An Ontology-based Approach for Modelling Architectural Styles

Claus Pahl[1] and Simon Giesecke[2] and Wilhelm Hasselbring[2]

[1] Dublin City University, School of Computing, Dublin 9, Ireland
cpahl@computing.dcu.ie
[2] University of Oldenburg, Software Engineering Group, D-26111 Oldenburg, Germany
[giesecke|hasselbring]@informatik.uni-oldenburg.de

Abstract. The conceptual modelling of software architectures is of central importance for the quality of a software system. A rich modelling language is required to integrate the different aspects of architecture modelling, such as architectural styles, structural and behavioural modelling, into a coherent framework. We propose an ontological approach for architectural style modelling based on description logic as an abstract, meta-level modelling instrument. Architectural styles are often neglected in software architectures. We introduce a framework for style definition and style combination. The link between quality requirements and conceptual modelling of architectural styles is investigated. The application of the ontological framework in the form of an integration into existing architectural description notations such as ACME and UML-based approaches, and also service ontologies is illustrated.

Keywords: Software architecture modelling, architecture ontology, architectural style, description logics, quality-driven development.

1 Introduction

Architecture descriptions are used as conceptual models in the software development process, capturing central structural and behavioural properties of a system at design stage [1]. The architecture of a software system is a crucial factor for the quality of a system implementation. The architecture influences a broad variety of properties such as the maintainability, dependability or the performance of a system [2]. While architecture description languages (ADLs) exist [3], these are not always suitable to support rich conceptual modelling of architectures [12]. Only a few, such as ACME [2], support the abstraction of architectures into styles and patterns. If formally defined, these can be used to reason about architectures and their properties [7].

We present an architectural style ontology to address this problem, which serves as a modelling basis. Beyond achievements in ACME, we aim to address

- a rich and easily extensible semantic style modelling language,
- operators to combine, compare, and derive architectural styles,

– an independent style language that can be applied to extend existing ADLs with style support.

For all three cases, an ontology-based approach to represent architectural knowledge – here in terms of a description logic, which is an underlying logic of ontology languages – is the ideal formal framework [14]. Our architectural style ontology focuses primarily on static, structural aspects of components and connectors. The terminological level of the ontology provides vocabulary and a type language for architectural styles. Instances of this type language are concrete architecture specifications. The structural modelling of architectures is currently adequately supported [5, 4, 3, 1, 2] and shall therefore not be addressed in this ontological framework.

The determination of an architectural style, based on a given set of quality requirements, should ideally be the first step in software design [11]. We use a description logic to define an ontology for the description and development of software architectures based on architectural styles that consists of

– an ontology to define architectural styles through a type constraint language,
– an operator calculus to relate and combine architectural styles.

Our aim is to present a conceptual, ontology-based modelling meta-level framework for software architectures, that allows the integration of style aspects into existing architectural description languages (ADLs) without an explicit notion of architectural styles.

We introduce the necessary ontology and description logic foundations in Section 2. We then present an ontology-based modelling approach for architectural styles in Section 3. Relating these styles is the focus of Section 4. We discuss possible extensions to deal with composition in Section 5 and relate the modelling approach to quality-driven development in Section 6. The application of the architectural style language is illustrated in Section 7, before discussing related work and ending with some conclusions.

2 Ontologies and Description Logic

Before presenting the style ontology, we introduce the core elements of the description logic language $\mathcal{ALC}$, which is an extension of the basic attributive language $\mathcal{AL}$ [14]. $\mathcal{ALC}$ provides a set of combinators and logical operators that suffices for the style ontology. Ontologies formalise knowledge about a domain (intensional knowledge) and its instances (extensional knowledge). A description logic, such as $\mathcal{ALC}$, consists of three types of basic notational elements.

– *Concepts* are the central entities. Concepts are classes of objects with the same properties. Concepts represent sets of objects.
– *Roles* are relations between concepts. Roles allow us to define a concept in terms of other concepts.
– *Individuals* are named objects.

Individuals can be thought of as constants, concepts as unary predicates, and roles as binary predicates. We can define our language through Tarski-style model semantics based on an interpretation I that maps concepts and roles to corresponding sets and relations, and individuals to set elements [16]. Properties are specified as *concept descriptions*:

- *Basic concept descriptions* are formed according to the following rules: A is an atomic concept, and if C and D are concepts, then so are $\neg C$ (negation), $C \sqcap D$ (conjunction), $C \sqcup D$ (disjunction), and $C \rightarrow D$ (implication).
- Value restriction and existential quantification, based on roles, are concept descriptions that extend the set of basic concept descriptions. A *value restriction* $\forall R.C$ restricts the value of role R to elements that satisfy concept C. An *existential quantification* $\exists R.C$ requires the existence of a role value.
- Quantified roles can be composed, e.g. $\forall R_1.\forall R_2.C$ is a concept description since $\forall R_2.C$ is one.

These combinators can be defined using their classical set-theoretic interpretations. Given a universe of values $\mathcal{S}$ of values, we define the model-based *semantics of concept descriptions* as follows[1]:

$$\begin{array}{ll} \top^I & = \mathcal{S} \\ \bot^I & = \emptyset \\ (\neg A)^I & = \mathcal{S} \backslash A^I \\ (C \sqcap D)^I & = C^I \cap D^I \\ (\forall R.C)^I & = \{a \in S \mid \forall b \in S.(a,b) \in R^I \rightarrow b \in C^I\} \\ (\exists R.C)^I & = \{a \in S \mid \exists b \in S.(a,b) \in R^I \wedge b \in C^I\} \end{array}$$

An *individual* x defined by $C(x)$ is interpreted by $x^I \in \mathcal{S}$ with $x^I \in C^I$. Structural subsumption is a relationship defined by subset inclusions for concepts and roles.

- A *subsumption* $C_1 \sqsubseteq C_2$ between two concepts C_1 and C_2 is defined through set inclusion for the interpretations $C_1^I \subseteq C_2^I$.
- A *subsumption* $R_1 \sqsubseteq R_2$ between two roles R_1 and R_2 holds, if $R_1^I \subseteq R_2^I$.

Structural subsumption (subclass) is weaker than logical subsumption (implication), see [14]. Subsumption can be further characterised by axioms such as the following for concepts C_1 and C_2: $C_1 \sqcap C_2 \sqsubseteq C_1$ or $C_2 \rightarrow C_1$ implies $C_2 \sqsubseteq C_1$. $C_1 \doteq C_2$ represents equality.

3 Modelling Architectural Styles

3.1 Basic Architectural Style Ontology

The $\mathcal{ALC}$ language shall now be used to define an architectural style ontology, providing a type and constraint language. The central concepts in this ontology are configuration, component, connector, role, and port types – all of which

[1] Combinators $\sqcap$ and $\rightarrow$ can be defined based on $\sqcup$ and $\neg$ as usual.

are derived from a general concept called an architectural type that captures all architectural notions. These are the elementary architectural types. Components and connectors are at the core of style definitions. Components encapsulate computation and connectors represent communication between the components. Components can communicate through ports. Connectors connect to other components through connectors via their ports, where each port plays a specific role in the context of a connector. Often, a provided and a required port interface is distinguished to add a direction to connectors. Configurations are compositions of components and connectors with their ports and roles.

This vocabulary consisting of five elements needs to be constrained in the ontology in order to ensure the desired semantics:

$$ArchType \sqsubseteq Component \sqcup Connector \sqcup Role \sqcup Port \sqcup Configuration$$

and

$$\begin{array}{ll} Configuration & \doteq \exists hasPart.(Component \sqcup Connector \sqcup Role \sqcup Port) \\ Component & \doteq ArchType \sqcap \exists hasInterface.Port \\ Connector & \doteq ArchType \sqcap \exists hasEndpoint.Role \end{array}$$

The roles *hasPart*, *hasEndpoint* and *hasInterface* are part of the basic vocabulary. This vocabulary of types can be extended to add further elements using the same mechanisms based on subsumption and concept descriptions.

3.2 Defining Architectural Styles

Defining architectural styles is actually done by extending the basic vocabulary of elementary architectural types. The subsumption relationship serves to introduce specific types that form an architectural style.

The Pipe-and-Filter Architectural Style. The specification of architectural styles shall be illustrated using the pipe-and-filter style. We start with an extension of the hierarchy of elementary architectural types in order to introduce style-specific components and ports:

$$\begin{array}{ll} PipeFilterComponent & \sqsubseteq Component \\ PipeFilterPort & \sqsubseteq Port \end{array}$$

These new elements shall be further detailed and restricted to express their connector semantics. Three types of pipe-filter components, *DataSource, DataSink* and *Filter*, shall be distinguished. Their respective connectivity through input and output ports is defined as follows:

$$\begin{array}{lll} DataSource & \doteq & \leq 1\ hasPort \sqcap \exists hasPort.Output \\ DataSink & \doteq & \leq 1\ hasPort \sqcap \exists hasPort.Input \\ Filter & \doteq & = 2\ hasPort \sqcap \exists hasPort.Input \sqcap \exists hasPort.Output \end{array}$$

DataSource, DataSink, and Filter are defined as components of a pipe-filter architectural style. Each of these components is characterised through the number and types of component ports using so-called predicate restrictions on a numerical domain ($\leq n$ and $= n$ are used to express $hasPort.(n|n \leq 1)$ for instance) and the usual concept descriptions. In addition to these more structural conditions that define the connections between the component types, a number of classification constraints shall be formulated that further refine the initial enumeration of pipe-filter components by describing how subtype classification is applied.

- *Disjointness* requires the individual components to be truly different:

$$DataSouce \sqcap DataSink \sqcap Filter = \bot$$

- *Completeness* requires pipe-filter components to be made up of only the three specified types:

$$PipeFilterComponent \doteq DataSource \sqcup DataSink \sqcup Filter$$

The Hub-and-Spoke Architectural Style. In addition to the well-known pipe-and-filter style [7, 2], we introduce another architectural style, the hub-and-spoke style. This style abstracts a system that manages a composition from a single location, the hub, which is normally the participant initiating the composition. The composition controller (the hub) is usually remotely accessed by the participants (the spokes). This is the most popular and usually default distribution configuration for service compositions. We would specify:

$$Hub \sqsubseteq Component \quad \text{and} \quad Spoke \sqsubseteq Component$$

with suitable completeness and disjointness constraints.

$$Hub \doteq \exists hasPort.Input \quad \text{and} \quad Spoke \doteq \exists hasPort.Output$$

explains that hubs receive incoming requests from spokes. Further constraints would limit the number of hubs to one, whereas spokes can be instantiated in any number.

3.3 Architectural Styles and Architecture Modelling

So far, we have addressed specifications of architectural properties at the architectural type level. These specifications are constraints that apply to concrete architecture descriptions formulated using the defined architectural types. The question is how these type-level specifications are applied to act as architectural styles. An instantiation of these type-level properties, i.e. an architecture, could be described by instantiating the elementary types only, fully ignoring any style-specific constraints. Thus, a specification of architectural properties is not what we would commonly see as an architectural style. The configuration type matches what an architectural style needs to express. It defines a specific vocabulary of components and other elements and their constraints. Therefore, we define an architectural style to be a subtype (subsumption) of the configuration type.

$$\begin{aligned} PipeFilterStyle &\sqsubseteq Configuration \\ PipeFilterStyle &\doteq \exists hasPart.(PipeFilterComponent \sqcup PipeFilterConnector \\ &\quad \sqcup Role \sqcup Port) \end{aligned}$$

is, together with related concept descriptions, a style definition. What clearly identifies a style is the configuration subtype that acts as a root of the style definition. An architecture description conforming to an architectural style is a subtype of the defined style configuration, e.g. *PipeFilterStyle*. All elements linked to the style (or its subtypes) directly or transitively through *hasPart* and the other predefined roles can be used to describe an architecture.

A distinguishing property of our approach is that the basic architecture vocabulary with notions like component or connector is defined with the same mechanism at the same layer as the architectural styles. The basic architectural style ontology itself is consequently an architectural style, albeit an abstract and unconstraining one – with the trivial equality as the required subsumption.

The ontology and the styles defined based on the ontology aim to provide a type language for architecture definitions. Components in an architecture definition are instances of the elements of an architectural style. In terms of description logics, the architecture elements are instances of the concepts that define an architectural style. The style constrains the use of the architecture elements. This architecture layer – the instances layer in terms of our ontology – shall not be addressed in terms of our framework. Instead we will demonstrate how this framework is independent of specific ADLs and can be applied to them as a style sublanguage in Section 7. Our aim is not to define yet another ADL.

4 Relating Architectural Styles

Each architectural style is defined by a separate specification as an extension of the basic ontology of elementary architecture elements. In order to reuse architectural styles as specification artefacts, these styles are often related to each other, e.g. to be compared to each other or to be derived from another [21]. Different styles can be related based on ontology relationships. We give an overview of the central operators restriction, union, intersection and refinement and define the semantics of this operator calculus. Instead of general ontology mappings, we introduce a notion of style specification and define style comparison and development operators on it.

4.1 Style Syntax and Semantics

Before defining the operators, the notions of architecture specification and styles and their semantics need to be made more precise. We assume a style to be a specification $Style = \langle \Sigma, \Phi \rangle$ based on the elementary type ontology with

– a signature $\Sigma = \langle C, R \rangle$ consisting of concepts C and roles R,
– concept descriptions $\phi \in \Phi$ based on Σ.

Style is interpreted by a set of models M. The model notion [16] refers to algebraic structures that satisfy all concept descriptions ϕ in Φ. The set M contains algebraic structures $m \in M$ with

– sets of objects C^I for each concept C and
– relations $R^I \subseteq C_i^I \times C_j^I$ for all roles $R : C_i \rightarrow C_j$

such that m satisfies the concept description. This satisfaction relation is as usual defined inductively over the connectors of the description logic $\mathcal{ALC}$.

The combination of two styles should be conflict-free, i.e. semantically, no contradictions should occur. A *consistency* condition can be verified by ensuring that the set-theoretic interpretations of two styles S_1 and S_2 are not disjoint, $S_1^I \cap S_2^I \neq \emptyset$, i.e. their combination is satisfiable and no contradictions occur.

Note, that this calculus of operators is not strictly an algebra in terms of styles – only in terms of specifications. A resulting specifications can be defined as a style by identifying a new root configuration.

4.2 Restriction

While often architectural styles are used as-is in combinations and relationships, it is sometimes desirable to focus on specific parts, before for instance refining an architectural style. Restriction is an operator that allows architectural style combinations to be customised and undesired elements (and their properties) to be removed. A *restriction*, i.e. a projection or view, can be expressed using the restriction operator $\langle \Sigma, \Phi \rangle_{|\Sigma'}$ for a specification, defined by

$$\langle \Sigma, \Phi \rangle_{|\Sigma'} \overset{\text{def}}{=} \langle \Sigma \cap \Sigma', \{\phi \in \Phi \mid \mathit{rls}(\phi) \in \mathit{rls}(\Sigma \cap \Sigma') \wedge \mathit{cpts}(\phi) \in \mathit{cpts}(\Sigma \cap \Sigma')\} \rangle$$

with the usual definition of role and concept projections $\mathit{rls}(\Sigma) = R$ and $\mathit{cpts}(\Sigma) = C$ on a signature $\Sigma = \langle C, R \rangle$. Restriction preserves consistency as constraints are, if necessary, removed.

4.3 Intersection and Union

Two architectural styles $S_1 = \langle \Sigma_1, \Phi_1 \rangle$ and $S_2 = \langle \Sigma_2, \Phi_2 \rangle$ shall be assumed.

– The *intersection* of S_1 and S_2, expressed by $S_1 \cap S_2$, is defined by

$$S_1 * S_2 \overset{\text{def}}{=} \langle \Sigma_1 \cap \Sigma_2, (\Phi_1 \cup \Phi_2)|_{\Sigma_1 \cap \Sigma_2} \rangle$$

Intersection is semantically defined based on an intersection of style interpretations, achieved through projection onto common signature elements.

– The *union* of S_1 and S_2, expressed by $S_1 \cup S_2$, is defined by

$$S_1 + S_2 \overset{\text{def}}{=} \langle \Sigma_1 \cup \Sigma_2, \Phi_1 \cup \Phi_2 \rangle$$

Union is semantically defined based on a union of style interpretations.

In the case of fully different architectural styles, their intersection results in the elementary architecture types and their properties. Both operations can result in consistency conflicts.

4.4 Refinement

Consistency is a generic requirement that should apply to all combinations of architecture ontologies. A typical situation is the derivation of a new architectural styles from an existing one [9]. The *refinement* operator that we are going to introduce is a consistent derivation. Refinement can be linked to the subsumption relation and semantically constrained by an inclusion of interpretations, i.e. the models that interpret a style. Refinement carries the connotation of preserving existing properties, for instance the satisfiability of the original style specification. In this terminology, the pipe-and-filter style is actually a refinement of the basic architectural type vocabulary. As the original types are not further constrained, the extension is consistent.

An explicit consistency-preserving refinement operator shall be introduced to provide a constructive subsumption variant that allows

– new subconcepts and new subrelationships to be added,
– new constraints to be added if these apply consistently to the new elements.

Assume a style $S = \langle \Sigma, \Phi \rangle$. For any specification $\langle \Sigma', \Phi' \rangle$ with $\Sigma \cap \Sigma' = \emptyset$, we define a *refinement* of S by $\langle \Sigma', \Phi' \rangle$ through

$$S \oplus \langle \Sigma', \Phi' \rangle \overset{\text{def}}{=} \langle \Sigma + \Sigma', \Phi + \Phi' \rangle$$

The precondition $\Sigma \cap \Sigma' = \emptyset$ implies $\Phi \sqcap \Phi' = \perp$, i.e. consistency is preserved. In this situation, existing properties of $S = \langle \Sigma, \Phi \rangle$ would be inherited by $S \oplus \langle \Sigma', \Phi' \rangle$. Existing relationships can in principle be refined as long as consistency is maintained – which might require manual proof in specific situations that go beyond the operator-based application.

4.5 Architectural Style Development

The main aim of these operators is to support the development of architectural styles. We imagine a catalogue of styles that is used by the software architect to describe architectures.

– The operator calculus allows individual styles from the catalogue to be compared. For instance, two styles can be united to test if the set of concepts they describe overlap. The consistency condition is used for this test.
– An existing style can be adapted. Refinement allows to add further elements and constraints, making the style more specific. Styles can also be made more general by removing constructs and properties through restriction.

This catalogue could be implemented as a repository.

The hub-and-spoke style shall be extended using the refinement operator. The idea is to add a broker component, which spokes would initially contact and which would assign a hub to them.

$$BrokeredHubSpokeStyle \doteq HubSpokeStyle \oplus \langle \Sigma, \Phi \rangle$$

where the signature Σ is defined by

$$\langle \ \{ \ BrokerComponent, BrokerSpokeConnector, BrokerHubConnector, \\ HubRegistrationRole, SpokeAllocationRole \ \} \ , \ \{ \ \} \ \rangle$$

and the properties Φ are defined by

$$\begin{aligned} BrokerComponent &\doteq HubSpokeComponent \sqcap \exists hasInterface.Port \\ BrokerSpokeConnector &\doteq HubSpokeConnector \sqcap \\ &\quad\ \exists hasEndpoint.SpokeAllocationRole \\ BrokerHubConnector &\doteq HubSpokeConnector \sqcap \\ &\quad\ \exists hasEndpoint.HubRegistrationRole \end{aligned}$$

We would automatically get $BrokeredHubSpokeStyle \sqsubseteq HubSpokeStyle$ as a consequence of the application of the refinement.

5 Composite Elements in Architectural Styles

An explicit support for composition is an important element of conceptual modelling languages. Composition is also central for software architectures. As an extension, we introduce two types of composite elements for architectural style specifications.

5.1 Components

Component hierarchies shall consist of unordered subcomponents, expressed using a component composition operator "$\models$", which adds another dimension to the subsumption-based subtype relationship. An example is $Configuration \models Port$, meaning that a *Configuration* consists of *Port*s as parts. This is actually a reformulation of the previously used *hasPart* relationship. In order to provide this with an adequate semantics, interpretations of configurations would have to be seen as tuple-structured elements.

5.2 Connectors

Connectors can be *process assemblies* that consist of ordered process elements, expressed using a set of process composition operators sequence ";", iteration "!", and choice "+". An example is $C \doteq D; E$, meaning that connector C is actually a process sequence of connectors D and E. This sequence can be semantically defined by requiring $D.in \doteq C.in$, $E.in \doteq E.out$, and $C.out \doteq E.out$ in order to express sequencing dependencies.

5.3 Discussion

Note, that these operators are specific to the respective architecture element. While the structural composition is often sufficient, full process specifications with interaction and data flow elements, however, cannot be expressed in the notational format introduced here. Ontological support for, for instance, the process combinators exists in description logics [14]. While this aspect of composition could not have been investigated here in detail, we felt it important to briefly discuss the benefits and also the potential of ontologies and description logics to provide adequate language support.

6 Quality-Driven Architecture

The use of styles in architecture design implies certain properties of software systems, as these styles are abstractions of successfully implemented systems that are usually easy to understand, to manage, or to maintain [11, 12]. While of course functional properties of components are vital, non-functional quality aspects ranging from availability, performance, and maintainability guarantees to costs are equally important and need to be captured to clearly state the quality-of-service (QoS) requirements. The reliability of a system, the availability of services, and the individual component and overall system performance are often crucial. Links exist between architecture models, that based on the component and connector view allocate function to structure, and QoS properties of these systems [8, 10]. A mere statement of required QoS properties is therefore often not sufficient to actually guarantee these properties. We look at architectural styles to illustrate this point.

6.1 Style-based Quality Description

A catalogue of *architectural styles* or *patterns* [15], consisting of styles such as pipe-and-filter and hub-and-spoke, may be utilised by software architects to build architectures that exhibit some desired quality properties. Each of the styles in the catalogue is associated with certain QoS characteristics, that would be exhibited during the deployment and execution of system compositions. The ISO 9126 standard for software product quality to support the evaluation of software can serve as a starting point here that defines quality attributes and metrics [19, 20].

We illustrate this using an architectural style. Some of the advantages of the hub-and-spoke architectural style in terms of QoS aspects are:

- Composition is easily *maintainable*, as composition logic is all contained at a single participant, the central hub.
- *Low deployment overhead* as only the hub manages the composition.
- Composition can include externally controlled participants. Web service technologies, for instance, would enable the *reuse* of existing service components.

The main disadvantages of this architectural style are:

- A single point of failure at the hub provides *poor reliability* and *availability*.
- A communication bottleneck at the hub results in *restricted scalability*. SOAP messages have considerable overhead for message deserialisation and serialisation.
- The high number of messages between hub and spokes is sub-optimal.

The style ontology can be extended by a quality ontology to capture a vocabulary of quality attributes and corresponding metrics using quality-specific properties.

$$\begin{aligned} \mathit{HubSpokeStyle} \doteq\ & \exists \mathit{hasAdvQual}.(\mathit{Maintainable} \sqcup \mathit{LowOverhead} \sqcup \mathit{Reusable}) \sqcap \\ & \exists \mathit{hasDisadvQual}.(\neg \mathit{Reliable} \sqcup \neg \mathit{Scalable} \sqcup \neg \mathit{Performant}) \end{aligned}$$

Some of these quality concepts are based ISO 9126. Further formalised descriptions such as the association of metrics, for instance in the format $\mathit{Performant} \doteq \exists \mathit{hasMetric}.\mathit{ResponseTime}$, are possible.

6.2 Quality Evaluation

Quality-driven development requires quality attributes to be evaluated and confirmed. The qualities of newly derived styles cannot always be taken for granted. Only through empirical evaluations can these expected qualities be confirmed.

A Goal-Question-Metric (GQM) approach to quality goal evaluation [18], a method which allows metric to be derived from abstract quality criteria, can support this quality evaluation endeavour. Implemented systems can be evaluated using the metrics derived from the quality goals via GQM.

7 Integration with Architecture Description Languages

Our aim is not to define yet another ADL. Our aim is to define a versatile architectural style language that can be combined with existing ADLs for a variety of reasons:

- to semantically define an existing style language and to allow reasoning within this semantic framework,
- to provide an ADL-independent style language that can be added to ADLs that do not have an explicit notion of styles,
- to provide a generic terminological framework into which quality aspects can be integrated.

We will look at ACME to illustrate the first point, at UML to illustrate the second point, and at service ontologies like WSMO to illustrate the third point. The architectural style ontology could be used in the first case to formally define the ACME style language. In the second case, the style ontology could be mapped to MOF, giving it an abstract syntactical definition through MOF. Equally,

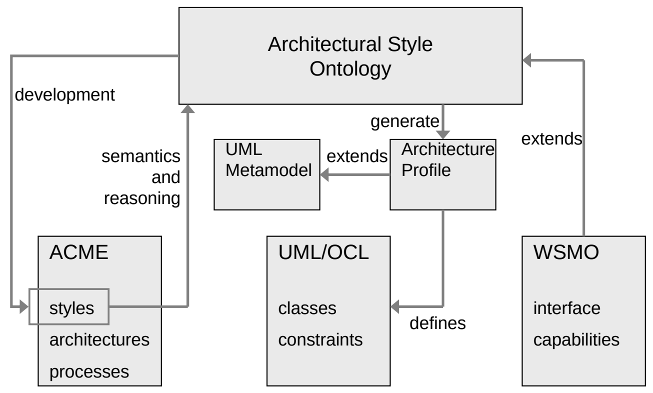

Fig. 1. Application of the Architectural Style Ontology to ACME, UML and WSMO.

an integration with a service ontology such as WSMO or OWL-S is another application of our approach.

We will not fully formalise these mappings for the three applications here – our aim is solely the motivation of these possibilities and the benefits from them. The focus of this paper is only on the definition of the style ontology.

7.1 ACME

ACME is an ADL that supports the component and connector view on architectures [2]. For that purpose, a basic set of architecture elements is introduced. These include the same terms that we have defined. ACME provides specific support to define architectural styles. The basic architecture elements are supported by the corresponding types. A style, called a family in ACME, is then a collection of constrained type definitions. Invariants can be expressed using a constraint language based on properties. Properties in ACME are name-value pairs. ACME does not provide native support for the interpretation these properties.

Our architectural style ontology can provide a standard semantics for properties. Due to the syntactic equality of the elementary types, a mapping from ACME into our ontological framework can easily be defined. The intended semantics of ACME types matches the formal semantics we have introduced here. This has the following benefits for ACME:

– The ACME type language is formally defined through the architectural style ontology.
– A framework for the analysis and reasoning about styles and their properties is introduced.
– The operator calculus enriches the mechanisms to develop architectural styles effectively and consistently.

7.2 UML and OCL

UML is often used to describe software architectures [22]. Class diagrams define components and connections between components through classes and associations. Additional constraints can be added using the Object Constraint Language OCL [23].

In terms of UML, architectural styles are MOF meta-level models, i.e. architectural style definitions correspond to the M2 level. Description logic can be translated to MOF easily, thanks to the Ontology Defintion Metamodel (ODM) [29], which defines a number of MOF-based metamodels including description logics and UML and a range to central transformations between them. This reference framework can be used to translate a given architectural style into a MOF-compliant metamodel. The difficulty here is only that this MOF metamodel is not UML-metamodel compliant. This means that compliance can only be achieved by adapting the standard transformation to define a suitable UML profile. The problem is similar to the need to clearly identify a style and to guarantee its correct application. The profile needs to provide UML-compliant model elements that must only be used in a style-conformant way.

7.3 WSMO

WSMO [24] is, like OWL-S [24], an ontology-based approach to describing services. In the traditional understanding, these two are not ADL [3]. Their aim is to provide a vocabulary that allows the description on functional and non-functional attributes of services and their operations in terms of pre- and post-conditions or quality attributes. Nonetheless, looking at service ontologies helps us to understand how quality attributes, possibly ISO 9126-compliant, can be integrated into an architectural style-driven ADL. This is also one of the reasons for us to use an ontological approach in the first place. Services and their operations are the concepts in WSMO (or OWL-S). Functionality information and quality attributes in WSMO are categorised into interface (syntax) and capability (semantics, quality) attributes and are described in terms of properties in the ontology.

8 Related Work

Formalising architectural styles is the first step of understanding their properties and the resulting impact on architectures and software systems. A seminal paper in this context is [7]. A formal framework based on the model-theoretic specification language Z is given. Abowd et al. introduce the detailed formal specification of architectural styles, e.g. for the pipe-and-filter style. This work has started the integration of semantics into architectural descriptions. The description logic we have used here provides the same expressive power to formulate structural architectural properties (we discuss the behavioural properties addressed by Abowd et al. below). The reason for choosing an ontological approach in our case are pragmatic. An ontological framework for this approach is an ideal choice since extension through subsumption is a natural choice to develop a catalogue of styles. The existence of meta-level frameworks such as the Ontology Definition Metamodel ODM with its predefined transformations makes ontologies and their dynamic logic foundations suitable as an interoperable notation that can be integrated with existing ADLs.

An ontology-based approach is also taken by work addressing service and process ontologies. OWL-S [24] and WSMO [24] are examples for service ontologies, which we have already discussed. WSPO [27, 26] and SWSF [28] are ontological frameworks with a stronger focus on service processes. These are of interest from an architectural perspective as they address service orchestration and choreography as two forms of architectural configuration in the form of component interaction. While we have not addressed this aspect and have rather limited our discussion to more structural properties, an integration of an architectural style ontology with these service ontologies is promising [17, 14].

Around the notion of an architectural style, similar concepts have emerged [6]. In [13], a notion of an architectural scenario is used to aid analyses in the design of architectures. Direct and indirect scenarios are used to view software systems as information processing software artefacts or to view these artefacts as subjects in a change and evolution process, respectively. The dynamic nature of software architectures is emphasised in contrast to the more static view of architectural styles and their application. A similar argumentation is followed by [30]. Associating a system to a single architectural style is often not sufficient. The notion of a mode, similar to a scenario, is introduced. Modes can be changed through structural and evolution constraints, which aims to support the self-organisation of service-based systems.

9 Conclusions

In addition to structural and behavioural properties of software architectures, meta-level constructs such as architectural styles, scenarios, or modes have recently received much interest in the software architecture community. Architectural styles have emerged as architecture abstractions that strongly influence the quality of architectures and their implementations. Our discussion of quality-of-service attributes reflects this observation. Architectural styles are often also linked to platforms; middleware platforms often support only specific styles. In this context, architectural styles help to determine essential aspects of software systems.

Using an ontological, description-logic-based setting for software architecture has a number of benefits, such as a concise and precise notation with formal semantics [7], an extensible type language based on subsumption and constraints [14], and a style combination algebra based on ontology technologies. The tractability of reasoning is a central issue for description logics. The logic $\mathcal{ALC}$ that we have used for this architectural style ontology is decidable, i.e. provides the basis for termination and reliable tool support.

Overall, ontology mechanisms provide an ideal conceptual modelling support, using a classical ontology approach. The notation is adequate, as the examples have demonstrated, to model architectural styles. While the notation is suited to formulate and relate architectural styles focusing on structural aspects, the introduction of composite element has demonstrated the lack of process modelling capabilities in the notation introduced here. Concepts are not meant to model the details of structured behaviour; using concepts to express structured processes is therefore not an adequate solution. While an integration with service or process ontologies is desirable, the seamless integration requires further investigations.

References

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition).* SEI Series in Software Engineering. Addison-Wesley, 2003.
2. David Garlan and Bradley Schmerl. Architecture-driven modelling and analysis. In Tony Cant, editor, *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, volume 69 of *Conferences in Research and Practice in Information Technology*, 2006.
3. N. Medvidovic and R.N. Taylor. A Classification and Comparison framework for Software Architecture Description Languages. In *Proceedings European Conference on Software Engineering / International Symposium on Foundations of Software Engineering ESEC/FSE'97*, pages 60–76. Springer-Verlag, 1997.
4. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
5. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153. Springer-Verlag, Berlin, Sitges, Spain, 1995.
6. C. E. Cuesta, M. del Pilar Romay, P. de la Fuente, and Manuel Barrio-Solorzano. Architectural Aspects of Architectural Aspects. In R. Morrison, B.C. Warboys, and F. Oquendo, editors, *2nd European Workshop on Software Architecture EWSA 2005*. Springer LNCS 3047, 2005.
7. Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.
8. Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, June 1998.
9. L. Baresi, R. Heckel, S. Thöne, and D. Varro. Style-based refinement of dynamic software architectures. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture WICSA4*, pages 155–164. IEEE, 2004.
10. Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. Software performance model-driven architecture. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1218–1223. ACM Press, 2006.
11. Simon Giesecke. A Method for Integrating Enterprise Information Systems based on Middleware Styles. In *International Conference on Enterprise Information Systems (ICEIS'06), Doctoral Symposium*, pages 24–37. INSTICC Press, 2006.

12. Simon Giesecke, Johannes Bornhold, and Wilhelm Hasselbring. Middleware-induced Architectural Style Modelling for Architecture Exploration. In *Proc. Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society Press. 2007.*
13. R. Kazman, S.J. Carriere, and S.G. Woods. Toward a Discipline of Scenario-based Architectural Evolution. *Annals of Software Engineering*, 9(1-4):5–33, 2000.
14. F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003.
15. R. Barrett, L. M. Patcas, J. Murphy, and C. Pahl. Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In *International Conference on Web Engineering ICWE'06. Palo Alto, US.* ACM Press, 2006.
16. D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier, 1990.
17. K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence, Sydney, Australia.* 1991.
18. V. Basili, G. Caldiera, and D. Rombach. The Goal/Question/Metric approach. In *Encyclopedia of Software Engineering, Volume I*, pages 528–532. Wiley, 1994.
19. Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software*, 21(5):88–92, 2004.
20. ISO/IEC. *Software engineering – Product quality – Part 1: Quality model*, June 2001. Published standard.
21. C. Canal, E. Pimentel, and J.M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:105–138, 2001.
22. F. Bachmann, L. Bass, P. Clements, D. Garlan, J. Ivers, J. Little, R. Nord and J. Stafford. *Documenting Software Architecture: Documenting Behavior.* Technical Report CMU/SEI-2002-TN-001. SEI, Carnegie Mellon University. 2002.
23. J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML.* Addison-Wesley, 2003. (2nd Edition).
24. R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.
25. DAML-S Coalition. DAML-S: Web Services Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
26. C. Pahl. An Ontology for Software Component Matching. *International Journal on Software Tools for Technology Transfer (STTT), Special Edition on Component-based Systems Engineering*, 7, 2007. (in press).
27. C. Pahl and M. Casey. Ontology Support for Web Service Processes. In *Proc. European Software Engineering Conference and Foundations of Software Engineering ESEC/FSE'03.* ACM Press, 2003.
28. Semantic Web Services Language (SWSL) Committee. *Semantic Web Services Framework (SWSF).* http://www.daml.org/services/swsf/1.0/, 2006.
29. Object Management Group. *Ontology Definition Metamodel - Submission (OMG Document: ad/2006-05-01).* OMG, 2006.
30. D. Hirsch, J. Kramer, J. Magee, S. Uchitel. Modes for Software Architectures. *Third European Workshop on Software Architecture EWSA 2006, Spinger-Verlag, LNCS Series*, 2006.