ODRL-Based Resource Definition in Business Processes

Zakaria Maamar[1][a], Amel Benna[2][b], Minglin Li[3], Huiru Huang[3] and Yang Xu[3][c]
[1]*College of Computing and IT, University of Doha for Science and Technology, Doha, Qatar*
[2]*Department of Multimedia and Information Systems, CERIST, Algiers, Algeria*
[3]*School of Computer Science, Fudan University, Shanghai, China*

Keywords: Asset, Business Process, ODRL, Resource, Service.

Abstract: Despite the popularity of Resource-as-a-Service (RaaS) model, it falls short of providing low-level, flexible control over resources in terms of which category of users can use them, when and where users can use them, and how much users need to pay for them. To provide such a control, this paper presents 2 intermediary models between resources and services referred to as Resource-as-an-Asset (RaaA) and Asset-as-a-Service (AaaS), respectively. First, resource-related constructs are mapped onto asset-related constructs and then, the same asset-related constructs are mapped onto service-related constructs. A projection operator is developed to support construct mappings based on a set of pre-defined rules. To illustrate asset-related constructs, the paper resorts to the Open Digital Rights Language (ODRL) defining what is permitted, forbidden, or obliged over an asset. A system implementing construct mappings is presented in the paper as well.

1 INTRODUCTION

In the literature, a Business Process (BP) is a set of activities/tasks connected together based on a process model that specifies who does what, where, when, and why. A plethora of BPs exist ranging from product procurement to loan approval and claim processing.

In addition to process model specification, BPs' owners proceed with identifying (processing, storage, and communication) resources that these BPs would use at run-time. Because resources are valuable assets, frequently costly, and limited (Arias et al., 2018), providers could be selective when provisioning their resources using criteria like urgency of demands and best offers. As a result, BPs' owners need to respond to changes of resources by avoiding for instance, to be locked into particular providers (i.e., BP-resource strong coupling). A widely adopted solution to address the lock-into concern is to abstract resources into services leading into what the ICT community refers to as Anything-as-a-Service (XaaS) model. Cloud computing has been an excellent showcase of the benefits of XaaS through SaaS, PaaS, and IaaS models. However, this abstraction does not offer providers the means nor the guidelines of how to either permit or prohibit the use of their resources to a specific category of users, a specific period of time, a specific location, etc. To address this lack of means and guidelines, we resort to the Open Digital Rights Language (ODRL, (W3C, 2018)) first, to elevate resource to the level asset and second, to specify the policies that would either permit or prohibit the use of assets abstracting resources.

Thanks to ODRL, we achieve XaaS over 2 stages where first, a resource is exposed as an asset requiring the development of Resource-as-an-Asset (RaaA) model, and second, an asset is exposed as a service also requiring the development of Asset-as-a-Service (AaaS) model. An immediate benefit of the RaaA model is the possibility of specifying ahead of time the context in which the use of a resource would be either permitted or prohibited without questioning this resource's availabilities at run time nor jeopardizing the execution of BPs. This would give providers full control of their resources by for instance, designating particular users of resources, e.g., BPs' owners, and even rectifying the prohibited uses of resources. Mire benefits of the RaaA model are constraining the use of resources with parameters like maximum capacity and allocated time. AaaS model would also achieve other benefits like hiding the intrinsic characteristics of resources exposed as as-

[a] https://orcid.org/0000-0003-4462-8337
[b] https://orcid.org/0000-0002-9076-5001
[c] https://orcid.org/0000-0002-0958-8547

Maamar, Z., Benna, A., Li, M., Huang, H. and Xu, Y.
ODRL-Based Resource Definition in Business Processes.
DOI: 10.5220/0012047500003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 225-232
ISBN: 978-989-758-665-1; ISSN: 2184-2833

sets and then, services, and allowing a platform-independent control of these resources. Section 2 defines resource and ODRL, and suggests a running example. Section 3 presents resource, asset, and service models and their mappings. Section 4 implements these mappings. Section 5 concludes the paper and discusses future work.

2 BACKGROUND

This section presents briefly resource and ODRL and then, suggests a running example.

2.1 Resource in Brief

The concept of resource is not new in the literature and has been adopted in different domains like distributed artificial intelligence, cloud computing, and service computing. From a broader perspective, Baker et al. specialize resource into computational, consumed, and produced, associating each type with a separate lifecycle that would stress out this type's unique characteristics (Baker et al., 2018). Lucchim et al. consider resources as *"directly-accessible components handled through a standard common interface"* (Lucchim et al., 2008). Finally, Hofman adopts everything-as-a-resource as a base for building seamless interoperable platforms in the IoT world (Hofman, 2015). To enforce the relationship between consumers of resources and providers of resources, Maamar et al. define 5 consumption properties referred to as *unlimited*, *shareable*, *limited*, *limited but-extensible*, and *non-shareable* (Maamar et al., 2016). Each property has a lifecycle consisting of states depicting each the respective actions that consumers and providers are expected to perform. By analogy with these properties, Pufahl et al. in (Pufahl et al., 2021) mention attributes that describe a resource in terms of capabilities, capacities, and availability at a given point in time, and decompose resources into active and passive.

2.2 ODRL in Brief

ODRL is an example of rights expression languages providing a flexible and interoperable information model, vocabulary, and encoding mechanisms to represent statements about the usage of assets. An asset is an identifiable resource (or collection of resources) such as data, media, applications, and services. Fig. 1 is an excerpt of ODRL information-model.

Policy could include one to many permission, prohibition, or duty rules. Permission allows an action over an asset if all constraints are satisfied and if all duties are fulfilled. Prohibition disallows an action over an asset if all constraints are satisfied. And, duty is to exercise an action that can be over an asset or not. It is fulfilled when all constraints are satisfied.

Party is an entity or collection of entities that could correspond to a person, group of persons, organisation, or agent. A party can fulfill different roles including assigner (issuer of the rule), assignee (recipient of the rule), and tracked party (indicating who is being traced).

Constraint is used to refine the specification of an action and a party/asset collection or to declare conditions applicable to a rule.

2.3 Running Example

It is about John who is visiting Melissa in Paris. They agree to meet in a coffee shop next to Melissa's office. John can either ride a taxi or take a bus to reach the coffee shop. It starts when John invokes an online program for itinerary planning so that he would approximately know how long it would take to reach the coffee shop. The program is integrated into a Web site. Prior to deciding on the transportation means, John checks the weather forecast by invoking another program that interacts with a dedicated database. Finally, he accesses the coffee shop's Web site deployed on a virtual server to submit a reservation.

In compliance with the RaaS model, existing practices would bind BPs to on-premise and in-the-cloud resources through services. However, this binding would not allow a low-level, flexible control of these resources. For instance, while databases support concurrency, identifying particular tasks that would be eligible for concurrent access as well as the authority that would approve this eligibility is not doable. Contrarily, ODRL constructs like assignee and assigner would help target these tasks and specify this authority. Another example is how to enforce specific time slots or locations. Contrarily, ODRL constructs like refinement/constraint would achieve this enforcement. Finally, while payment covers a resource as a whole, paying for specific operations using a resource as well as handling the consequences of not fulfilling a payment are not possible. Contrarily, ODRL constructs like duty and action with refinement/constraint would help achieve this targeted payment.

3 CONNECTING CONCEPTS

This section presents the mappings of RaaA onto AaaS and the operator supporting these mappings.

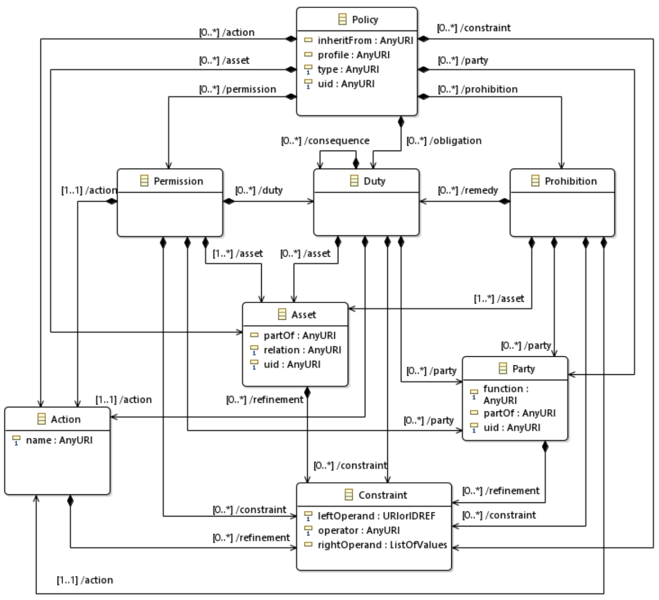


Figure 1: Excerpt of ODRL information model diagram.

3.1 Overview

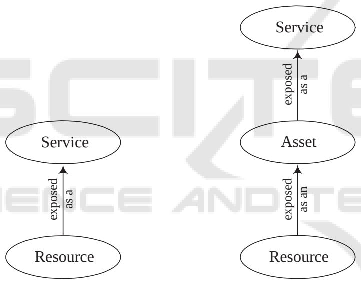


Figure 2: From RaaS to RaaA and then, AaaS.

Fig. 2 (a) is the current practice of concretizing the RaaS model. Contrarily, Fig. 2 (b) is the future practice that implements first, the RaaA model and then, the AaaS model. We proceed in Section 3.2 with developing and/or adopting resource, asset, and service models. Then, we do the same in Section 3.3 by creating 3 separate layers, one per model, in a way that the layers would be semantically connected together using a specific operator that would project a construct from one model to another. Having the AaaS model on top of the RaaA model addresses some of the RaaS model's shortcomings like lack of means for resource use. Contrarily, the AaaS model accommodates these means using policies that could integrate permission, prohibition, and obligation rules.

3.2 Model Development

Resource Model. In Fig. 3, Resource as a core class is connected to Provider class and is described with a set of functional characteristics captured through a relevant class. These characteristics depict the purpose of using a resource in terms of processing, storing, or communicating. Resource is also connected to another class that is Interface acting as an entry point (Hintsch et al., 2015) to a particular instantiated resource at run-time. Practically speaking, an entry point directs a particular demand of use to a resource and consists of a set of operations identified with Operation class. To have a controlled use over instantiated resources at run-time, both Interface and Operation classes are connected to Non_Functional Characteristic class depicting by whom/how/when/where an instantiated resource is used. Examples of non-functional characteristics are reported in Table 1.

Table 1: Examples of non-functional characteristics.

Non-functional characteristic		Concerned	
Name	Description	Interface	Operation
Occurrence	single use/multiple uses	✓	
Payment	with fees/without fees	✓	
Availability	unlimited/limited/... use	✓	✓
Chronology	specific order/random order		✓

Asset Model. We adopt the asset-model diagram depicted in Fig. 1 and explained in Section 2.2. Worth mentioning, in compliance with Fig. 2 (b), the role of Action class in first, identifying what is permitted/prohibited/obliged over an asset and second, setting the

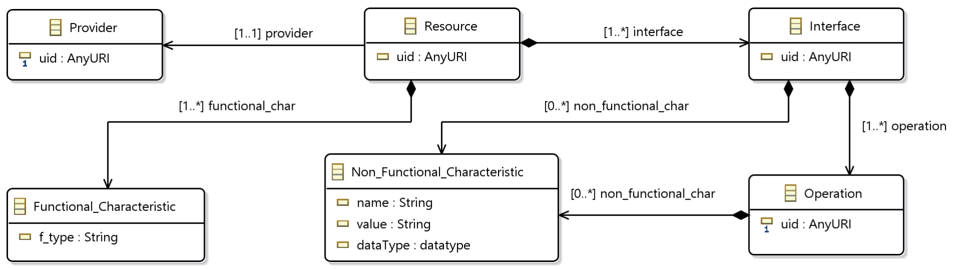


Figure 3: Representation of the resource-model diagram.

stage of exposing an asset as a service.

Service Model. In Fig. 4, Service as a core class has a provider identified with Provider class, is described with a set of non-functional characteristics captured with QoS class, and is associated with several Service Level Agreements (SLA) that track the instantiated services' ongoing provisioning levels to users at run-time. These users are identified with Consumer class. To frame the interactions between consumers of services and providers of services, SLA are established referring to specific non-functional characteristics. Finally, in Fig. 4, Operation class refers to how an instantiated service is performed at run time.

3.3 Model Overlay

To achieve the transition from the RaaA model to the AaaS model as per Fig. 2 (b), we associate this transition with a projection operator (σ). This operator would permit in a bottom-up way to semantically correspond a construct, either class or relation along with their relevant attributes, in a source model into another either a simple or a complex construct in a destination model. First, simple construct would be either class or relation along with their relevant attributes. Second, complex construct would be a set of classes and relations connecting these classes together. In the following, $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{A}$ stand for class, relation, and attribute constructs, respectively.

Prior to proceeding with the different projections, the following points are worth mentioning:

1. When rule class is used, it encompasses permission, prohibition, and duty classes.
2. When rel(A,B) is used, rel as a relation has A class as source and B class as destination. The connection between the 2 classes is set by having rel included as an attribute of B datatype in A class.
3. Because party class has a dual role being assigner and assignee, this role is differentiated using *function* attribute that could be set to either *assigner* or *assignee*. In addition, during the projection of the asset model's constructs, party class appears twice because of the dual role.

Projecting Resources Onto Assets. The resource model (*res*) is the source and asset model (*ass*) is the destination. Applying σ to the resource model's constructs, i.e., classes, attributes, and relations, would lead to the following semantic correspondences in the asset model's constructs:

1. σ(*res*:$\mathcal{C}$.Provider) → *ass*:$\mathcal{C}$.Party allowing to indicate those provisioning resources.
   (a) Party.*uid* ← Provider.*uid* allowing to uniquely link the provider to a party.
   (b) Party.*function* ← "assigner" allowing to set the role of the party using an ODRL-predefined value that is "assigner".
2. σ(*res*:$\mathcal{C}$.Resource) → ϕ allowing to hide a resource from any potential uses.
3. σ(*res*:$\mathcal{C}$.Interface) → *ass*:$\mathcal{C}$.Asset allowing to treat an interface as an asset, which means controlling the use of a resource through its different interfaces.
   (a) Asset.*uid* ← Interface.*uid* allowing to uniquely link the interface to an asset.
   (b) Asset.*relation* ← "target" allowing to confirm the control of the interface using an ODRL-predefined value that is "target".
4. σ(*res*:$\mathcal{C}$.Operation) → *ass*:$\mathcal{C}$.Action allowing to treat an interface's operation as an asset's action.
   (a) Action.*name* ← *any uid* allowing to set the name of the action with either a predefined value in compliance with ODRL core-vocabulary or a new value in compliance with ODRL profile mechanism.
5. σ(*res*:$\mathcal{C}$.Functional_Characteristic) → ϕ. The lack of corresponding constructs is justified by the fact that ODRL does not "focus" on assets' functional characteristics like what assets are meant for.
6. σ(*res*:$\mathcal{C}$.Non_Functional_Characteristic) → *ass*:$\mathcal{C}$.Constraint allowing to treat a non-functional characteristic as a constraint.
   (a) Constraint.*leftOperand* ← *any uid* allowing to set the name of the constraint with either a pre-

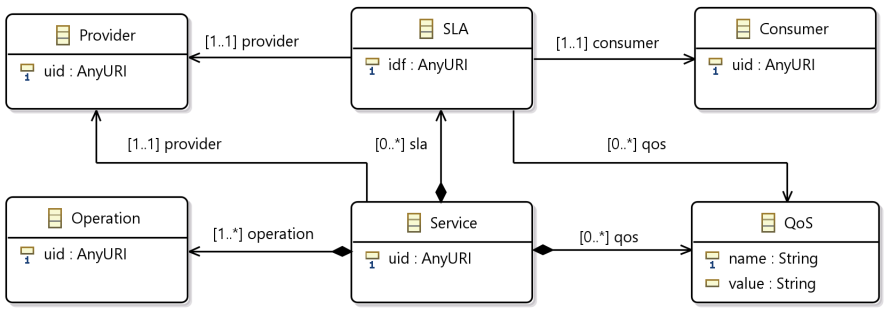


Figure 4: Representation of the service-model diagram.

defined value in compliance with ODRL core-vocabulary or a new value in compliance with ODRL profile mechanism. This name should semantically correspond to the non-functional characteristic's name to be used as an *uid*.

(b) Constraint.*operator* ← "eq" ⊕ "gt" ⊕ ··· allowing to select an operator function that will set the constraint's condition to satisfy using an ODRL-predefined value.

(c) Constraint.*rightOperand* ← Non_Functional_Characteristic.*value* allowing to set the constraint's value that will be compared to the *leftOperand*.

(d) Constraint.*unit* ← Non_Functional_Characteristic.*dataType* allowing to adopt the data type of the value of the non-functional characteristic.

The projection of the resource model's aforementioned classes onto the asset model's classes results into a partial overlay of this asset model's constructs. Indeed, many classes, of which some are mandatory[1] are missing. To consider ODRL information-model's mandatory classes and their attributes and to achieve a complete overlay, Policy, Rule, and Party[2] classes are considered with focus on updating their attributes as follows:

1. Policy.*uid* ← *any uid* allowing to set the identifier of the policy that will integrate the necessary rules that will control the use of an asset related to an interface.
2. Policy.*type* ← *"Agreement"*⊕ *"Offer"* ⊕ ··· allowing to set the type of the policy that will integrate the necessary rules using an ODRL-predefined value like "Agreement" and "Offer".
3. Rule.*uid* ← *any uid* allowing to set the identifier of a rule.
4. Party.*uid* ← *any uid* allowing to indicate those who will be either granted or denied using resources in contrast to those provisioning resources. This assumes that users, referred to as party, of resources are considered as a non-functional characteristic.
5. Party.*function* ← "assignee" allowing to set the role of the party that will be either granted or denied the use of a resource exposed as an interface using an ODRL-predefined value that is "assignee". This assumes that the party is considered as a non-functional characteristic.

After examining the different classes, we now project the relations in the resource model onto the relevant constructs in the asset model.

1. $\sigma(res{:}\mathcal{R}$.provider($\mathcal{C}$.Resource,$\mathcal{C}$.Provider)) $\rightarrow \phi$ allowing to hide the relation that exists between a resource and its provider.
2. $\sigma(res{:}\mathcal{R}$.interface($\mathcal{C}$.Resource,$\mathcal{C}$.Interface)) $\rightarrow \phi$ allowing to hide the relation that exists between a resource and its interface.
3. $\sigma(res{:}\mathcal{R}$.operation($\mathcal{C}$.Interface,$\mathcal{C}$.Operation)) $\rightarrow$ $ass{:}\mathcal{R}$.action($\mathcal{C}$.Rule,$\mathcal{C}$.Action) $\cup$ $ass{:}\mathcal{R}$.asset($\mathcal{C}$.Rule,$\mathcal{C}$.Asset) allows to link an action corresponding to an operation to an asset corresponding to an interface through a rule.
   (a) Rule.*action* ← Action allowing to include the reference of the action in a rule.
   (b) Rule.*asset* ← Asset allowing to include the reference of the asset in a rule.
4. $\sigma(res{:}\mathcal{R}$.functional_char($\mathcal{C}$.Resource, $\mathcal{C}$.Functional_Characteristic)) $\rightarrow$ ϕ allowing to hide the relation that exists between a resource and its functional characteristics.
5. $\sigma(res{:}\mathcal{R}$.non_functional_char($\mathcal{C}$.Operation, $\mathcal{C}$.Non_Functional_Characteristic)) $\rightarrow$

[1]Using "should" and "may" to describe ODRL constructs, we label them as either mandatory or optional.

[2]Though Party is included in the first round of projection as "assigner", it is included again to identify "assignee".

ass:$\mathcal{R}$.refinement($\mathcal{C}$.Action,$\mathcal{C}$.Constraint)) allowing to link a constraint corresponding to a non-functional characteristic to an action corresponding to an operation.

(a) Action.*refinement* ← Constraint allowing to include the reference of the constraint in an action.

6. σ(res:$\mathcal{R}$.non_functional_char($\mathcal{C}$.Interface, $\mathcal{C}$.Non_Functional_Characteristic)) → ass:$\mathcal{R}$.constraint($\mathcal{C}$.Rule,$\mathcal{C}$.Constraint) ∪ ass:$\mathcal{R}$.asset($\mathcal{C}$.Rule,$\mathcal{C}$.Asset) allowing to link a constraint corresponding to a non-functional characteristic to an asset corresponding to an interface through rules.
   (a) Rule.*constraint* ← Constraint allowing to include the reference of the constraint in a rule.
   (b) Rule.*asset* ← Asset allowing to include the reference of the asset in a rule.

By analogy with the partial overlay of the asset model due to the absence of some classes, we proceed with the same by considering more relations, permission, prohibition, obligation, and party, with focus on updating their attributes as follows:

1. Policy.*permission* ← Permission allowing to include the reference of the permission rule in a policy. This reference corresponds to the relation permission(Policy,Permission).
2. Policy.*prohibition* ← Prohibition allowing to include the reference of the prohibition rule in a policy. This reference corresponds to the relation prohibition(Policy,Prohibition).
3. Policy.*obligation* ← Duty allowing to include the reference of the duty rule in a policy. This reference corresponds to the relation obligation(Policy,Duty).
4. Policy.*asset* ← Asset allowing to include the reference of the asset in a policy. This reference corresponds to the relation asset(Policy,Asset). Indeed, as asset is common to all rules belonging to the same policy, we associate it with the Policy and not Rule construct.
5. Policy.*party* ← Party allowing to include the reference of the party in a policy. This reference corresponds to the relation party(Policy,Party) where the party is set as assigner.
6. Rule.*party* ← Party allowing to include the reference of the party in a rule. This reference corresponds to the relation party(Rule,Party) where the party is set as assignee and considered as a non-functional characteristic.

Projecting Assets Onto Services. After completing the projection of the resource model's constructs onto the asset model's constructs, we now project these asset model's constructs onto the service model's constructs. Thus, the asset model (ass) is the source and service model (ser) is the destination. Applying σ to the asset model's constructs, i.e., first, classes, attributes, and relations, would lead to the following semantic correspondences in the service model's constructs:

1. σ(ass:$\mathcal{C}$.Asset) → ser:$\mathcal{C}$.Service allowing to expose an asset as a service, which means "shielding" future users from any technical specifications related to the asset.
   (a) Service.*uid* ← Asset.*uid* allowing to uniquely link the asset to a service.
2. σ(ass:$\mathcal{C}$.Party) → ser:$\mathcal{C}$.Provider subject that Party.*function* is set to "assigner". This projection allows to expose the party as a provider.
   (a) Provider.*uid* ← Party.*uid* allows to uniquely identify the provider of a service.
3. σ(ass:$\mathcal{C}$.Party) → ser:$\mathcal{C}$.Consumer subject that Party.*function* is set to "assignee". This projection allows to expose the party as a consumer.
   (a) Consumer.*uid* ← Party.*uid* allows to uniquely identify the consumer that will be either granted or denied the invocation of a service.
4. σ(ass:$\mathcal{C}$.Policy) → ser:$\mathcal{C}$.SLA allowing to define under the guidance of a service provider the necessary clauses of an agreement between this service and a future consumer.
   (a) SLA.*uid* ← Policy.*uid* allowing to uniquely link a policy to an agreement.
5. σ(ass:$\mathcal{C}$.Rule) → ϕ. The lack of corresponding constructs is justified by the fact that the service model does not put any restrictions on for instance, when and where SLAs should be permitted, prohibited, or enforced.
6. σ(ass:$\mathcal{C}$.Action) → ser:$\mathcal{C}$.Operation allowing to associate an action with a particular operation for invoking a service.
   (a) Operation.*uid* ← Action.*name* where *name* would be used as the operation's id.
7. σ(ass:$\mathcal{C}$.Constraint) → ser:$\mathcal{C}$.QoS allowing to develop the non-functional characteristics featuring the quality-of-service of a service.
   (a) QoS.*name* ← Constraint.*leftOperand* allowing to set the non-functional characteristic's name.
   (b) QoS.*value* ← Constraint.*rightOperand* allowing to set the non-functional characteristic's value.

(c) QoS.*operator* ← Constraint.*operator* allowing to set the condition to satisfy the non-functional characteristic's value.

(d) QoS.*unit* ← Constraint.*unit* allowing to set the unit used for the value of the non-functional characteristic.

After examining the different classes, we now project the relations in the asset model onto the relevant constructs in the service model.

1. σ(*ass*:$\mathcal{R}$.permission($\mathcal{C}$.Policy,$\mathcal{C}$.Permission)) $\rightarrow \phi$. The lack of corresponding constructs confirms that SLAs are free of any permissions permitting their enablement. The same applies to σ(*ass*:$\mathcal{R}$.prohibition($\mathcal{C}$.Policy,$\mathcal{C}$.Prohibition)) and σ(*ass*:$\mathcal{R}$.obligation($\mathcal{C}$.Policy,$\mathcal{C}$.Duty)).

2. σ(*ass*:$\mathcal{R}$.asset ($\mathcal{C}$.Policy,$\mathcal{C}$.Asset)) $\rightarrow$ *ser*:$\mathcal{R}$.sla($\mathcal{C}$.Service,$\mathcal{C}$.SLA) allowing to link a SLA corresponding to a policy to a service corresponding to an asset.

   (a) Service.*sla* ← SLA allowing to include the reference of the SLA in a service.

3. σ(*ass*:$\mathcal{R}$.party($\mathcal{C}$.Policy,$\mathcal{C}$.Party)) $\rightarrow$ *ser*:$\mathcal{R}$.provider($\mathcal{C}$.SLA,$\mathcal{C}$.Provider) $\cup$ *ser*:$\mathcal{R}$.provider($\mathcal{C}$.Service,$\mathcal{C}$.Provider) allowing to ensure that the provider that sets a service's SLAs corresponding to an assigner will be referenced in both the SLAs and service associated with these agreements.

   (a) SLA.*provider* ← Provider allowing to include the reference of the provider in a SLA.

   (b) Service.*provider* ← Provider allowing to include the reference of the provider in a service.

4. σ(*ass*:$\mathcal{R}$.party($\mathcal{C}$.Rule,$\mathcal{C}$.Party)) $\rightarrow \phi$ allows to hide the relation that exists between the rule and party, where Party.*function* is set to "assignee".

5. σ(*ass*:$\mathcal{R}$.constraint($\mathcal{C}$.Rule,$\mathcal{C}$.Constraint)) $\rightarrow \phi$ allowing to hide the relation that exists between the rule and constraint.

6. σ(*ass*:$\mathcal{R}$.action($\mathcal{C}$.Rule,$\mathcal{C}$.Action)) $\rightarrow \phi$ allowing to hide the relation that exists between the rule and action.

7. σ(*ass*:$\mathcal{R}$.refinement($\mathcal{C}$.Action,$\mathcal{C}$.Constraint)) $\rightarrow \phi$ allowing to hide the relation that exists between the action and constraint.

8. σ(*ass*:$\mathcal{R}$.permission($\mathcal{C}$.Policy,$\mathcal{C}$.Permission)) $\rightarrow \phi$ allowing to hide the relation that exists between the policy and permission.

9. σ(*ass*:$\mathcal{R}$.prohibition($\mathcal{C}$.Policy,$\mathcal{C}$.Prohibition)) $\rightarrow \phi$ allowing to hide the relation that exists between the policy and prohibition.

10. σ(*ass*:$\mathcal{R}$.obligation($\mathcal{C}$.Policy,$\mathcal{C}$.Duty)) $\rightarrow \phi$ allowing to hide the relation that exists between the policy and permission.

    (a) SLA.*consumer* ← Consumer allowing to include the reference of a consumer corresponding to an assignee in a SLA. This reference corresponds to the relation $\mathcal{R}$.consumer($\mathcal{C}$.SLA,$\mathcal{C}$.Consumer)

    (b) Service.*operation* ← Operation allowing to include the reference of an operation in a service. This reference corresponds to the relation $\mathcal{R}$.operation($\mathcal{C}$.Service,$\mathcal{C}$.Operation)

    (c) Service.*qos* ← QoS allowing to include the reference of quality of service in a service. This reference corresponds to the relation $\mathcal{R}$.qos($\mathcal{C}$.Service,$\mathcal{C}$.QoS)

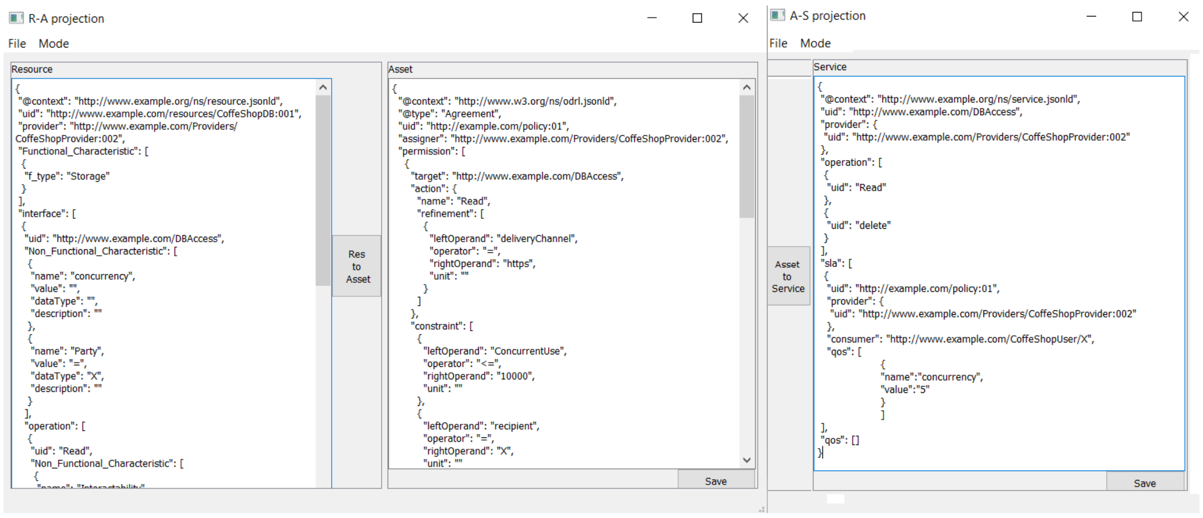
    (d) SLA.*qos* ← Qos allowing to include the reference of quality of service in a SLA. This reference corresponds to the relation $\mathcal{R}$.qos($\mathcal{C}$.SLA,$\mathcal{C}$.Qos)

4 PROTOTYPE

Developed in Python 3.0 along with using sys, json, and pyqt5 libraries, the system implementing the projection operator σ consists of 5 modules, *instantiation*, *R-A projection*, *A-S projection*, *A-storage*, and *S-storage*, and 5 repositories, *resources*, *R-A projection rules*, *A-S projection rules*, *assets*, and *services*. *resource instance*, *asset instances*, and *service instances* are the system's outcomes.

To demonstrate how the system is put into action, we adopt coffee-shop-database as a resource from the running example. This resource includes many interfaces with focus here on DBaccess that implements read, update, delete, and save operations (Fig. 5's left panel). As per Fig. 3, on top of an interface's non-functional characteristics like concurrency for DBaccess, an operation can also have non-functional characteristics like interactability and availability.

Following resource specification through a dedicated editor, *R-A projection* module is initiated to map resources specified in JSON files onto assets. Fig. 5's center panel partially shows the outcome of projecting the coffee-shop-database resource into ODRL constructs such as asset, action, and rule. Among these constructs that could be set manually, we mention for instance, a permission rule to read concurrently the database, a prohibition rule to proscribe updating the database concurrently and during a specific period of time, and an obligation rule to backup the database using the *archive* action. Finally, *A-S projection* mod-



```
{
 "@context": "http://www.example.org/ns/resource.jsonld",
 "uid": "http://www.example.com/resources/CoffeShopDB:001",
 "provider": "http://www.example.com/Providers/
CoffeShopProvider:002",
 "Functional_Characteristic": [
  {
   "f_type": "Storage"
  }
 ],
 "interface": [
  {
   "uid": "http://www.example.com/DBAccess",
   "Non_Functional_Characteristic": [
    {
     "name": "concurrency",
     "value": "",
     "dataType": "",
     "description": ""
    },
    {
     "name": "Party",
     "value": "=",
     "dataType": "X",
     "description": ""
    }
   ],
   "operation": [
    {
     "uid": "Read",
     "Non_Functional_Characteristic": [
      {
       "name": "Interactability"
```

```
{
  "@context": "http://www.w3.org/ns/odrl.jsonld",
  "@type": "Agreement",
  "uid": "http://example.com/policy:01",
  "assigner": "http://www.example.com/Providers/CoffeShopProvider:002",
  "permission": [
    {
      "target": "http://www.example.com/DBAccess",
      "action": {
        "name": "Read",
        "refinement": [
          {
            "leftOperand": "deliveryChannel",
            "operator": "=",
            "rightOperand": "https",
            "unit": ""
          }
        ]
      },
      "constraint": [
        {
          "leftOperand": "ConcurrentUse",
          "operator": "<=",
          "rightOperand": "10000",
          "unit": ""
        },
        {
          "leftOperand": "recipient",
          "operator": "=",
          "rightOperand": "X",
          "unit": ""
```

```
{
 "@context": "http://www.example.org/ns/service.jsonld",
 "uid": "http://www.example.com/DBAccess",
 "provider": {
  "uid": "http://www.example.com/Providers/CoffeShopProvider:002"
 },
 "operation": [
  {
   "uid": "Read"
  },
  {
   "uid": "delete"
  }
 ],
 "sla": [
  {
   "uid": "http://example.com/policy:01",
   "provider": {
    "uid": "http://www.example.com/Providers/CoffeShopProvider:002"
   },
   "consumer": "http://www.example.com/CoffeShopUser/X",
   "qos": [
                {
                "name":"concurrency",
                "value":"5"
                }
                ]
 ],
 "qos": []
}
```

Figure 5: Screenshots of resource-asset-service projections.

ule is initiated to project assets into services as per Fig. 5's right panel. Here, DBAccess asset has become a service having in compliance with Fig. 4 a provider, an operation, and a sla, for example. While some details during *A-S projection* to specify services are set automatically because of the pre-defined rules, the rest like reference to a particular consumer and QoS are set manually by the end-user.

5 CONCLUSION

This paper presented a customized low-level control over resources that BPs execution would use at runtime. We elevated resource to the level of asset and then, specified policies that would either permit or prohibit the use of resources exposed as assets and afterward, as services. We resorted to ODRL to specify these policies. We also presented how the double exposure of resource into asset and then, service was achieved by developing 2 models, RaaA and AaaS. A projector operator achieved the double exposure whose technical doability was verified based on an in-house system. In term of future work, we would like to apply the system to more case studies along with checking the soundness of RaaA and AaaS models.

REFERENCES

Arias, M., Saavedra, R., Marques Samary, M., Munoz-Gama, J., and Sepulveda, M. (2018). Human Resource Allocation in Business Process Management and Process Mining: A Systematic Mapping Study. *Management Decision*, 56.

Baker, T., Ugljanin, E., Faci, N., Sellami, M., Maamar, Z., and Kajan, E. (2018). Everything as a Resource: Foundations and Illustration through Internet-of-Things. *Computers in Industry*, 94.

Hintsch, J., Schrödl, H., Scheruhn, H., and Turowski, K. (2015). Industrialization in Cloud Computing with Enterprise Systems: Order-to-Cash Automation for SaaS Products. In *Proceedings of WI'2015*, Osnabrück, Germany.

Hofman, W. (2015). Towards a Federated Infrastructure for the Global Data Pipeline. In *Proceedings of I3E'2015*, Delft,The Netherlands.

Lucchim, R., Millot, M., and Elfers, C. (2008). Resource oriented Architecture and REST: Assessment of Impact and Advantage on INSPIRE. JRC Scientific and Technical Reports EUR 23397 EN - 2008, JRC European Commission.

Maamar, Z., Faci, N., Sakr, S., Boukhebouze, M., and Barnawi, A. (2016). Network-based Social coordination of Business Processes. *Information Systems*, 58.

Pufahl, L., Ihde, S., Stiehle, F., Weske, M., and Weber, I. (2021). Automatic resource allocation in business processes: A systematic literature survey. *CoRR*, abs/2107.07264.

W3C (2018). ODRL Information Model 2.2. https://www.w3.org/TR/2018/REC-odrl-model-20180215/. (Visited January 2021).