Towards a Complete and Minimal Basis for the Definition of Software Architecture Languages

Hüseyin Yüksel[1], Johannes Reichardt[1], and Chris Johnson[2]

[1]h_da – University of Applied Sciences Darmstadt, Germany
h.yueksel@fbi.h-da.de
j.reichardt@fbi.h-da.de
[2]University of Plymouth, Plymouth, UK
c.johnson@plymouth.ac.uk

Abstract: A meta-metamodel, which we refer to as the ADL-Kernel, for the definition of architecture description languages is presented. The ADL-Kernel is complete and minimal in the sense that each of its instantiations contains a minimal set of architectural elements capable of describing any software architectural solution. The ADL-Kernel is presented in terms of a hypothesis that is based on a stakeholder-oriented definition of software architecture and validated against current UML, as far as it is conformant with the underlying definition of software architecture.

1 Introduction

Existing Architecture Description Languages (ADLs) such as UML (see [Ob03]) provide a large repertoire of language elements for both the architectural and algorithmic descriptions of software systems. In the present paper, we investigate the question of when such a set of language elements is sufficient (or complete), and necessary (or minimal) for the purpose of providing such descriptions. The question of completeness and minimality of language elements for the description of algorithms has already been answered by the introduction of Turing machines and the establishment of the Turing-Church hypothesis. In analogy to this, we will in the present paper present a complete and minimal set of description elements for the architectural level. We present our architectural elements as a meta-meta model (which we refer to as the "ADL-Kernel") which provides a complete and minimal basis for the definition of architecture description languages. As is the case with Turing machines, the proof of completeness and minimality can only be expected in an empirical form, i.e. in terms of a hypothesis. For this reason, the completeness and minimality of the ADL-Kernel will be demonstrated by validating it against recent development of UML 2.0.

One difficulty with the formulation of our ADL-Kernel is the question – as yet unresolved in the literature – of the proper purpose and definition of software architecture (see [Ca07]). This includes, in particular, the question of whether, or to what extent, the specification of the functional requirements should be regarded as part of software architecture. From a Turing viewpoint, which we will take in the course of our discussion, the structural emphasis in the de-facto standard definition of software architecture (according to [BCK03]) appears inappropriate. This stance is supported by Rice's theorem [HMU01] on Turing machines, which states that the function of a Turing machine cannot be read from its syntactical structure, i.e. from an unlabelled Turing table. Due to this undecideable classification problem, textual and, thus, conceptual elements are necessary to define and explain the functionality of a program. Structure can only serve the purpose of adding a positional value to textual elements. By "textual description elements" we understand "terms", representing concept designators, and "texts", representing concept definitions. All of these textual elements are capable of specifying functional requirements. The intent to combine structural and textual elements in the architectural description of a program is to specify its "syntax and semantics in the large" by a simultaneous refinement of requirements, function and structure (see [Re03]). Thus, our view of software architecture implies that its primary goal is to support human understanding of a program, which is considered a prerequisite of any activity of the stakeholders during software development. Furthermore, by the inclusion of textual elements in the meta-metamodel the specification of functional requirements becomes an integral part of software architecture.

2 The ADL-Kernel

In this section the language elements of the ADL-Kernel are introduced. The ADL-Kernel is an architecture description language definition which has been developed with twofold aims: 1.) to provide a set of architecture description elements which is *complete* and *minimal*; and 2.) to provide a language from which ADLs can be developed. The ADL-Kernel has been designed on the basis of previous works on ADLs, including the visual modelling language FracTool® (see [Re06]). However, while it adopts concepts from FracTool®, such as the relationship between requirements, function and architectural structure (see [Re03] and [Re05]), the ADL-Kernel also addresses the notion of completeness and minimality for ADLs.
In order to provide a language from which ADLs can be developed, the metamodelling approach[1] of the OMG has been applied to specify the language elements of the ADL-Kernel (see [Ob03]). The syntax and semantics of the language elements are specified as meta-metaclasses within a meta-metamodel. The interrelationships among the language elements are modelled by inheritance-, association-, aggregation-, and composition-relations. The following figure visualises the essential meta-metaclasses of the ADL-Kernel.

The key features of the ADL-Kernel can be summarized as follows:

1. The ADL-Kernel provides constructs for documenting requirements in the architectural structure (see the modelling elements "Requirement" and "ArchitecturalElement" in Figure 1 and their "concept"-association). An architectural element may be associated with several concepts representing requirements of different levels of abstraction, from user requirements to the name of the architectural element.
2. In the ADL-Kernel, terms are bearers of the functional purpose of architectural

1 The meta-modelling approach is an object-oriented specification approach, where elements in a given conceptual layer, e.g. meta-meta-layer, describe elements in the next layer down, e.g. meta-layer. In our case, we have defined a meta-meta-model which can be used to define Architecture Description Languages.

elements. They define and explain the functionality of architectural elements (see Figure 1 attribute "Naming" of the meta-metaclass "ArchitecturalElement").

3. The ADL-Kernel requires the parallel refinement of the functional specification and the architectural structure, so that the functional refinement is always based upon the architectural structure (highlighted as a constraint definition on the "structure"- and "refinement"-association in Figure 1).
4. The ADL-Kernel supports modelling, i.e. it supports a structure-preserving mapping from any application domain onto the software architecture (see meta-metaclasses "Designatum" and "ArchitecturalElement" in Figure 1 and their "type"-association)

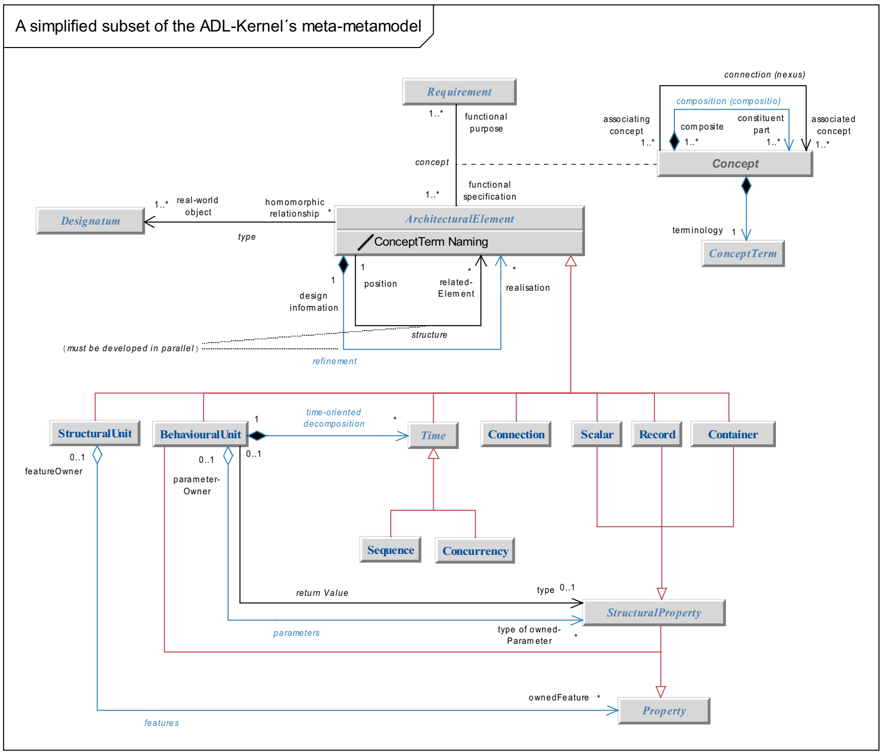

Figure 1: A simplified subset of the ADL-Kernel's meta-metamodel[2]

The remainder of this section discusses the main concepts behind the ADL-Kernel and presents each of its language elements.

2 The ADL-Kernel contains various abstract concepts which are applied to reduce the extent of the language specification. For the sake of simplicity, however, they were omitted in this paper (for more detail see [Yu09]).

The root concepts of the ADL-Kernel

The ADL-Kernel provides a number of root meta-metaclasses which define the ADL-Kernel software architecture concept (depicted as abstract meta-metaclasses in Fig.1). In simple terms, the root modelling elements and the four association definitions of the meta-metaclass "ArchitecturalElement"[3] specify, for any architectural element at a certain position in the architecture description, the following information (cp. [Re03]):

1. The type of that architectural element (via the "type"-association)
2. The location of that architectural element relative to other architectural elements (via the "structure"-association)
3. The purpose of this architectural element at this position (via the role definition "purpose" at the "concept"-association)
4. The functional refinement of this architectural element or, in other words, the interior structure and how it functions (via the "refinement"-association).

The "concept"-association and the modelling elements "Concept", "ConceptTerm" and "Requirement"

The principal purpose of an architectural element in the ADL-Kernel is to define and explain the function of a software system partly or entirely. This functional information is specified by naming the architectural element (via the attribute "Naming" of the modelling element "ArchitecturalElement" in Figure 1). For example, the label "Fax machine" of an architectural element defines that it provides fax functionalities. The naming is specified in the ADL-Kernel as an attribute of the meta-metaclass "ArchitecturalElement" which is derived from the "ConceptTerm" meta-metaclass[4]. The "ConceptTerm" meta-metaclass specifies a term definition in the context of the "Concept" meta-metaclass. A term is a shortened form of a concept and distinguishes unambiguously one concept from another. The "Concept" meta-metaclass defines primarily the relations by which concepts are developed (via the "composition"-association in Figure 1) and ordered (via the "nexus"-association in Figure 1). Furthermore, it is specified as an association-metaclass which ensures that in the context of software architecture descriptions only such concepts are developed or used which are relevant for the specification of functional requirements (cp. Figure 1).

The "refinement" -association

The "refinement"-association captures and provides the functional refinement of a software system. For example, considering a fax software system, the "refinement"-association would provide the individual components of such a software system, e.g. the scanner, printer, and the modem component.

The "structure"-association

The "structure"-association captures all potential interrelationships of architectural elements in a software system. Herewith the position[5] of an architectural element in the architectural structure carries a meaning (highlighted by the role definition "position" at

3 The four associations have been developed in dependence on the semiotic model introduced in [Re03]. The semiotic model is concerned with the process of understanding computer programs. In this context, a computer program is considered as a semiotic sign describing syntactic, semantic, and other important sign aspects.

4 The derivation is indicated by the slash and the type of the attribute definition (cp. Figure 1).

the reflexive “structure”-association in Figure 1). For example, the position of a layer within a layered architecture provides essential information. Here, the position reflects the degree of abstraction and places the layer of abstraction onto a scale between a basic machine and a user’s application[6] (cp. [Re03]).

The “type”-association and the “Designatum” modelling element
The “type”-association specifies a structure-preserving mapping from real-world-concepts onto the constructs of the ADL-Kernel. In particular, the type and the module structure of architectural elements are derived from this association. The “Designatum” meta-metaclass stands abstractly for that part of a real-world concept that is denoted by an architectural element (cp. [Re03])[7]. For example, a fax software system can be modularised by observing the real world construction of a fax machine, i.e. the individual components, their interrelationships and the communication among them can be derived from its real-world counterpart.

The tangible meta-metaclasses of the ADL-Kernel
The tangible meta-metaclasses can be subdivided into groups which abstractly represent operations, operands and the references between them. In this section we will present each modelling element.
StructuralUnit. A structural unit represents a node in the module structure of the architecture, including modules, components and layers. Each structural unit consists of: 1) a set of structural sub-units and 2) a set of global data elements or properties. The elements of these two sets, as well as the structural unit itself, are positioned according to the “structure”-association of the ADL-Kernel (cp. Figure 1).

BehaviouralUnit. A behavioural unit represents an abstract functional description of a process, including sequential and parallel activities. Behavioural units are used to specify parameters and method signatures of interfaces.

Sequence. This modelling element specifies a concept for modelling sequencing in the context of behavioural units. For example, the chronology of a business process could be specified by the “Sequence” meta-metaclass.

Concurrency. This modelling element specifies a concept for modelling parallelism in the context of behavioural units. For example, parallel activities of a business process could be specified by the “Concurrency” meta-metaclass.

5 The meaning and importance of the position is introduced and discussed in [Re03]. The location is considered as “a *concept of order* [...] capable of inducing partial orders on the [... architecture], reflecting uses-relations, inheritance-relations and others. Therefore, the location of [... an architectural element] within a structure conveys essential information for the abstract understanding of a program.”
6 An example of a layered architecture is the OSI-reference-model. The layers in the OSI-reference-model are ordered, with hardware-dependent layers at the one end of the system hierarchy and application specific layers at the other end (cp. [Ta02]).
7 Sometimes, architectural descriptions are termed as “models of the real world”.

Connection. This modelling element is used to interconnect abstract operations such as structural and behavioural units with abstract operands such as scalars, records and containers. This means that the communication among structural and behavioural units, respectively, is always specified by using at least one connection, where one end of the connection is attached to an operation and the other end to a global operand. This implies strictly hierarchical cross-references among structural and behavioural units.

Scalar, Record and Container. These modelling elements specify operand types. They are used to specify "uses"-relations and communications among architectural elements. In addition, they can represent complex data structures, e.g. a database. A scalar is a simple operand type whose internal structure is not visible. Unlike scalars, the internal structure of records and containers are visible. The main distinction between a record and a container is that a record contains a fixed number of different operand types, whereas a container contains any number of the same operand type.

3 Empirical validation of the completeness and minimality of the ADL-Kernel against UML 2.0

This chapter demonstrates the empirical validity of the completeness and the minimality of our ADL-Kernel we have introduced in chapter 2, against the modelling elements of UML 2.0 (see [Ob03]). UML is widely accepted as the standard modelling language for representing various software artifacts generated during a development process (cp. [Ob03]). UML version 2.0 has created expectations about the expressive power of the language to capture software architecture more precisely (cp. [Ob03]). For example, UML 2.0 provides new concepts, such as the structured class and connector concept, which are explicitly targeted at the description of architectural issues of a software system (cp. [Ob03]). Furthermore, the existing component concept has been revised in order to model architectural components (cp. [Ob03]). In addition to the support of modelling architectural issues, UML 2.0 provides modelling elements, such as the use case and class, for specifying functional requirements of a software system (cp. [Ob03]). For this purpose UML has also been used as an object-oriented functional requirement specification notation (cp. [KGL+06]). Therefore, we consider UML 2.0 as a modelling language that has the capability to model architectural and functional requirements issues of any application domain, i.e. UML 2.0 provides a complete set of modelling elements for these purposes (cp. [BK03] and [KGL+06]). For this reason, we want to demonstrate the empirical validity of the completeness of the ADL-Kernel based on UML 2.0 by defining mapping-relations between the metaclasses of UML 2.0 and the meta-metaclasses of the ADL-Kernel. In order to demonstrate that the ADL-Kernel has a similar expressive power for modelling architectural and requirements issues of a software system like UML 2.0, we have to focus on the modelling elements which deal with architectural and functional requirements issues. Therefore, we provide in [Yu09] a simplified UML 2.0 metamodel that represents only those metaclasses of UML 2.0, which are relevant for modelling architectural and requirements issues. Despite its capability as a modelling language for architectural and requirements issues, UML 2.0 has a number of drawbacks regarding its richness. In particular, it provides in some cases two or more different types of modelling elements for modelling the same concept. In our investigation we have encountered that UML 2.0 provides the structured class and component modelling element for modelling computational units and their hierarchical decomposition. Hence, in the language reference [RJB04] it is cited that: “*The distinction between a structured class and a component is somewhat vague and more a matter of intent than firm semantics.”* Therefore, we want to demonstrate, based on these drawbacks of UML 2.0, empirically that our ADL-Kernel provides a minimal set of language elements for specifying architectural and requirements issues of a software system. This means, in contrast to UML, none of the language elements can be expressed by another language element in that set or by a combination of language elements of this set. Thus, all language elements of the ADL-Kernel are necessary, which is in simple terms our definition of minimality.

The validation strategy
In order to demonstrate the empirical validity of the completeness and minimality of the ADL-Kernel based on UML 2.0 we have pursued the following strategy:

1. In order to show that the ADL-Kernel provides a similar expressive power as UML 2.0 for specifying architectural and requirements issues of a software system, we provide a set of mapping relations between the metaclasses of the simplified UML 2.0 metamodel (see [Yu09]) and the meta-metaclasses of the ADL-Kernel (see [Yu09]). A mapping relation shows for a given UML metaclass the adequate ADL-Kernel meta-metaclasses that provides a similar expressive power. In our case the mapping between the syntax and semantics of the simplified UML 2.0 metamodel and the syntax and semantics of the tangible meta-metaclasses of the ADL-Kernel are defined in natural language. Thus, if we provide for all metaclasses of the simplified UML 2.0 metamodel at least one mapping relation onto the meta-metaclasses of the ADL-Kernel, then we can ensure the empirical validity of the completeness of our ADL-Kernel based on UML 2.0.
2. In order to verify the minimality of our ADL-Kernel, we have pursued following strategy:
    a. There are no two mapping relations that map one UML 2.0 metaclass specified in the simplified UML 2.0 metamodel onto two distinct meta-metaclasses of the ADL-Kernel. Thus, in contrast to UML 2.0, the ADL-Kernel does not provide for the same concept two distinct meta-metaclasses
    b. We will show that the structured class and component concepts, can be mapped onto the same meta-metaclass of the ADL-Kernel, namely the “StructuralUnit” meta-metaclass. Thus, the ADL-Kernel provides a more minimal set of language elements as UML 2.0.

   Based on these considerations, we can empirically demonstrate that the ADL-Kernel provides a minimal set of architectural elements.

In [Yu09] the results of this investigation are discussed in detail. In this section we present a brief overview of the results. For this purpose, we will show the mapping for UML’s structured class, port, interface connector and component concepts. These are modelling elements which are relevant for modelling architectural issues. Finally, we will show the mapping for the class concept. This is relevant for modelling requirements issues.

Mapping of the UML 2.0 metaclasses "StructuredClass", "Port" and "Interface"
In order to map the metaclasses "StructuredClass", "Port" and "Interface", we use the meta-metaclasses "StructuralUnit", "BehaviouralUnit" and "StructuralProperty" of the ADL-Kernel as the base classes (see chapter 2).
The modelling element "StructuredClass" in UML 2.0 is a subclass of the modelling element "Classifier" so that it can define attributes and operations (cp. [Ob03]). An attribute is specified via the metaclass "Property" (see [Yu09]), whereas an operation is specified via the metaclass "Operation" (see [Yu09]). These characteristics are similar to that of a structural unit in the ADL-Kernel (see chapter 2). The meta-metaclass "StructuralUnit" defines attributes and operations based on the "features"-association to the modelling element "Property" (see chapter 2). A property in the ADL-Kernel specifies a behavioural or structural characteristic of a structural unit. In particular:

1. attributes are structural features which are specified by sub-classes of the meta-metaclass "StructuralProperty" (similar to that of UML's "Property" metaclass) and
2. operations are behavioural features which are specified by the modelling element "BehaviouralUnit" (similar to that of UML's "Operation" metaclass).

A structured class in UML 2.0 can be a simple unit or a composite unit. This means that it can contain other descriptional elements, such as part, port and connector (cp. [Yu09]). This characteristic is similar to that of the "refinement"-association of the "StructuralUnit" meta-metaclass, which provides the internal construction of a structural unit[8].
In UML 2.0 a structured class provides interfaces via one of its ports. An interface in UML 2.0 specifies a set of public operations assigned to a structured class or component port. The modelling element "Interface" in UML 2.0 is a subclass of the modelling element "Classifier" so that it can define attributes and operations. These characteristics are similar to that of a structural unit (cp. Figure 1). Therefore, in contrast to UML 2.0, the ADL-Kernel does not contain an explicit interface language element. We assume that interface definitions are a special kind of structural unit definition, i.e. an interface definition is the definition of a structural unit which highlights public features or required features of structural units. A port is specified in [Ob03] as a structural feature of a component that defines the interaction points between the component and its environment. A structural feature of a structural unit is specified in the ADL-Kernel via the sub-classes of the meta-metaclass "StructuralProperty" (cp. Figure 1). Therefore, we assume that a port is a special kind of structural property definition, i.e. a port definition is the definition of a structural property that defines the interaction points of a structural unit (cp. chapter 2).

Mapping of the UML 2.0 metaclass "Component"
In order to map the syntax and semantics of a component we use the meta-metaclass "StructuralUnit" of the ADL-Kernel as the base class (see chapter 2). The metaclass

8 The meta-metaclass "StructuralUnit" inherits this feature from the abstract meta-metaclass "ArchitecturalElement" (cp. generalisation relationship between these two meta-metaclasses in Figure 1).

"Component" is a sub-class of the metaclass "StructuredClass" (cp. [Yu09]). Thus, it has the same expressive power to specify its internal construction (decomposition) by defining the parts, ports and connectors it contains. A further feature of the "Component" metaclass is specified via its relationship "ownedMember" to the "PackagableElement" metaclass (see [Ob03]). This relation extends the component concept in order to define grouping aspects for modelling elements (like packages). This characteristic is similar to that of the meta-metaclass "StructuralUnit". A structural unit can serve as specification unit which groups architectural elements of a software system that define a certain functional module of a software system. In this context, the elements of such a functional unit can be derived via its "refinement"- association. Therefore, the component concept of UML 2.0 can be mapped with the same systematic as the structured class concept. Hence, the mapping relations defined for these two concepts can be considered as identical. Therefore, two distinct UML 2.0 concepts can be mapped onto one ADL-Kernel concept with the same systematic, and thus, the set of language elements that the ADL-Kernel provides for specifying architectural issues of a software system are more minimal than the set of language elements which UML provides for this purpose.

Mapping of the UML 2.0 metaclass "Connector"
In order to map the syntax and semantics of the "Connector" metaclass we use the meta-metaclass "Connection" of the ADL-Kernel as the base class (see chapter 2). In UML 2.0 two different connector types are specified in [Ob03], the assembly and delegation connectors. An assembly connector is defined as follows: "is a connector between two components that defines that one component provides the services that another component requires" (see [Ob03]). A delegation connector defined as follows: "is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behaviour by the component's parts." These characteristics are similar to that of the "Connection" meta-metaclass (see chapter 2). A connection specifies the interaction of two or more structural or behavioural units. A structural unit in this context can represent a component, and a behavioural unit can represent an operation of an interface. The interaction between structural or behavioural units is specified by using one or more declaration types, such as scalar, container or record. Each declaration defines the certain role of a structural unit in this context, i.e. in UML 2.0 terms the parts of a component.
In order to define an assembly connector in the ADL-Kernel, the appropriate parts are specified as "StructuralProperties", whereas the type information references the respective component. The connection among these parts is specified via the "Connection" meta-metaclass, where each part is connected with one end of a respective connection instance. The other end of the connection instance is connected with a behavioural unit, whereas the behavioural unit can either define explicitly which service is used in this context or can stand abstractly for the interface between these parts (similar to the metaclass "Connector").
In order to define a delegation connector in the ADL-Kernel, the appropriate port is specified via the "StructuralProperty" meta-metaclass. The internal part to which the messages or call is delegated can either be a behaviour of the component itself, e.g. an operation, or a part, i.e. a role of another component in this context. The first case can be

specified by using the “BehaviouralUnit” meta-metaclass and the latter case can be specified by using the “StructuralProperty” meta-metaclass (cp. chapter 2). The connection between the external port and the internal part is specified via the “Connection” meta-metaclass, where one end of the connection is attached to the port and one end to the internal part.

Mapping of the UML 2.0 metaclasses “Class”, “Operation” and “Property”
In order to map the syntax and semantics of a class and its attributes or operations we use the meta-metaclasses “StructuralUnit”, “BehaviouralUnit” and “StructuralProperty” of the ADL-Kernel as the base classes (cp. chapter 2).
A class that defines requirements is (typically) called “analysis class” (cp. [AN05]). An analysis class is distinguished from another analysis class by its name. Hence, the name identifies its functional intent. The operations of an analysis class define the desired functionality. These characteristics are similar to that of a structural unit. The naming of a structural unit serves as a functional identifier, which distinguishes unambiguously its functional purpose from other structural units (cp. chapter 2). In this context, its association to the “Requirement” meta-metaclass determines for which functional requirements it serves as functional specification (cp. Figure 1). The behavioural units of a structural unit define the desired behavioural features, similar to the operation definitions of a class. In UML the operation signature is specified by the operation name, the types of all parameters it contains, and its return type. These characteristics are similar to the “BehaviouralUnit” meta-metaclass. In the ADL-Kernel the signature of behavioural unit is determined by:
1. its name (specified via the “Naming” attribute; see Figure 1),
2. the type of all its contained structural properties (specified via the relation to the “StructuralProperty” meta-metaclass; see Figure 1), and
3. its return values (specified via its “return” association” to the “StructuralProperty” meta-metaclass; see Figure 1).

The attributes of a class in UML defines its structure. This characteristic is also similar to that of a structural unit, because of its “structural features” aggregation to the “StructuralProperty” meta-metaclass which specifies its structural features, like attributes of a class. The name of an attribute in UML serves as an identifier string. Each attribute has a type that denotes a class (cp. [Ob03] and [RJB04]). These characteristics are similar to that of a structural property in the ADL-Kernel (specified via the “StructuralProperty” meta-metaclass; see Figure 1). A structural property has a name which defines unambiguously its functional purpose and its implementation relation to the meta-metaclass “StructuralUnit” denotes the type of it (see role definition “type” in Figure 1).

4 Summary and Conclusion

We have presented a meta-metamodel (the ADL-Kernel) for the definition of architecture description languages. Essentially, the ADL-Kernel provides for each of its instantiations a minimal set of architectural elements sufficient for describing arbitrary software architectures. This hypothesis is based on the assumption that software architecture primarily addresses the human stakeholders in the development process by defining the function of a program before its implementation and – based on the same architectural information – by explaining its function after implementation. This implies that requirements specifiers of different levels of abstraction are part of software architecture including textual and conceptual architectural elements. At the same time, this excludes implementation-specific elements from being part of the ADL-Kernel. On the other hand, all of the UML language elements conformant with our architectural view are represented by the ADL-Kernel.

5 References

[AN05] Arlow, J. and I. Neustadt: UML 2.0 and the Unified Process. 2 ed. 2005: Addison-Wesley-Longman. ISBN-10: 0321321278

[BCK03] Bass, L., P. Clements, and R. Kazman: Software Architecture in Practice. 2003:Addison-Wesley. ISBN-10: 0321154959

[BK03] Björkander, M. and C. Kobryn: Architecting Systems with UML 2.0. IEEE Computer Society Press, 2003. 20(4): p. 57-61.

[Ca07] Carnegie Mellon University: How Do You Define Software Architecture? 2007 [cited 2010 3 April]; Available from: http://www.sei.cmu.edu/architecture/definitions.html.

[HMU01] Hopcroft, J.E., R. Motwani and J.D. Ullman: Introduction to automata theory, languages and computation 2001: Addison-Wesley. ISBN-10: 0201441241

[KGL+06] Konrad, S., Goldsby, H., Lopez, K. Cheng, B.H.C.: Visualizing Requirements in UML Models. in Requirements Engineering Visualization, 2006. REV '06. First International Workshop on 2006. Minneapolis, MN, USA IEEE Computer Soceity.

[Ob03] Object Management Group (OMG). UML 2.0 Superstructure Specification OMG Adopted Specification. 2003 [cited 2010 3 April]; Available from: http://www.omg.org.

[Re03] Reichardt, J.: Semiotical aspects of the grid calculus – Part I: Terminology and models. 2003, University of Applied Sciences, Computer Science Department: Darmstadt. [cited 2010 3 April]; Available from: http://www.fbi.h-da.de/fileadmin/personal/j.reichardt/Sonstiges/semiotics-part-1.pdf

[Re05] Reichardt, J.: Equivalence and refinement problems in the software development process. 2005, University of Applied Sciences, Computer Science Department: Darmstadt. p. 24. [cited 2010 3 April]; Available from: http://www.fbi.h-da.de/fileadmin/personal/j.reichardt/Sonstiges/refinement.pdf

[Re06] Reichardt, J.: Two-dimensional C++. in ACM. 2006. Brighton, United Kingdom: ACM Press New York, NY, USA.

[RJB04] Rumbaugh, J., I. Jacobson, and G. Booch: The Unified Modeling Language Reference Manual. Second Edition ed. 2004: Addison Wesley. 752. ISBN-13: 0-321-24562-8

[Ta02] Tanenbaum, A.S.: Computer Networks. 4 ed. 2002: Prentice Hall International. ISBN-10: 0130384887

[Yu09] Yüksel, H.: Complete and minimal basis for the definition of Architecture Description Languages, in School of Computing and Mathematics. 2009, University of Plymouth: Plymouth-UK. p. 390. PhD-thesis; [cited 2010 3 April]; Available from: http://eurasia-it.de/reports/20091005_Entire_Thesis%20-%20Print%20-%20Version.pdf