The Anatomy of the ArchiMate Language

M.M. Lankhorst[1], H.A. Proper[2,3] and H. Jonkers[4]
[1] Novay, Enschede, The Netherlands
[2] Radboud University Nijmegen, Nijmegen, The Netherlands
[3] Capgemini, Utrecht, The Netherlands
[4] BiZZdesign, Enschede, The Netherlands

Abstract

In current business practice, an integrated approach to business and IT is indispensable. In many enterprises, however, such an integrated view of the entire enterprise is still far from reality. To deal with these challenges, an integrated view of the enterprise is needed, enabling impact/change analysis covering all relevant aspects. This need sparked the development of the ArchiMate language, which was developed with the explicit intention of becoming an open standard, and as such has been designed such that it is extendable while still maintaining a clear and orthogonal structure.

This paper is concerned with documenting some of the key design decisions and design principles underlying the ArchiMate language. ArchiMate is designed as an architecture description language (ADL) for enterprise architectures. We will start by discussing the challenges facing the design of an architecture description language. Consequently we discuss the way how the design principles of the ArchiMate language aim to tackle these challenges. We then continue with a discussion of the modelling concepts needed. In this, we make a distinction between concepts needed to model domains in general, the modelling of dynamic systems, and the modelling of enterprise architectures.

Introduction

Applying information technology effectively requires a company to have a clear, integrated vision on the relation between its business and IT. Without such a vision, the IT infrastructure will never adequately support the business, and vice versa, the business will not optimally profit from IT developments. A vast amount of literature has been written on the topic of strategic alignment, underlining the significance of both “soft” and “hard” components of an organisation. Organisational effectiveness is not obtained by local optimisations, but is realised by well-orchestrated interaction of organisational components (Nadler et al., 1992).

In current business practice, an integrated approach to business and IT is therefore indispensable. In many enterprises, however, such an integrated view of the entire

enterprise is still far from reality. This is a major problem, since changes in an enterprise's strategy and business goals have significant consequences within all domains of the enterprise, including organisational structures, business processes, software systems, data management and technical infrastructure (Lankhorst et al., 2005a, Op 't Land et al., 2008). Enterprises find themselves confronted with the need to adjust processes to their environment, open up internal systems and make them transparent to both internal and external parties. To deal with the challenges brought forward by these developments, an integrated view of the enterprise is needed, enabling impact/change analysis covering all relevant aspects.

Consider for example a (business unit of an) enterprise that needs to assess the impact of introducing a new product offering. This introduction may require the definition of additional business processes, hiring extra personnel, changing the supporting applications, and augmenting the technological infrastructure to support the additional load of these applications. Perhaps this may even require a change of the organisational structure. Many stakeholders within and outside the company can be identified, ranging from top-level management to software engineers. Each stakeholder requires specific information presented in an accessible way, to deal with the impact of such wide-ranging developments. It is very difficult to obtain an overview of these changes and their impact on each other, and to provide both decision makers and engineers implementing the changes with the information they need.

To manage the complexity of any large system, be it an enterprise, an organisation, an information system or a software system, an architectural approach is needed. As IEEE Std 1471 (IEEE, 2000) puts it: "*Architecture is the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principle guiding its design and evolution*".

Enterprise architecture is an important instrument in executing a company-wide, integrated strategy (Ross et al., 2006). It is a coherent whole of principles, methods and models that are used in the design and realisation of the enterprise's organisational structure, business processes, information systems, and infrastructure (Bernus et al., 2003). However, in practice these domains are often not approached in an integrated way. Every domain speaks its own language, draws its own models, and uses its own techniques and tools. Communication and decision making across domains is seriously impaired.

To be able to represent "*the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment*", an architecture description language for enterprise architectures is needed. At an enterprise level, it is of the utmost importance to be able to represent the core structures of different aspects of the enterprise, such as business processes, applications and infrastructures, as well as the coherence between these aspects.

As discussed in more detail in (Op 't Land et al., 2008), enterprise architecture is a steering instrument enabling *informed governance*. Important applications of enterprise architecture are therefore the analysis of problems in the current state of an enterprise, determining the desired future state(s), and ensuring that the development projects within transformation programs are indeed on-track with regards to the desired future states. This implies that in enterprise architecture models, coherence and overview are more important than specificity and detail. This also implies the need for more coarse-

grained modelling concepts than the finer grained concepts that can typically be found in modelling languages used at the level of specific development projects, such as e.g. UML (OMG, 2009) and BPMN (OMG, 2008). Although several vendor- or tool-specific solutions to this problem have been around for a while, no open and neutral enterprise modelling language was available. Therefore a new language was needed, leading to the development of ArchiMate. In this paper, we will put ArchiMate on the slab and dissect it layer by layer to show you its anatomy.

Background of the ArchiMate Language

The ArchiMate language was realised as part of a collaborative research project on enterprise architecture, funded partly by the Dutch government and involving several Dutch research institutes, as well as governmental and financial institutions (Lankhorst et al., 2005a). ArchiMate was developed with the explicit intention of becoming an open standard, and as such has been designed such that it is extendable while still maintaining a clear and orthogonal structure. The results of the project in general are described in detail by in (Lankhorst et al., 2005a) as well as several papers (Jonkers et al., 2003, Steen et al., 2004, Jonkers et al., 2004, Lankhorst et al., 2005b, Arbab et al., 2007). An illustrative example of an ArchiMate model is provided in Figure 1.

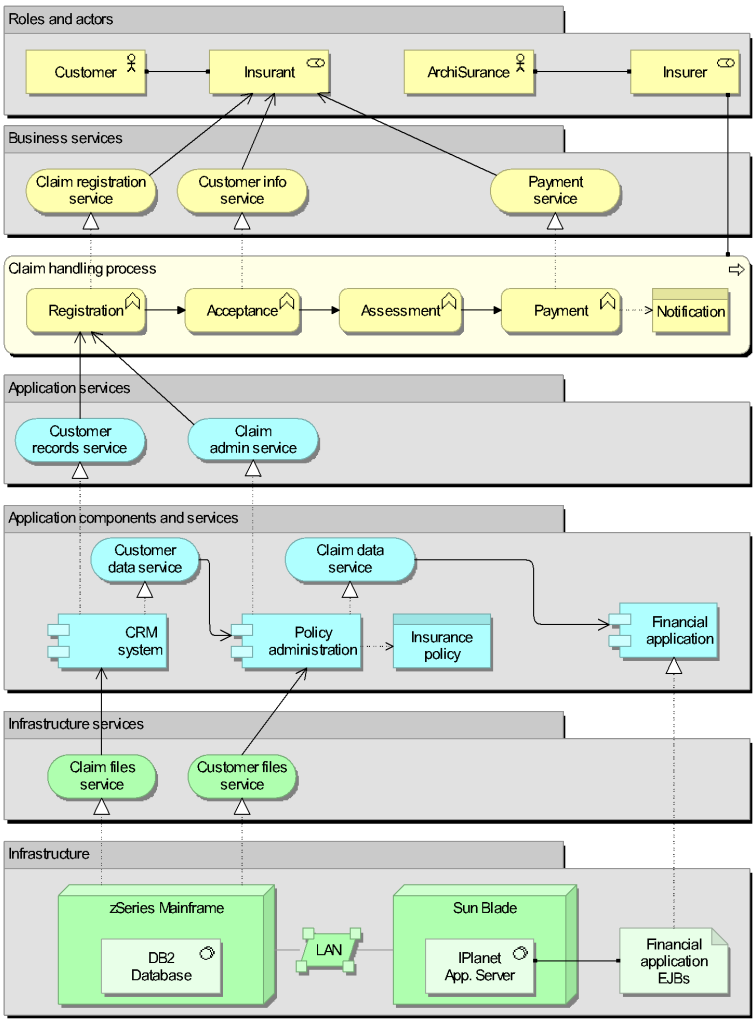


Figure 1. An example ArchiMate model

Meanwhile, the ArchiMate language has been transferred to the Open Group (http://www.opengroup.org/archimate), who adopted it as a technical standard in February 2009 (The Open Group, 2009b). It is slated to become the standard for architectural description accompanying the Open Group's architecture framework TOGAF (The Open Group, 2009a).

The ArchiMate standard consists of the following primary components:

A framework – A conceptual framework consisting of rows (layers) and columns (aspects), which facilitates classification of architectural phenomena. This performs a similar role to the Zachman framework (Zachman, 1987). It defines a theory or "world view" about the way enterprises are structured.

Modelling concepts – A set of modelling concepts allowing for the description of relevant aspects of enterprises at the enterprise level. This set underlies the abstract syntax, focussing on the *concepts* and their meaning, separate from the language constructs in which they are used.

An abstract syntax – This component contains the formal definition of the language in terms of a metamodel. The ArchiMate metamodel defines the characteristics of

each language construct, and its relationships to other language constructs. Besides, the metamodel positions the different language constructs in the cells of the ArchiMate framework and specifies the relationships that may exist not only between constructs but also between cells. This feature distinguishes ArchiMate from both UML (OMG, 2009) and the Zachman framework (Zachman, 1987). In ArchiMate it is therefore possible to explicitly model the dependencies between the different layers, domains and views of the enterprise architecture, which thus becomes a coherent whole instead of a collection of isolated diagrams of different kinds.

The language semantics – This component defines the meaning of each language construct and relation type.

A concrete syntax in terms of a graphical notation – The concrete syntax defines how the language constructs defined in the metamodel are represented graphically. Even though the standard suggests a specific notation, other notations are possible. Changing an icon or other graphical element does not lead to a different language.

A viewpoint mechanism – This corresponds to the idea of diagram types in UML, though it is much more flexible as there is not a strict partitioning of constructs into views. The separation between the abstract and concrete syntaxes of the language as noted above facilitates the creation of different views for different stakeholders.

The first four components (framework, concepts, abstract syntax, and semantics) form the core of the ArchiMate language. The other two components (graphical notation and viewpoint mechanism) are crucial in making the standard usable in practice. The focus of this paper is on documenting some the key design decisions and design principles underlying the language structure, i.e., the first four components.

Until now, publications on the ArchiMate language have focussed mainly on its expressiveness and its applicability. Now that the ArchiMate language has been transferred to the Open Group, the standard is expected to evolve hand-in-hand with the further development of TOGAF. This also provides a trigger to more explicitly discuss the extensible nature of the language. In the remainder of this paper, we therefore start by discussing the challenges facing the design of an architecture description language, while consequently discussing the way in which the design of the ArchiMate aims to tackle these. We then continue with a discussion of the modelling concepts needed to domain models in general, which we then first refine to the modelling of dynamic systems, and finally to the modelling of enterprise architectures.

Requirements on an Architecture Modelling Language

The design of the ArchiMate language was based on an extensive requirements study. In this requirements study, both practical requirements from the client organisations involved in the ArchiMate project (ABN AMRO, ABP Pension Fund, Ordina, and the Dutch Tax and Customs Administration), as well as general requirements on the soundness and other qualities of the language were taken into account (Bosma et al., 2002). Client-specific requirements were elicited by interviewing and studying architectural material from the client organisations. These were then generalised,

combined and augmented with relevant quality requirements from e.g. (Lindland et al., 1994, Krogstie et al., 1995).

The resulting requirements were grouped into the following areas: *modelling*, *analysis*, *visualisation*, *process and tool support*, and *organisational implementation*. The first three types of requirements were a primary source for the design of the language. Below we outline the underlying challenges and describe the resulting 'must haves' for the language.

Modelling

From a modelling perspective, the essential requirements were the following:

Concept coverage – Several domains for grouping concepts have been identified, such as product, process, organisation, information, application, technology (infrastructure, system development, maintenance). The concepts in the language must at least cover the concepts in these domains.

Enterprise level concepts – At an enterprise level, it is important to be able to represent the core elements from the different domains such as product, process, et cetera, as well as the coherence between these aspects. In enterprise architecture models, coherence and overview are more important than specificity and detail. This also implies the need for more coarse grained modelling concepts. As mentioned before, the concepts which can typically be found in modelling language used at the level of specific development projects, such as UML and BPMN, were found to be too fine-grained.

Concept mapping – Organisations and/or individual architects must be able to keep using their own concepts and descriptions in development projects. This requires a mapping from the coarse grained concepts in ArchiMate to the fine-grained concepts used in languages at the project level.

Unambiguous definitions of concepts – The meaning and definition of the modelling concepts offered by the language must be unambiguous. Every concept must be described taking into account: informal description, specialisation, notation, properties, structuring, rules and restrictions and guidelines for use.

Conformance to international standards – The architecture description language must follow, and whenever possible, influence international standards.

Structuring mechanisms – Composition & decomposition, generalisation & specialisation, and aggregation of concepts must be supported.

Abstraction – It must be possible to model relations at different abstraction levels. For example, relations can be formulated between concepts, groups of concepts or different architectural domains.

Consistency – It must be possible to perform consistency checking of architectures.

Tracing of design decisions – It must be possible to register, trace and visualise the requirements, constraints, design decisions and architectural principles that are used in the construction of the architecture.

Extensibility – The language should be easy to maintain and extend, should the need for new concepts arise.

Analysis

The ability to perform various kinds of analyses was also recognised as an important benefit of using architecture models. These benefits also contribute towards the *return on modelling effort* (RoME) of the creation of architectural models. The following demands were therefore also taken into account in designing the modelling language:

Analysis of architectural properties – It must be possible to perform qualitative and quantitative analysis of properties of architectures.

Impact of change analysis – Impact of change analysis must be supported. In general, such an analysis describes or identifies effects that a certain change has on the architecture or on characteristics of the architecture. We identify three types of change analysis:

- The impact of a change in an architectural element on other architectural elements, e.g. what is the impact of the introduction of new products or modifications in products on processes, applications and infrastructure?
- The impact of a change in a characteristic of the architecture on the architecture itself, e.g. what is needed to improve the required throughput by 20%?
- The impact of an event or change in the architecture on the characteristics of the architecture e.g. what are the consequences of a fatal failure in a technological component?

Visualisation

Requirements concerning visualisation are about the way architectural results are presented to stakeholders. Following (IEEE, 2000), the information need of stakeholders is addressed by viewpoints consisting of views and models.

Representation of concepts – The visual representation of concepts must be easily adaptable.

Consistency of presentation – The visual presentation needs to be consistent and unambiguous.

Visualisation independence – Visualisation techniques and solutions should be independent of the actual concepts used in a model, i.e., it should be possible to modify/add the concepts in a model without consequences for visualisation techniques.

Visualisation generation – Automatic generation of visualisations from architecture models must be supported.

Viewpoint definition – A viewpoint definition must

- state the stakeholder(s) it is created for,
- define the concerns covered by the viewpoint, and
- explain how views are created for this viewpoint (in terms of the concepts to be presented and the format of the presentation).

Adaptability of viewpoints – Viewpoints must be adaptable and extensible independent of visualisation techniques.

Viewpoint coverage – ArchiMate has to support often used 'general' viewpoints, i.e., viewpoints for frequently occurring stakeholders.

Several of these requirements imply a clear separation between concepts and their notation(s) in the modelling language. This is different from e.g. UML, in which the notation is tied closely to the individual concepts. Separating these makes it much easier to create architectural views for specific stakeholders, using symbols that these stakeholders are familiar with.

Since this paper is concerned with the structure of the ArchiMate language and not with its notation, most of these requirements on visualisation will not be dealt with here, although we will explain the language symbols in later sections. Visual aspects of enterprise architecture modelling have been addressed in (Lankhorst et al., 2005, chapter 7).

General Quality Criteria

In addition to the specific requirements outlined above, the ArchiMate language also needed to address general quality criteria for modelling and modelling languages. Pioneers in the design of complex systems (Dijkstra, 1968, Brooks Jr., 1987) have described principles to ensure the conceptual integrity of a design: '*It is not enough to learn the elements and rules of combination; one must also learn idiomatic usage, a whole lore of how the elements are combined in practice. Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. (…) Ease of use, then, dictates unity of design, conceptual integrity*' (Brooks Jr., 1987).

Conceptual integrity is the degree to which a design can be understood by a single human mind, despite its complexity. The core idea of conceptual integrity is that any good design exhibits a single, coherent vision, which is easy to understand by others. This allows someone with a limited knowledge and understanding of a model to understand easily yet unknown parts of the model. To ensure conceptual integrity, one can use subordinate design principles such as: do not link what is independent (orthogonality), do not introduce multiple functions that are slightly divergent (generality), do not introduce what is irrelevant (economy; sometimes denoted as parsimony), and do not restrict what is inherent (propriety).

These principles not only apply to the design of architectures, but also to the design of the modelling languages in which these architectures are expressed. To convey this notion of conceptual integrity, the language should help the architect in expressing his design in such a way that it conforms to these design principles. If a language only allows a convoluted way of describing an architecture, the resulting design will never achieve this conceptual integrity. Furthermore, the language itself should exhibit conceptual integrity: similar things should be expressed in a similar way, using a simple set of core concepts that are easy to learn and understand. Cobbling together a language from many different sources (viz. UML) is not the ideal way to achieve this.

The literature on quality requirements for models shows a broad consensus about the general applicability of these heuristics (Lindland et al., 1994, Krogstie et al., 1995, Teeuw & Berg, 1997). Applying these design principles increases the internal quality of a design (Teeuw & Berg, 1997).

Additionally, the quality of an architecture is also determined by its stakeholders: an architecture that is a 'correct' and 'complete' representation of the real-life enterprise that is being modelled, given the objectives of stakeholders, has a high external quality. In short, external quality refers to the fitness for use of a model (Biemans et al., 2001). This implies that the modelling language should also be fit for use by these stakeholders: its expressive power and ease of use should cover their needs. This is something we will address in the following sections.

Fulfilling the Requirements

In this section we start with a discussion of the key design principles used in the construction of the ArchiMate language, together with their motivations as well as their actual impact on the design of the language. This is followed by the introduction of the concept of a stack of metamodels, forming the backbone of the definition of the ArchiMate language.

Key Design Principles

The requirements posed in the previous section were the basis for formulating a set of key design principles for the structure of our language:

The language should be as compact as possible – The most important design restriction on the language was that it was explicitly designed to be as compact as possible, while still being usable for most enterprise architecture related modelling tasks. Many other languages, such as UML, try to accommodate as much as possible all needs of all possible users. In the case of UML, this has resulted in a language specification of nearly 1000 pages that has become extremely hard to implement and difficult to learn. In the interest of simplicity of learning and use, ArchiMate has been limited to the concepts that suffice for modelling the proverbial 80% of practical cases. When it is clear for each of the concepts what its contribution is, the language becomes easier to use and easier to learn (Proper et al., 2005). Furthermore, having a large number of concepts makes it difficult to extend the language in the future, potentially requiring an exponentially rising number of connections to existing concepts. Finally, adding concepts later on is much easier than removing something, since somewhere someone may depend on that specific concept; the continuing growth of UML is witness to this bane of downwards compatibility.

Core concepts should not dependent on a specific framework – Many architecture frameworks are in existence. Therefore, it is not desirable for a general purpose architecture description language to be too dependent on a specific architecture framework. Doing so will also make the language more extendible in the sense that can more easily be adapted to other frameworks.

However, in order to provide a useable architecture description language, the ArchiMate standard does include the definition of a (general) framework as well, to ensure the standard is specific enough to be applicable in practice. Nevertheless, the core concepts are independent of the framework as will be illustrated later.

Concepts should be mapped easily to and from those used in project level languages – To enable traceability from the enterprise level to the project level, a strong relationship should exist between the modelling concepts used at project level and those used in the enterprise architecture. Therefore, the ArchiMate language needed to be set up in such a way that project level modelling concepts be expressed easily in terms of the more general concepts defined in the language (e.g., by specialisation or composition of general concepts).

A Stack of Metamodels

The key challenge in the development of the language metamodel was actually to strike a balance between the specific concepts used by project-level modelling languages on one extreme, and the very general modelling concepts suggested by general systems theory (Beer, 1985). The triangle in Figure 2 illustrates how concepts can be described at different levels of specialisation.

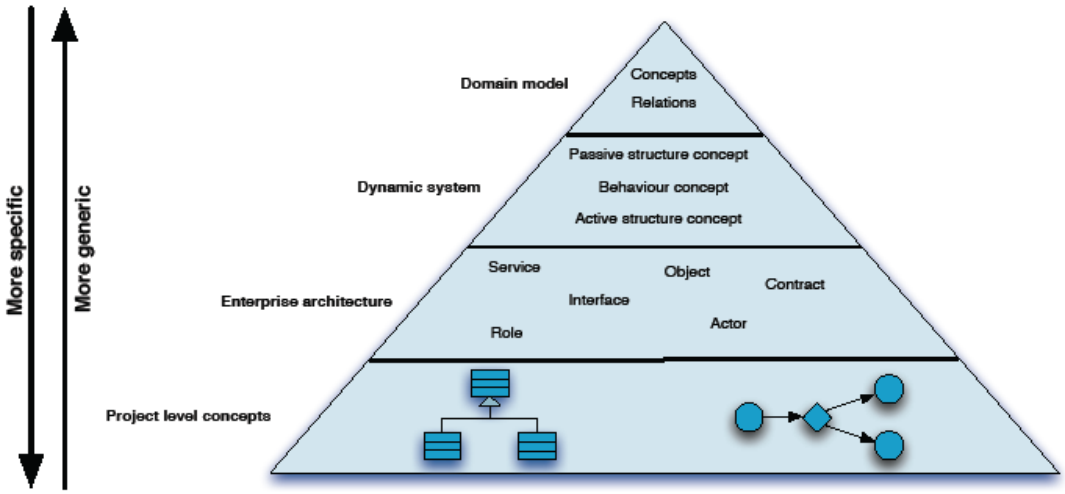


Figure 2. Conceptual hierarchy

To meet this challenge of balancing generality and specificity, the design of the ArchiMate language started from a set of relatively generic concepts (higher up in the triangle) focussing on domain modelling in general. These were then specialised towards the modelling of dynamic systems (at a course grained level), and consequently to enterprise architecture concepts. At the base of the triangle, we find the metamodels of the modelling concepts used by project-level modelling languages such as UML, BPMN, et cetera. The ArchiMate metamodel defines the concepts somewhere between these two extremes. In moving from the top of the triangle to the lower levels, we will each time argue what the utility (towards our general modelling goals) of the added concepts (Proper et al., 2005). This makes the exercise of creating such a stack of metamodels comparable to the creation of a metamodel hierarchy (Falkenberg & Oei, 1994). However, we do not use a branching structure here, and hence not create a hierarchy but merely a stack. When defining the language in this way, it also becomes easier to position and discuss possible extensions of the language in relation to higher level core concepts and/or the specialisations of these at the lower levels.

In this paper we have chosen to use Object-Role Modelling (ORM2 to be precise) (Halpin & Morgan, 2008) as a metamodelling language, since it allows for precise modelling and elaborate verbalisations, making it well suited for the representation of metamodels. Furthermore, due to its elaborate formalisation (Hofstede & Weide, 1993, Hofstede et al., 1993), the graphical ORM models (and any textual constraints) can be regarded as a graphical representations of underlying logical theory. This provides a better foundation for logic reasoning on models and metamodels than given by e.g. the Meta Object Facility (MOF) (OMG, 2006), the metamodelling language underpinning UML.

The two main elements in ORM are entity types, representing 'things' being modeled, and fact types, representing facts about (i.e., relationships between) these things. Entity types are depicted as named rectangles with rounded corners. Fact types (relationship types) are shown as named sequences of role boxes, with the predicate name in or beside the first role of the predicate. Bars over or under one or more role boxes indicate a uniqueness constraint over the(se) role(s). Black dots on the connection between an entity type and a role-box indicate a mandatory role constraint, whereby all instances of the involved entity type are required to play the specific role. A solid arrow from one entity type to another indicates that the first is a (proper) subtype of the second. Each instance of a subtype is also an instance of it supertype. Similarly, dotted arrows represent subset relations on the populations of relationships.

In the remainder of the paper, we discuss the stack of metamodels taking us from the top of the triangle to the level of the ArchiMate metamodel. At each level, we will present a metamodel of the additional modelling concepts provided by this level. Each level also inherits the concepts from the previous level, while also providing specialisations of the existing concepts. Formally, each level l involves the specification of a metamodel $Schema(l)$ and a mapping $Mapping(l)$, where at the base of the stack the mapping is empty: $Mapping(0) = \emptyset$. The total metamodel at each level can be defined recursively as:

$$TotalSchema(0) \equiv Schema(0)$$
$$TotalSchema(n) \equiv TotalSchema(n-1) \cup Schema(n) \cup Mapping(n)$$

As an example metamodel stack, involving two levels, consider Figure 3.

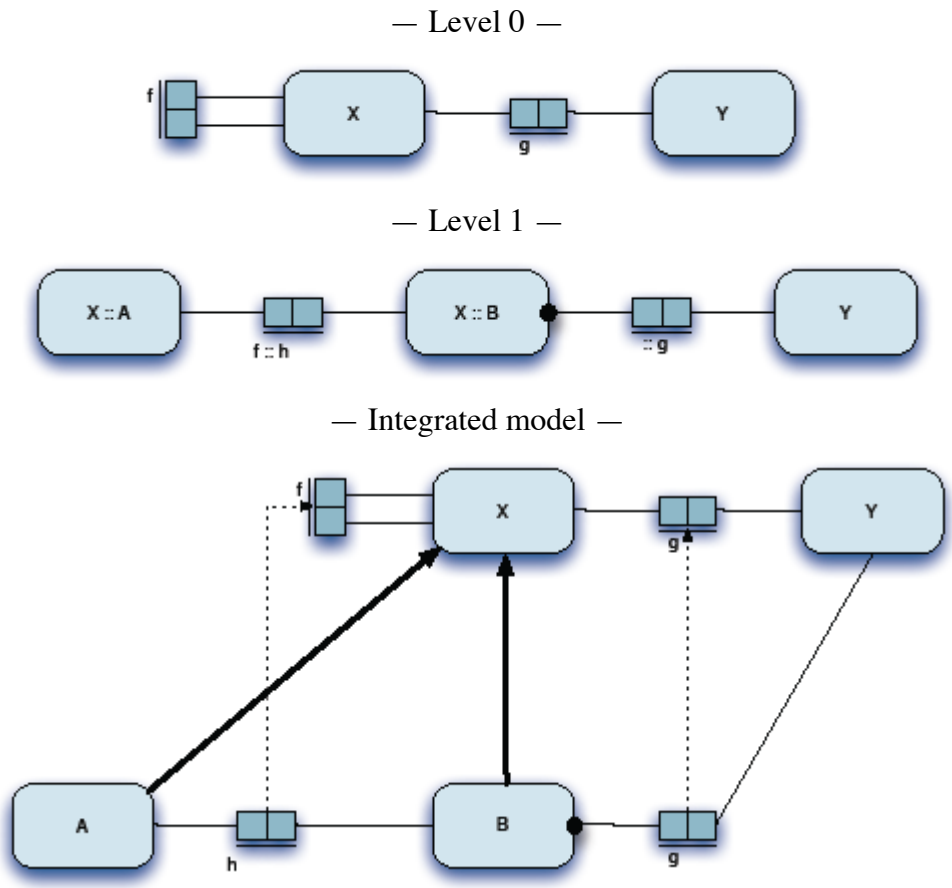


Figure 3. Example metamodel

The mappings between the levels are represented as $a :: b$. As a rule we will required that if $Mapping(n + 1) \square a :: b$, then a must be a type from $TotalSchema(n)$ and b must be a type in $Schema(n + 1)$. Furthermore, as also illustrated in Figure 3, if a and b are both object types, b is subtype of a, while if both are fact types, b is a subset of a. More specifically, in Figure 3, *A* and *B* are a subtype of *X*, while fact type *h* is a subset of fact type *f*.

Sometimes we will want to repeat fact types that already exist between two supertypes for subtypes of these supertypes. This may be needed as a clarification, or to define more specific constraints that hold for the instances of the subtypes participating in the fact type. In this case we will write $:: a$ as a shorthand for $a :: a$. In the example shown in Figure 3 we see how *g* is repeated at level 1, while the mandatory role (the black dot on entity type *B*) requires the instances of subtype *B* to all play a role in fact type *g* (which is not required for all instances of supertype *X*).

It should be noted that, although the triangle in Figure 2, as well as the stack of metamodels to be discussed in the next sections, played an important part in the construction of the current ArchiMate standard, they are not formally part of the standard. Making explicit these intermediate structures, however, will also support the future evolution ensuring the clarity, coherence and orthogonality of the concepts of the language are maintained.

Domain Modelling

In this section we are concerned with the establishment of a metamodel covering a set of modelling concepts that would allow us to model domains in general. We do so by defining three levels as depicted in Figure 4.

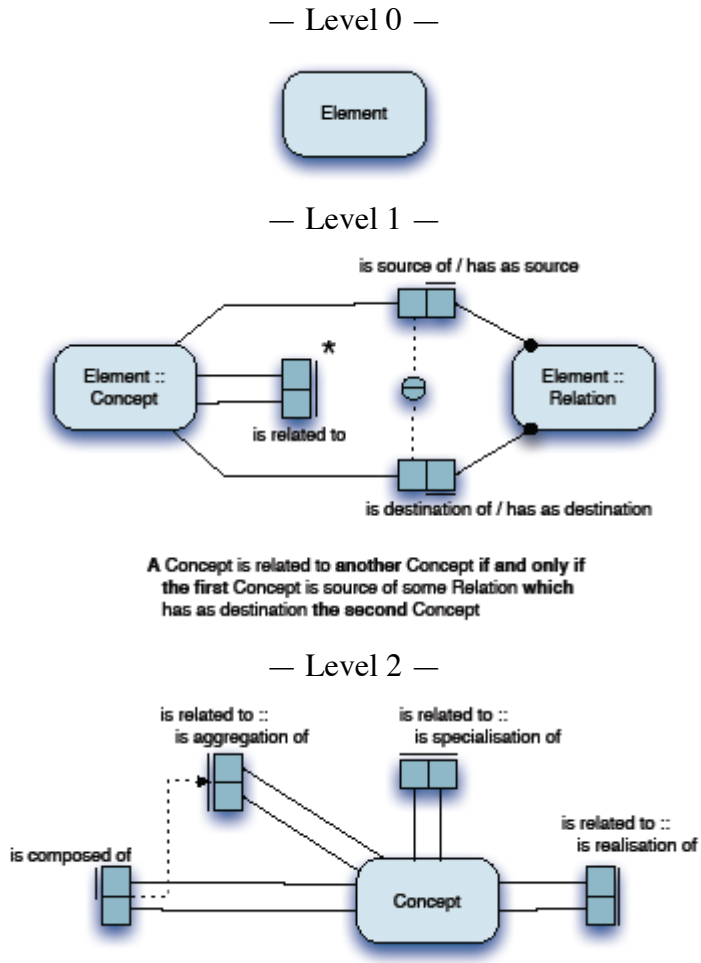


Figure 4. Basic layers: Levels 0, 1 and 2

The first level in Figure 4 shows a metamodel comprising a single modelling concept: *Element*. This constitutes a first important step. The domains we want to model are not just black boxes. We want to discern several elements within each domain (and its environment). On its own, this is of course still highly impractical. For practical reasons we need the ability to identify relations between these elements. This, therefore, leads to the refinement suggested by level two.

At the second level, we identify two kinds of elements: Concepts (graphically represented by 'boxes') and Relations (graphically represented by lines or arrows between the boxes). Concepts are the source of Relations as well as the destination of Relations. In other words, Concepts can be related by way of a Relation. This is abbreviated by the derived (as marked by the asterisk) fact type is related to. The definition of this derived fact type is provided in the style of SBVR (SBVR Team, 2006).

The domains we are interested in tend to be large and complex. To be able to harness this complexity we need special relationships between *Concepts* which provide us with abstraction, aggregation and specialisation mechanisms. This leads to three specialisations of the *is related to* fact type: *is realisation of* representing the fact that one concept may be the physical realisation of another concept, *is specialisation of* dealing with specialisation of concepts, and *is aggregation of* concerned with the aggregation of multiple concepts into a complex concept. A special class of aggregations are compositions, as signified by the is composition of fact type.

Distinguishing Extensional and Intentional Perspectives

Next, we make a distinction between the real-world things we want to model and the intentions we have with these things. These real-world entities are modelled with so-called *extensional* concepts, which provide an objective view on the world, i.e., they describe what is 'out there'. In addition, we discern (in line with e.g. the *why* perspective in the Zachman framework), a subjective, *intentional*[1] perspective, to capture the intentions that stakeholders have with these real-world entities and hence with the extensional concepts. This leads to a separation between extensional and intentional concepts, as shown in Figure 5.

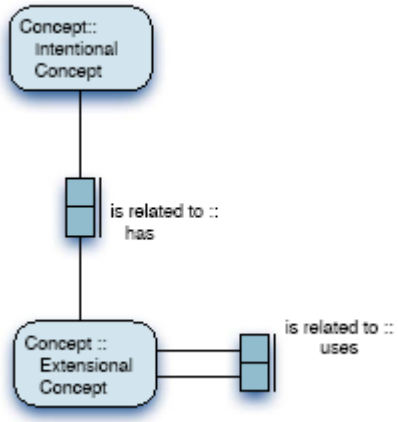


Figure 5. Level 3 – Intensional and extensional concepts

Modelling Dynamic Systems

Based on the foundation established in the previous sections, we will now describe general concepts for modelling dynamic systems. In this context, a dynamic system is any (discrete-event) system in which one or more subjects (actors or agents) display certain behaviour, using one or more objects. Examples of dynamic systems are business systems, information systems, application systems, and technical systems.

The concepts we define here should be suitable for modelling a variety of information-intensive organisations. If desired, they can be further specialised or composed to form concepts tailored towards a more specific context. The structure of the metamodel

[1] Note that this is not an intensional perspective!

is consistent with the structure of several common architectural frameworks or methods, such as TOGAF (The Open Group, 2009a), the Zachman framework (Zachman, 1987), and DYA (Wagter et al., 2005), and the architectural practice within user organisations. We have also re-used concepts from existing languages (such as UML and several business process modelling languages) as much as possible, although we use many of these concepts in a more abstract sense.

In this section, we gradually extend the set of concepts, using three more or less orthogonal aspects or 'dimensions'; we distinguish:

1. the aspects *active structure*, *behaviour* and *passive structure*,
2. an *internal* and an *external* view, and
3. an *individual* and a *collective* view.

In the following three subsections, we introduce the concepts that follow from this. We also motivate the relevant design decisions (e.g., not all the possible combinations of the aspects lead to new concepts).

Active Structure, Behaviour and Passive Structure

Next, we distinguish *active structure concepts*, *behavioural concepts* and *passive structure concepts* within the extensional view. These three classes have been inspired by structures from natural language. When formulating sentences concerning the behaviour of a dynamic system, concepts will play different roles in the sentences produced. In addition to the role of a *proposition* dealing with some activity in the dynamic system (selling, reporting, weighing, et cetera), two other important roles are the role of *agens* and the role of *patiens* (note, further refinements of these roles do indeed exist, but we regard these two as the primary roles in addition to the *proposition*). The *agens* role (the active structure) refers to the concept which is regarded as executing the activity, while the *patiens* role (the passive structure) refers to the concept regarded as undergoing/experiencing the activity.

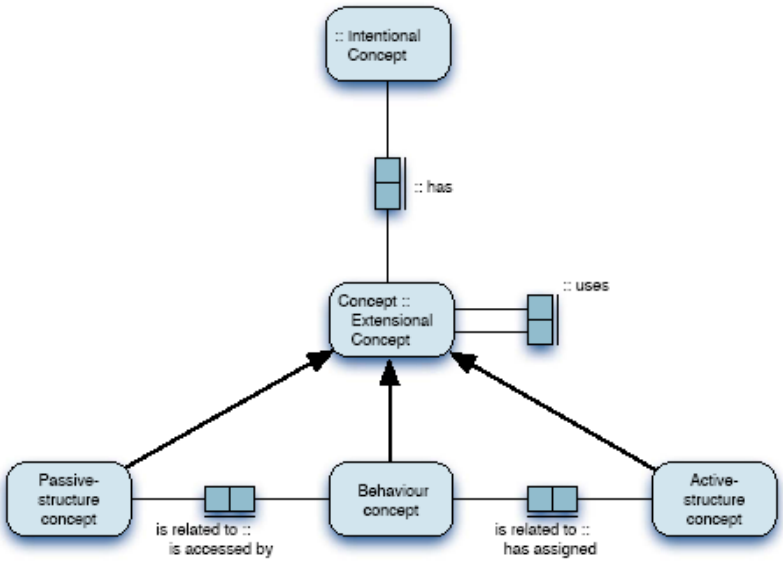


Figure 6. Level 4 – Passive structure, behaviour and active structure

Active structure concepts are concepts concerned with the execution of behaviour; e.g., (human) actors, software applications or devices that display actual behaviour. *Behaviour concepts* represent the actual behaviour, i.e., the processes and activities that are performed. The active structure concepts can be *assigned to* behaviour concepts, to show who (or what) performs the behaviour. *Passive structure concepts* are the concepts upon which behaviour is performed. In the domain that we consider, these are usually information or data objects, but they may also be used to represent physical objects. Even more, concepts may play the role of active structure in one occasion, but the role of passive structure in another. For example, an actor may execute a process, but that same actor may be managed by another one. This is shown in Figure 6.

In the standard ArchiMate notation, the convention is that active and passive structure concepts are represented by rectangular shapes, while behavioural concepts are represented by round or oval shapes, or rectangular shapes with rounded corners.

Meaning, Value and Reason

Mirroring the structure of the previous section, we subdivide the intentional perspective into three different concepts. We identify the *Meaning* concept to express the meaning attached by stakeholders to extensional concepts and in particular to passive structure concepts. The *Value* concept expresses the value exchange or addition that may be associated with concepts, e.g. with the performance of the behaviour or with passive structure concepts. The *Reason* concept, expresses the rationale underlying the design of concepts. This leads to the refined metamodel as is shown in the top half of Figure 7.

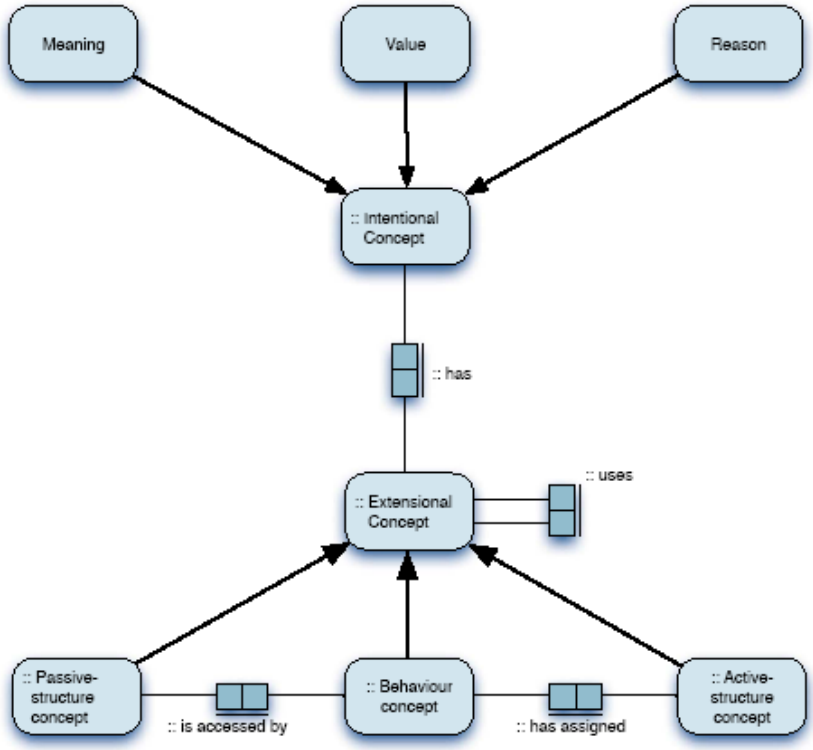


Figure 7. Level 4 – Meaning, value and reason added

Unlike *Meaning* and *Value*, *Reason* is not (yet) a concept that is available to the users of the ArchiMate language; it rather remains 'hidden' under the surface. However, it is an important extension point for future augmentation of ArchiMate with requirements modelling concepts.

Internal and External Perspective

In the ArchiMate language a distinction is made between an *external* perspective and an *internal* perspective on a system. When looking at the behaviour aspect, these perspectives reflect the principles of service orientation. The *Service* concept represents a unit of essential functionality that a system exposes to its environment. This leads to the extension as depicted in Figure 8.

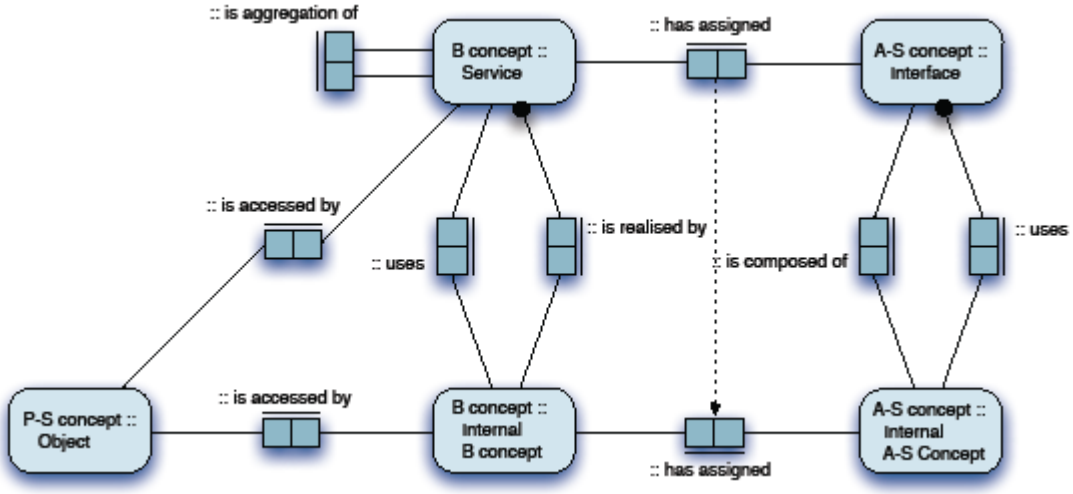


Figure 8. Level 5 – Internal and external concepts

A *Service* is accessible through an *Interface*, which constitutes the external view on the active structural concept. An interface is a (physical or logical) location where the functionality of a service is exposed to the environment. When a service has assigned an interface, then this assignment must be mirrored by the assignment of relevant internal active structure concepts to the internal behaviour concepts involved in the realisation of the service (the dotted arrow between the two has assigned fact types).

For the passive structural concepts, the internal and external views are collapsed: objects may be accessed by both internal behaviour concepts and services. This actually suggests a possible extension of the language, where a distinction is made between objects that are internally accessible and those that are externally accessible. Note that with the *Object* concept we do not only refer to an informational abstraction of objects. For example, at the business level, the objects are likely to include physical objects as well.

Individual and Collective Behaviour

Going one level deeper in the structure of the language, we distinguish between the *individual* behaviour, performed by a single active structure concept, and the *collective* behaviour performed by multiple active structure concepts which collaborate. This leads to the refinements shown in Figure 9.

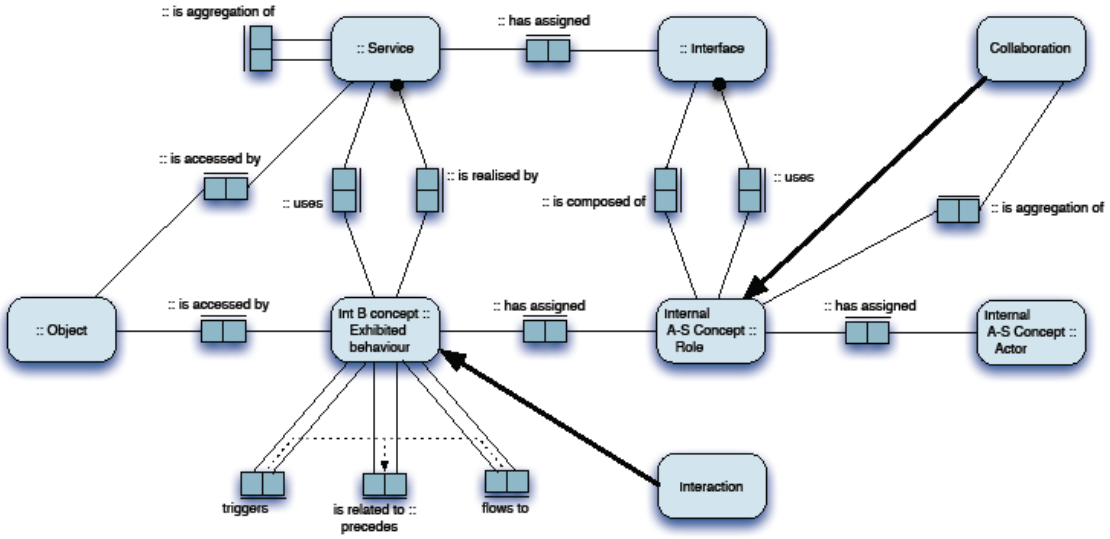


Figure 9. Level 6 – Individual and collective concepts

In describing individual and/or collective behaviour in more detail, the internal behaviour concept needs refinement in terms of temporal ordering of *Exhibited behaviour*. This leads to the *precedes* fact type and its subsets: *triggers* (for activities) and *flows to* (for information processing). A further refinement needed is the distinction between *Role* and *Actor* as active structure concepts. An *Actor* represents an essential identity that can ultimately be regarded as executing the behaviour, e.g. an insurance company, a mainframe, a person, et cetera. The actual execution is taken to occur in the context of a *Role* played by an *Actor*; a *Role* represents the responsibility for performing the behaviour. Needless to say that actors can fulfil multiple roles.

A collective of co-operating *Roles* is modelled by the *Collaboration* concept: a (possibly temporary) aggregation of two or more active structure concepts, working together to perform some collective behaviour. A *Collaboration* is defined as a specialisation of a *Role*. The collective behaviour itself is modelled by the *Interaction* concept, which is defined as a specialisation of the *Exhibited behaviour* concept.

In the current version of the language, the distinction between individual and collective behaviour is only made for the internal view: both individual internal behaviour concepts and interactions may realise services. The reason for this is that we focus on models from the perspective of a single system (i.e., one system realises services which may be used by other systems). Also for passive structural aspects, the individual and collective views are collapsed.

An extension of the language is conceivable in which a collective external view is also explicitly described. For example, the *Transaction* concept, which is used in some modelling languages to express the mutual exchange of services or value items (see, e.g., (Dietz, 2006)), could be considered an external collective behaviour concept.

Modelling Enterprise Architectures

In this section we further extend the metamodel stack to arrive at the actual ArchiMate language. Two steps remain. The first step involves the introduction of an architecture

framework allowing us to consider enterprises as a layered set of systems. The final step is to refine the metamodels to the specific needs of each of these layers.

Multiple Layers of Systems

As a common denominator of the architecture frameworks in use by the client organisations participating in the ArchiMate project, as well as a number of standard frameworks used in the industry, with of course TOGAF as a prominent example, a framework was created involving three layers of systems:

Business layer This layer is concerned with the products and services offers to external customers, which are realised in the organisation by business processes performed by business actors and roles.

Application layer This layer supports the business layer with application services which are realised by (software) application components.

Technology layer This layer offers infrastructural services (e.g., processing, storage and communication services) needed to run applications, realised by computer and communication hardware and system software.

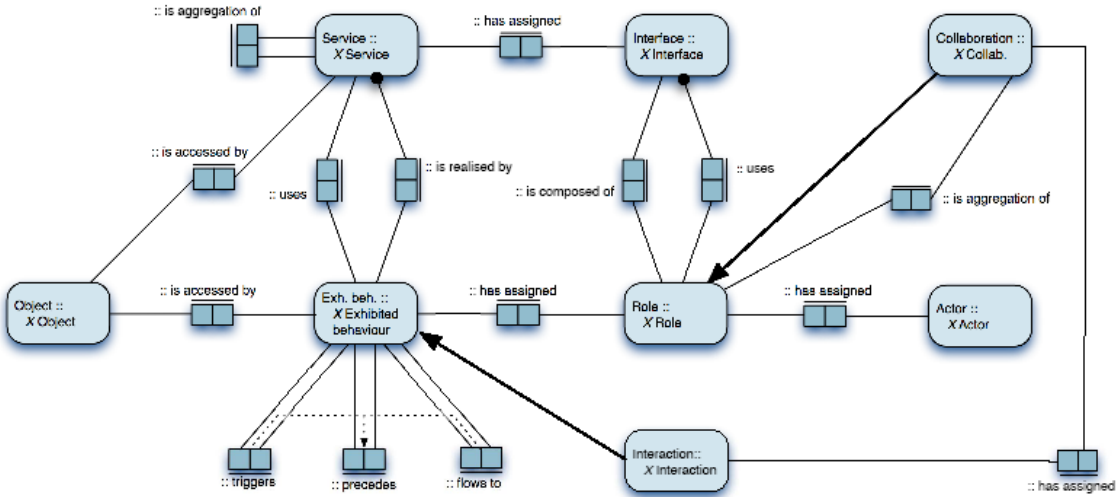


Figure 10. Level 7 – Generic layer

Since each of these layers involves a dynamic system, the metamodel at level 7 comprises three copies of the fragment depicted in Figure 10 for *Business*, *Application* and *Technology* respectively (each time replacing the X in Figure 10). These fragments, however, need to be connected as well, therefore for each of the $\langle X, Y \rangle$ two combinations: $\langle Business, Application \rangle$ and $\langle Application, Technology \rangle$ the fragments shown in Figure 11 should be added.

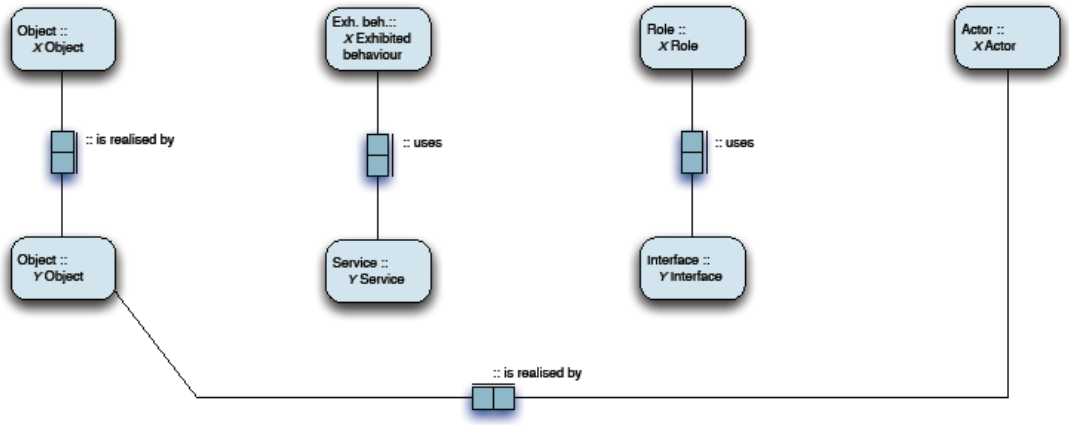


Figure 11. Level 7 – Inter-layer links

As can be seen in Figure 11, we distinguish two flavours of inter-layer relationships:

- Relationships related to the use of services, providing a rather loose coupling between layers. These are expressed by *used by* relations, between a service and an internal behaviour concept, and between an interface and an active structure concept.
- Implementation relationships, providing a tighter coupling between layers. These are expressed by *realisation* relations.

Although, in principle, both types of relations can be used for all three aspects, *used by* relations are mainly used to link layers for the active structure and behaviour aspects, while *realisation* relations are mainly used to link layers for the passive structure aspect.

Layer-Specific Refinements

Given the focus of each of the layers, further refinements are needed to better cater for the specific needs of the respective layers.

Business layer

For the business layer, as shown in Figure 12, the concepts of *Contract* and *Product* have been introduced. At the business level, *Business services* may be aggregated to form *Products*, which are treated as (complex) services. A *Business service* offers a certain *Value* (economic or otherwise) to its (prospective) users, which provides the motivation for the service's existence. For the external users, only this external functionality and value, together with non-functional aspects such as the quality of service, costs, et cetera, are of relevance. These can be specified in a *Contract* or Service Level Agreement (SLA). This leads to the situation as depicted in Figure 12. The concepts of *Meaning* and *Value* (see Figure 7) have been repeated to stress the fact that they specifically play a role in the business layer.

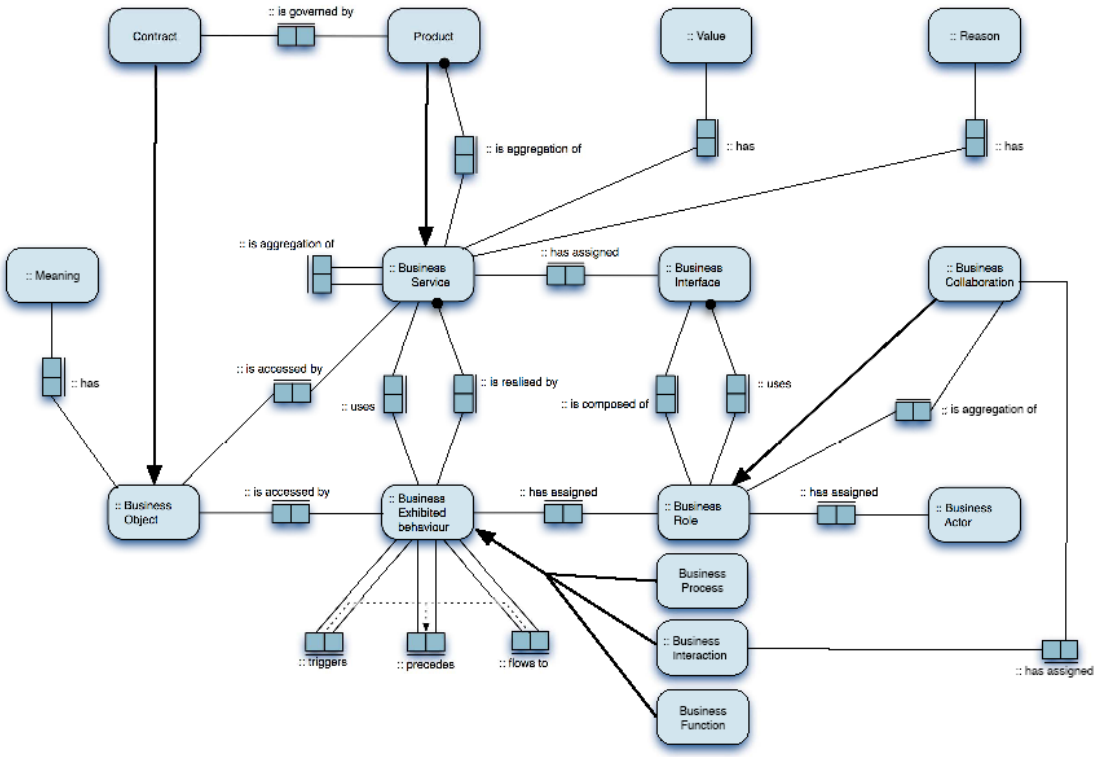
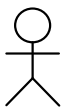
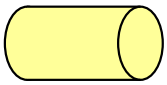
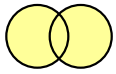


Figure 12. Level 8 – Business layer

Table 1 summarises the business layer concepts, including their default graphical notations.

Table 1. Business layer concepts

Concept	Description	Notation
Business actor	An organisational entity that is capable of performing behaviour.	Business actor 
Business role	A named specific behaviour of a business actor participating in a particular context.	Business role 
Business collaboration	A (temporary) configuration of two or more business roles resulting in specific collective behaviour in a particular context.	Business collaboration 
Business interface	Declares how a business role can connect with its environment.	Business interface 
Business object	A unit of information that has relevance from a business perspective.	Business object

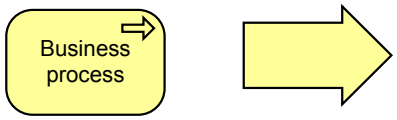
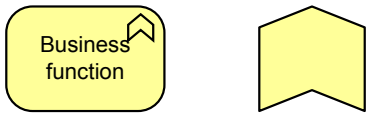
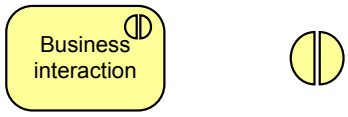
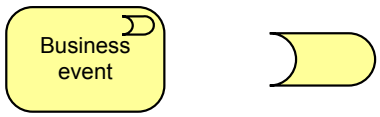
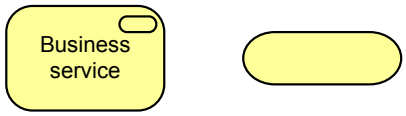
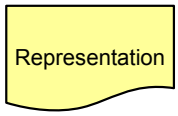
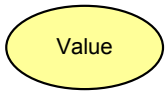
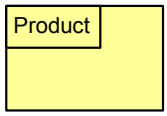
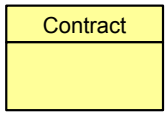
Concept	Description	Notation
Business process	A unit of internal behaviour or collection of causally related units of internal behaviour intended to produce a defined set of products and services.	
Business function	A unit of internal behaviour that groups behaviour according to, for example, required skills, knowledge, resources, etc., and is performed by a single role within the organisation.	
Business interaction	A unit of behaviour performed as a collaboration of two or more business roles.	
Business event	Something that happens (internally or externally) and influences behaviour.	
Business service	An externally visible unit of functionality, which is meaningful to the environment and is provided by a business role.	
Representation	The perceptible form of the information carried by a business object.	
Meaning	The knowledge or expertise present in the representation of a business object, given a particular context.	
Value	That which makes some party appreciate a service or product, possibly in relation to providing it, but more typically to acquiring it.	
Product	A coherent collection of services, accompanied by a contract/set of agreements, which is offered as a whole to (internal or external) customers.	
Contract	A formal or informal specification of agreement that specifies the rights and obligations associated with a product.	

Figure 13 illustrates the use of the main business layer concepts with a small self-explanatory example of how an insurance claim is processed.

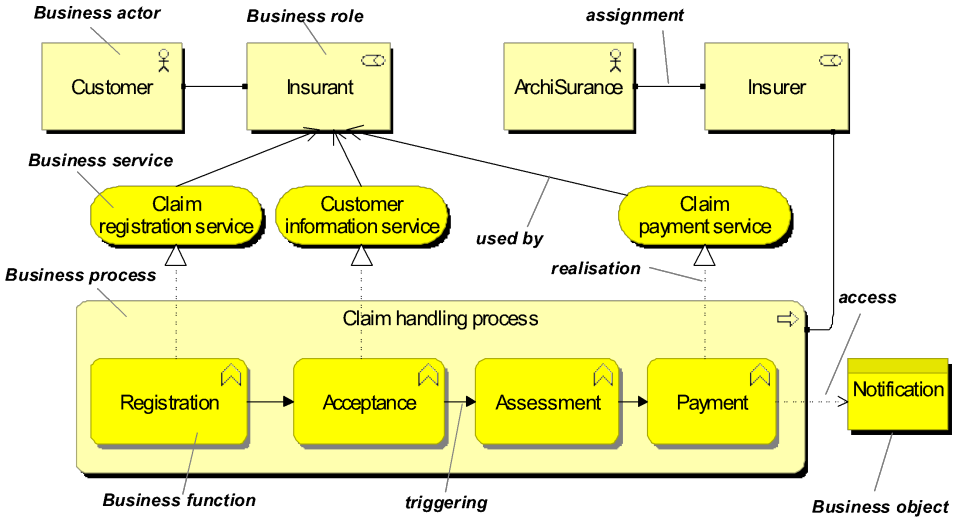


Figure 13. Business layer example model

Application layer

The application layer, shown in Figure 14, does not lead to the introduction of additional concepts, and only involves the re-naming of some of the existing concepts. The renamings results in new names for existing concepts, which correspond better to the names already used by the partners participating in the ArchiMate project, as well as to existing standards such as the UML. For example, an *Application component* is the application-layer specialisation of the *Role* concept. Not shown in the figure is that *Application services* may also be aggregated in a *Product*.

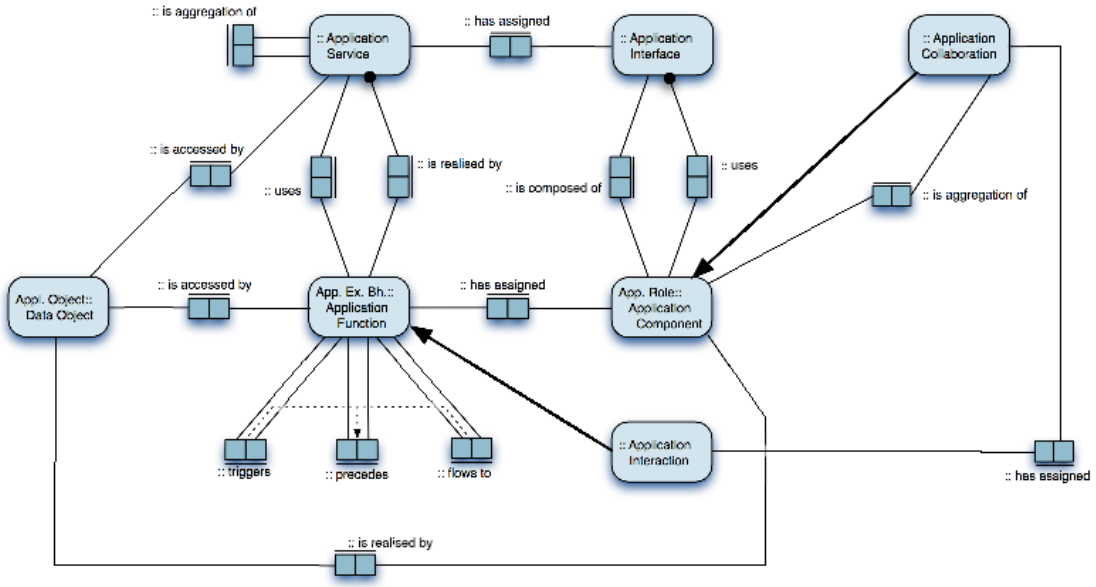


Figure 14. Level 8 – Application layer

Table 2 summarises the application layer concepts, including their default graphical notations.

Table 2: Application layer concepts

Concept	Definition	Notation
Application component	A modular, deployable, and replaceable part of a system that encapsulates its contents and exposes its functionality through a set of interfaces.	Application component
Application collaboration	An application collaboration defines a (temporary) configuration of two or more components that co-operate to jointly perform application interactions.	Application collaboration
Application interface	An application interface declares how a component can connect with its environment.	Application interface
Data object	A coherent, self-contained piece of information suitable for automated processing.	Data object
Application function	A coherent group of internal behaviour of a component.	Application function
Application interaction	A unit of behaviour jointly performed by two or more collaborating components.	Application interaction
Application service	An externally visible unit of functionality, provided by one or more components, exposed through well-defined interfaces, and meaningful to the environment.	Application service

Figure 15 illustrates the use of the main application layer concepts with a small example.

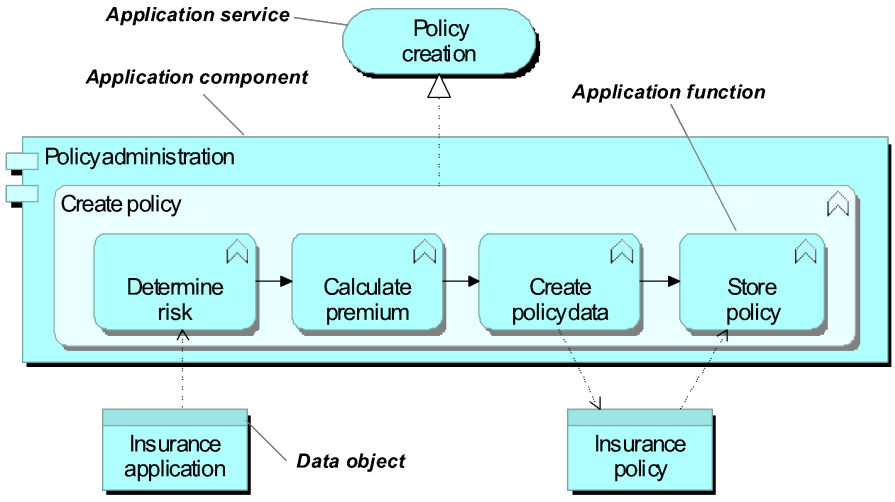


Figure 15. Application layer example model

Technology layer

The technology (or infrastructure) layer also involves some renamings of existing concepts. In addition, some further refinements of existing concepts were needed as well, as depicted in Figure 16. The newly introduced concepts deal with the different kinds of elements that may be part of a technology infrastructure: *System software*, *Device* and *Network*, which are all specialisations of the *Node* concept, itself an *Infrastructure role*.

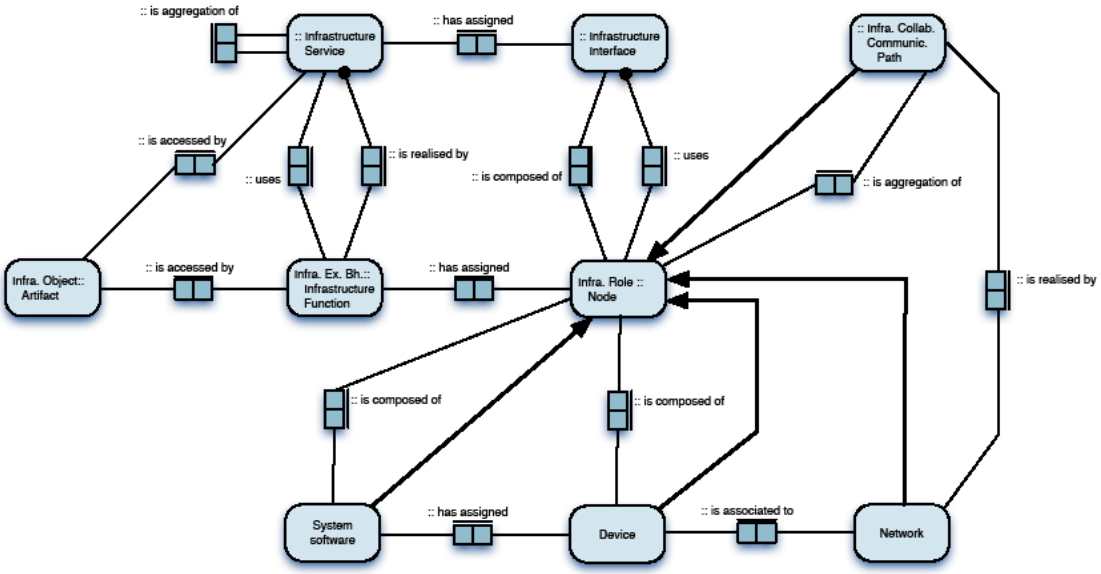


Figure 16. Level 8 – Technology layer

Table 3 summarises the technology layer concepts, including their default graphical notations.

Table 3: Technology layer concepts

Concept	Definition	Notation
Node	A computational resource upon which artifacts may be deployed for execution.	Node
Device	A physical computational resource upon which artifacts may be deployed for execution.	Device
Network	A physical communication medium between two or more devices.	Network
Communication path	A link between two or more nodes, through which these nodes can exchange information.	Communication path

Concept	Definition	Notation
Infrastructure interface	A point of access where the functionality offered by a node can be accessed by other nodes and application components.	Infrastructure interface
System software	A software environment for specific types of components and objects that are deployed on it in the form of artifacts.	System software
Infrastructure service	An externally visible unit of functionality, provided by one or more nodes, exposed through well-defined interfaces, and meaningful to the environment.	Infrastructure service
Artifact	A physical piece of information that is used or produced in a software development process, or by deployment and operation of a system.	Artifact

Figure 16 illustrates the use of the main technology layer concepts with a small example.

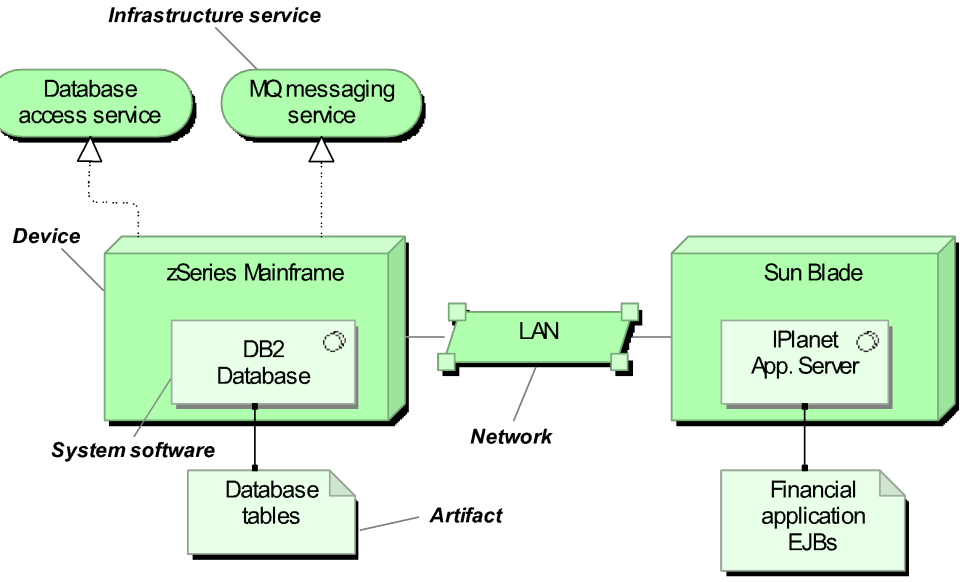
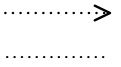
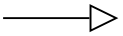


Figure 17. Technology layer example model

Relations

As outlined in the previous sections, ArchiMate defines a limited set of relation types, which are summarised in Table 4**Error! Reference source not found.** (including their default graphical notation as used in the previous example models).

Table 4. Relation types

Structural Relations		**Notation**
Association	Models a relation between concepts that is not covered by another, more specific relationship.	
Access	Models the access of behaviour concepts to business or data objects.	
Used by	Models the use of services by behaviour concepts and the access to interfaces by structure concepts.	
Specialisation	Indicates that an object is a specialisation of another object.	
Realisation	Links a logical entity with a more concrete entity that realises it.	
Assignment	Links behaviour concepts with active structure concepts (e.g., roles, components) that perform them, or roles with actors that fulfil them.	
Aggregation	Indicates that a concept groups a number of other concepts.	
Composition	Indicates that a concept consists of a number of other concepts.	
Behavioural Relations		**Notation**
Flow	Describes the exchange or transfer of, for example, information or value between behaviour concepts.	
Triggering	Describes a temporal or causal ordering of behaviour concepts.	

In addition to the 'direct' relations in the metamodel described in the previous sections, we have defined a property on the structural relations that allows us to compose them and define the indirect relations between concepts 'at a distance'. The structural relations are listed in Table 4 in ascending order by binding strength: *association* is the weakest structural relations; *composition* is the strongest. Part of the language definition is an abstraction rule that states that two relations that join at an intermediate element can be combined and replaced by the weaker of the two. If two structural relationships $r : R$ and $s : S$ are permitted between elements a, b, and c such that $a\ r\ b$ and $b\ s\ c$, then a structural relationship $t : T$ is also permitted, with $a\ t\ c$ and type T being the weakest of R and S.

Transitively applying this property allows us to replace a chain of structural relations (with intermediate concepts) by the weakest structural relation in the chain. A more extensive description and derivation of this property is given by (Buuren et al., 2004). With this rule, it is possible to determine the 'indirect' relationships that exist between model elements without a direct relationship, which may be useful for, among other things, impact analysis. By deriving these indirect relations, one can obtain an overview of the impact of e.g. changes to a business process or failure of a hardware device on the entire architecture.

This property was specifically designed into the language because ArchiMate focusses strongly on the relationships and coherence within enterprise architectures. It

has been defined as an integral part of the ArchiMate language. Thus, all these derived relations are also valid in ArchiMate (i.e., part of the language metamodel), although they were not shown in the successive metamodels explained in the previous sections. To our knowledge, this transitivity of relations is unique to ArchiMate no other modelling languages exhibit such a property.

Conclusions and Future Work

In this paper we have discussed the key structures and principles underlying the design of the ArchiMate language. We have reviewed the challenges confronting an architecture description language for enterprise architecture, as well as the design principles aiming to meet these challenges. We then discussed the modelling concepts needed in the ArchiMate language, where we made a distinction between concepts needed to model domains in general, the modelling of dynamic systems, and the modelling of enterprise architectures.

Recently, the ArchiMate language has been adopted by the Open Group. It is expected that the language will evolve further to better accompany future versions of the Open Group's architecture framework (TOGAF). This can easily be accommodated by taking the metamodel at level 6 as a common denominator. At level 7 a choice has to be made for a specific architecture framework; in the case of TOGAF this corresponds to a *business architecture*, an *information systems architecture* (comprising a *data architecture* and an *application architecture*) and a *technology architecture*. These largely correspond to the existing three ArchiMate layers, but some adjustments might be in order. TOGAF version 9 (The Open Group, 2009a) now also includes a content metamodel that defines a formal structure for the products and artefacts produced by its Architecture Development Method (ADM). Many of the core concepts of this metamodel can easily be mapped to ArchiMate and vice versa, which makes ArchiMate well-suited for usage in combination with TOGAF 9. More research is needed especially in determining a suitable way of modelling TOGAF's requirements, principles and constraints in ArchiMate.

Related to this is the issue of modelling business rules, which can be viewed as a declarative way of operationalising principles and constraints. Linking ArchiMate to emerging business rules standards such as SBVR (SBVR Team, 2006) may also require adding one or more concepts to the metamodel at level 6 or 7.

Furthermore, we also envisage the support of other architecture frameworks. As also advocated by TOGAF's ADM, TOGAF can be used with other architecture (content) frameworks as well, such as GERAM (IFIP-IFAC Task Force, 1999), Zachman (Zachman, 1987) and IAF (Capgemini, 2007, Goedvolk et al., 1999). It is only natural for ArchiMate to mirror this ability.

Finally, we expect that concepts such as goal modelling (Yu & Mylopoulos, 1996), value modelling (Gordijn et al., 2006) and transaction modelling (Dietz, 2006) will also lead to further refinements of the language, primarily requiring further extensions at level 7 that relate to our intentional concepts. This would enhance ArchiMate's suitability as a language for early-stage and business-oriented modelling.

Acknowledgments

This research reported in this paper resulted from the ArchiMate project. The project consortium consisted of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Novay (formerly Telematica Instituut), Centrum Wiskunde & Informatica, Radboud Universiteit Nijmegen, and the Leiden Institute of Advanced Computer Science.

References

Arbab, F., Boer, F.S. de, Bonsangue, M., Lankhorst, M.M., Proper, H.A., & Torre, L. van der (2007). Integrating Architectural Models. *Enterprise Modelling and Information Systems Architectures*, 2(1), 40–57.

Beer, S. (1985). *Diagnosing the System for Organizations*. New York, New York, USA: Wiley.

Biemans, F.P.M., Lankhorst, M.M., Teeuw, W.B., & Wetering, R.G. van de (2001). Dealing with the Complexity of Business Systems Architecting. *Systems Engineering*, 4(2), 118—133.

Bosma, H., Doest, H. ter, & Vos, M. (2002). *Requirements*. ArchiMate Deliverabe D4.1, TI/RS/2002/112. Enschede: Telematica Instituut.

Brooks Jr., F.P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10–19.

Buuren, R.van, Jonkers, H., Iacob, M.-E., Strating, P. (2004). Composition of Relations in Enterprise Architecture. In Ehrig, H. et al., *Proceedings of the Second International Conference on Graph Transformation*, pp. 39–53, Rome, Italy.

Capgemini. (2007). *Enterprise, Business and IT Architecture and the Integrated Architecture Framework*. White paper. Utrecht, The Netherlands: Capgemini.

Dietz, J.L.G. (2006). *Enterprise Ontology – Theory and Methodology*. Berlin: Springer.

Dijkstra, E.W. (1968). Structure of the 'THE'-Multiprogramming System. *Communications of the acm*, 11(5), 341–346.

Falkenberg, E.D., & Oei, J.L.H. (1994). Meta Model Hierarchies from an Object–Role Modelling Perspective. In: Halpin, T.A., & Meersman, R. (eds), *Proceedings of the First International Conference on Object–Role Modelling (ORM–1)*, pp. 218–227. Magnetic Island, Queensland, Australia: Key Centre for Software Technology, University of Queensland, Brisbane, Australia.

Goedvolk, J.G., Bruin, H. de, & Rijsenbrij, D.B.B. (1999). Integrated Architectural Design of Business and Information Systems. *In:* Bosch, J. (ed), *Proceedings Of The Second Nordic Workshop on Software Architecture (NOSA'99)*. Research Report, vol. 1999, no. 13. Ronneby, Sweden: University of Karlskrona/Ronneby.

Gordijn, J., Petit, M., & Wieringa, R. (2006). Understanding Business Strategies of Networked Value Constellations Using Goal and Value Modeling. *Pages 126–135 of: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*. Washington DC, USA: IEEE Computer Society.

Halpin, T.A., & Morgan, T. (2008). *Information Modeling and Relational Databases*. 2nd edn. Data Management Systems. Morgan Kaufman.

Hofstede, A.H.M. ter, & Weide, Th.P. van der (1993). Expressiveness in Conceptual Data Modelling. *Data & Knowledge Engineering*, 10(1), 65–100.

Hofstede, A.H.M. ter, Proper, H.A., & Weide, Th.P. van der (1993). Formal Definition of a Conceptual Language for the Description and Manipulation of Information Models. *Information Systems*, 18(7), 489–523.

IEEE (2000). *Recommended Practice for Architectural Description of Software Intensive Systems*. IEEE Std 1471–2000. The Architecture Working Group of the Software Engineering Committee, Standards Department, IEEE. Piscataway, New Jersey, USA: IEEE Computer Society.

IFIP-IFAC Task Force (1999). *GERAM: Generalised Enterprise Reference Architecture and Methodology*. Version 1.6.3, Published as Annex to ISO WD15704.

Jonkers, H., Veldhuijzen van Zanten, G.E., Buuren, R. van, Arbab, F., Boer, F. de, Bonsangue, M., Bosma, H., Doest, H. ter, Groenewegen, L., Guillen Scholten, J., Hoppenbrouwers, S.J.B.A., Iacob, M.-E., Janssen, W., Lankhorst, M.M., Leeuwen, D. van, Proper, H.A., Stam, A., & Torre, L. van der (2003). Towards a Language for Coherent Enterprise Architecture Descriptions. In: Steen, M., & Bryant, B.R. (Eds.), *7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pp. 28–39, Brisbane, Queensland, Australia. Los Alamitos, California, USA: IEEE.

Jonkers, H., Lankhorst, M.M., Buuren, R. van, Hoppenbrouwers, S.J.B.A., Bonsangue, M., & Torre, L. van der (2004). Concepts for Modeling Enterprise Architectures. *International Journal of Cooperative Information Systems*, 13(3), 257–288.

Krogstie, J., Lindland, O.I., & Sindre, G. (1995). Defining Quality Aspects for Conceptual Models. *Pages 216–231 of:* Falkenberg, E.D., Hesse, W., & Olivé, A. (eds), *Information System Concepts: Towards a consolidation of views – Proceedings of the third IFIP WG8.1 conference (ISCO–3)*. Marburg, Germany, EU: Chapman & Hall/IFIP WG8.1, London, United Kingdom, EU.

Lankhorst, M.M., et al. (2005). *Enterprise Architecture at Work: Modelling, Communication and Analysis*. Berlin, Germany, EU: Springer.

Lindland, O.I., Sindre, G., & Sølvberg, A. (1994). Understanding Quality in Conceptual Modeling. *IEEE Software*, 11(2), 42–49.

Nadler, D., Gerstein, M., Shaw, R., (1992). *Organisational Architecture: Designs for Changing Organisations*. San Francisco: Jossey-Bass.

OMG (2006). *Meta Object Facility (MOF) Core Specification, Version 2.0*. OMG Available Specification formal/06-01-01. Needham, Massachusetts: Object Management Group.

OMG (2008). *Business Process Modelling Notation, V1.1*. OMG Available Specification formal/2008-01-17. Needham, Massachusetts: Object Management Group.

OMG (2009). *UML 2.2 Superstructure Specification*. OMG Available Specification formal/2009-02-02. Needham, Massachusetts: Object Management Group.

Op 't Land, M., Proper, H.A., Waage, M., Cloo, J., & Steghuis, C. (2008). *Enterprise Architecture – Creating Value by Informed Governance*. Berlin: Springer.

Proper, H.A., Verrijn-Stuart, A.A., & Hoppenbrouwers, S.J.B.A. (2005). Towards Utility-based Selection of Architecture Modelling Concepts. *Pages 25–36 of:* Hartmann, S., & Stumptner, M. (eds), *Proceedings of the Second Asia–Pacific Conference on Conceptual Modelling (APCCM2005), Newcastle, New South Wales, Australia*. Conferences in Research and Practice in Information Technology Series, vol. 42. Sydney, New South Wales, Australia: Australian Computer Society.

SBVR Team (2006). *Semantics of Business Vocabulary and Rules (SBVR)*. Tech. rept. dtc/06–03–02. Needham, Massachusetts: Object Management Group.

Steen, M.W.A., Doest, H.W.L. ter, Lankhorst, M.M., & Akehurst, D.H. (2004). Supporting Viewpoint–Oriented Enterprise Architecture. *Pages 20–24 of: Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*.

Teeuw, W.B., & Berg, H. van den (1997). On the Quality of Conceptual Models. *In:* Liddle, S.W. (Ed.), *Proceedings Of The ER'97 Workshop on Behavioural Models And Design Transformations: Issues And Opportunities in Conceptual Modeling*, Los Angeles, California: UCLA.

The Open Group (2009a). *The Open Group Architecture Framework (TOGAF) Version 9*. Reading, UK: The Open Group.

The Open Group (2009b). *ArchiMate 1.0 Specification, Technical Standard*. Reading, UK: The Open Group.

Wagter, R., Berg, M. van den, Luijpers, J., Steenbergen, M. van (2005). *Dynamic Enterprise Architecture: How to Make It Work*. Hoboken, New Jersey: Wiley.

Yu, E., & Mylopoulos, J. (1996). Using Goals, Rules, and Methods to Support Reasoning In Business Process Reengineering. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 5(1), 1–13. Special issue on Artificial Intelligence in Business Process Reengineering.

Zachman, J.A. (1987). A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3).