Towards a MaaS Service for Cloud Service Interoperability

Nour El Houda Bouzerzour¹,², and Yahya Slimani¹,³
¹LISI Laboratory of Computer Science for Industrial Systems, Carthage University, Tunis, Tunisia
²ENSI, La Manouba University, Manouba, Tunisia
³ISAMM, La Manouba University, Manouba, Tunisia

Keywords: Cloud Computing, Cloud Service Description, Service Interoperability, Transformation Rules, ATL, WSDL.

Abstract: Cloud computing is an emerging computing paradigm, which provides high service availability, high scalability as well as low usage costs. This has encouraged enterprises and individual users to embrace cloud technology. However, the lack of service interoperability (also known as the vendor lock-in) issue still persists. The vendor lock-in is caused by the cloud service providers who aim to prevent the clients from switching to other clouds or providers. The solutions to overcome the vendor lock-in addressed a specific cloud actor or a specific cloud model, which makes them not generic. Thus, we present in this paper, our Cloud Interoperability Pivot Model (CIPiMo). CIPiMo is a Model-as-a-Service, which standardizes the cloud service description languages by transforming them into a Generic Cloud Service Description model (GCSD) to make them interoperable. We rely on MDE techniques to achieve a Model-to-Model transformation. Therefore, we define mappings between the source description languages (OWL-S and WSDL) and the target language (GCSD). Furthermore, we illustrate our proposed meta-models for each language and we implement our transformations using ATL with OCL. Eventually, we use a static analyzer (AnATLyzer IDE) to validate the correctness of our transformations. We provide use cases to demonstrate the applicability of our approach.

1 INTRODUCTION

Cloud computing has known a huge spread and popularity among providers and clients. The reason being that cloud computing offers an on-demand access to a pool of shared computing resources (Liu et al., 2011) with high availability and scalability, as well as in a "pay-as-you-go" fashion. Despite the numerous advantages of cloud computing, providers and clients are still hesitant to adopt it because of its prominent obstacles. The security and privacy concerns, the service recovery from disastrous situations and the SLA management are among these obstacles. However, the most prominent issue is the lack of cloud service interoperability. The latter is defined as the ability of heterogeneous systems to communicate, whether they are deployed on the same cloud or on multiple clouds (Opara-Martins et al., 2014). The cloud providers, who offer proprietary services and decline standardization, are causing the service interoperability issue. This is also known as the vendor lock-in, where the providers prevent the clients from switching from one provider to another or from one cloud to another in interconnected cloud environments (Opara-Martins et al., 2014). The solutions that were proposed in the literature (such as brokers, standards and semantic approaches) either target a single cloud actor (client or provider), they are specific to a certain technology, or they do not address all the cloud models (SaaS, PaaS, IaaS). Therefore, these approaches are not generic (Bouzerzour et al., 2020b). Other solutions such as RESTful APIs and gRPC are widely adopted to enable systems’ interoperability. However, they are still faced with issues. The semantic REST APIs are not adopted or recognized by commonly used tools (Cheron et al., 2019). Furthermore, the communication between two cloud systems using different protocols (e.g: REST HTTP protocol and MQTT (Message Queuing Telemetry Transport) protocol) requires a protocol adapter to happen. Eventually, The use of RESTful APIs does not solve the semantic inconsistencies, which are created by adopting heterogeneous cloud systems’ descriptions (Baudoin et al., 2014). Therefore, in this paper, we propose a cloud service interoperability approach based on the standardiza-
standardization of cloud service descriptions. Thus, we define the following research hypothesis:

**RH1:** the heterogeneity of cloud service descriptions is among the reasons causing vendor-lock-in.

**RH2:** standardizing service descriptions will allow their interoperability.

To enable the standardization of cloud service descriptions, the authors of (Ghazouani and Slimani, 2017) proposed a Generic Cloud Service Description model (GCSD). The GCSD describes cloud services in a standardized and comprehensive manner (from multiple aspects), which was demonstrated by use case scenarios that describe different services from different delivery models (SaaS, PaaS, IaaS) (Ghazouani and Slimani, 2017). Therefore, we rely on an MDE model-to-model transformation technique to transform heterogeneous description languages to the GCSD to make them standardized and interoperable. MDE is a software engineering approach, which uses human and machine-readable models. MDE considers models as first class entities to express specifications of systems at different levels of abstraction. The model’s elements, the relations between them and their constraints are described by a meta-model (Bezivin et al., 2005). We rely on MDE because it is expected to gain more growth in the software industry, as it was stated and approved by various high-quality research and studies that MDE is indeed able to provide effective and helpful solutions to improve the software development process (Brambilla et al., 2017).

Whereas Section 3, we present the mappings of the source description languages to the GCSD as well as their meta-models. In Section 4 we implement our transformation rules, and we discuss the results. Section 5 discusses the static validation of our transformations using AnATLyzer. In Section 6, we present the related works and in Section 7 we conclude this paper.

## 2 CLOUD SERVICE INTEROPERABILITY PIVOT MODEL (CIPiMo)

CIPiMo is an MDE-based MaaS (Model as a Service) service (aka. Modeling as a Service), which enables cloud service interoperability by transforming heterogeneous service descriptions into a GCSD. MaaS is a SaaS variant (Software as a Service), which allows users to deploy and execute model-driven and modeling services over the Internet, and it provides an interface for the client to communicate with the services. Among the main contributions of MaaS is enabling interoperability between tools and systems by bridging the gaps between their specifications (Bruneliere et al., 2010). Thus, CIPiMo aims to overcome the vendor-lock-in by promoting: (i) client-centric interoperability, which allows enterprises or end-users to adopt a multi-cloud strategy to interoperate their services, which are deployed on different providers or clouds, and it encourages enterprises to migrate their legacy systems to the cloud (Bouzerzour et al., 2020b); and (ii) provider-centric interoperability, which enables cloud federations between SME (Small and Medium Enterprises) service providers to gain more computing power and scalability, or hybrid cloud strategy for bursting the resources at peak moments (Bouzerzour et al., 2020b).

This transformation mediator maps the concepts and elements of source languages to the GCSD concepts. Then, it applies a set of transformation rules to transform the source language into the generic description. Hence, the service descriptions will be unified and interoperable. We chose a transformation through a pivot rather than a direct transformation from a source language to a target language because it requires fewer transformations, especially if the number of languages to be transformed is substantial (Boukhari et al., 2012). Figure 1 presents an overview of CIPiMo: in the cloud environment, service providers describe their services using different CSD languages (such as OWL-S and WSDL), which will be transformed using CIPiMo by applying the transformations corresponding to the proposed map-

![Figure 1: Our proposed interoperability model CIPiMo (Bouzerzour et al., 2020a).](image-url)
The mappings for each language. Then, the resulting descriptions will be standardized and ready to be published. The mapping phase is achieved by the manual extraction of different elements and their corresponding definitions from the corresponding specification document of each language. Then, we align the source element with its equivalent target element.

To demonstrate the functionality of our proposed pivot model we provide use cases, which transform WSDL and OWL-S service descriptions to the GCSD. We fully recognize that WSDL and OWL-S were not originally created for describing cloud services. However, both of these languages were used by researchers, who considered cloud services as web services, to describe cloud services (Ghazouani and Slimani, 2017). In (Zhou et al., 2011), the authors used WSDL-S for semantically describing services for SaaS discovery. Whereas, in (Goscinski and Brock, 2010), a WSDL file extension was proposed to take into consideration cloud characteristics. OWL-S was used in (Martino et al., 2014) to semantically describe Microsoft Azure API functional and non-functional properties and it was also used in (Karim et al., 2014) to define cloud services and their Quality of Service (QoS). Moreover, the available cloud service description and modeling approaches that were presented in the literature (such as (Bergmayr et al., 2014) (Perez and Rumpe, 2014) (Andrikopoulos et al., 2014)) addressed the description of the deployment, the configuration, and the provisioning of the cloud service rather than the description of the service properties and functionality. Furthermore, the majority of the cloud description and modeling languages lack formal documentation or specification documents to define in details their constructs and elements. Unlike that, WSDL and OWL-S are thoroughly documented in their specification documents (Christensen et al., 2001) (Martin et al., 2004). Moreover, many legacy systems were based upon these languages and they provide all the basic required concepts to describe a service and to demonstrate our proposal. Therefore, in the aim of demonstrating the genericity of our pivot model, we chose to run our interoperability model on WSDL (which is a syntactic language) and on OWL-S (which a semantic language), respectively, to unify their description and make them interoperable.

3 PROPOSED MAPPING RULES

In this section we present the mappings from WSDL and OWL-S to the GCSD. Furthermore, we illustrate our proposed exhaustive meta-models for each language. To model the description languages meta-models we use Eclipse Modeling Framework (EMF) ECORE meta-model as the de-facto standard modeling framework in the industry.

3.1 Mapping WSDL to GCSD

WSDL (Web Services Description Language) (Christensen et al., 2001) is an XML format to describe web services as collections of network endpoints operating on messages. To define network services, WSDL 1.1 uses, mainly, six elements, which are Types, Message, PortType, Binding, Port; and Service. However, WSDL only describes services from a technical aspect. Figure 2 depicts, our proposed WSDL meta-model, which illustrates all WSDL elements required to describe a service and the relations between them. Table 1 depicts the proposed mapping from WSDL to GCSD.

As depicted in Table 1, the WSDL operation element, which describes the operations that are performed by the service, is transformed into the Function concept in the GCSD, which in its turn describes a course of actions to be performed by the service. Each message part element is transformed into a Parameter instance as they both describe the parameters required by the operation/Function. The input and output are also mapped to Parameter concept because they define the abstract messages formats. PortType describes a set of abstract operations and it describes the abstract message that is involved in the operation. Therefore, it is mapped to the abstract class Interface. Binding, which defines a concrete protocol for the operations defined in a portType is mapped to Protocol. The operation name and the part name are mapped to the concept Description, which attaches typed textual descriptions to the other concepts (in this case, names are Description objects of type name). Import location, port location, and definition targetNameSpace are all mapped to Artifact URI, which represents a link to a resource that can be located using a URI.

We do not define mappings for message because it provides an abstract definition of the data being transmitted by its parts; it is presented as an argument, which is mapped to a method invocation. Therefore, the mapping of the different part elements of the enclosing message implies the mapping of the message element.
Table 1: Mapping WSDL elements and OWL-S classes into the GCSD concepts.

<table>
<thead>
<tr>
<th>WSDL elements</th>
<th>Description</th>
<th>GCSD concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition</td>
<td>It is the root element of all WSDL documents. It defines the name of the web service.</td>
<td>-</td>
</tr>
<tr>
<td>service</td>
<td>A service groups a set of related ports together</td>
<td>ServiceModule</td>
</tr>
<tr>
<td>part</td>
<td>It defines the parameters of the web service function.</td>
<td>FunctionalModule</td>
</tr>
<tr>
<td>operation</td>
<td>It is a transmission primitive that an endpoint can support.</td>
<td>Function</td>
</tr>
<tr>
<td>input</td>
<td>It specifies the abstract message format for an operation.</td>
<td>Parameter</td>
</tr>
<tr>
<td>output</td>
<td>It specifies the abstract message format for an operation.</td>
<td>Parameter</td>
</tr>
<tr>
<td>fault</td>
<td>It specifies the abstract message format for any error messages that may be output as the result of the operation.</td>
<td>Fault</td>
</tr>
<tr>
<td>portType</td>
<td>It is a set of abstract operations.</td>
<td>TechnicalModule</td>
</tr>
<tr>
<td>binding</td>
<td>It specifies the concrete protocol and data format specifications for the operations and messages defined by a particular portType.</td>
<td>Protocol</td>
</tr>
<tr>
<td>soap:binding: transport</td>
<td>It indicates the SOAP transport that the binding corresponds to.</td>
<td>AccessProfile</td>
</tr>
<tr>
<td>operation:name</td>
<td>It is an operation name, which is not required to be unique.</td>
<td>FoundationModule</td>
</tr>
<tr>
<td>types</td>
<td>It is a data type definitions used to describe the messages exchanged.</td>
<td>TypeReference</td>
</tr>
<tr>
<td>import: location</td>
<td>It associates a namespace with a document location.</td>
<td>Artifact: URI</td>
</tr>
<tr>
<td>port: location</td>
<td>It is an address for a binding and it defines a single communication endpoint.</td>
<td>Artifact: URI</td>
</tr>
</tbody>
</table>

Figure 2: Our proposed WSDL meta-model.

3.2 Mapping OWL-S to GCSD

OWL-S (Martin et al., 2004) is an ontology for web service description, which is based on three sub-ontologies: (i) serviceProfile, (ii) serviceModel; and, (iii) Grounding. However, it lacks the business aspect of the service. Our proposed OWL-S meta-model is depicted in Figure 3. Whereas, the mappings of OWL-S classes to GCSD concepts were detailed in (Bouzerzour et al., 2020a).
We also define a meta-model for the GCSD. Given that the GCSD is based on Unified Service Description Language (USDL) (Barros and Oberle, 2012), it has nine concepts to represent USDL’s modules. Furthermore, each concept regroups sub-concepts. Therefore, we do not include it the GCSD meta-model in this paper for space reasons.

4 IMPLEMENTING TRANSFORMATION RULES

We rely on Atlas Transformation Language (ATL) (Jouault et al., 2008) to implement the transformation rules. ATL is a domain-specific language for unidirectional Model-to-Model (M2M) transformations, which provides declarative and imperative constructs. Moreover, ATL is built upon Object Constraint Language (OCL) (Cabot and Gogolla, 2012), which is an OMG standard and a typed declarative language.

Therefore, in this section we present our implementation using ATL and we define OCL invariants for WSDL and OWL-S. However, we do not define any constraints for the GCSD, as it is based on USDL, which does not specify any constraint to allow openness and genericity. Otherwise, specifying constraints for specific types will result in enlarging USDL core model with the complex and overlapping constraints (Barros and Oberle, 2012). To implement our transformations we use ATL toolkit (Version 4.2.0) on top of Eclipse IDE (Version: 2020-12 (4.16.0)). We also use ATLauncher (Guana, 2015) to programmatically launch our transformations. ATLauncher is standalone Java class that runs ATL M2M transformations outside the Eclipse to promote the integration of MDE tools with other software engineering solutions. Therefore, using ATLauncher, our transformations will be integrated into a JAVA application, which will be offered as a MaaS service to allow cloud service interoperability.

4.1 Transformation Rules from WSDL to GCSD

Figure 4 depicts the implemented transformation rules from WSDL to GCSD and Figure 5 presents the OCL constraints that we implemented for our transformation.
As it is shown in Figure 4, rule definition2GCDSservice transforms the WSDL definition, which is the root element of the service description to GCDSservice concept with the definition name represented by the GCDSservice name and the targetNamespaces is represented by the Artifact concept’ attribute URI. The constraint states that a definition may have an optional but unique name. Next, rule import2Artifact transforms the import element to the Artifact concept, which provides a URI property to locate a resource. The rule part2Parameter depicts the transformation of the WSDL part class and its type to Parameter concept and TypeReference concept, which represents the parameter type using unitSymbol property. The part name is transformed to the Description concept’s value attribute. Then, the rule Operation2Function transforms WSDL operation to Function concept. The operation’s input, output and fault properties are described by the GCSD relations inputs, outputs, and faults, respectively; and the operation name is described by the value property of the Description concept.

For the concrete description of WSDL services, the rule service2Service transforms of the service class and the service name to the GCSD Service concept and the property serviceName respectively. The rule soapbinding2Protocol transforms the binding transport to the Protocol concept identifier. Eventually, the PortType2Interface rule transforms WSDL portType and port elements to Interface concept, the port binding is transformed to Interface’s implementation-
Figure 6: WSDL description of StockQuoteService service.

```xml
<definitions name="StockQuote">
  <targetNamespace>http://example.com/stockquote.wsd1</targetNamespace>
  <types>
      <element name="TradePriceRequest" type="string"/>
      <complexType name="TradePriceRequest">
        <element name="price" type="float"/>
      </complexType>
    </schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="inputBody" element="xsd:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="outputBody" element="xsd:float"/>
  </message>
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>
  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <input>
        <soap:Envelope>
          <soap:Body>
            <GetLastTradePriceInput/>
          </soap:Body>
        </soap:Envelope>
      </input>
      <output>
        <soap:Body>
          <GetLastTradePriceOutput/>
        </soap:Body>
      </output>
    </operation>
  </binding>
  <service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>
</definitions>
```

Figure 7: StockQuoteService service description in GCSD.

```
<gcsl:service name="StockQuoteService">
  <gcsl:port name="StockQuotePort" binding="StockQuoteSoapBinding">
    <gcsl:interface declaration="GetLastTradePriceInput" type="tns:TradePriceRequest"/>
    <gcsl:interface declaration="GetLastTradePriceOutput" type="float"/>
  </gcsl:port>
</gcsl:service>
```

Specification and the portType name is transformed to the implementationTypeId property, which references the service interface.

To test our transformation, we apply our trans-
4.2 Transformation Rules from OWL-S to GCSD

Figure 8 presents the OCL constraints that we implemented for our transformation, and Figure 9 depicts the implemented transformation rules from OWLS to GCSD.

The rule service2GCSDService transforms OWL-S service and Profile classes into Service and ContactProfile concepts to describe the service’s name and general information. The rule Process2Function transforms OWL-S Process (of type AtomicProcess) and Participant (of type TheClient) into Interaction, Function and Consumer concepts, respectively. The hasParticipant property is transformed into involvedRoles to describe the participant interacting with the process and it also specifies the roleDescription for the Consumer concept. Whereas, the hasInputs, hasOutput, ProcessPrecondition, and ProcessResult properties are transformed into input, outputs, preconditions, and postconditions, respectively. The Parameter2Parameter rule transforms OWL-S Parameter and its type (described by ParameterType attribute) into GCSD Parameter and TypeReference (which describes the type using unitSymbol attribute) concepts.

The precondition2precondition rule transforms OWL-S Condition and Expression into Condition concept, the OWL-S condition’s expression (described by expr property) is transformed into the conditionExpression property. The OWL-S condition’s expression body (expressionBody attribute) and expression language (expressionLanguage attribute) are transformed into GCSD Expression value and languageID, respectively.

The rule result2postcondition transforms the OWL-S Result into GCSD Condition and the resultID attribute into Description’s value attribute. The rule grounding2Interface transforms OWL-S grounding into Interface concept and the grounding wsdlDocument attribute is transformed into implementationTypeId attribute. The rule theServer2provider transforms OWL-S participant of type theServer into GCSD Provider. Eventually, the rule CompositeProcess2Phase transforms OWL-S Process of type CompositeProcess into GCSD Phase and the Process name is transformed into Description’s value attribute.

We apply our transformation rules on OWL-S ”Con-goBuyService” service instance, as shown in Figure 10. The resulting description in GCSD is depicted in Figure 11. The GCSDService describes the OWL-S service and its name. The capabilities describe the Process achieved by the OWL-S service. ContactProfile provides the contact information (described by Description). The Interaction describes the participants involved in the atomicProcess and the Process is described by the Function and its inputs, outputs, preconditions, and postconditions. Consumer describes the participants and their types (client or provider). Two Parameters are described with their types and names (described by TypeReference and Description) and one Condition is described alongside its Expression body and language. The Interface describes the WSDL document for the grounding using the implementationTypeId. OWL-S CompositeProcess is composed of subprocesses (atomic and composite) and it is constructed using control constructs and references to processes called PERFORMs. Therefore, the transformation of OWL-S CompositeProcess requires defining UML activity for describing the control constructs, UML sequence diagram for describing the compositions, and OCL to specify the pre- / post-conditions required for the composition (Bouzerzour et al., 2020a). Thus, this transformation is out of the scope of this paper, and it will be included in a future work.
Figure 9: OWL-S transformation rules.

```xml
<owl version="1.0" encoding="UTF-8">
  <service rdf:ID="#CongoBuyService">
    <profile rdf:resource="#CongoBuyProfile"/>
    <process rdf:resource="#CongoBuyProcess"/>
    <interface rdf:resource="#CongoBuyInterface"/>
  </service>
</owl>
```

Figure 10: CongoBuy Bookselling service description in OWL-S.
4.3 Results Discussion

The whole process of implementing and testing our proposed transformations proves the correctness of our research hypothesis. Indeed, the heterogeneity of service descriptions hinders the interoperability. Offering services described using different, proprietary specifications promotes the vendor lock-in issue (Opara-Martins et al., 2014). Furthermore, the preliminary results that we obtained from transforming WSDL to GCSD and OWL-S to GCSD are an evidence that the unification of service descriptions enables their interoperability, combination, customization, and composition (Nguyen et al., 2012).

5 STATIC VALIDATION OF ATL TRANSFORMATIONS

Regardless of EMF being the de-facto standard modeling framework in the industry, it is still faced with a lack of an official validation and verification tool for EMF models with OCL constraints (González et al., 2012). Only two tools were found in the literature: (i) EMFtoCSP (González et al., 2012), which translates the model and its constraints into a constraint satisfaction problem, which is then solved by a constraint solver. EMFtoCSP checks the strong satisfiability, the weak satisfiability, the lack of constraint subsumptions and the lack of constraint redundancies in a model. However, the tool, which was developed as a research project, was not updated for several years. Therefore, the tool stopped from running at times, or it generated ambiguous results at other times; and (ii) EFinder (Cuadrado and Gogolla, 2020), which is a model finding tool that automatically searches for models satisfying a set of the model’s OCL constraints. The approach enables EMF meta-model verification, verification of model transformations, and model synthesis. However, we did not get any results from it. EFinder was originally running over AnATLyzer (Cuadrado et al., 2016), which is a transformation validation tool for the static analysis of ATL model transformation. AnATLyzer is integrated with ATL environment, and it is available as an Eclipse plug-in, which enables the detection of typing and rule errors, as well as a support for pre-conditions and post-conditions. Therefore, we used AnATLyzer to correct errors (such as unresolved bindings, uninitialized features, rule conflicts, and others) in our transformations.
6 RELATED WORK

MDE-based approaches were proposed as solutions to achieve the cloud service interoperability. In (Alipour and Liu, 2018) the authors applied an MDE technique to manage auto-scaling services. Their approach enables the migration, deployment and configuration of services on multiple clouds. Ferry et al. proposed CloudMF (cloud modeling framework) (Ferry et al., 2013a), and they extended it in (Ferry et al., 2014) with: (i) a provider-agnostic management solution for applications that are deployed on IaaS and PaaS; (ii) an eclipse-based editor for the textual syntax alongside a web-based editor for the graphical syntax; and (iii) remote access to models@run-time reasoning engines. Furthermore, the authors proposed a domain-specific modeling language (DSML) called CloudML (Ferry et al., 2013b). Uni4Cloud was proposed in (Sampaio and Mendonca, 2011) to automatically configure and deploy applications across multiple clouds in an IaaS provider-agnostic manner. Whereas, MODAClouds (Ardagna et al., 2012) was proposed for service migration between clouds, and it provides a cloud-agnostic software design, a decision system to determine the most suitable cloud to deploy a given component and a support for migrating legacy software to the cloud. The approach in (Alipour and Liu, 2018) is different from our approach because the authors applied the transformation to transform a CPIM (Cloud Platform Independent Model) to a CPSM (Cloud Specific Platform Model) using EMF model generation abilities. In (Ferry et al., 2013a) (Sampaio and Mendonca, 2011), the authors adopted MDE techniques to enable the management of deployment, configuration and provisioning of PaaS and IaaS resources in multicloud. Whereas, in (Ardagna et al., 2012), the authors used MDE to enable the design and execution of applications on multicloud. Therefore, none of the aforementioned approaches applied MDE M2M technique to transform an input model to an output model based on a set of transformation rules and constraints.

7 CONCLUSION

This paper presents CIPiMo, which is a MaaS based on MDE to enable interoperability between cloud services described using heterogeneous CSD languages. The model enables the interoperability between services of different cloud models and for different target actors. The preliminary results that we obtained from our use case scenarios are promising, and they reveal that our model is capable of standardizing the descriptions of heterogeneous cloud services. Therefore, it enables their interoperability. As for future work, we will extend our model by defining the transformations for other CSD languages. Eventually, using a cloud simulation tool (for example: CloudSim) we will test the performance criteria of our MaaS model.

REFERENCES


