assert(alreadyAsked(WebID,personal,personalization)),
    !,
    searchHelper(WebID,Q,From,Result,DWP,In).
```

This alternative defines the actions when the user is already logged in and hasn't been previously asked about her consent for personal data process, i.e. the predicates loggedIn/3 and not(alreadyAsked(WebID,personal,personalization)) succeed. First, a message is sent back to the user, expecting to receive a yes/no answer (the relevant Prova code for alice agent is omitted). Afterwards, the fact stating that the user is already asked is asserted in to the KB, followed by the cut operator to prevent the evaluation of other search/6 alternatives.

Finally, the helper predicate searchHelper(WebID,Q,From,Result,DWP,In) is evaluated, binding the results in the variable Result, while taking in to account the consent demonstration (or, the lack of it) of the user:

```
searchHelper(WebID,Q,From,Result,_,no) :-
    nonPersonalizedSearch(Q,Result).

searchHelper(WebID,Q,From,Result,DWP,yes) :-
    consent(WebID,personal,personalization),
    personalizedSearch(WebID,Q,From,Result).

searchHelper(WebID,Q,From,Result,DWP,yes) :-
    not(consent(WebID,personal,personalization)),
    assert(consent(WebID,personal,personalization)),
    !,
    personalizedSearch(WebID,Q,From,DWP,Result).
```

As seen above, the searchHelper/6 has three alternatives, covering all possible cases: 1. the user refuses to provide consent, where the nonPersonalizedSearch/2 is called and binds the non-personalized search results in to the second argument, 2. the user has already provided consent and the personalizedSearch/5 is called, and 3. the user just provided consent and this fact should be asserted to the KB, followed by the cut operator that stops the evaluation of the second alternative, and then calling `personalizedSearch/5`. The rest of the Prova code for `personalizedSearch/5` is omitted here.

5. Comparison of Approaches

In general, the two approaches used in this paper differ in their intended application scopes and hence their syntax, semantics, and expressiveness. However, interestingly, both could be used to represent a significant part of the domain under consideration.

The agent-based Prova implementation with the message passing primitives was able to model the entire workflow. The reactive rules, combined with assertions and retractions were adequate to model all possible states. Moreover, it was possible to simulate real-world practices such as storing hashed passwords instead of plain-text, and session cookies to facilitate logins.

This is clearly out of the scope of static knowledge-representation formalisms such as OWL, as the latter lacks the necessary interaction primitives with the outside world, such as reaction rules or data stream processing facilities. It is, however, conceivable to employ some sort of dynamically updating ABox and let our AspectSWRL rules run on new ABox axioms as they are added to the knowledge base.

An obvious difference between Prova and OWL is that the latter operates in a strictly monotonic fashion without any notion of knowledge retraction. AspectOWL is able to circumvent this to a certain extent since its ability to create contexts which restrict the validity of exiting axioms introduces a way of mimicking non-monotonicity. However, AspectOWL is still a monotonic formalism, and while knowledge may be retracted from the global scope by restricting it to a local context, the context itself (containing the knowledge) can never be retracted. In other words, the history of emerging and disappearing knowledge can never be erased in AspectOWL. This can be regarded either as an advantage or as a disadvantage, depending on the requirements of the application.

Prova, OWL, and AspectOWL permit the choice of the level of expressiveness and the selection of semantics. OWL 2, for example, introduced different profiles, such as OWL 2 DL, OWL 2 EL, OWL 2 RL, and OWL 2 QL. OWL 2 semantics correspond to the semantics of description logics and the set of language primitives used determines the particular description logic in which an OWL ontology can be expressed. Prova and AspectOWL permit the selection of semantic profiles, which, in the case of AspectOWL, determines the model-theoretic semantics under which a theory is interpreted. These choices lead to different computational properties of each of the formalisms.

AspectOWL, being a static KR formalism, requires recomputation of all inferences as facts are added to the knowledge base. Incremental reasoning has not been implemented but might be in the future.

In the case of the use case presented in this paper, the resulting AspectOWL ontology has an expressiveness that corresponds to the description logic $\mathcal{ALCROIQ}$, which means that the problems of concept satisfiability and consistency checking are NExpTime-hard. Description logics are by design decidable, but AspectOWL, the semantics of which divert from pure DLs, are not guaranteed to be decidable. The non-decidability comes from the multi-dimensional interpretation. Since, however, in the underlying use case there is no interaction between the different context levels (the object and the temporal level have both access to the time individual T_DC_1, but this individual is rigid, i. e. its interpretation is context-independent), the ontology is decidable even under the AspectOWL full semantics. Prova is a combination of rule and scripting language. The semantics of the rule language correspond to Prolog, which makes Prova generally undecidable. Prova extends the declarative rule language by procedural attachments, a mechanism for making calls to procedures written in an imperative language, such as Java, which makes it impossible to make a generalized statement about the computational properties of the Prova system as a whole. Prova's messaging system, of which we primarily make use in the context of this research, is pattern-based with selectable inference regimes, such as DL reasoning.

A clear advantage of AspectOWL is its full backwards-compatibility with OWL 2 and the resulting ability to import and reuse existing knowledge from the many OWL ontologies that are publicly available as we did with PrOnto and the ontology design patterns while the Prova implementation was basically built from scratch. In in the context of the data wallet scenario existing standards such as Solid use RDF as their data model and are thereby directly compatible with AspectOWL.

6. Conclusions and Future Work

We described a GDPR-related use case for a distributed data wallet. The use case defines all typical stakeholders, namely an Identity Provider, a Data Wallet Provider, Relaying Parties as applications (a PictureApp and a SearchApp), and users. The main concepts of the use case are data access (from Relaying Parties or from users), and demonstration of consent that enables this data sharing. Depending on consent demonstrating or consent revoking actions, different possible states can emerge, rendering the use case non-monotonic.

We provided two implementations, one in AspectOWL, one in Prova. Both approaches were able to result in a representation of the problem domain which is sufficiently adequate for GDPR-related inference tasks, especially deontic state transitions resulting from actions such as demonstration of user consent. Prova with its reaction rule style messaging system and procedural attachments is capable of implementing the entire workflow of our given use cases.

Specifically, the Prova implementation is able to fully model the interaction of all parties (represented as agents) in real-time. The reactive messaging capabilities, the Java object support and the non-monotonic state transition semantics can model all possible states of the use case. The AspectOWL implementation provides both the ontology with types and description of the domain concepts, and the state transition modeling required by the use case. However, since AspectOWL is a monotonic formalism the representation of non-monotonic states using contexts may lead to an indefinite growth of the knowledge base.

While AspectOWL proved to be a suitable approach for the specification of the domain model Prova is the more practical approach for a real-world application. Therefore, the two implementations are complementing each other.

Future work may consist in combining the two approaches, by integrating and reusing the existing AspectOWL ontology in the Prova runtime system, combining the strengths of the two systems. This combined system may serve as a real-world back-end of an ecosystem of a distributed data wallet and applications. Investigating and implementing methods of incremental reasoning for the AspectOWL reasoner is also planned.

Acknowledgments

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project RECOMP (DFG – GZ: PA 1820/5-1).

References

[1] European Commission, Regulation (EU) 2016/679 of the European Parliament and of the Council, 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[2] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, T. Berners-Lee, A demonstration of the solid platform for social web applications, in: Proceedings of the 25th International Conference Companion on World Wide Web, WWW '16 Companion, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 2016, p. 223–226. URL: https://doi.org/10.1145/2872518.2890529. doi:10.1145/2872518.2890529.

[3] R. Schäfermeier, A. Paschke, Aspect-oriented ontology development, in: G. J. Nalepa, J. Baumeister (Eds.), Synergies Between Knowledge Engineering and Software Engineering, volume 626 of *Advances in Intelligent Systems and Computing*, Springer, 2018, pp. 3–30. URL: https://doi.org/10.1007/978-3-319-64161-4_1. doi:10.1007/978-3-319-64161-4_1.

[4] R. Schäfermeier, A. Paschke, Aspect-Oriented Ontologies: Dynamic Modularization Using Ontological Metamodeling, in: P. Garbacz, O. Kutz (Eds.), Proceedings of the 8th International Conference on Formal Ontology in Information Systems (FOIS 2014), volume 267 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2014, pp. 199 – 212.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Aksit, S. Matsuoka (Eds.), ECOOP'97 — Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1997, pp. 220–242.

[6] R. Filman, D. Friedman, Aspect-Oriented Programming Is Quantification and Obliviousness, Workshop on Advanced Separation of Concerns, OOPSLA (2000).

[7] K. Schild, A Correspondence Theory for Terminological Logics: Preliminary Report, in: J. Mylopoulos, R. Reiter (Eds.), Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991, Morgan Kaufmann, 1991, pp. 466–471.

[8] A. Kozlenkov, R. Penaloza, V. Nigam, L. Royer, G. Dawelbait, M. Schroeder, Prova: Rule-Based Java Scripting for Distributed Web Applications: A Case Study in Bioinformatics, in: T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, J. Wijsen (Eds.), Current Trends in Database Technology – EDBT 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 899–908.

[9] G. Kober, L. Robaldo, A. Paschke, Modeling medical guidelines by prova and shacl accessing fhir/rdf. use case: The medical abcde approach, in: dHealth 2022, IOS Press, 2022, pp. 59–66.

[10] A. Paschke, Rules and Logic Programming for the Web, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 326–381. doi:10.1007/978-3-642-23032-5_6.

[11] A. Kozlenkov, Prova Rule Language version 3.0 User's Guide (2010). URL: https://github.com/prova/prova/tree/master/doc.

[12] A. Paschke, M. Bichler, Knowledge representation concepts for automated SLA management, Decision Support Systems 46 (2008) 187–205. URL: https://www.sciencedirect.com/science/article/pii/S0167923608001206. doi:https://doi.org/10.1016/j.dss.2008.06.008.

[13] M. Palmirani, M. Martoni, A. Rossi, C. Bartolini, L. Robaldo, PrOnto: Privacy Ontology for Legal Reasoning, in: A. Kő, E. Francesconi (Eds.), Electronic Government and the Information Systems Perspective, Springer International Publishing, Cham, 2018, pp. 139–152.