A comparison of two-level and multi-level modelling for cloud-based applications

Alessandro Rossini[1], Juan de Lara[2], Esther Guerra[2], and Nikolay Nikolov[1]

SINTEF, Oslo, Norway {firstname.lastname}@sintef.no
Universidad Autónoma de Madrid, Spain {Juan.deLara,Esther.Guerra}@uam.es

Abstract The Cloud Modelling Framework (CloudMF) is an approach to apply model-driven engineering principles to the specification and execution of cloud-based applications. It comprises a domain-specific language to model the deployment topology of multi-cloud applications, along with a *models@run-time* environment to facilitate reasoning and adaptation of these applications at run-time. This paper reports on some challenges encountered during the design of CloudMF, related to the adoption of the two-level modelling approach and especially the *type-instance* pattern. Moreover, it proposes the adoption of an alternative, multi-level modelling approach to tackle these challenges, and provides a set of criteria to compare both approaches.

Keywords: domain-specific languages, metamodelling, multi-level modelling, multi-level reasoning, cloud computing, CloudMF, CloudML, MetaDepth

1 Introduction

Model-driven engineering (MDE) aims at improving the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code to model-centric, whereby models and modelling languages are the main artefacts of the development process. In MDE, the abstract syntax of a modelling language is defined by its metamodel, which describes the set of concepts, properties and relations of a domain, as well as the rules for combining them. Based on this paradigm, a software system is represented by a model that conforms to a metamodel. This approach, hereafter called *two-level modelling*, may have limitations [15,5,21] when the metamodel includes the *type-instance* pattern [5,10], which requires an explicit modelling of types and their instances at the same metalevel. In this case, an alternative approach that employs more than two levels, hereafter called *multi-level modelling*, yields simpler models [5,21,10]. However, while some recent studies show the potential applicability of multi-level modelling [10], there are still scarce works showing its benefits for real-life projects.

Cloud computing provides a ubiquitous networked access to a shared and virtualised pool of computing capabilities (*e.g.*, network, storage, processing, and memory) that can be provisioned with minimal management effort. MDE has been applied in the field of cloud computing, where models and modelling languages enable developers and reasoning engines to work at a high level of abstraction and focus on cloud concerns rather than implementation details. One notable example in this area is the Cloud Modelling Framework (CloudMF) [12,13,14], which consists of: *(i)* the Cloud Modelling Language (CloudML), a domain-specific language (DSL) to model the deployment of multi-cloud applications (*i.e.*, applications that can be deployed across multiple private, public, or hybrid cloud infrastructures and platforms); and *(ii)* a *models@run-time* environment to enact the deployment and adaptation of these applications. The run-time environment provides a model-based representation of the underlying running system, which facilitates reasoning and adaptation of multi-cloud applications.

This paper reports on some challenges encountered during the design of CloudMF, related to the adoption of the two-level approach and especially the type-instance pattern. Moreover, it proposes an alternative, multi-level approach, and provides a detailed comparison of both approaches along six criteria, which aims to serve as a guideline for prospective adopters of the multi-level solution.

Paper organisation. Sec. 2 outlines the current design of CloudML and its models@run-time environment. Sec. 3 presents a case study used throughout the paper. Secs. 4 and 5 compare how to model the case study using two-level and multi-level approaches. Sec. 6 discusses the pro and contra of the two approaches. Finally, Sec. 7 compares with related work and Sec. 8 ends with conclusions and future work.

2 CloudMF

CloudMF is being developed in the context of the EU projects MODAClouds and PaaSage[1], where several industrial partners are adopting it to specify and execute the multi-cloud applications of their use cases. In this section, we outline its two main ingredients: CloudML and its models@run-time environment.

2.1 CloudML

CloudML has been designed based on the following requirements, among others:

Separation of concerns (R_1): CloudML should support a modular, loosely-coupled specification of the deployment. This will facilitate the maintenance as well as the dynamic adaptation of the deployment model.

Reusability (R_2): CloudML should support the specification of types that can be seamlessly reused to model the deployment. This will ease the evolution as well as the rapid development of different variants of the deployment model.

Abstraction (R_3): CloudML should provide an up-to-date, abstract representation of the running system. This will facilitate the reasoning, simulation, and validation of the adaptation actions before their actual enactments.

CloudML implements a component-based approach [14], which facilitates separation of concerns (R_1) and reusability (R_2). Hence, deployment models can be regarded as assemblies of components and relations between them.

[1] `http://www.modaclouds.eu/`, `http://www.paasage.eu/`

2.2 Models@run-time

Models@run-time [6] is an architectural pattern for dynamically adaptive systems that leverages upon models at both design-time and run-time. In particular, models@run-time provides an abstract representation of the underlying running system, whereby a modification to the model is enacted on-demand in the system, and a change in the system is automatically reflected in its model.

In CloudMF, the models@run-time environment provides a model causally connected to the running cloud-based application (addressing requirement R_3). On the one hand, any modification to a CloudML model is enacted on-demand in the running application. On the other hand, any change in the running application is automatically reflected in its CloudML model.

Fig. 1 depicts the architecture of CloudMF. A reasoning engine reads the current model (step 1) and produces a target model (step 2). Then, the run-time environment computes the difference between the current model and the target one (step 3). Finally, the adaptation engine enacts the adaptation by modifying only the parts of the cloud-based application necessary to account for the difference and the target model becomes the current model (step 4).

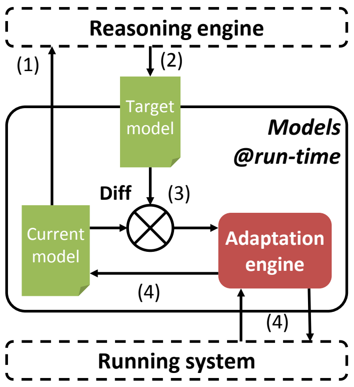


Figure 1. CloudMF architecture

3 Case study

The adoption of CloudML as the DSL for specifying models in the models@run-time environment of CloudMF introduces some challenges in its design and implementation. In this section, we present these challenges through a case study.

SensApp[2] is an open-source, service-oriented application for storing and exploiting large data sets collected from sensors and devices. Suppose that, at design-time, we would like to model the deployment of SensApp on a single cloud, whereby a SensApp cluster should be hosted on a Tomcat container cluster, which in turn should be hosted on a Ubuntu virtual machine cluster, which in turn should be provisioned on a private OpenStack cloud in Norway. Moreover, the SensApp cluster should be load balanced by an HAProxy load balancer and should have from one to four instances.

Fig. 2(a) shows the deployment model specified using a graphical syntax for CloudML. The left part depicts the available reusable types, while the right part depicts instances of these. The range in instances range represents that a minimum of one and a maximum of four instances of SensApp can be executed at run-time. We assume the range attached to sensApp1 also applies to its (indirect) hosts, *i.e.*, tomcat1 and ubuntu1.

[2] `http://sensapp.org`

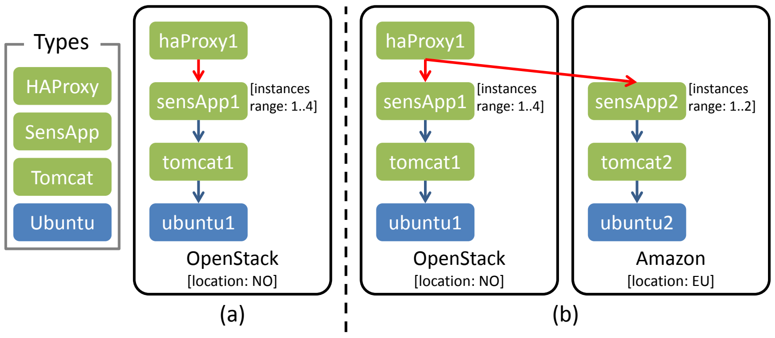


Figure 2. Deployment model at design-time: (a) with single cloud, (b) with multiple clouds

We could have considered the deployment of SensApp on multiple clouds, whereby a second SensApp cluster is deployed and provisioned on a public Amazon cloud in Europe. Fig. 2(b) shows the deployment model with multiple clouds. In the remainder of the paper, we only consider the single-cloud scenario to keep the models simple and retain only the details that are relevant for the discussion.

Then, suppose that, at run-time, we would like to dynamically adapt the deployment of the application in order to meet service-level objectives (*e.g.*, response time < 50ms) and goals (*e.g.*, minimise cost). A reasoning engine would first read the number of current instances, and then enact adaptations based on these service-level objectives and goals. Therefore, the deployment model above is insufficient, and an additional property is needed to represent the number of current instances. Fig. 3(a) shows the deployment model before and after the adaptation.

Finally, suppose that, also at run-time, we would like to dynamically adapt the deployment of the application in order to prevent impending failures and recover from occurred ones. A reasoning engine would first read, *e.g.*, the CPU load of each individual Ubuntu virtual machine, and then enact adaptations based on scalability rules.

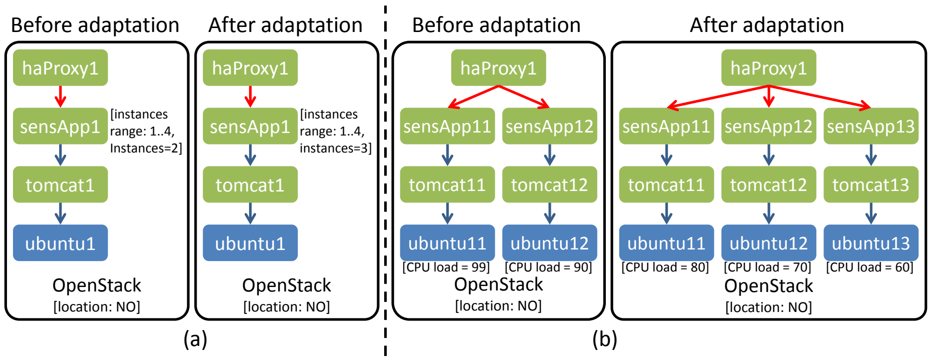
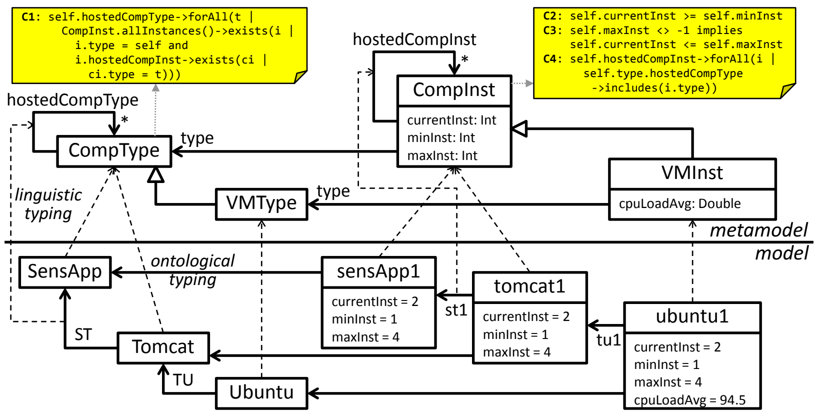


Figure 3. Deployment model at run-time: (a) with number of current instances, (b) with explicit instances and CPU loads



```
C1: self.hostedCompType->forAll(t |
      CompInst.allInstances()->exists(i |
        i.type = self and
        i.hostedCompInst->exists(ci |
          ci.type = t)))
```

```
C2: self.currentInst >= self.minInst
C3: self.maxInst <> -1 implies
    self.currentInst <= self.maxInst
C4: self.hostedCompInst->forAll(i |
      self.type.hostedCompType
       ->includes(i.type))
```

Figure 4. Deployment model in abstract syntax at run-time, with number of current instances and average CPU load

Therefore, the deployment in Fig. 3(a) is too high-level (or coarse-grained), and a low-level (or fine-grained) deployment model is needed to represent each individual instance of SensApp along with the underlying Tomcat container and Ubuntu virtual machine. Fig. 3(b) shows the deployment with explicit instances before and after the adaptation.

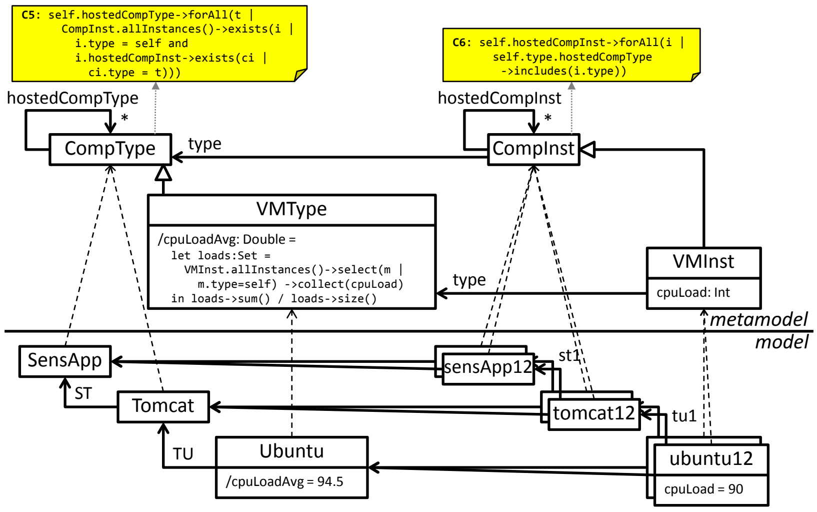
The current version of CloudML is based on two-level modelling. It supports the specification of single components within a deployment model, but it does not support the specification of clusters of components with ranges.

In the following section, we present a proof of concept of how the current version of CloudML could be extended to support the case study.

4 Two-level approach

CloudML implements the type-instance pattern [4]. To declare and instantiate types (*e.g.*, the type Ubuntu and its instance ubuntu1 in Fig. 2(a)), this pattern requires both types and instances to be represented by classes in the metamodel. This pattern also exploits two flavours of typing: *ontological* and *linguistic* [17]. The latter is the relation between a model and its metamodel, while the former is the relation between elements within a model.

Fig. 4 shows a simplified version of the CloudML metamodel along with the model in abstract syntax corresponding to the one in graphical syntax in Fig. 3(a), where several concepts such as the life-cycle scripts attached to the components, the ports provided and required by components, the communications between ports, and the cloud providers are omitted for brevity (see [14] for a detailed description of the CloudML metamodel). SensApp represents a reusable type of SensApp. It is linguistically typed by the class CompType (short for component type). sensApp1 represents an instance of SensApp. It is ontologically typed by SensApp and linguistically typed by CompInst (short for component instance). Similarly, Fig. 5 shows the model in abstract syntax corresponding to the one in graphical syntax in Fig. 3(b), before adaptation.



```
C5: self.hostedCompType->forAll(t |
      CompInst.allInstances()->exists(i |
        i.type = self and
        i.hostedCompInst->exists(ci |
          ci.type = t)))
```

```
C6: self.hostedCompInst->forAll(i |
      self.type.hostedCompType
       ->includes(i.type))
```

```
/cpuLoadAvg: Double =
  let loads:Set =
    VMInst.allInstances()->select(m |
      m.type=self) ->collect(cpuLoad)
  in loads->sum() / loads->size()
```

Figure 5. Deployment model at run-time, with two explicit instances and CPU loads

Fig.s 4 and 5 depict one possible approach for allowing CloudML to support the specification of clusters of components with ranges. However, it implies the use of two syntactically and semantically disjoint models: one representing the aggregated view of each cluster (Fig. 4) and one representing each individual instance in each cluster (Fig. 5). In order to avoid this, a naive solution could be to merge the two models by applying the type-instance pattern twice, which would lead to a new *type-template-instance* pattern. Fig. 6 shows this merged model. Unfortunately, this solution is both ineffective and insufficient.

First, it is ineffective since applying the type-instance pattern twice leads to six classes to represent components and their instances, and even more references between

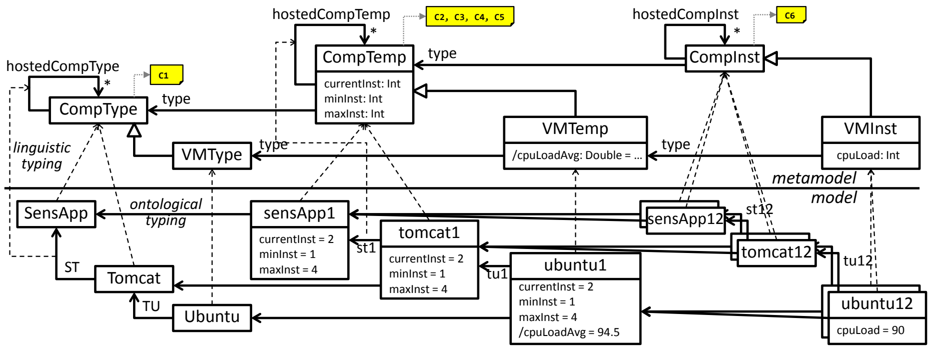


Figure 6. Proof-of-concept deployment model at run-time, with both number of current instances as well as explicit instances and CPU loads

them in both the metamodel and the model (*e.g.*, CompType, CompTemp, and CompInst in the metamodel and their instances in the model). Please note that the metamodel in Fig. 6 only contains the classes and references necessary to represent components and virtual machines, while the model only contains the elements needed to represent a SensApp application cluster, a Tomcat container cluster, and a Ubuntu virtual machine cluster. The figure omits the classes and references needed to represent the life-cycle scripts attached to the components, the ports provided and required by components, the communication between ports, and the cloud providers. Applying the type-instance pattern twice would lead to an explosion of elements for each of these concepts in both the metamodel and the model. Moreover, this solution is ineffective since checking the type-instance conformance within the model requires complex OCL constraints in the metamodel, while applying the type-instance pattern twice requires replicating these constraints (*e.g.*, C1/C5 and C4/C6).

Second, this solution is insufficient, as we are not modelling the allowed number of component instances within a host, but the restriction on instances is checked globally. Please note that this could be naturally expressed if we were able to put cardinalities on the references st1 and tu1.

Altogether, we need to apply the type-instance pattern twice and add complex OCL constraints to the metamodel in order to emulate three ontological levels within a single linguistic level. This makes the two-level approach convoluted and less usable [5].

As an alternative design strategy, we could merge the three classes CompType, CompTemp, and CompInst into one class Component (and similar for VMType, VMTemp, and VMInst). The resulting class Component would have a reference type to itself with optional cardinality as well as a property level to distinguish whether an instance of Component belongs to the type, template, or instance level. In addition, we could add OCL constraints to ensure the correctness of the ontological typing. While this solution would make the metamodel more compact, it would lead to a higher complexity of OCL constraints. Moreover, it would also lead to the misuse of model elements, as the properties currentInst, minInst, and maxInst would be present in instances of Component, independently of their level, while they are only necessary at the template level.

In the following section, we present a proof of concept of how CloudML could be defined and used adopting a multi-level approach.

5 Multi-level approach

Multi-level modelling extends traditional two-level modelling by enabling the use of an arbitrary number of levels (rather than just two) in a modelling stack. In scenarios where the type-instance pattern or one of its variants arise [10], this solution yields simpler models, since the additional classes to specify instances become unnecessary.

Fig. 7 shows CloudML organised in four levels. The top level contains an excerpt of the refactored CloudML metamodel, while the subsequent levels contain the definition of types (*e.g.*, Tomcat, Ubuntu), a high-level deployment model, and a low-level deployment model with explicit instances and CPU loads at run-time. In this solution, it is not necessary to have classes CompType, CompTemp, and CompInst at the top level, but a single class Component is sufficient (and similar for VirtualMachine).

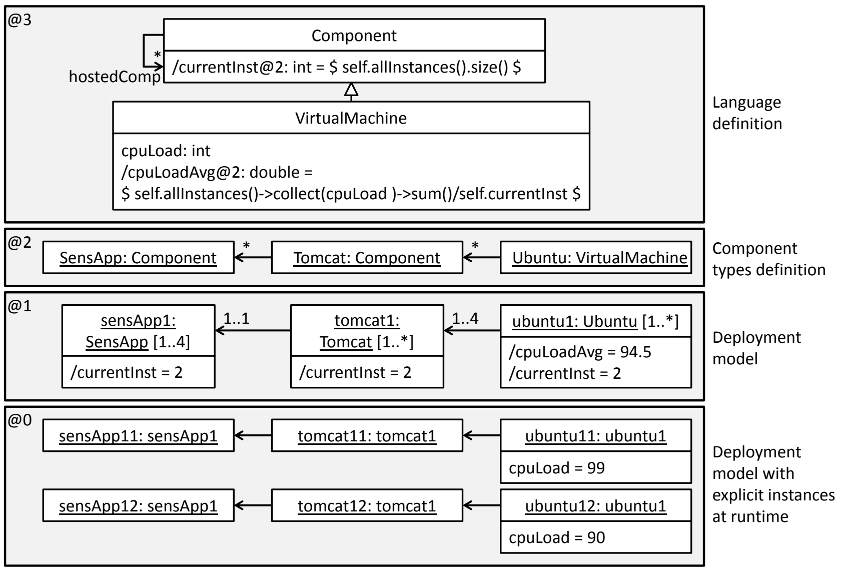


Figure 7. Simplified multi-level model for CloudML

In this approach, elements are called *clabjects* (by the contraction of *cla*ss+o*bject*), as they have both a type and an instance facet. For example, Ubuntu is an instance of VirtualMachine and the type of ubuntu1. Furthermore, clabjects can specify the features of their instances beyond the next level. For example, VirtualMachine specifies that three levels below, its indirect instances have a cpuLoad, while two levels below, their instances have a cpuLoadAvg. The mechanism used for this deep characterisation of instances beyond the next level is *potency* [4]. A potency is a natural number (or zero) indicating at how many levels an element can be instantiated (*cf.* [21] for a formal discussion of different types of potency). In Fig. 7, the potency is denoted by the @ symbol. At every lower level, the potency decreases, and when it reaches zero, the element cannot be instantiated further. If an element does not declare a potency, it inherits it from its container, and eventually from the enclosing model. Hence, potency is a generalisation of the standard instantiation mechanism in the two-level approach, where types in the metamodel have potency 1, and instances in the model have potency 0.

For the case study, we use multi-level modelling as realised in our MetaDepth tool [9]. The tool offers textual modelling and integrates the Epsilon languages[3] for model manipulation. MetaDepth also supports derived properties, whose calculation expression can be specified using the Epsilon Object Language (EOL, a variant of OCL). The top model in Fig. 7 contains two derived properties: currentInst on Component, and cpuLoadAvg on VirtualMachine. Both are calculated at level 1: the first counts the number of instances of each deployed component at level 1, and the second computes the average of the cpuLoad of all the instances at the bottom level. We adapted EOL for its use in a multi-level setting [11]. For example, we allow *indirect type referencing*.

[3] http://eclipse.org/epsilon/

Because the type names at intermediate levels are unknown when specifying the top level, we can refer to instances of instances of a type by using the type name. Hence, at level 0, an expression like Component.allInstances() returns the set {sensApp11, sensApp12, tomcat11, tomcat12, ubuntu11, ubuntu12}. Since clabjects with potency 1 or more retain a type facet, it is possible to apply the operation allInstances on them.

Similar to clabjects, properties and references also have type and instance facets. Thus, references at intermediate levels can specify cardinalities, as is the case at levels 1 and 2 in the figure. In addition, MetaDepth supports clabject cardinalities. We have used this feature at level 1 to specify the allowed scaling for ubuntu1, tomcat1 and sensApp1.

Finally, multi-level modelling permits linguistic extensions, *i.e.*, elements which are typed linguistically but not ontologically [4]. The typing relation between the elements of consecutive levels in Fig. 7 is ontological, while all the elements are typed by a linguistic metamodel (not shown, but it contains classes like Clabject and Property). In the case study, this feature would allow adding new clabjects and properties with no ontological typing at levels 1 and 2.

6 Comparison

In this section, we discuss the advantages and disadvantages of each approach. We base our comparison on a set of criteria that demonstrate the expressiveness and usability of the examined modelling solutions.

Size of language definition. A prominent aspect that affects the choice of a modelling approach is the size of the language definition. We claim that a large and verbose language definition is generally undesirable because it is harder to comprehend. This negative effect also projects into the resulting models, since more elements need to be used in order to represent an intended expression. As evidenced by the models in Sec.s 4 and 5, the language definition is three times larger in the two-level approach. For example, the clabject Component in the multi-level approach needs to be unfolded into CompType, CompTemp, and CompInst in the two-level approach. The situation is similar with references (*e.g.*, hostedComp gets tripled). This is so because the two-level approach requires emulating the ontological typing by adding the references type between classes at the type, template, and instance levels, as well as the corresponding OCL constraints. Instead, the ontological typing in the multi-level approach is native.

Complexity of OCL constraints. Both multi-level and two-level modelling take advantage of OCL constraints to ensure the well-formedness of the produced models. However, there are significant differences related to the complexity and count of the constraints needed to achieve the targeted outcomes. Constraints in the two-level approach tend to be more complex because they can only use the linguistic types (CompType, CompTemp, CompInst), but not the reusable types at the model level (SensApp, Tomcat, Ubuntu). Nevertheless, there are a number of other factors contribute to this additional complexity:

- *Type-conformance constraints*. The two-level approach requires defining OCL constraints to check, *e.g.*, type conformance between the instances of CompTemp and the instances of CompType, and between the instances of CompInst and CompTemp. In the case study, these constraints check the correctness of the references hostedCompType and hostedCompInst (see constraints C1 and C4 in Fig. 4, and constraints C5 and C6 in Fig. 5). The multi-level approach does not need to include such constraints, because the type conformance check is embedded into the paradigm.
- *Access to lower levels*. In the two-level approach, to obtain the instances of a class representing a type (*e.g.*, the instances of SensApp), we need to navigate the reference type (*e.g.*, CompInst.allInstances()->select(ci | ci.type = SensApp)). This access is simpler in the multi-level approach, as it is possible to obtain the direct or indirect instances of a clabject using the operation allInstances on the clabject (*e.g.*, sensApp.allInstances()). In the case study, cpuLoadAvg needs to access all instances at the bottom level to obtain the cpuLoad, and currentInst counts the number of instances at the bottom level.
- *Transparent navigation to upper levels*. In the multi-level approach, the expression o.feature looks up the value of feature in the direct type of o, or in some indirect type in upper levels. In the two-level approach, this needs to be done explicitly by using o.type.feature or o.type.type.feature.
- *Constraints on reusable types*. Suppose we need to specify a constraint on some reusable type (*e.g.*Ubuntu). In the multi-level approach, the constraint would be directly specified on the context of the clabject. For example, defining self.cpuLoad < 80 with potency 2 on the clabject Ubuntu ensures that all its instances at level 0 have a cpuLoad lower than 80. In contrast, in the two-level approach, we should add the following constraint to the class VMInst: self.type.type.name = 'Ubuntu' implies self.cpuLoad < 80. In addition, we should add a property name to VMType to be able to identify the instance of Ubuntu. This is necessary because Ubuntu lacks a type facet in the two-level approach, and hence, it cannot specify constraints. Moreover, the constraint should be added to the metamodel, which may not be allowed as metamodel changes are frequently restricted to language developers.
- *Instantiability of classes*. To restrict the number of instances of a certain class (*e.g.*, CompInst), the two-level approach requires adding properties to the class to specify the minimum and maximum number of allowed instances, as well as OCL constraints checking their fulfilment (*e.g.*, properties minInst and maxInst, and constraints C2 and C3, in Fig. 4). Instead, the three-level approach does not require to specify any OCL constraint, but only the clabject cardinality.

Precision. We define precision as a measure that reflects how accurately, in semantic terms, a model can represent an intended expression. Thus, we compare how reference cardinalities are specified. The multi-level approach supports reference cardinalities at level 1, to gain a fine-grained control of the allowed scaling at level 0. The presented two-level approach uses ranges for this purpose. However, it does not constrain *e.g.*, how many instances of SensApp are allowed in a Tomcat, but only that a global maximum of four are allowed, either residing in a single Tomcat or in several ones. To enable this feature, the two-level approach would require to emulate reference cardinalities by adding extra classes to the metamodel (the *Relation Configurator* pattern in [10]).

Extensibility. We define extensibility as the ability to extend a language while minimising changes to its metamodel. This is because languages, like any other software artefact, need to evolve in response to changing requirements. In this respect, the multi-level approach allows adding new properties at levels 1 and 2, due to its support for linguistic extensions (*i.e.*, elements without ontological type). Thus, we could add a property maxSensors to SensApp to configure the maximum number of sensors in SensApp applications. Defining this property on Component at level 3 is less appropriate, as some component instances at level 2 may lack the property. In the two-level approach, being able to specify new properties in models would require emulating the infrastructure for property specification/instantiation at the metamodel level.

Flexibility. We define flexibility as the degree of expressiveness of the chosen abstraction in terms of the model element relationships and level of encapsulation. In the multi-level approach, clabjects at level 1 or above can specify operations. Thus, the engineer designing the deployment model could add the following EOL operation to Ubuntu, with a condition for scaling out:

```
operation Ubuntu scaleOut(): Boolean { return self.cpuLoadAvg >= 90 }
```

A reasoner could use this operation as a trigger for scaling out. This flexibility is not possible in the two-level approach, because elements at model level cannot specify operations. Instead, a workaround would be to design a dedicated DSL to express such conditions.

Also concerning flexibility, the multi-level approach supports inheritance at any level. For example, in Fig. 7, the model at level 2 could specify a hierarchy of application servers. In the two-level approach, the semantics of inheritance at the model level would need to be emulated. Nonetheless, the two-level approach allows customising the semantics of the conformance relation (*e.g.*, to permit two ontological types for an element) and the inheritance relation (*e.g.*, to (dis)allow multiple inheritance), while the semantics of these relationships is fixed in the multi-level approach.

Tooling. Lastly, we examine the tool support for each of the methodologies. The defacto standard in modelling frameworks in the state-of-the-art is the Eclipse Modelling Framework (EMF)[4]. Whereas EMF provides a large ecosystem of tools and languages for two-level modelling, the support for multi-level modelling is limited. MetaDepth is compatible with the Epsilon languages but does not rely on EMF. Similarly, other multi-level tools, like XModeler [7] or Nivel [1] do not rely on EMF. A notable exception is Melanee [2], which is built upon EMF. However, in this case model manipulation languages would need to be adapted to work with it. Alternatively, at the implementation level one could use programming languages enhanced with multi-level concepts, like DeepJava [18], or with strong reflection capabilities [16,8].

Table 1 summarises the studied aspects. Altogether, a multi-level approach for the case study leads to a smaller language definition, with less OCL constraints, and less complex. To achieve the same degree of precision, extensibility and flexibility, the two-level approach would need to include in the metamodel many features that are native

[4] https://www.eclipse.org/modeling/emf/

Table 1. Comparison criteria

Dimension	**Two-Level**	**Multi-Level**
Size	× 3-fold replication of classes and references	√ Compact language definition: clabjects and potencies
	× Explicit modelling of type relations	√ Type relations are native
OCL complexity	× Explicit type-conformance constraints	√ Constraints can use ontological types
	× Constraints cannot use ontological types	√ Transparent navigation across ontological levels
	× Explicit navigation through type relations	
Precision	× Lack of cardinality constraints at model level	√ Fine-grained control by cardinality constraints at level 1
Extensibility	× Dynamic properties need to be emulated	√ New properties can be added at intermediate levels
Flexibility	√ Customisable conformance/inheritance rel.s	√ Operations can be added at intermediate levels
	× Lack of inheritance at model level	√ Inheritance at the model level is native
Tooling	√ Large ecosystems of tools and languages	× Limited tool choice, increased integration effort

in a multi-level framework, like reference cardinalities, properties, operations, or inheritance. Nonetheless, as metamodelling features (like ontological typing or inheritance) are explicitly specified, they can be customised. The case study needed no customisation though. Finally, a richer set of tools is currently available for a two-level approach.

7 Related Work

In the cloud community, frameworks such as Cloudify, Puppet or Chef[5] provide DSLs that facilitate the specification and enactment of provisioning, deployment, monitoring, and adaptation of cloud-based applications, without being language-dependent. Moreover, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [20] is a specification developed by the OASIS consortium, which provides a language for specifying the components comprising the topology of cloud-based applications along with the processes for their orchestration. Similar to CloudMF, the aforementioned solutions are based on a two-level modelling approach, so an alternative, multi-level modelling approach could also be considered for these solutions.

There are scarce works comparing two-level and multi-level solutions for given problems. In [3], some comparison criteria for multi-level approaches (*e.g.*, powertypes and deep modelling) are proposed. The criteria include language size and intended audience. Instead, our criteria are directed to evaluate solutions to a modelling problem. In our previous work [10], we identified patterns that signal the need for a multi-level solution, and analysed their occurrence on a set of metamodels, including an early version of CloudML. In [5], the authors use a simple example to discuss the benefits (regarding size) of a potency-based multi-level approach compared to powertypes [15] and a two-level approach. In contrast, we use a real-life example, and discuss other dimensions beyond size. In [19], the authors detect the multi-level nature of the MARTE profile, and use an embedding of a potency based multi-level approach using stereotypes to refactor its definition. Instead, we use a native multi-level framework like MetaDepth and compare with a two-level solution.

[5] `http://www.cloudifysource.org/`, `https://puppetlabs.com/`, `http://www.opscode.com/chef/`

8 Conclusions and Future Work

In this paper, we have compared two metamodelling techniques for the purpose of providing CloudML with features that facilitate reasoning and adaptation of multi-cloud applications at multiple levels of abstraction. The results show a smaller language definition in the multi-level case, with some other benefits regarding extensibility, flexibility, and precision.

In the future, we intend to conduct a user-based empirical study on differences between the two-level and multi-level modelling approaches discussed in this paper. The goal is to collect quantitative and qualitative measurements on the two alternatives with respect to the criteria described in Sec. 6, which will allow us to verify if there is any statistically significant difference and evaluate the viability of each. Moreover, we plan to take advantages of the best features from the two-level and multi-level approaches. In this respect, we are building a compiler from MetaDepth to EMF, which, given the top model in Fig. 7, would produce the metamodel in Fig. 6, including the OCL constraints. Hence, designers would deal with a reduced language definition, and the resulting framework would be easy to integrate with the EMF tooling.

Acknowledgements. The research leading to these results has received funding from the European Commission's Seventh Framework Programme (FP7/2007-2013) under grant agreement numbers 317715 (PaaSage), 318392 (Broker@Cloud), and 611125 (MONDO), the Spanish Ministry under project Go Lite (TIN2011-24139), and the Madrid Region under project SICOMORO (S2013/ICE-3006).

References

1. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. Software and System Modeling 8(4), 521–549 (2009)
2. Atkinson, C., Gerbig, R., Kennel, B.: Symbiotic general-purpose and domain-specific languages. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE 2012: 34th International Conference on Software Engineering. pp. 1269–1272. IEEE (2012)
3. Atkinson, C., Gerbig, R., Kühne, T.: Comparing multi-level modeling approaches. In: MULTI 2014: 1st International Workshop on Multi-Level Modelling, co-located with MODELS 2014: 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. CEUR Workshop Proceedings, vol. 1286, pp. 53–61. CEUR (2014)
4. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation 12(4), 290–321 (2002)
5. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and Systems Modeling 7(3), 345–359 (2008)
6. Blair, G., Bencomo, N., France, R.: Models@run.time. IEEE Computer 42(10), 22–27 (2009)
7. Clark, T., Gonzalez-Perez, C., Henderson-Sellers, B.: A foundation for multi-level modelling. In: MULTI 2014: 1st International Workshop on Multi-Level Modelling, co-located with MODELS 2014: 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. CEUR Workshop Proceedings, vol. 1286, pp. 43–52. CEUR (2014)

8. Cointe, P.: Metaclasses are First Class: the ObjVlisp Model. In: Meyrowitz, N.K. (ed.) OOPSLA 1987: Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 156–167. ACM (1987)
9. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. In: Vitek, J. (ed.) TOOLS 2010: 48th International Conference on Objects, Models, Components, Patterns. Lecture Notes in Computer Science, vol. 6141, pp. 1–20. Springer (2010)
10. de Lara, J., Guerra, E., Cuadrado, J.S.: When and How to Use Multi-Level Modelling. ACM Trans. on Software Eng. and Methodology 24(2), 1–46 (2014)
11. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific metamodelling languages. Software and System Modeling 14(1), 429–459 (2015)
12. Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A.: Managing multi-cloud systems with CloudMF. In: Solberg, A., Babar, M.A., Dumas, M., Cuesta, C.E. (eds.) NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies. pp. 38–45. ACM (2013)
13. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: O'Conner, L. (ed.) CLOUD 2013: 6th IEEE International Conference on Cloud Computing. pp. 887–894. IEEE Computer Society (2013)
14. Ferry, N., Song, H., Rossini, A., Chauvel, F., Solberg, A.: CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In: Bilof, R. (ed.) UCC 2014: 7th IEEE/ACM International Conference on Utility and Cloud Computing. pp. 269–277. IEEE Computer Society (2014)
15. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Software and Systems Modeling 5(1), 72–90 (2006)
16. Kiczales, G., des Rivieres, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)
17. Kühne, T.: Matters of (meta-)modeling. Software and Systems Modeling 5(4), 369–385 (2006)
18. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style: multi-level programming with deepjava. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) OOPSLA 2007: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 229–244. ACM (2007)
19. Mallet, F., Lagarde, F., André, C., Gérard, S., Terrier, F.: An Automated Process for Implementing Multilevel Domain Models. In: van den Brand, M., Gasevic, D., Gray, J. (eds.) SLE 2009: 2nd International Conference on Software Language Engineering. Lecture Notes in Computer Science, vol. 5969, pp. 314–333. Springer (2009)
20. Palma, D., Spatzier, T.: Topology and Orchestration Specification for Cloud Applications (TOSCA). Tech. rep., Organization for the Advancement of Structured Information Standards (OASIS) (June 2013), http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf
21. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. Formal Aspects of Computing 26(6), 1115–1152 (2014)