A Deep Perspective on the ArchiMate Modeling Language and Standard

Colin Atkinson[*,a], Thomas Kühne[b]

[a] University of Mannheim, Germany
[b] Victoria University of Wellington, New Zealand

Abstract. *Given the scale, complexity and variety of enterprise architectures, approaches for modeling them need to be as simple and flexible as possible in order to minimize the accidental complexity within enterprise architecture models. Multi-level modeling techniques offer an effective way of achieving this but to date there has been little research into how they could contribute to enterprise architecture modeling. In this article we therefore explore how the former could be best leveraged within the latter by considering the modeling goals, architecture and principles of one of the most concrete and widely used enterprise architecture modeling standards: ArchiMate. More specifically, we discuss how the conceptual integrity of the ArchiMate standard and modeling experience could be enhanced using multi-level modeling principles. In our discussions, we focus on a specific variant of multi-level modeling, called deep modeling, which is based on the notions of orthogonal classification and deep instantiation.*

Keywords. Enterprise Architecture Modeling • Multi-Level Modeling • Deep Visualization • ArchiMate

Communicated by Ulrich Frank. Received 2017-03-01. Accepted on 2019-09-08.

1 Introduction

Enterprise Architectures play a pivotal role in enabling companies to align their processes with their IT infrastructures. With the increasing trend towards digitization and automation, companies need effective enterprise architectures to remain competitive and respond rapidly to change. Poorly understood and/or aligned processes, information systems and IT infrastructures significantly reduce a company's ability to respond agilely to change and deliver services in a cost-effective way.

In general, the notion of "Enterprise Architecture" (EA) encompasses all aspects of a company's assets, relationships, stakeholders and processes over its entire lifetime, from inception and design to operation and retirement. Approaches for *Enterprise Architecture Modeling* (EAM) therefore need to have a broad scope and be able to portray enterprise architectures in the large variety of forms expected by their many different stakeholders. In other words, they need to be *multi-view* approaches which allow an enterprise architecture and/or its parts to be described from multiple *viewpoints* using a suite of different (sub)languages. In addition, given the large number of different kinds of stakeholders and tasks that EA models in different companies need to support, it is important that EAM frameworks allow new view types and view-representation languages to be added by users (Frank 2002). Defining a single language framework to support all view languages needed out-of-the-box is not a realistic proposition.

Several EAM approaches have become de facto standards over recent years. Some, such as TOGAF (The Open Group 2010) and Zachmann (Zachman 1987), do not prescribe the specific languages to be used to portray information in different kinds of views (e. g., processes, data types

* Corresponding author.
E-mail. atkinson@informatik.uni-mannheim.de

etc.) while others, such as RM-ODP (ISO/IEC 1997) and ArchiMate (The Open Group 2017a), define their own specialized languages to represent their specific view types. However, only one of these, ArchiMate, explicitly encourages the definition of new viewpoints and language variants by end users through a dedicated "extension" mechanism.

Of the well-known EAM standards, therefore, ArchiMate may be regarded as the most advanced in terms of supporting viewpoint/language engineering. However, like most modeling environments today, the ArchiMate approach to modeling and language engineering is rooted in modeling infrastructure principles that go back to the first generation of modeling tools. In particular, its definition and use is based on the traditional four level hierarchy popularized by the UML infrastructure (Object Management Group 2007), where the bottom level is considered to be the "real world" and the top two levels are language definitions, i. e., a language for defining modeling languages (i. e., the meta-meta model) and a language for defining EA models (i. e., the ArchiMate meta model). This leaves only one level to accommodate all domain modeling content, including instances, classes and potentially domain meta classes.

Numerous authors have pointed out that multi-level modeling environments generally offer a better platform for flexible, domain-specific language engineering than two-level modeling environments (Atkinson and Kühne 2003). Frank, in particular, has specifically made this case in the context of EAM by clarifying the requirements EAM languages should support and developing a new prototype EAM modeling environment to showcase the benefits of multi-level modeling in this domain (Frank 2014).

The use of multi-level modeling holds the promise of economic benefits (Frank 2016) due to reducing accidental complexity (Atkinson and Kühne 2007), which

- minimizes the effort involved in understanding and changing models,
- promotes adaptability of models, and
- helps to protect the integrity of models.

In combination, the above aid the maintainability of models and thus lower typical maintenance costs, as mistakes that are often provoked by overly complex models and may be costly to fix, can be avoided in the first place.

In this article we specifically aim to reinforce the arguments for multi-level modeling in the domain of EAM by investigating how a multi-level framework could better support the modeling goals and principles outlined by the ArchiMate standard (The Open Group 2017a) and its designers (Lankhorst 2013; Lankhorst et al. 2010). As part of this investigation we are interested in the way the ArchiMate language is defined as well as used. We focus on ArchiMate for this study because (a) it includes one of the most comprehensive and well-defined languages and (b) it is defined as a meta-model in a publicly available standard.

The remainder of this article is organized as follows. In the next section we provide an introduction to ArchiMate and summarize the language designers' explicitly stated goals in relation to viewpoint language usage and definition. After that we provide a general introduction to multi-level modeling, followed by a description of the specific variant we use in this article known as deep modeling (Atkinson and Kühne 2003). The following two sections then analyze the pros and cons of deep modeling approaches in terms of the requirements outlined in the ArchiMate standard and by its designers. Sect. 4 does so in terms of the underlying concepts involved in language definition and use (i. e., the abstract syntax) while Sect. 5 does so in terms of the presentation (or visualization) of those concepts. Sect. 6 then continues by considering how, in the context of ArchiMate, deep modeling could provide support for other desirable modeling features identified by Frank and others, but not yet included as explicit goals of ArchiMate. Sect. 7 concludes with final remarks and observations for future work.

2 ArchiMate Language Goals

ArchiMate is an EAM standard managed by the Open Group, a vendor and technology-neutral industry consortium that manages a wide range of open standards. The Open Group characterizes ArchiMate as "*. . . the standard visual language for communicating and managing change and complexity through architecture descriptions development*" (The Open Group 2017b). It is complemented by the TOGAF standard (The Open Group 2010) which provides a broader picture of how to create, evolve and leverage enterprise architectures in a disciplined way. This article focuses on ArchiMate version 3.0 released in June 2016 (The Open Group 2017a).

The main value of ArchiMate is the set of modeling concepts it provides for representing different aspects of enterprise architectures from multiple viewpoints. These modeling concepts are organized in a two-dimensional "core framework", illustrated in Fig. 1. The rows of this figure, referred to as *layers*, represent the different levels of abstraction at which properties of the enterprise are described (*business*, *application* and *technology*), while the columns, referred to as *aspects*, represent the kinds of concepts used to represent that information (*passive structure*, *behavior* and *active structure*). These concepts, along with the rest of ArchiMate are defined by a metamodel in an analogous way to the UML.

ArchiMate also places a great deal of emphasis on defining a "*default iconography for describing, analyzing, and communicating many concerns of Enterprise Architectures as they change over time*". Most other open EAM approaches such as TOGAF and Zachman do not define any iconography while others such as RM-ODP do so in a limited way. ArchiMate is the only major EAM *standard* that attempts to define a comprehensive range of symbols specifically for modeling EAs from multiple viewpoints using metamodel-based language engineering techniques.

It is not the goal of this article to critique the concrete EAM concepts defined in the ArchiMate language which have been carefully refined over a

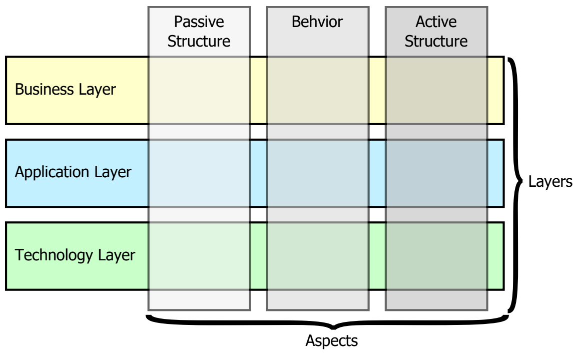


Figure 1: ArchiMate Core Framework (The Open Group 2017a)

period of years based on user experience and research. Rather the goal of this article is to critique the architecture of the language infrastructure and discuss whether the goals of the language designers could be better satisfied using multi-level modeling. Thus, the focus is more on *how* the language is defined and what overall properties it exhibits rather than *what* it actually contains in terms of abstract and concrete syntax. This requires close scrutiny of the ArchiMate standard which constitutes the official definition of the language. In the following subsections we clarify the main sets of goals outlined by the language designers. The standard itself does not devote a lot of space to explaining ArchiMate's concrete goals, instead it explicitly refers to other sources where a full description of the ArchiMate language design goals are described (Lankhorst 2013; Lankhorst et al. 2010).

Overall, the language designer's articulated three main kinds of requirements. These are discussed in the following subsections.

2.1 Conceptual Integrity

According to Lankhorst (2013) one of the core goals of the ArchiMate language is to maximize the conceptual integrity of EA models. Brooks describes conceptual integrity as resulting from "*simplicity and straightforwardness*" and from ensuring "*unity of design*" in which "*every part must reflect the same philosophies and the same balancing of desiderata*" (Brooks 1975). Lankhorst

(2013) characterize conceptual integrity as "*the degree to which a design can be understood by a single human mind, despite its complexity*" and state that it has the following subordinate design principles:

- orthogonality - do not link what is independent,
- generality - do not introduce multiple functions that are slightly divergent,
- economy (aka parsimony) - do not introduce what is irrelevant,
- propriety - do not restrict what is inherent.

The core idea of simplicity and making the language as "*compact as possible*" (Lankhorst et al. 2010) is explicitly highlighted in the standard which states that: "*the most important design restriction on the language is that it has been explicitly designed to be as small as possible, but still usable for most Enterprise Architecture modeling tasks*" (The Open Group 2017a). These authors also explain that "*similar things should be expressed in a similar way, using a simple set of core concepts that are easy to learn and understand*" (Lankhorst 2013).

Another set of principles that aim to maximizing conceptual integrity by ensuring "*economy of communication*" are Grice's Maxims (Grice 1975):

- Maxim of Quantity
  - make your model as informative as necessary.
  - do not make your model more informative than necessary.
- Maxim of Quality
  - do not model what you believe to be false.
  - do not model that for which you lack adequate evidence.
- Maxim of Relevance
  - be relevant (i. e., model things related to the modeling goal).
- Maxim of Manner
  - avoid obscurity of expression.
  - avoid ambiguity.
  - be brief (avoid unnecessary concepts and relations).
  - be orderly.

Of these, the fourth, "Maxim of Manner", is the most relevant to language design. As Lankhorst et al. point out, in order to avoid ambiguity and ensure concepts can be "*mapped easily to and from those used in the project level*" a strong relationship should exist between the modeling concepts used at the project level and those used in the enterprise architecture (Lankhorst et al. 2010). It is therefore important that the language be "*set up in such a way that project level modeling concepts be expressed easily in terms of the more general concepts defined in the language (e. g.,, by specialization or composition of general concepts)*" (Lankhorst et al. 2010).

2.2 Customizability

The designers of ArchiMate faced the same dilemma as the designers of the UML when deciding what to include in the language standard since they both have huge numbers of users with greatly diverging requirements. The ArchiMate designers therefore took the same approach as the OMG by defining a small core language (to maximize conceptual integrity) accompanied by extension mechanisms to allow users to customize the language to their own domain-specific needs. However, in contrast to the UML, which only has one built-in extension mechanism (the profile mechanism), ArchiMate defines two:

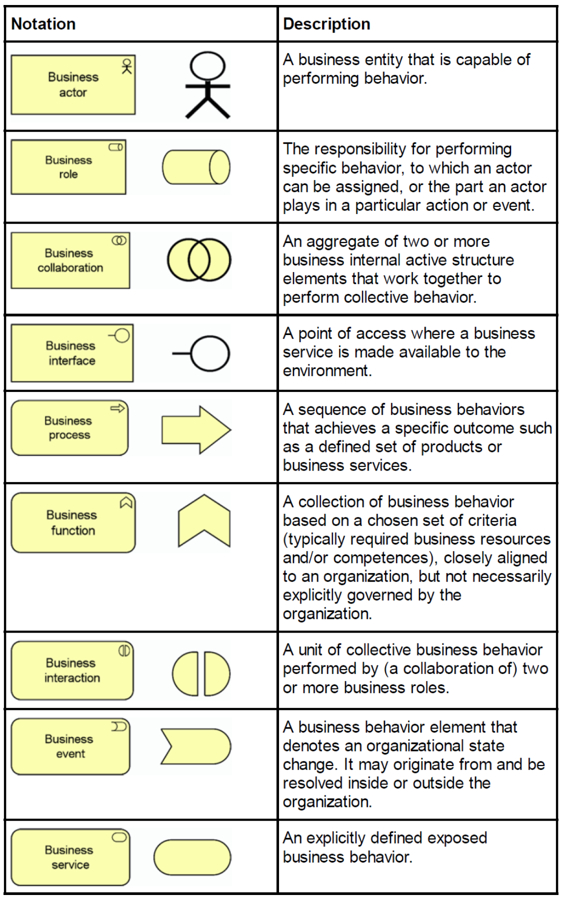
1. adding attributes to ArchiMate elements and relationships, and
2. specialization of elements and relationships.

The first is seen as essential for supporting special kinds of analyses on models and allowing more information to be communicated about the model elements. This can be done at modeling time by a modeler (called user-defined customization), or it can be done when a modeling tool is first configured for use (called pre-defined customization). The purpose of the second form of customization is "*to define new elements or relationships based on the existing ones*" by logically defining subclasses of the model elements in the predefined metamodel. The ArchiMate standard states that "*specialized elements inherit the properties of their generalized elements (including the relationships that are allowed for the element), but some of the relationships that apply to the specialized element need not be allowed for the generalized element*".

The ArchiMate standard envisages that both forms of customization are defined using some kind of "profiling" mechanism such as the textual profiling language described by Eertink et al. (1999) or the UML profiling mechanism. However, the standard gives very little information about the precise nature of the mechanisms, and equivocates on whether, in the second case, the mechanism is meant to be exactly the same as the UML profile mechanism or just "similar". For example it states that "*A specialized element or relationship strongly resembles a stereotype as it is used in UML*" (The Open Group 2017a, p. 111). Profiles are also described in extension packages that can be loaded and unloaded at modeling time as desired.

2.3 Visualization Flexibility

The extensions mechanisms discussed in the previous section are concerned with extensions of the abstract syntax of the language (i. e., what information the extensions capture). However, the ArchiMate standard also envisages extensions to the concrete syntax (i. e., how language concepts are represented). For example, the standard states that the aforementioned profile mechanism is intended to make it possible to "*define a specific notation to denote the specialization*" (The Open Group 2017a, p. 112). Moreover, it explains that "*new graphical notation could be introduced for a specialized concept, but preferably with a resemblance to the notation of the generalized concept; e. g., by adding an icon or other graphical marker, or changing the existing icon.*" This means that end users should be able to extend the iconography of the ArchiMate language, as



Notation	Description
Business actor	A business entity that is capable of performing behavior.
Business role	The responsibility for performing specific behavior, to which an actor can be assigned, or the part an actor plays in a particular action or event.
Business collaboration	An aggregate of two or more business internal active structure elements that work together to perform collective behavior.
Business interface	A point of access where a business service is made available to the environment.
Business process	A sequence of business behaviors that achieves a specific outcome such as a defined set of products or business services.
Business function	A collection of business behavior based on a chosen set of criteria (typically required business resources and/or competences), closely aligned to an organization, but not necessarily explicitly governed by the organization.
Business interaction	A unit of collective business behavior performed by (a collaboration of) two or more business roles.
Business event	A business behavior element that denotes an organizational state change. It may originate from and be resolved inside or outside the organization.
Business service	An explicitly defined exposed business behavior.

Figure 2: Business Layer Notation (Excerpt of tab. 6 from The Open Group (2017a))

well as the abstract syntax, but that they should "ideally" do so in a way that specializes the existing notation. The standard also clarifies that the "*default is the guillemet notation of UML for stereotypes ('«specialization name»')*" and that other "*options include specific icons, colors, fonts, or symbols.*"

Fig. 2 shows a summary of the built-in notation used to represent the static structural and behavioral elements of the behavioral layer (an excerpt of tab. 6 from the ArchiMate Standard (The Open Group 2017a)). As can be seen from this figure, model elements are either visualized by (a) a single icon corresponding to the element's type (the icon notation) or (b) a rectangle with the element's type icon in the top right hand corner (the box notation). This generic visual syntax, however, is mainly intended for use by enterprise architects. Various end users (i. e., stakeholders of a system) are intended to use *viewpoints* offering domain-specific visualizations.

Although no semantics are formally assigned to colors, in practice they are frequently used in the standard to distinguish between the layers of the ArchiMate Core Framework as follows:

1. yellow for the Business Layer,
2. blue for the Application Layer,
3. green for the Technology Layer.

In addition to colors, other notational cues can be used to distinguish between the layers. For example, a letter such as 'B', 'A' or 'T' can be placed in the top-left corner of an element to denote a Business, Application or Technology element, respectively. The standard notation also uses a convention based on the corners of symbols to distinguish element types. More specifically:

1. square corners indicate structure elements,
2. round corners indicate behavior elements,
3. slanted corners indicate motivation elements.

3 Deep Modeling

Deep modeling describes a family of modeling approaches that support seamless, level-agnostic modeling across an unlimited number of classification levels. The approach uses a linear hierarchy of models in which the elements at one level are all instances of model elements at the next level, organized according to the principles of strict modeling (Atkinson and Gerbig 2016). As a consequence, all classification levels are equally accessible to modelers for viewing and modification, and changes made to one classification level are immediately available at all other classification levels without the need for recompilation or re-deployment steps.

This flexibility brings additional economic benefits beyond those provided by reduced model complexity and error proneness because it makes EAM modeling environments much more open and extensible. Tools based on two-level modeling technologies usually hardwire their metamodels into their code (such as contemporary ArchiMate tools), and thus reduces opportunities for exchanging, reusing and extending models. This in turn tends to lock users into particular vendor solutions leading to significantly higher costs and potential platform migration changes in the future.

One way of realizing such a level-agnostic modeling environment is through the Orthogonal Classification Architecture (OCA) shown in Fig. 3. In such an architecture, all model elements are typed by two types, their ontological type which originates from the problem domain, and their linguistic type which is defined in the deep modeling language. Another key ingredient of deep modeling is the notion of "potency" which essentially captures the "characterizing" power of concept over multiple classification levels. There are several deep modeling approaches offering slightly differing forms of potency. However, these differences are irrelevant to the arguments presented in this article. For the purpose of presenting concrete examples in the rest of the article we used the deep modeling variant developed by the authors of this paper. It supports three distinct variants of potency: clabject potency which establishes an upper bound on the instantiation depth of a clabject, attribute existence potency (aka "durability") which defines over how many instantiation levels an attribute has to exist, and attribute value potency (aka "mutability") which specifies over how many instantiation levels the value of an attribute can be changed. Any instances created beyond that level must retain the value set at the lowest changeable level.

A schematic example of the use of the OCA is displayed in Fig. 3 where linguistic classification is indicated by dotted arrows and ontological classification is indicated by dashed arrows. *Manager* in the middle level, O_1, has *Clabject* as its linguistic type and *ActorType* as its ontological type. Note that *Manager* is not only an instance of *ActorType* but also a type for *Ann*. This type/instance duality of model elements residing in the middle levels motivates the term "*clabject*" - a portmanteau of the terms "*class*" and "*object*".

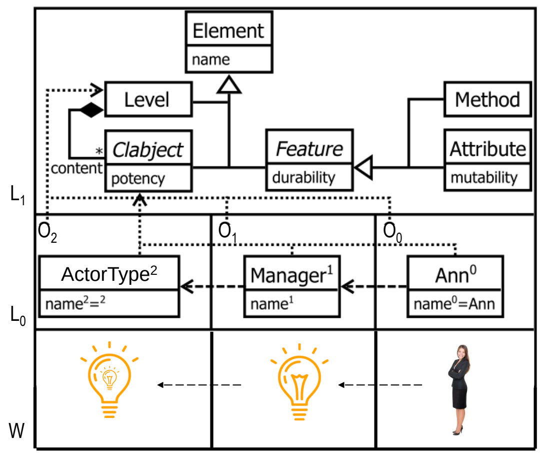


Figure 3: Example of a Deep Model in an Orthogonal Classification Architecture

The potency of a clabject can be displayed next to its name as a superscript. *ActorType* has a potency of two allowing it to have (deep) instances two ontological levels below it, in this case *Ann*. The potency of an instance of a type must be lower than the potency of the type, but cannot be negative, so the clabject *Ann*, with potency zero, cannot have instances itself.

Attribute and value potencies (i. e., mutabilities) are displayed as superscripts next to attribute names and values respectively. In the case of *ActorType*, the *name* attribute has an attribute potency of two and a mutability of two. Hence, instances of *ActorType* instances (i. e., two levels down from *ActorType*) have to have name values and these name values may differ. To minimize the amount of information displayed in a deep model, deep modeling tools may apply elision rules for all forms of potency. For example, one approach is to display attribute potency (i. e., durability) only when it is not equal to the potency of the clabject containing the attribute and to display mutability only when it is not equal to the durability.

4 Conceptual Integrity in ArchiMate

In the next three sections we consider how well ArchiMate's core language and architecture satisfies the requirements discussed in section 2 and investigate whether multi-level modeling could lead to improvements. In this section we start with the goal of conceptual integrity.

As explained in the introduction, the definition and use of the ArchiMate core language essentially takes place in the context of a traditional two-level modeling platform or infrastructure. This is illustrated schematically in Fig. 4. The left hand side of this figure depicts a traditional modeling stack such as the MOF/UML as usually presented by the OMG (e. g., the UML Infrastructure specification (Object Management Group 2006, 2007)). The two levels explicitly supported by ArchiMate are the two non-shaded levels in Fig. 4. The upper of these two levels, labeled "Metamodel" is the place where the abstract syntax of the core language is modeled using a language akin to the MOF. It is called a "metamodel" because its instances are parts of a model. The lower of these two levels, labeled "Model", is the level where end users of an ArchiMate modeling tool add their domain content. Note that, in general, ArchiMate's model level is used to represent both domain types and domain instances.

The meta-meta model is not part of the ArchiMate language per se, it is the language used to describe (i. e., model) the abstract syntax of the ArchiMate language. Nevertheless, since it is explicitly used in the standard to model the ArchiMate language, it conceptually constitutes the top of the stack of languages depicted on Fig. 4, just as the MOF constitutes the top level of the OMG's four level modeling infrastructure. The role of the meta-metamodel used in the ArchiMate standard is therefore exactly the same as the role of the MOF in the UML specification, even though it is not explicitly represented in the ArchiMate standard or documentation. In fact, the ArchiMate meta-meta model appears to be a simplified version of the MOF.

Level contents at both the model and metamodel levels of the ArchiMate modeling infrastructure are represented using concepts which originate from the same underlying object-oriented modeling principles that underpin the UML, and thus also the MOF - namely, the concepts of classes

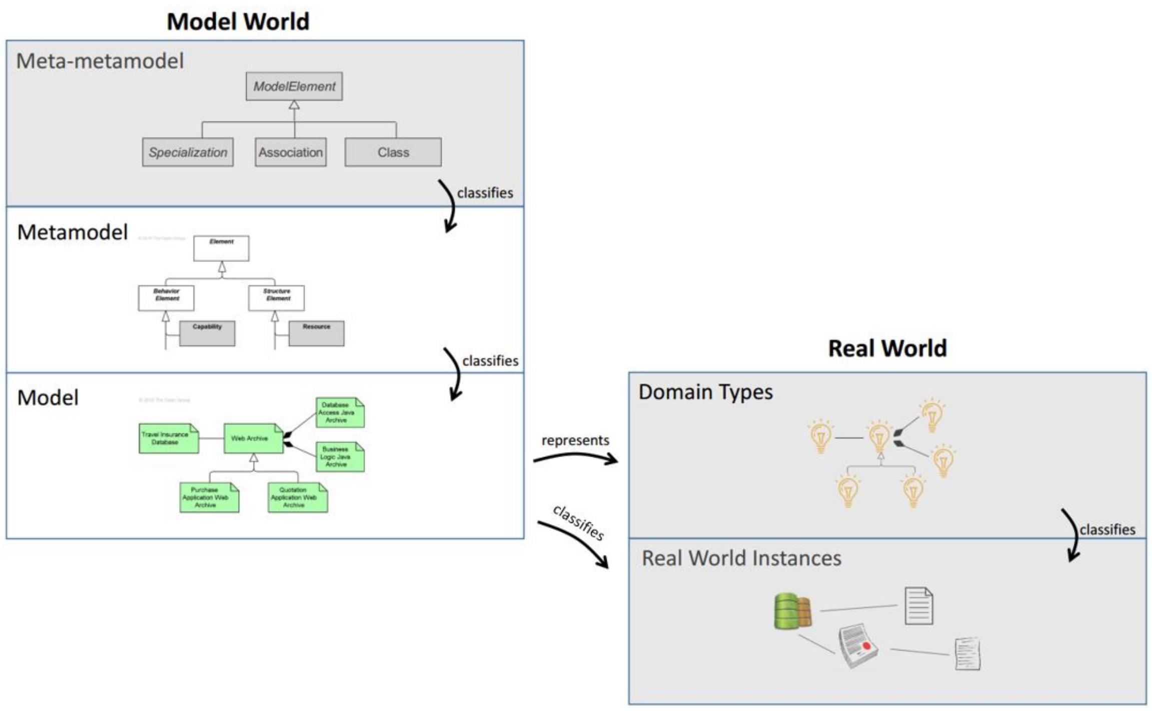


Figure 4: ArchiMate Modeling Infrastructure

(whose "standard" presentation is roughly a rectangle), specialization/generalization (whose "standard" presentation is an arrow with a triangle head), and associations of various kinds (whose "standard" presentation is a line annotated with certain symbols to provide particular meanings). We can therefore observe that the ArchiMate metamodeling language and modeling language are based on the same foundation.

4.1 Language Duplication

Although the ArchiMate standard makes no reference to the language used to define the metamodel, the fact that the metamodel is explicitly modeled in the standard using an object-oriented modeling language means that, conceptually at least, there is an additional level above the ArchiMate metamodel describing the language used to define it. In Fig. 4 this is referred to as the meta-meta model. Although the ArchiMate standard only refers to, and only intends tools to support, the middle two levels, ArchiMate users must at least conceptually be aware of all four levels when working with ArchiMate. This leads to the first, *language duplication* issue:

The ArchiMate language (*i. e., metamodel*) *is defined using a language* (*i. e., meta-metamodel*) *which appears to have "very similar" core features to the language itself* (*i. e., classes, associations, inheritance etc.*). *However, the meta-metamodel is not explicitly defined in the ArchiMate standard or literature. The ArchiMate standard therefore faces a situation similar to that which existed between the MOF and UML before the OMG's initiative to align them in the* 2.0 *versions of the languages. In the worst case, features in the meta-metamodel may only be "similar" to, but not exactly the same as, corresponding features in the metamodel* (*e. g., specialization*) *leading to confusion and errors.*

Ideally, the semantics of corresponding features should therefore be "aligned", as with the UML core and MOF since UML 2.0, but there is currently no indication that this is intended to be the case in the ArchiMate standard. Either way, the conceptual integrity of the standard is compromised. At the very least, when there is alignment, the infrastructure fails to live up to the spirit of minimality and unity of design that underpins concept integrity, and in the worst case, when there is no alignment, two of the aspects of the Maxim of Manner - avoiding “ambiguity” and avoiding “unnecessary concepts and relationships” - are directly contradicted. One of the four subordinate principles of conceptual integrity identified by Lankhorst et al. (2010) explicitly warns against introducing “multiple functions that are slightly divergent”.

The deep modeling approach naturally addresses this issue because the OCA can be used to unify the definition of these common concepts and make them available at both the metamodel and model levels, as shown in Fig. 5. In this figure, instead of a meta-meta model which defines the MOF-like modeling feature just for the metamodel level, a level-spanning linguistic model (or “metamodel”) exists which defines the concepts for use in both the metamodel and model levels. This not only avoids redundancy, it ensures, and explicitly declares, that a given linguistic concept has the same meaning at both levels. Thus, the two left-hand levels in Fig. 5 are guaranteed to have the same semantics for the common elements they share (i. e., classes/object, inheritance, associations/links, attributes/slots) because they are defined in the single, overarching linguistic model. Moreover, there is no need for a meta-meta model at the top of Fig. 4 because this role is performed by the linguistic model.

4.2 Type/Instance Ambiguity

Sect. 3.6 of the ArchiMate standard states that “*The ArchiMate language intentionally does not support a difference between types and instances. At the Enterprise Architecture abstraction level, it is more common to model types and/or exemplars rather than instances. Similarly, a business process in the ArchiMate language does not describe an individual instance (i. e., one execution of that process). In most cases, a business object is therefore used to model an object type (cf. a UML class), of which several instances may exist within the organization. For instance, each execution of an insurance application process may result in a specific instance of the insurance policy business object, but that is not modeled in the Enterprise Architecture*”.

An analysis of the examples in the standard shows that, across all levels, behavior and passive structure views indeed typically show types. However, active structure views typically show instances and there are many examples of models that contain a mixture of types and instances. In general, ArchiMate is rather relaxed about the difference between instances and types and how they may be mixed in a domain model. The standard even explicitly states (sect. 8.2.1) that “*Business actors may be specific individuals or organizations; e. g., ‘John Smith’ or ‘ABC Corporation’, or they may be generic; e. g., ‘Customer’ or ‘Supplier’*”. An example model illustrating this is shown in Fig. 6 (Example 22 from the standard). Here *Travel Insurance Claims Analyst*, *Home Insurance Product Specialist* and *Customer Service Representative* are clearly types since they all inherit from *Specialist*, while *Greg*, *Joan* and *Larry* are individuals. Although the latter are clearly instances of the former the assignment relationships used to connect them in the model does not convey classification, but rather the assignment of responsibilities. Finally, all the structural elements in the model (both types and instances) are regarded as instances of the same metamodel element - *Active Structure Element*. ArchiMate therefore has a *type/instance ambiguity* issue:

It is not, in general, clear whether a model element in an ArchiMate model (*i. e., at the model level in the infrastructure*) *represents an instance or a type in the domain. Moreover, types and instances can be mixed and interrelated in uncontrolled ways. This lack of discrimination directly conflicts with the Maxim of Manner* (*avoid ambiguity*) *and the propriety subprinciple of conceptual integrity which warns against restricting “what is inherent”. In Fig.* 6 *the most natural* (*i. e., inherent*) *relationship between the types and instances is “instance of” but there is no way of representing such “ontological” instance of relationships in ArchiMate.*

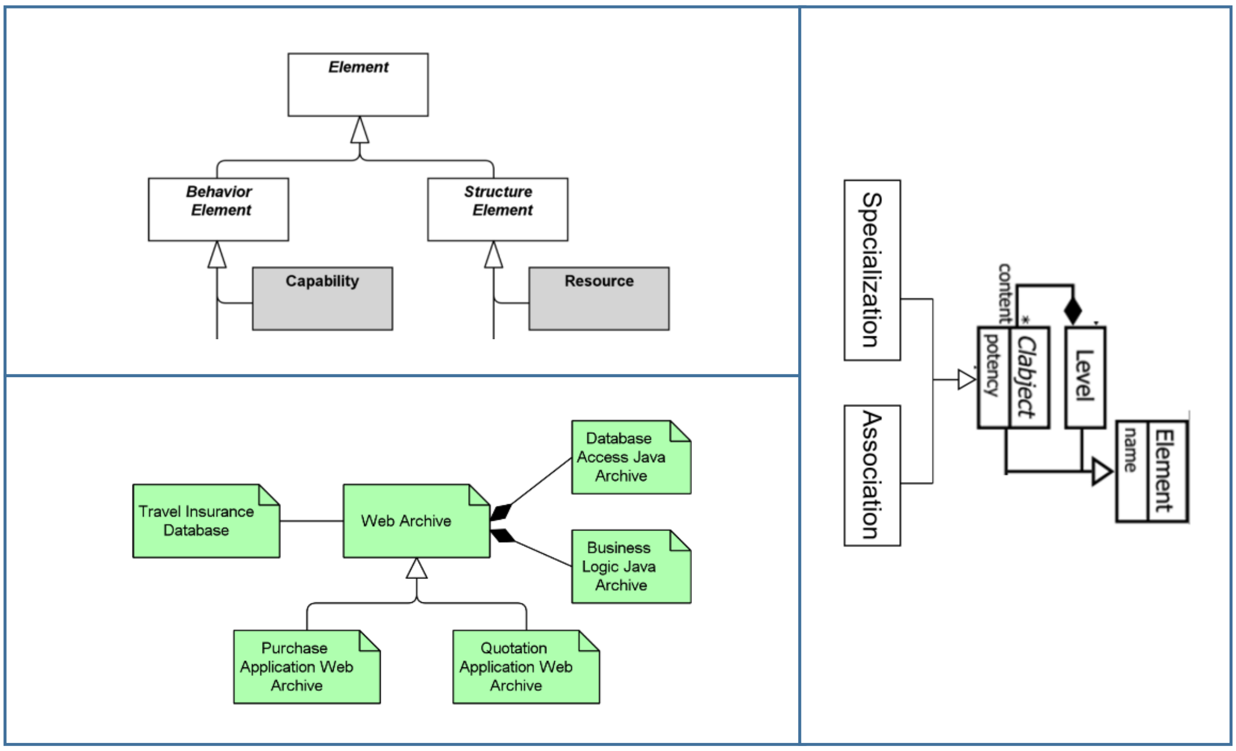

Element
Behavior Element
Structure Element
Capability
Resource
Database Access Java Archive
Travel Insurance Database
Web Archive
Business Logic Java Archive
Purchase Application Web Archive
Quotation Application Web Archive
Specialization
Association
content
*
Clabject
potency
Level
Element
name

Figure 5: OCA Core Language Definition (Atkinson and Kühne 2003)

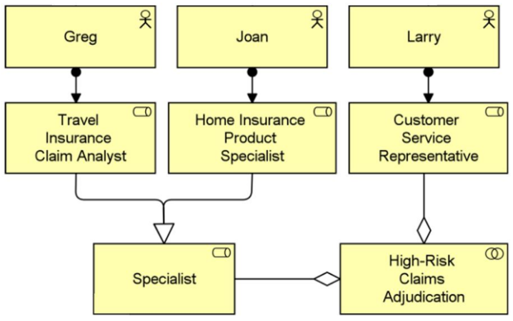

Greg
Joan
Larry
Travel Insurance Claim Analyst
Home Insurance Product Specialist
Customer Service Representative
Specialist
High-Risk Claims Adjudication

Figure 6: Example Business Active Structure Elements (The Open Group 2017a, Example 22)

Mixing instances and types in the same level is not a problem per se as long as the types of the instances do not appear together (i. e., as long as a type in a level has no instances in that level) (Atkinson and Kühne 2002). However, when types and instances can be arbitrarily mixed within levels, without any notational support for discrimination, modellers can easily become confused and fail to adhere to basic modeling principles. The lack of distinction between types and instances also means that modeling well-formedness constraints may not apply as intended. For example, sect. 5.4.1 of the specification states "*A specialization relationship is always allowed between two instances of the same element*", where "element" here refers to a metamodel element such as Business Actor. However, since instances of Business Actor can also be individuals (i. e., uninstaniatable objects) in the domain of discourse (e. g., Greg, Joan and Larry in Fig. 6) this rule unintentionally enables specialization relationships between individuals.

Note that while most two-level approaches, such as the UML, strictly separate instances (as modeled by objects) from types (as modeled by classes), they fail to properly distinguish between types and metatypes (Atkinson and Kühne 2007). ArchiMate's single-level approach to accommodating both types and instances therefore essentially transposes all issues arising from the conflation of levels to a lower part of the modeling stack than is typically the case.

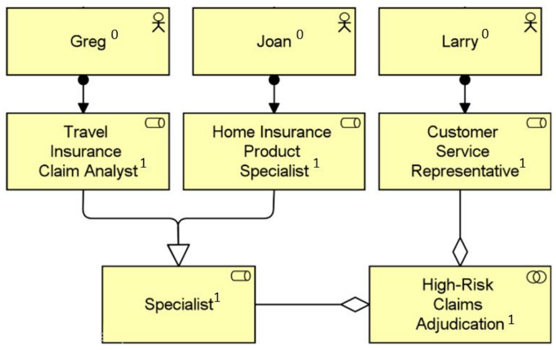

Greg 0
Joan 0
Larry 0
Travel Insurance Claim Analyst 1
Home Insurance Product Specialist 1
Customer Service Representative 1
Specialist 1
High-Risk Claims Adjudication 1

Figure 7: Business Active Structure Elements with Potency

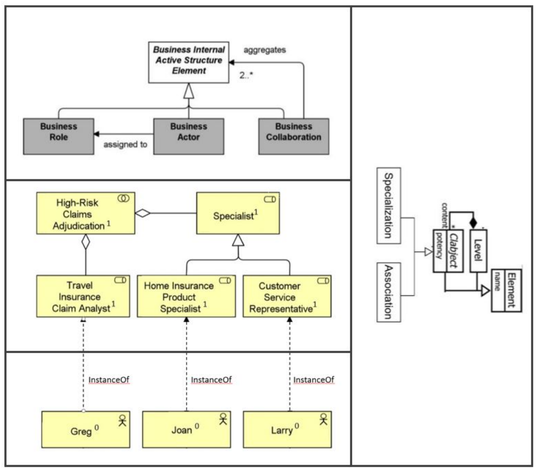


Figure 8: Domains Types and Instance related by Ontological Classification

Deep modeling can be used to address issues caused by an insufficient separation of elements at different classification levels by introducing a notation to explicitly indicate whether a model element represents a type that classifies multiple instances in the modeled subject domain or represents an instance that does not. In other words, deep modeling can be used to explicitly show the location of model elements in the ontological classification hierarchy, since the location is implied by the ontological classification relationships in the subject domain.

This can be seen in Fig. 7 which shows the same model as Fig. 6 but using potencies to show whether elements represents types or instances. The names of model elements representing types have a potency value “1” appearing as a superscript after their name, while model elements representing instances have “0”as a superscript. Note that the inheritance relationships exist only between types. The explicit presentation of type-/instance properties in this way makes it easier for tools to ensure that specialization relationships do not erroneously exist between model element that represent instances, such as *Greg*, *Joan* or *Larry*. Fig. 8 goes one step further and shows the model content from 6 separated into distinct ontological levels with the natural ontological *instanceOf* relationship between them. This multi-level version also avoids the problem of domain instances and types being instances of the same metamodel element.

5 Customizability in ArchiMate

As explained in Sect. 2, ArchiMate identifies two customization mechanisms, one designed to extend the predefined core modeling concepts with attributes and the other designed to extend the core language with specializations of the predefined core concepts using nominal (i. e., attribute-free) typing. Apart from calling them both “profile” mechanisms, the standard provides little detail about how they would work and what detailed features they provide. The standard only hints at what they might be like by, in the first case referencing the textual profiling mechanism proposed by Eertink et al. (1999) and, in the second case referencing the graphical/textual profiling mechanism of the UML. There are however many forms a “lightweight” extension mechanism could take that does not require full metamodel editability. For example Brunelière et al. (2015) describe a lightweight approach based on the use of a textual DSL for metamodel extension that has been used in the EAM domain. Like the UML extension mechanism, this is a form of non-invasive model decoration approach which avoids making direct changes to the underlying metamodel because it is often hardwired in a propriety tool (Kolovos et al. 2010). Although these approaches differ in terms of the exact set of operations they offer, their strengths and weaknesses relative to an approach that allows full access and changeability of the metamodel are basically the same. Therefore, without loss of generality, in the remainder of the this article we will assume the UML extension mechanism when discussing the pros and cons of deep modeling. The UML extension mechanism is the most well-known, and subsumes Eertink et al.'s approach (since it is able to add attributes to base types).

5.1 Superfluous Languages

The main problem with the ArchiMate approach to customization is that it requires users to essentially learn an additional language in order to understand and define an extension - namely the profile modeling features (Atkinson et al. 2013). Thus, the meta-metamodel depicted in Fig. 4 actually has to contain two language definitions, the core "metamodeling" language used to define the standard metamodels and the profile language used to define profiles.[1] This lowers conceptual integrity because the core language already contains a simpler and more fundamental mechanism for extending models and metamodels - the specialization (i. e., inheritance) mechanism. As mentioned before, just to understand the metamodel, ArchiMate users have to learn the meaning of the specialization relationship that exist between the metamodel elements anyway. However, this relationship is exactly the mechanism used in the second extension mechanism. Thus, if the core language were extended to support attributes, which most object-oriented modeling languages do, no additional mechanism beyond specialization would be needed to support extensions.

The reason why direct specialization of the metamodel is not available in existing tools such as ArchiMate is their reliance on two-level modeling platforms in which the metamodel is hardwired. Specialization can only be used for language extension if the language is truly represented as changeable data (i. e., a metamodel) so that any changes are automatically recognized by the tool. The ArchiMate standard's implicit adoption of two-level modeling, and its lack of support for direct, user-defined specialization of the metamodel, therefore leads to a *superfluous language* issue:

Because direct, user-defined specialization of the metamodel is not supported or envisioned by the ArchiMate standard - despite the fact that specialization is used in the definition of the metamodels - an additional set of modeling concepts, the profile definition and application concepts, have to be provided to users and implemented by tools. This directly contravenes the goal of brevity in the Maxim of Manner and thus lowers the conceptual integrity of the language.

For metamodels to be directly extensible using the regular specialization mechanisms they cannot be hardwired into tools, as is the common practice today (Atkinson et al. 2013). They must be represented as "soft" model content that can be changed just like the normal "models" at the level below. This, in turn, means that the metamodel elements need to be explicit instances of a "high-level model", such as the metamodel in Fig. 4 or the level-spanning linguistic model shown in the schematic representation of the deep modeling approach in Fig. 5.

For the reasons explained above, a level-spanning approach is to be preferred since it avoids the language duplication problem discussed earlier. With such an architecture, extensions can be easily defined by explicitly extending the metamodel using specialization as shown in Fig. 9. This shows how the business layer metamodel as described in the ArchiMate specification (The Open Group 2017a) can be extended to support the business layer specialization example shown in sect. 15.2.1 of the ArchiMate standard. The top package contains the generic business layer metamodel consisting of *BusinessInternalActiveStructureElement* (that represent the static structure of an organization), *BusinessInterface* (that expose functionality to other roles or actors), *BusinessInternalBehviorElement* (that represent the behavior of active structure elements), *BusinessEvent* (that trigger or interrupt behavior), *BusinessService* (that expose business

[1] This resembles the top-level UML Infrastructure library, used to define the concepts shared between the UML and the MOF (i. e., a core modeling package and a profile packages).

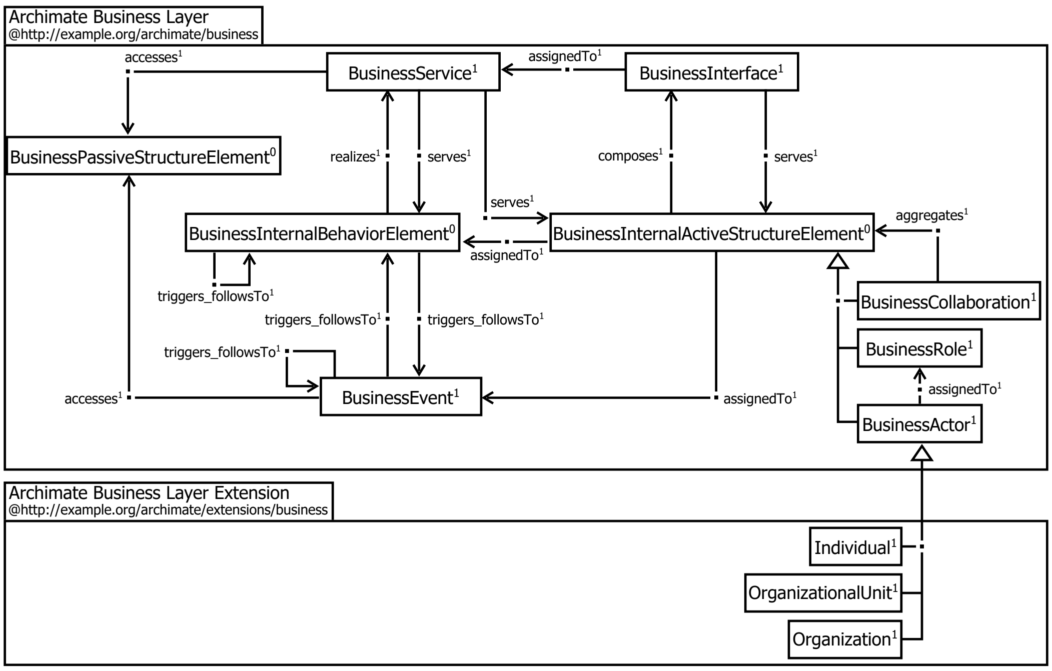


Figure 9: Excerpt of the Business Layer Metamodel and suggested Extensions.

behavior), *BusinessPassiveStructureElement* (that represent business objects which are manipulated by behavior). *BusinessInternalActiveStructureElements* are further divided into *BusinessActor* (that represent entities capable of performing behavior), *BusinessRole* (that represent entities responsible for performing specifically assigned behavior) and *BuinsessCollaboration* (two or more active structure elements that perform a behavior). *BusinessPassiveStructureElement*, *BusinessInternalBehaviorElement* and *BusinessInternalActiveStructureElement* are abstract, indicated by their potencies of zero, so only their subclasses can be instantiated.

Although the ArchiMate standard conceptually envisages the specialization of metamodel elements, in common with prevailing "two-level modeling" practice it does not view the actual use of specialization relationships as a practical mechanism for achieving this. This necessitates the inclusion of extra extension mechanisms. In contrast, since all classification levels are soft in a deep modeling framework (i. e., not hard-coded into tools) metamodel customizations can be performed out-of-the-box without the need for special profile mechanisms.

In the example shown in Fig. 9, a package called *ArchiMate Business Layer Extension* is added to the model. This package contains the extensions for *BusinessActor* suggested by the ArchiMate standard (sect. 15.2.1), that is -

- *Individual* - a natural person capable of performing behavior.
- *OrganizationalUnit* - a subdivision (e. g., department) of an organization.
- *Organization* - an institution, corporation or association that has a collective goal linked to an external environment.

Both packages, the *ArchiMate Business Layer Package* and *ArchiMate Business Layer Extension*, display a URL beneath their package name. They can therefore be stored on the internet and imported when needed as described in (Atkinson et al. 2015).

Note that the extensions shown in Fig. 9 show another example of ArchiMate's lack of distinction between, and clear treatments of, types and instances. The existence of *Individual* in the metamodel suggests that its instances at the level below are individuals. However, *OrganizationalUnit* is a concept that would often be instantiated to a type at the level below (e. g.,"Marketing Department") which in turn would have instances at the level below that. For example, large international companies will usually have several marketing departments (i. e., instances) in each of the countries where it operates. This situation can not be handled cleanly in ArchiMate.

5.2 Deep Characterization

An important notion in EAM is the notion of "integrity", i. e., the fact that certain principles can be relied on with respect to both the static structure and the dynamic behavior (Lankhorst 2013). Thus, in the previous example revolving around business actors, while specific business actors may vary, there are common principles which ought to hold in all cases. For example, a common principle is that all business actors are assigned a business role, regardless of what type of business actor they are. Although these properties ultimately materialize at the instance level they have to be defined for all kinds of business actor types.

The top package in Fig. 9 shows an excerpt of the ArchiMate metamodel that deals with business roles and business actors. This includes a relationship *assignedTo* that requires business roles be assigned to business actors. However, this constraint applies generically regardless of what kind of business role or business actor is involved. Most enterprises, however, require that business roles only be assigned to business actors that are qualified to perform them. Thus, in the example in 6, one would hope that Larry has been assigned the customer service representative role because he is a customer service expert. However, this is not guaranteed.

This gives rise to the *shallow characterization* issue:

A simple profiling mechanism that merely supports specialization is incapable of concisely defining constraints that guarantee the presence of features (e. g., attributes, associations, etc.) two or more levels down.

A straightforward way to achieve the aforementioned requirement in a deep modeling approach is to use so-called "deep characterization". The latter can be achieved through potency values higher than one and/or using variants of the powertype pattern as popularized by the UML. As Gonzalez-Perez and Henderson-Sellers have pointed out (Gonzalez-Perez and Henderson-Sellers 2007), the application of the powertype pattern is a form of multi-level modeling, at least conceptually. Powertypes are essentially domain metatypes whose instances are subtypes of predefined metamodel elements such as *BusinessActor*.

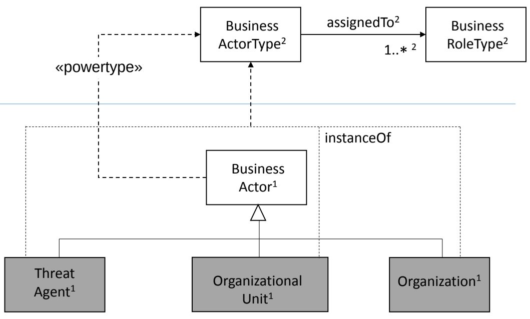


Figure 10: BusinessActor Powertype

An example of the use of the deep modeling approach to achieve deep characterization is shown in Fig. 10. This shows the *BusinessActor* metamodel element along with three of the specializations suggested in sect. 15.2.1 of the standard. As can be seen from this figure, an additional domain-specific level has been added in which *BusinessActorType*^{2} and *BusinessRoleType*^{2} have been defined, where the former is the powertype of *BusinessActor*. This means that any instance of *BusinessActorType*^{2} must specialize supertype *BusinessActor*. Supertype *BusinessActor* may thus prescribe certain attributes that all its subtypes automatically inherit, i. e., force them to have mandatory relationships with other types such as *BusinessRole*. Fig. 10 shows an alternative way of achieving the aforementioned mandatory association to *BusinessRole* by a deep association and deep multiplicity constraint (Kühne et al. 2015).

Either way, all specializations of *BusinessActor* are thus required to have regular (potency-one) *assignedTo* associations to instances of *BusinessActorType*[2]. Carvalho and Almeida (2016) show that by extending standard multi-level modeling with some additional kinds of cross-level relationships, it is possible to provide even richer controls over the nature of the extensions that can be made to metamodels.

6 Visualization Flexibility in ArchiMate

The previous section discussed potential enhancements that multi-level modeling can offer to the ArchiMate language and standard in terms of abstract syntax (i. e., supported modeling concepts). In this section we consider what benefits multi-level modeling can offer the ArchiMate language and standard when applied to the visualization mechanism (i. e., concrete syntax).

As the notation summary in Fig. 2 shows, the ArchiMate core language envisages two ways of visualizing most of the element types that are used to construct ArchiMate models. One uses a standard rectangle-based shape (e. g., round cornered rectangle = behavioral element) to identify the basic nature of the element, colors to denote the layer in which element appears (e. g., business = yellow) and small icons to identify the particular type of element (e. g., event, process etc.). The other approach uses just the icons together with the color conventions to present the particular element types. We refer to the former as the "rectangular" form and the latter as the "iconographic" form. Fig. 2 shows only a part of the business layer notation, but the principles are the same for all other levels.

6.1 Notation Interaction

ArchiMate intends that both forms be arbitrarily mixeable within the same model. For example, Fig. 11, which is Example 23 from the ArchiMate standard, shows both notational forms in use at

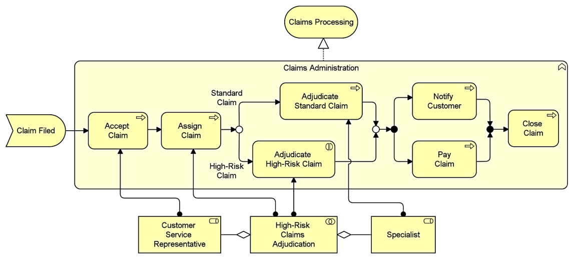


Figure 11: Business Behavioral Elements Example (The Open Group 2017a, Example 23)

the same time. In this figure, the *Claims Processing* business service and the *Claim Filed* business event are presented in the iconographic form and the other elements are presented in the rectangular form.

Supporting the arbitrary, on-demand switching between notational forms is not as trivial as it may at first appear, especially in a tool that aims to allow users to define new notations themselves. For example, the UML only defines one standard notation for each model element type defined in the metamodel, and requires all other notation forms to be defined using the profiling mechanism. However, profiles usually can only be applied in their totality to a diagram. Either the whole diagram is shown in the standard form or the whole diagram is shown in the profile notation. This leads to the *notation interaction* issue:

The ArchiMate language supports two different notations for model elements out-of-the-box, but lacks a systematic way of managing them.

Deep modeling infrastructures have a natural and simple way of separating "standard" and "domain-specific" visualizations of model elements thanks to the fundamental separation of linguistic and ontological concerns embodied by the OCA. As a consequence of this separation model elements inherently have two direct types - a linguistic one and an ontological one. By exploiting both of the dimensions to assign visualization symbols to model elements, a distinction between the fundamental linguistic notation and the domain-oriented ontological notation is naturally available. Moreover, since both forms of classification are always present, they naturally

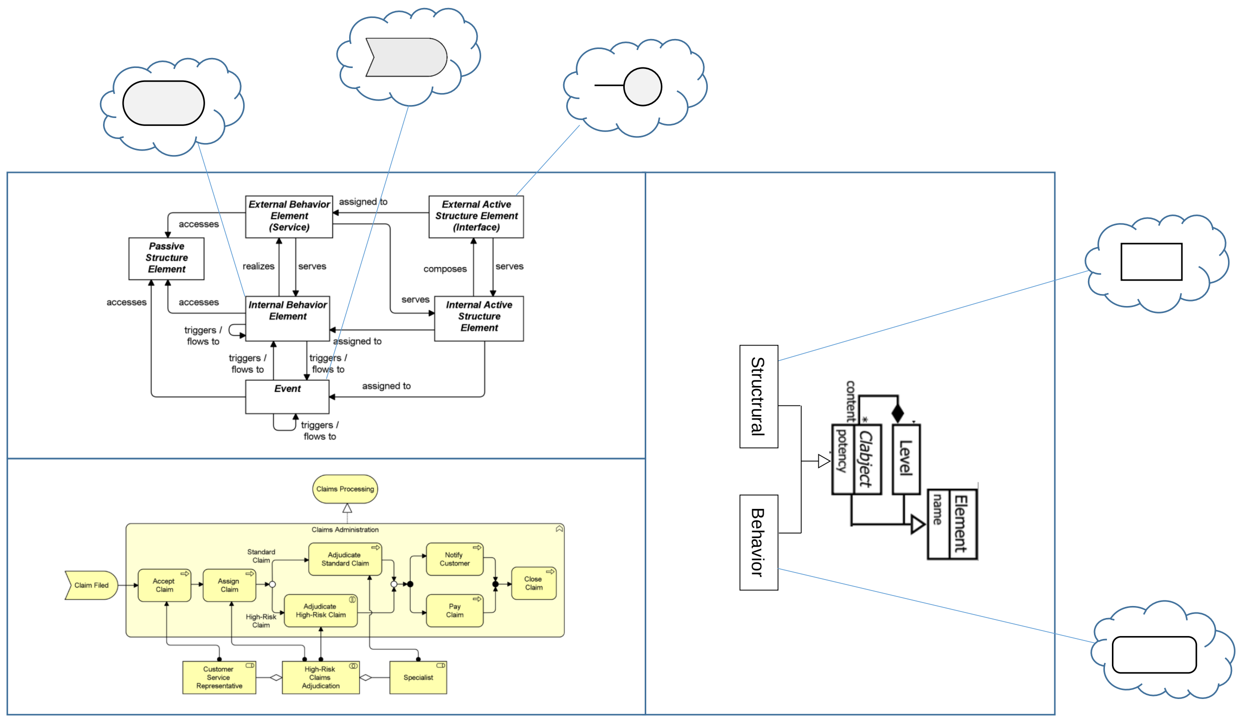


Figure 12: *OCA Based Visualization Definition*

allow the arbitrary interchange and mixing of notations. This is true regardless of what concrete mechanisms are used to assign domain-specific visualizations (e. g., icons) to model elements in the ontological dimensions and how many notations are assigned. It is also true regardless of how many ontological classification levels exist.

A visualization management approach that exploits linguistic and ontological classification is shown schematically in Fig. 12. Here the basic "rectangular" forms of structural and behavior elements (i. e., square cornered and round cornered rectangles respectively) are defined via the linguistic metamodel, while the icon forms of the different types of elements are defined via the ontological metamodel.

In the depicted scenario, two visualizations (i. e., notations) are available for all elements in the user-defined model at the bottom left of the figure which can be switched at will - a generic one originating from the linguistic dimension, and an ontological one originating from the ontological dimensions. This inherent ability of deep modeling environments to support switching between ontologically defined visualizations and linguistically defined visualization is sometimes referred to as "language symbiosis" (Atkinson et al. 2012).

Although it is theoretically possible to allow end users to change the linguistic visualization symbols, in practice these are usually hardwired into tools along with the rest of the linguistic model. The use of the cloud notation in Fig. 12 to represent visualizer assignment in both the linguistic and ontological level should not therefore be taken to mean that the mechanisms are the same, or are equally flexible from the user point of view. In Melanee, the only tool which currently allows the visualization of model elements to be controlled both by linguistic and ontological classification (Atkinson and Gerbig 2016), an arbitrary number of domain-specific visualization symbols can be defined for visualizing model elements.

6.2 Language Customization

As mentioned previously, in terms of extension mechanisms sect. 5.2 of the ArchiMate standard states that *“new graphical notation could be introduced for a specialized concept, but preferably with a resemblance to the notation of the generalized concept; e. g.„ by adding an icon or other graphical marker, or changing the existing icon.”*. This means that end users should not only be able to extend the iconography of the ArchiMate language, like the abstract syntax, they should be able to do so in a way that specializes the existing notation. Sect. 5.2 goes on to say *“The profile may also define a specific notation to denote the specialization. The default is the guillemet notation of UML for stereotypes (“«specialization name»”). Other options include specific icons, colors, fonts, or symbols.”*

The clear intent of the standard is that the iconography of specific model element types should be derived from the iconography of the more general model element types they are derived from. In other words, the visualizations of model elements, as well as the semantics of model elements, should adhere to the principle of specialization. However, this is more easily said than done. Although the visualization mechanisms of many of today’s modeling tools recognize specialization hierarchies in the sense that they are able to find and apply more general symbols to specialized classes, none currently provides systematic support for building new icons from pre-existing icon fragments. Unfortunately, the ArchiMate standard provides no details about the features of its envisaged profile mechanism, but no profile mechanism available at the present time allows the specialization of visualization symbols. This leads to a *visualization specialization* issue:

The ArchiMate language explicitly calls for users to be able to add new visualizations to the ArchiMate language in a way that specializes the existing notation (i. e., concrete syntax). However, such a mechanism is not supported by any existing profiling mechanism, nor indeed by any contemporary modeling tool.

The key requirements supporting a flexible approach to visualization specialization is decoupling the definitions of concrete syntax from abstract syntax (Espinazo-Pagán et al. 2008), and allowing the former to be attached to the latter in a simple and changeable way. This, in turn, requires unfettered access to the abstract syntax (i. e., metamodel) of the language which is a fundamental strength of multi-level modeling approaches (de Lara et al. 2015) where languages definitions can be created and edited as easily as normal model content.

In order to support the specialization of visualization (i. e., concrete syntax) an additional ingredient is needed to allow new icons to be adapted from existing one - namely, an aspect-oriented mechanism for notation definition (Gerbig 2017). Such an approach allows a notation to be customized by replacing parts declared as join points. Fig. 13 shows the aspect-oriented syntax customization applied to the example of *BusinessInternalActiveStructureElement*.

In this example, a generic diagrammatic notation is defined for *BusinessInternalActiveStructureElement* via a visualizer shown as a cloud. This notation, realizing the ArchiMate *box notation*, is named *box*. It consists of a box which displays the *name* attribute of *BusinessInternalActiveStructureElements* in its center. Two join points are placed in the shape for further customization. One, named J_A, in the upper left of the figure and one, named J_B, in the upper right of the figure.

The subtype *BusinessActor* contributes to the diagrammatic visualization through the definition of aspects. A stick man is defined for placement in join point J_B in the visualizer for the box notation. The *icon notation* is additionally defined on *BusinessActor* via a visualizer displaying *icon* in the upper right. This notation shows only a stick man for visualization and defines a join point in the visualization’s upper right for further customization. The subtype’s of *BusinessActor* further refine the *icon* and *box* notation by providing aspects to join point J_A. *Individual* contributes an *I* to the visualization whereas *OrganizationalUnit* contributes a “*U*” and *Organization* an “*O*”.

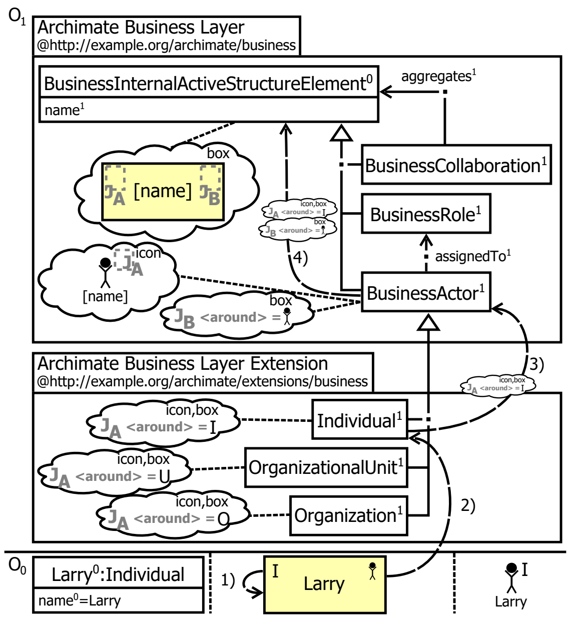


Figure 13: Aspect-oriented Notation Definition for BusinessActor.

When visualizing a model element, the visualization search algorithm first searches the model element to be visualized and then its inheritance hierarchy for a visualizer. If no visualizer is found the search algorithm continues at the level of the type of the model element to be visualized. Aspects are collected during the visualization search and merged into a full visualizer.

The search conducted by the visualization algorithm when applied to *Larry* in the bottom center part of the diagram is indicated by the dashed arrows in Fig. 13. The algorithm first searches for a visualizer starting at *Larry*. As no such visualizer is attached to *Larry*, the latter's type - *Individual* - is searched for a visualizer. Since *Individual* provides an aspect for J_A, the search algorithm collects this aspect and continues by visiting the supertypes of *Individual*. As *BusinessActor* defines an aspect for J_B, the search algorithm also collects this aspect and continues the search at *BusinessInternalActiveStructureElement*, the supertype of *BusinessActor*. *BusinessInternalActiveStructureElement* has a visualizer attached which defines the join points J_A and J_B. The aspects are merged into this visualizer resulting in the box displaying *Larry* in the bottom center of the diagram with an"*I*" in the upper left corner and a stick man in the upper right corner. In addition to this *box notation* in the bottom center of the diagram, the *icon* visualizer is shown on the right and the predefined notation is shown on the left.

The visualization of each model element can be toggled between all user-defined notations and the predefined notation while modeling. The visualizer concept also supports visualizations in the form of tables, forms, text and diagrammatic formats (cf. Fig. 15). It also enables dynamic choices because a modeler can decide which format and notation a part of the model should be displayed in, allowing the format and notation to be used that best fits the current task at hand.

Although this visualization approach cannot entirely prevent counter-intuitive overriding of base visualizations, its aspect-oriented features encourage conservative notation extensions, i. e., extensions that retain base visualizations by adding features to them at defined extension points, rather than indiscriminately overriding them. The deep visualization approach therefore supports the monotonic visualization extension approach foreseen by the ArchiMate designers.

7 Further Potential Benefits of Deep Modeling

The previous two sections focused on describing the advantages of deep modeling from the perspective of the requirements and vision explicitly outlined by the ArchiMate designers. The goal of the discussion in those sections was not to question the assumptions and decisions made in defining the modeling experience ArchiMate offers to end users, but to show that the intended user experience could be supported in a simpler and more intuitive way. In this section, however, we go beyond the explicit vision laid out in the ArchiMate standard and show how the users' modeling experience could be further enhanced by deep modeling in ways not envisaged by the authors of the ArchiMate standard (The Open Group 2017a).

7.1 Explicitly Modeling Ontological Classification

The ArchiMate standard states that "*The ArchiMate language intentionally does not support a difference between types and instances*". In other words, since it sees no reason to distinguish between types and instances ArchiMate sees no need to represent ontological classification in user-defined models. While this may appear justifiable in terms of keeping the language simple, it explicitly goes against some of the subprinciples of conceptual integrity such as the principle of propriety (do not restrict what is inherent) and the Maxim of Manner (avoid ambiguity). Not only do many papers on multi-level modeling present examples from the enterprise modeling domain (Atkinson and Kühne 2001; de Lara and Guerra 2010; Jordan et al. 2014), a comprehensive study by de Lara et al. (2014) on the use of the type-instance pattern[2] in modeling repositories shows that one of the applications where this pattern is most commonly used is the business process/enterprise modeling area.

In (Frank 2002), Frank lays out a fundamental set of requirements that EAM modeling environments should ideally fulfill based on a careful consideration of needs and practices in EAM projects. Two of his so-called technical "Requirements for Modeling Environments" essentially call for multi-level modeling and the ability to support the editing of all ingredients of a deep modeling language. Quoting from (Frank 2002):

- *Requirement TR1: A tool environment for enterprise modeling should include a metamodel editor for specifying and modifying metamodels.*
- *Requirement TR2: A metamodel editor should efficiently support the creation of a model editor from a metamodel. This includes the implementation of the abstract syntax and semantics as well as the additional definition of the concrete syntax.*

[2] The type-instance pattern is a workaround technique for representing ontological classification in two-level modeling frameworks.

In (Frank 2014) and (Frank 2016), Frank furthermore proposes a prototype multi-level meta modeling language - Flexible Executable Multi-level Modelling Language (FMMLx) - to support the explicit and level-agnostic representation of ontological classification in enterprise architecture models. FMMLx was implemented by extending the XCore metamodel of the Xmodeler tool (Clark and Willans 2014). As discussed in Sect. 2, Frank's requirements reinforce the argument for an explicit and specially designed ontological meta-metamodel ensuring the metamodels describing the languages for the different viewpoints are "soft" and can be created/edited by users. When new view types, and thus view languages, are needed they can be added by simply defining new metamodels.

Some of the extra capabilities that explicit support for modeling ontological classification would offer ArchiMate modelers are shown in Fig. 14. It shows what a deep version of the "ArchiSurance Contact Center" model from Fig. 7 could look like. The left hand side of the diagram (a) shows the traditional "two-level" version from Fig. 7 while the right hand side (b) shows an alternative deep version which displays potency values and uses the well-known UML colon-notation to show the type of a model element.

The top part of Fig. 14 (b) shows the type information from the original version while the bottom part shows the instance information. The essential difference is that the "fulfills role" is now represented by the ontological classification relationship and a new *Contact Center* class has been added, of which the *ArchiSurance Contact Center* is defined to be an instance.

At first sight, the deep version might appear to offer few advantages over the single-level version and may even appear even a bit more verbose (i. e., requiring more modeling elements). However, the enhanced conceptual integrity more than justifies the addition of a single new modeling element.

First, the deep model leaves much less room for misunderstanding than the single-level version because the types and instances are clearly separated

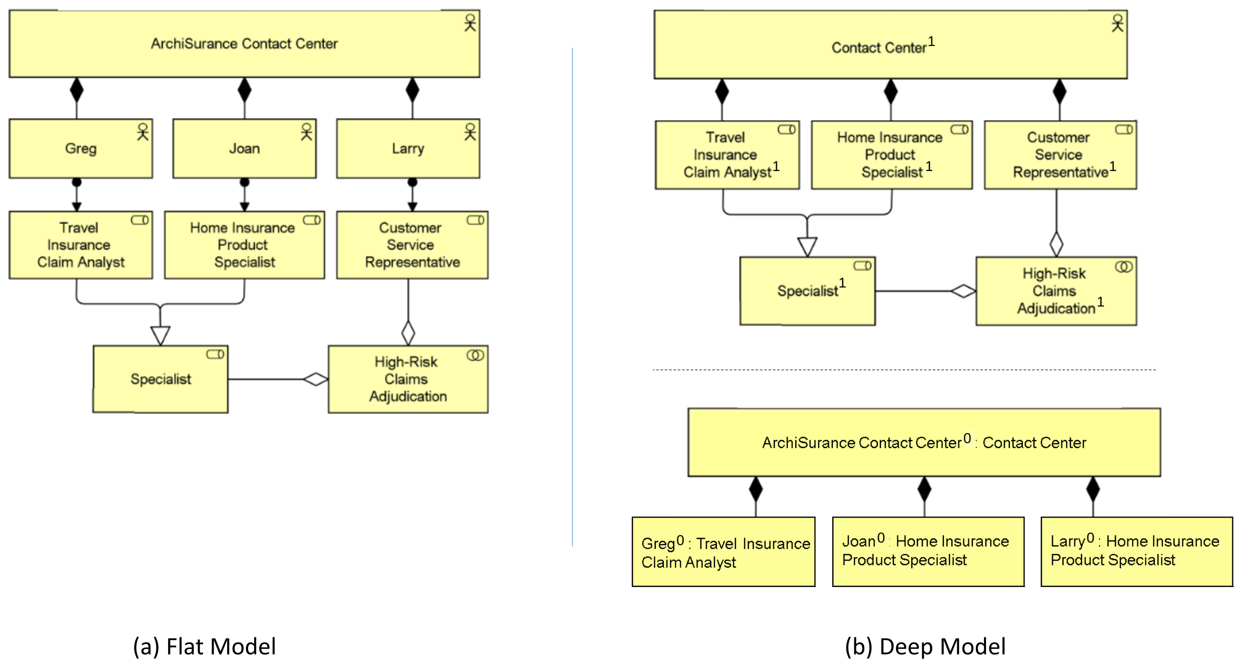

ArchiSurance Contact Center
Greg
Joan
Larry
Travel Insurance Claim Analyst
Home Insurance Product Specialist
Customer Service Representative
Specialist
High-Risk Claims Adjudication
(a) Flat Model
Contact Center^1
Travel Insurance Claim Analyst^1
Home Insurance Product Specialist^1
Customer Service Representative^1
Specialist^1
High-Risk Claims Adjudication^1
ArchiSurance Contact Center^0 : Contact Center
Greg^0 : Travel Insurance Claim Analyst
Joan^0 : Home Insurance Product Specialist
Larry^0 : Home Insurance Product Specialist
(b) Deep Model

Figure 14: Flat versus Deep Contact Center Models

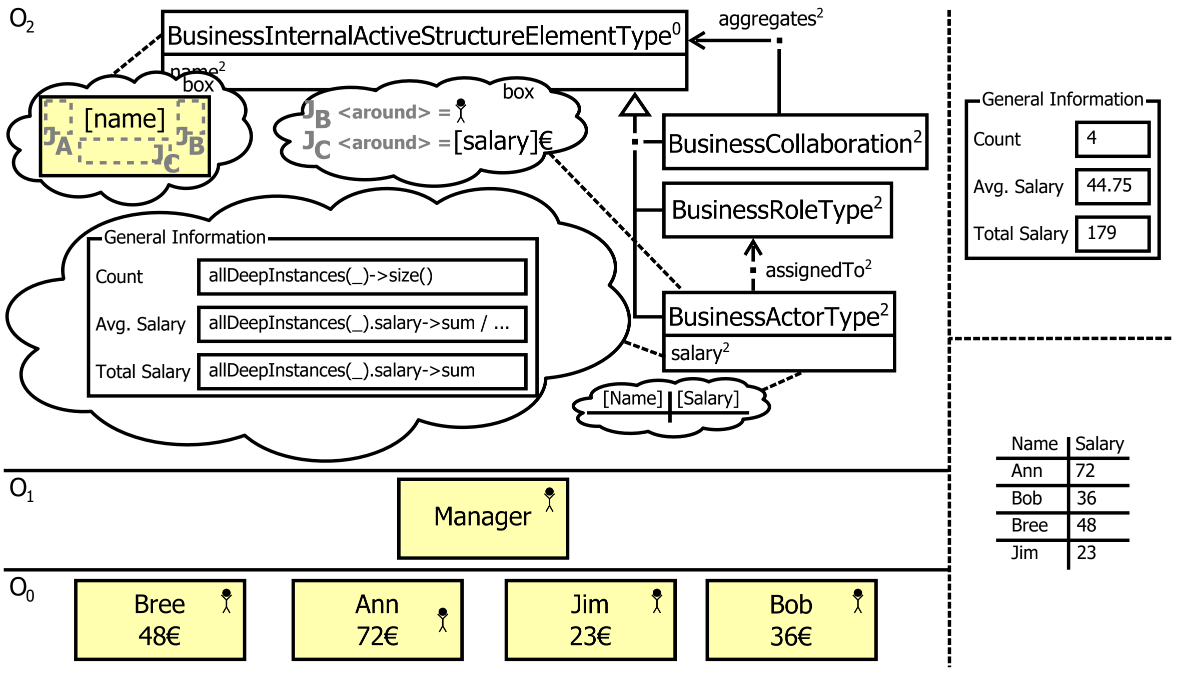
and labeled explicitly as types or instances (a *clarity* benefit). While "*Greg*" may unambiguously refer to an instance, it is much less clear whether the potential alternative "*Manager*" refers to "Manager" the type, or to a single "Manager", i. e., an individual performing the role of a manager. Arguably, if the latter meaning is intended the use of "a Manager" (rather than "Manager") would be more appropriate, but it is understandable that ArchiMate did not chose this option.

Second, *Contact Center* in the top ontological level in Fig. 14 (b) captures the notion of a type abstraction whereas *ArchiSurance Contact Center* at the bottom level captures the notation of a specific incarnation of this abstraction. Clearly, *Greg*, *Joan* and *Larry* are associated with such a specific incarnation, not with its Platonic idea (as suggested by Fig. 14 (b)) (a *separation of concerns* benefit). In general, it is highly useful to separate common information and constraints that are applicable to all instances (which may be a large number over the lifetime of the system and can be applied to them without duplication or change), from specific scenarios in order to promote reuse and avoid the confounding of particular circumstances with universal rules.

Third, type-level descriptions of the constraints and properties that all business roles in the diagram should have, together with the separation of concerns provided by the explicit levels for instances and types, reduces the scope for modelers to introduce errors. This benefit would be even greater if ArchiMate supported attributes and multiplicities, but even with the current modeling features, the ability to define allowed patterns at the type level reduces the error proneness of modelers using ArchiMate (a *quality* benefit).

Frank views the lack of this kind of support as one of the greatest weaknesses of ArchiMate. In (Frank 2016), he states that:

Apart from the unusual conceptualization, the downside of this kind of flexibility is obvious: ArchiMate allows for creating models that are wrong in the sense that they are counter-factual,


O_2
$\text{BusinessInternalActiveStructureElementType}^0$
name^2
box
[name]
J_A
J_B
J_C
aggregates^2
box
J_B <around> =
J_C <around> = [salary]€
$\text{BusinessCollaboration}^2$
$\text{BusinessRoleType}^2$
assignedTo^2
$\text{BusinessActorType}^2$
salary^2
[Name] [Salary]
General Information
Count
allDeepInstances(_)->size()
Avg. Salary
allDeepInstances(_).salary->sum / ...
Total Salary
allDeepInstances(_).salary->sum
O_1
Manager
O_0
Bree 48€
Ann 72€
Jim 23€
Bob 36€

General Information

Count	4
Avg. Salary	44.75
Total Salary	179

Name	Salary
Ann	72
Bob	36
Bree	48
Jim	23

Figure 15: Multi-format and Notation Views of Model Content

e. g., an ERP system could be modeled as being part of a DBMS, or that do not make any sense, because properties were added that are meaningless.

7.2 Multi Format, Multi Notation, Projective Editing

With the possible exception of the profile mechanism which may introduce additional visualization types, ArchiMate only supports a graph-based concrete syntax. However, the ArchiMate standard itself points out that “it should be possible to visualize the same model in different ways, tailored towards specific stakeholders with specific information requirements”. Since many stakeholders in enterprise architecture are business users without experience of formal languages, text-based, table-based and form-based visualizations should also be supported (Gerbig 2017).

Fig. 15 shows how a deep modeling tool - in this case the Melanee tool (Atkinson and Gerbig 2016) - can be naturally extended to support multi-format modeling (Gerbig 2017) such that form- and table-based presentations of model content can be displayed next to a graph-based model. The metamodel at O_2 is extended by adding one deep attribute, *salary*^{2}, to *BusinessActorType*^{2} for demonstration purposes. An additional join point is added to the diagrammatic box notation definition of *BusinessInternalActiveStructureElementType*^{0} to display such additional information as *salary* figures. The visualizer of *BusinessActorType*^{2} provides two aspects to this visualizer, one for displaying the stick man in the upper right and one for displaying the *salary* below the *name* attribute.

In addition to the diagrammatic visualizer a form and table visualizer are defined on *BusinessActorType*^{2} to enable so-called dashboards. The form visualizer shows the total number of all business actors (*Count*), the average salary of all business actors (*Avg. Salary*) and the total salary of all business actors (*Total Salary*) calculated through OCL expressions. The table visualization gives a condensed overview of all business actor instances, providing information about their names and salaries.

The demonstrated capabilities provide the basis for supporting the notion of dashboards described by Frank (2016) as the ultimate goal of EAM approaches - namely, the notion of concise, integrated, information summaries, providing stakeholders (primarily managers) with information across all abstraction and classification levels in their preferred format and notation.

Deep modeling environments such as Melanee and (FMMLx) essentially achieve their flexibility by making what the users sees and interacts with as malleable as possible rather than hard-coded. This principle of modeling rather than hard-coding choices and behavior can be extended to all elements that a user sees when interacting with a modeling tool, including ultimately the whole environment in which models exist and are edited, including menus, toolbars, views, notations, formats, model elements, modeling layout environment view and editor arrangement, the modeling palette available to the user and the property sheet. AtomPM (Syriani et al. 2013) is another example from the domain of metacase tools that has taken the "modeling over coding" principle to heart as almost all its components are entirely modeled and can thus be comparatively easily adapted to different requirements. By making as many features as possible "soft" and configurable, the whole EAM environment used by stakeholders can be configured on a view-by-view basis, thereby supporting the overall vision of viewpoint engineering promoted by ArchiMate and other EAM approaches.

8 Conclusion

In recent years there has been increasing interest in the potential benefits of multi-level modeling in a variety of domains, and the technology has been used successfully in industry for a number of tasks (Aschauer et al. 2009; Igamberdiev et al. 2018). However, deep modeling concepts are still not being applied in earnest in the enterprise architecture modeling industry, despite the fact that existing EA models exhibits some of the most frequent uses of the type-instance pattern (de Lara et al. 2014), an approach that is used as a workaround for deep modeling in environments where there is no explicit support for ontological classification. This is a lamentable situation since the need to use workarounds directly conflicts with the goal of maximizing conceptual integrity. As explained in this article, this need could easily be reduced by employing fully-fledged, multi-level modeling features.

One of the reasons for the lack of adoption of deep modeling in the EAM domain may be that current literature arguing the case for deep modeling in EAM does not present the benefits in the context of existing EAM standards. This article has thus aimed to demonstrate the benefits that deep modeling can bring to a concrete and widely used EAM standard, the ArchiMate approach. We fully embrace the notion of conceptual integrity identified by Lankhorst (2013) and Lankhorst et al. (2010) as the underlying language design principle for ArchiMate and support the view that any introduction of further features must be justified by benefits. The designers of any language must therefore make a fundamental trade-off between the expressiveness of a language in terms of what users can model and the complexity of a language in terms of how complicated it is to learn and apply.

Our contribution is therefore twofold: First, we investigated how the modeling experience explicitly described and envisaged by the ArchiMate standard (The Open Group 2017a) could be better supported and described. To this end, we deliberately restricted ourselves to using deep modeling techniques to support this experience, without enhancing it. Second, in Sect. 7, we outlined additional potential benefits of deep modeling that go beyond the features and capabilities explicitly envisioned by the designers of ArchiMate. As a result, we believe we have presented cogent arguments as to why ArchiMate's current realization of its targeted feature set could be improved by using deep modeling techniques.

We have shown that deep modeling can both –

- reducing the complexity involved in understanding and using the ArchiMate language as currently defined in the standard.

- enhance the expressiveness of the language with minimal additional complexity.

We therefore hope that this article will be helpful not only to the ArchiMate designers in terms of suggesting potential concrete enhancements for future versions, but also to the EAM community in general by helping to reinforce the message that deep modeling approaches and tools can offer benefits in EAM projects.

References

Aschauer T., Dauenhauer G., Pree W. (2009) Multi-level Modeling for Industrial Automation Systems. In: Proceedings of EUROMICRO 2009: 35th EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE, pp. 490–496

Atkinson C., Gerbig R. (2016) Flexible Deep Modeling with Melanee. In: Modellierung 2016 - Workshopband Vol. 255. Gesellschaft für Informatik e.V., pp. 117–122

Atkinson C., Gerbig R., Fritzsche M. (2013) Modeling Language Extension in the Enterprise Systems Domain. In: 2013 17th IEEE International Enterprise Distributed Object Computing Conference. IEEE, pp. 49–58

Atkinson C., Gerbig R., Fritzsche M. (2015) A multi-level approach to modeling language extension in the Enterprise Systems Domain. In: Information Systems 54, pp. 289–307

Atkinson C., Gerbig R., Kennel B. (2012) Symbiotic general-purpose and domain-specific languages. In: ICSE ’12: Proceedings of the 34th International Conference on Software Engineering, pp. 1269–1272

Atkinson C., Kühne T. (2001) Processes and Products in a Multi-Level Metamodeling Architecture. In: International Journal of Software Engineering and Knowledge Engineering 11(6), pp. 761–783

Atkinson C., Kühne T. (2002) Rearchitecting the UML infrastructure. In: ACM Transactions on Modeling and Computer Simulation 12(4), pp. 290–321

Atkinson C., Kühne T. (2003) Model-Driven Development: A Metamodeling Foundation. In: IEEE Software 20(5), pp. 36–41

Atkinson C., Kühne T. (2007) Reducing Accidental Complexity in Domain Models. In: Software and System Modeling 7(3), pp. 345–359

Brooks Jr. F. P. (1975) The Mythical Man-month: Essays on Software Engineering. Addison-Wesley

Brunelière H., García J., Desfray P., Khelladi D., Hebig R., Bendraou R., Cabot J. (2015) On Lightweight Metamodel Extension to Support Modeling Tools Agility. In: Modelling Foundations and Applications. ECMFA 2015. Lecture Notes in Computer Science Vol. 9153. Springer, pp. 62–74

Carvalho V., Almeida J. (2016) Toward a well-founded theory for multi-level conceptual modeling. In: Software & Systems Modeling 17, pp. 205–231

Clark T., Willans J. (2014) Software Language Engineering with XMF and XModeler In: Computational Linguistics: Concepts, Methodologies, Tools, and Applications IGI, pp. 866–896

de Lara J., Guerra E. (2010) Deep Meta-modelling with MetaDepth. In: Objects, Models, Components, Patterns. TOOLS 2010. Lecture Notes in Computer Science Vol. 6141. Springer, pp. 1–20

de Lara J., Guerra E., Cuadrado J. S. (2014) When and How to Use Multilevel Modelling. In: ACM Transactions on Software Engineering and Methodology 24(2), pp. 1–46

de Lara J., Guerra E., Cuadrado J. S. (2015) Model-driven engineering with domain-specific metamodelling languages. In: Software & Systems Modeling 14(1), pp. 429–459

Eertink H., Janssen W., Luttighuis P. O., Teeuw W., Vissers C. (1999) A Business Process Design Language. In: FM’99 — Formal Methods. FM 1999. Lecture Notes in Computer Science Vol. 1708. Springer, pp. 76–95

Espinazo-Pagán J., Menárguez M., García-Molina J. (2008) Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. In: Model Driven Architecture – Foundations and Applications. ECMDA-FA 2008. Lecture Notes in Computer Science Vol. 5095. Springer, pp. 185–199

Frank U. (2002) Multi-perspective enterprise modeling (MEMO) conceptual framework and modeling languages. In: Proceedings of the 35 Annual Hawaii International Conference on System Sciences. IEEE, pp. 1258–1267

Frank U. (2014) Multilevel Modeling. Toward a New Paradigm of Conceptual Modeling and Information Systems Design. In: Business & Information Systems Engineering 6, pp. 319–337

Frank U. (2016) Designing Models and Systems to Support IT Management: A Case for Multilevel Modeling. In: MULTI 2016 – Multi-Level Modelling. Proceedings of the Workshop in Saint-Malo. Atkinson, Grossmann, and Clark, pp. 3–24

Gerbig R. (2017) Deep, Seamless, Multi-format, Multi-notation Definition and Use of Domain-specific Languages. Dr. Hut Verlag

Gonzalez-Perez C., Henderson-Sellers B. (2007) Modelling software development methodologies: A conceptual foundation. In: Journal of Systems and Software 80(11), pp. 1778–1796

Grice H. P. (1975) Logic and conversation. In: Syntax and semantics 3: Speech arts, pp. 41–58

Igamberdiev M., Grossmann G., Selway M., Stumptner M. (2018) An integrated multi-level modeling approach for industrial-scale data interoperability. In: Software and Systems Modeling 17(1), pp. 269–294

ISO/IEC (1997) RM-ODP. Reference Model for Open Distributed Processing. In: ISO/IEC 10746, ITU-T Rec. X.901-X.904

Jordan A., Selway M., Grossmann G., Mayer W., Stumptner M. (2014) Re-engineering the ISO 15926 Data Model: A Multi-Level Metamodel Perspective. In: Service-Oriented Computing – ICSOC 2013 Workshops. ICSOC 2013. Lecture Notes in Computer Science Vol. 8377. Springer, pp. 248–255

Kolovos D. S., Rose L. M., Matragkas N. D., Paige R. F., Polack F. A., Fernandes K. J. (2010) Constructing and navigating non-invasive model decorations. In: Theory and Practice of Model Transformations. ICMT 2010. Lecture Notes in Computer Science Vol. 6142. Springer, pp. 138–152

Kühne T., Atkinson C., Gerbig R. (2015) A Unifying Approach to Connections for Multi-Level Modeling. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems. IEEE, pp. 216–225

Lankhorst M. M. (2013) Enterprise Architecture at Work: Modeling, Communication, and Analysis. Springer

Lankhorst M. M., Proper H. A., Jonkers H. (2010) The anatomy of the archimate language. In: International Journal of Information System Modeling and Design 1(1), pp. 1–32

Object Management Group (2006) Meta Object Facility (MOF) 2.0 Core Specification

Object Management Group (2007) Unified Modeling Language Infrastructure, Version 2.1.2

Syriani E., Vangheluwe H., Mannadiar R., Hansen C., Van Mierlo S., Ergin H. (2013) AToMPM: A Web-based Modeling Environment. In: CEUR Workshop Proceedings Vol. 1115. Liu et al., pp. 21–25

The Open Group (2017a) ArchiMate 3.0 Specification http://pubs.opengroup.org/architecture/archimate3-doc/toc.html Last Access: 2017-02-24

The Open Group (2017b) ArchiMate 3.0 Specification Launch http://www.opengroup.org/news/press/The-Open-Group-Launches-ArchiMate-3 Last Access: 2017-02-24

The Open Group (2010) TOGAF 9 - The Open Group Architecture Framework Version 9 http://www.bibsonomy.org/bibtex/24bc77b548cf18e9b08955140208cf2a1/amitgoel Last Access: 2010-02-26

Zachman A. (1987) A framework for information systems architecture. In: IBM Systems journal 26(3), pp. 276–292