Ontology-based modelling of architectural styles

Claus Pahl [a,*], Simon Giesecke [b], Wilhelm Hasselbring [c]

[a] Dublin City University, School of Computing, Dublin 9, Ireland
[b] BTC Business Technology Consulting AG, Kurfüstendamm 33, D-10719 Berlin, Germany
[c] University of Kiel, Software Engineering Group, D-24118 Kiel, Germany

ARTICLE INFO

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. Architectural styles are often neglected in software architectures. We propose an ontological approach for architectural style modelling based on description logic as an abstract, meta-level modelling instrument. We introduce a framework for style definition and style combination. The application of the ontological framework in the form of an integration into existing architectural description notations is illustrated.

© 2009 Elsevier B.V. All rights reserved.

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 [7]. 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 [10]. While architecture description languages (ADLs) exist [22], these are not always suitable to support rich conceptual modelling of architectures [12]. Only a few, such as ACME [10], support the abstraction of architectures into styles or patterns. If formally defined, these can be used to reason about architectures and their properties [1].

We present an architectural style ontology, which serves as a modelling language for formally defined architectural styles and patterns. We address a number of aspects that go beyond ADLs such as ACME in terms of style description

- a rich and easily extensible semantic style modelling language,
- operators to combine, compare, and derive architectural styles,
- a composition technique that incorporate behavioural composition.

The result is an independent style language that can be applied to extend existing ADLs to include style support. For all three as-

pects, an ontology-based approach to represent architectural knowledge – here in terms of a description logic, which is an underlying logic of ontology languages – is a highly suitable formal framework [3]. Ontologies provide modelling and reasoning support for information structured in terms of taxonomies and described in terms of abstract properties.

The modelling of basic structural connectivity of architectures is currently adequately supported [21,2,22,7,10] and shall therefore not be the primary concern in this ontological framework. We use ontologies as a conceptual modelling approach with reasoning support to represent architectural styles in terms of style hierarchy construction and the formulation of architecture concepts and their relationships. Our architectural style ontology focuses primarily on abstractions of structural aspects of components and connectors in the form of styles. 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 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 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

* Corresponding author.
E-mail addresses: cpahl@computing.dcu.ie, Claus.Pahl@dcu.ie (C. Pahl), Simon.Giesecke@btc-ag.com (S. Giesecke), wha@informatik.uni-kiel.de (W. Hasselbring).

languages (ADLs) without an explicit notion of architectural styles. We extend our previous work in this area [26] through an extension of the core ontology by more elaborate development and composition operators (in particular including behaviour in this context to the previous focus on structural aspects) and by providing an extensive application in the ACME context that illustrates its benefits.

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 define an advanced composition approach in Section 5. The application of the architectural style language is illustrated in Section 6. We discuss style-based architectural modelling in terms of applications beyond ACME, quality-driven development, and advanced behavioural specification in Section 7, before reviewing related work in Section 8 and ending with some conclusions in Section 9.

2. Ontologies and description logic

Before presenting the architectural style ontology, we introduce the core elements of the description logic language $\mathscr{ALC}$, which is an extension of the basic attributive language $\mathscr{AL}$ [3]. $\mathscr{ALC}$ has been selected since it is the most simple language that 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 $\mathscr{ALC}$, 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 [18]. Properties are specified as *concept descriptions*:

- *Basic concept descriptions* are formed according to the following rules: A denotes an atomic concept, and if C and D are any (atomic or composite) 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 $\mathscr{S}$, we define the model-based *semantics of concept descriptions* as follows:[1]

$$\top^I = \mathscr{S}$$
$$\bot^I = \emptyset$$
$$(\neg A)^I = \mathscr{S} \setminus A^I$$

$$(C \sqcap D)^I = C^I \cap D^I$$
$$(\forall R.C)^I = \{a \in S | \forall b \in S.(a,b) \in R^I \rightarrow b \in C^I\}$$
$$(\exists R.C)^I = \{a \in S | \exists b \in S.(a,b) \in R^I \land b \in C^I\}$$

An *individual x* defined by $C(x)$ is interpreted by $x^I \in \mathscr{S}$ 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 [3]. 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 $C_2 \sqsubseteq C_1$. The expression $C_1 \equiv C_2$ represents equality.

The concept descriptions can be mapped to a predicate logic, which clarifies the reasoning capabilities of the approach. A concept C can be thought of as a unary predicate $C(x)$ for a variable x and roles R as binary predicates $R(x, y)$, i.e. concept descriptions like $\exists R.C$ are mapped to $\exists y.R(x, y) \land C(x)$.

3. Modelling architectural styles

3.1. The basic architectural style ontology

The $\mathscr{ALC}$ language shall now be used to define an architectural style ontology, thus providing a type and constraint language for ADLs. The central concepts in this ontology are configuration, component, connector, role, and port types – all of which are derived from a general concept called an architectural type that captures all architectural notions. These are the elementary architectural types. The architectural types configuration, component and connector are at the core of style definitions [32]. Ports and roles are used in a range of ADLs such as ACME, Darwin, Wright or AADL. 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. Configurations are compositions of components and connectors with their ports and roles. Often, a provided and a required port interface is distinguished to add a direction to connectors, which can be clarified in terms of roles. Ports enhance component descriptions and roles enhance connector descriptions.

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

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

and

$Configuration \equiv \exists hasPart.(Component \sqcup Connector \sqcup Role \sqcup Port)$
$Component \equiv ArchType \sqcap \exists hasInterface.Port$
$Connector \equiv ArchType \sqcap \exists hasEndpoint.Role$

The roles *hasPart*, *hasEndpoint* and *hasInterface* are part of the basic vocabulary. The *hasPart* role will be defined formally later on. The other two represent structural links from connectors to roles and from components to their ports; they ensure that roles and ports are associated to the core architectural types. This vocabulary of types can be extended to add further elements using the same mechanisms based on subsumption and concept descriptions.

---

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

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.

3.2.1. 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:

$$PipeFilterComponent \sqsubseteq Component$$
$$PipeFilterConnector \sqsubseteq Connector$$
$$PipeFilterPort \sqsubseteq Port$$

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:

$$DataSource \equiv \leqslant 1 hasPort \sqcap \exists hasPort.Output$$
$$DataSink \equiv \leqslant 1 hasPort \sqcap \exists hasPort.Input$$
$$Filter \equiv = 2 hasPort \sqcap \exists hasPort.Input \sqcap \exists hasPort.Output$$

DataSource, *DataSink*, and *Filter* are defined as components of a pipe-and-filter architectural style. We assume Input and Output to be defined as ports. Each of these components is characterised through the number and types of component ports using so-called predicate restrictions on a numerical domain (for instance, $\leqslant n$ and $= n$ are used to express $hasPort.(n|n \leqslant 1)$ for a non-negative integer n) and the usual concept descriptions (such as *hasPort*). 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-and-filter components by describing how subtype classification is applied.

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

$$DataSouce \sqcap DataSink \sqcap Filter \equiv \perp$$

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

$$PipeFilterComponent \equiv DataSource \sqcup DataSink \sqcup Filter$$

Similarly, we can define disjointness $Input \sqcap Output \equiv \perp$ and completeness $PipeFilterPort \equiv Input \sqcup Output$ for ports.

3.2.2. The hub-and-spoke architectural style

In addition to the well-known pipe-and-filter style [1,10], 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. The expressions

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

explain that hubs receive incoming requests from spokes. Further constraints could limit the number of hubs to one:

$$HubSpokeConfiguration \equiv = 1 hasPart.Hub$$

with $HubSpokeConfiguration \sqsubseteq Configuration$, whereas spokes can be instantiated in any number. A standard connector, called *Hub–Spoke*, with $Hub-Spoke \sqsubseteq Connector$ connects hubs and spokes.

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.

$$PipeFilterStyle \sqsubseteq Configuration$$
$$PipeFilterStyle \equiv \exists hasPart.(PipeFilterComponent$$
$$\sqcup PipeFilterConnector \sqcup Role \sqcup Port)$$

This 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. Generally, styles are defined through existential quantification. This is consistent with the aim of supporting the composition and hierarchies of styles. Architectures can belong to several styles.

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 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. The style constrains the use of the architecture elements. This architecture layer – the instances layer in terms of our ontology – shall not be addressed here. Instead we will demonstrate how the framework is independent of specific ADLs in Section 6. It can be applied to general ADLs as a style sublanguage. It is not our aim 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 [8]. Different styles can be related based on ontology relationships. We give an overview of the central operators renaming, restriction, union, intersection and refinement and define the semantics of this oper-

ator calculus. Instead of general ontology mappings, we introduce a notion of a 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 [18] 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,
- 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 defined inductively over the connectors of the description logic $\mathscr{ALC}$ as usual [18,3].

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 specification can be defined as a style by identifying a new root configuration.

4.2. Renaming

Style development might require syntactical elements to be renamed. A *renaming* operator can be defined elementwise for a given signature Σ. By providing mappings for the elements that need to be modified, a new signature Σ' is defined:

$$\Sigma' = \Sigma[n_1 \mapsto n_1', \ldots, n_m \mapsto n_m']$$

for all names of concepts or roles n_i ($i = 1, \ldots, m$) of Σ that need to be modified.

4.3. Restriction

While often architectural styles are used unchanged 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, can be expressed using the restriction operator $\langle \Sigma, \Phi \rangle_{|\Sigma'}$ for a specification, defined by

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

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

4.4. Intersection and union

Adding elements of one style to another (or removing specific style properties from a style) is often required. Union and intersec-

tion deal with these situations, respectively. 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

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

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

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

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.

4.5. 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 [5]. 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

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

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.

4.6. Architectural style development

The main aim of these operators is to support the development of architectural styles. We imagine a catalogue of styles, for example similar to those developed for design patterns, 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.

The hub-and-spoke style, which might be included in the catalogue, 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.

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

where the signature Σ is defined by

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

and the properties Φ are defined by

$$BrokerComponent \equiv HubSpokeComponent \sqcap \exists hasInterface.Port$$
$$BrokerSpokeConnector \equiv HubSpokeConnector$$
$$\sqcap \exists hasEndpoint.SpokeAllocationRole$$
$$BrokerHubConnector \equiv HubSpokeConnector$$
$$\sqcap \exists hasEndpoint.HubRegistrationRole$$

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 three types of composite elements for architectural style specifications.

5.1. Architectural composition principles

Subsumption is usually the central relationship in ontology languages, which allows concept taxonomies to be defined in terms of subtype or specialisation relationships. In the wider context of conceptual modelling, composition is another fundamental relationship that focuses on the part–whole relationship between concepts or objects. In ontology languages, subsumption is well understood and well supported. Composition is less often used in ontological modelling languages [3].

The notion of composition is applied in the context of software architectures in two different ways:

- *Structural composition.* Structural hierarchies of some architectural elements define an important aspect of architectures. Structural composition can be applied to components and configurations.
- *Sequential composition.* Dynamic elements can be composed to represent sequential behaviour. Connectors are usually seen as dynamically oriented architectural elements.
- *Behavioural composition.* Extending the idea of sequential composition, a number of behavioural composition operators including choice and iteration are introduced to describe interaction behaviour.

We use the symbol "▷" to express the composition relationship. Composition is syntactically used in the same way as subsumption "⊑" to relate concept descriptions.

- *Component and configuration hierarchies* shall consist of unordered subcomponents, expressed using the component composition operator "▷". An example is *Configuration ▷ 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, components and configurations are interpreted by unordered multisets.
- *Connectors* can be *sequences* or *complex behaviours* that consist of ordered process elements, again expressed using the composition operator "▷". An example is *Connector ▷ Transformation*, meaning that *Connector* is actually a composite process, which contains for instance a *Transformation* element. We see composite connector implementations as being interpreted as ordered tuples providing a notion of sequence. For more complex behavioural compositions, graphs serve as models to interpret this behaviour.

Composition shall only be applied to components, configurations and connectors. The other architectural elements, i.e. ports and roles, are atomic. Although internal structuring of ports can be imagined by providing operations, hierarchies, as we intend to build them through the composition construct, are not necessary for ports.

5.2. Basic architectural composition

The composition construct is based on the operator "▷". We introduce two basic syntactic forms, before looking at behavioural composition as an extension of sequential composition in the next subsection:

- The *structural composition* between concepts C and D is defined through $C \triangleright \{D\}$, i.e. C is structurally composed of D if $type(C) = type(D) = Component \lor Configuration$.
- The *sequential composition* between concepts C and D is defined through $C \triangleright [D]$, i.e. C is sequentially composed of D if $type(C) = type(D) = Connector$.

Note, that the composition operators are specific to the respective architecture element. We can allow the composition type delimiters, i.e. $\{\ldots\}$ and $[\ldots]$, to be omitted if the type of the part–element D is clear from the context.

This basic format that distinguishes between the two composition types shall be complemented by a variant that allows several parts to be associated to an element in one expression.

- The structural composition $C \triangleright \{D_1, \ldots, D_n\}$ is defined by $C \triangleright \{D_1\} \sqcap \cdots \sqcap C \triangleright \{D_n\}$. The parts $D_i, i = (1, \ldots, n)$ are not assumed to be ordered.
- The sequential composition $C \triangleright [D_1, \ldots, D_n]$ is defined by $C \triangleright [D_1] \sqcap \cdots \sqcap C \triangleright [D_n]$. The parts D_i with $i = (1, \ldots, n)$ are assumed to be ordered with $D_1 \leqslant \cdots \leqslant D_i \leqslant \cdots \leqslant D_n$ prescribing an execution ordering $\leqslant$ on the D_i.

The intended semantics of the two composition operators shall now be formalised. So far, models $m \in M$ are algebraic structures consisting of sets of objects C^I for each concept C in the style signature and relations $R^I \subseteq C^I \times C^I$ for roles R. We now consider objects to be composite:

- Structurally composite concepts $C \triangleright \{D_1, \ldots, D_n\}$ are interpreted as multisets $C^I = \left\{ \left\{ D_1^{I^{i_1}}, \ldots, D_1^{I^{i_k}}, \ldots, D_n^{I^{i_1}}, \ldots, D_n^{I^{i_l}} \right\} \right\}$. We allow multiple occurrences for each concept D_i $(i = 1, \ldots, n)$ that is a part of concept C. With $c \in C^I$ we denote set membership.
- Sequentially composite concepts $C \triangleright [D_1, \ldots, D_n]$ are interpreted as tuples $C^I = \left[D_1^I, \ldots, D_n^I \right]$. Tuples are ordered collections of sequenced elements. In addition to membership, we assume index-based access to these tuples in the form $C^I(i) = D_i^I$ $(i = 1, \ldots, n)$, selecting the ith element in the tuple.

This means that while subsumption as a relationship is defined through subset inclusion, composition relationships are defined through membership in collections (multisets for structural composition and tuples for behavioural composition).

5.3. Behavioural composition

The introduction of behavioural specification depends in our approach on the composition operator applied to connectors. This operator allows us to refine a connector and specify detailed behaviour. While a basic from of behaviour in the form of sequencing has been defined above, we now introduce a more comprehensive approach that requires a more complex semantic model (graphs).

Connectors were originally defined as atomic concepts, explained in terms or their endpoints: $Connector \equiv \exists hasEndpoint.Role$ where $Role$ refers to a component. We now define a connector C through a behavioural specification: $C \triangleright [B]$ where B is a behavioural expression consisting of

- a basic connector C or
- a unary operator '!' applied to a behavioural expression $!B$, expressing *iteration*, or
- a binary operator '+' applied to two behavioural expressions $B_1 + B_2$, expressing *non-deterministic choice*, or
- a binary operator ';' applied to two behavioural expressions $B_1 ; B_2$, expressing the previously introduced *sequencing*.

In line with the basic forms of composition:

- the iteration $C \triangleright [!B]$ is defined by $C \triangleright [B, \ldots, B]$
- the choice $C \triangleright [B_1 + B_2]$ is defined by $C \triangleright [B_1] \sqcup C \triangleright [B_2]$
- the sequence $C \triangleright [B_1 ; B_2]$ is defined as in Section 5.2

We extend the semantics by interpreting behaviourally composite connectors through graphs (N, E) where connectors are represented by edges $e \in E$ and nodes $n \in N$ represent connection points for sequence, choice and iteration. The three operators are defined through simple graphs: $(\{n_1, n_2\}, \{(n_1, n_2)\})$ for a sequence, $(\{n\}, \{(n, n)\})$ for an iteration, and for choice we define $(\{n_1, n_2, n_3, n_4\}, \{(n_1, n_2), (n_2, n_4), (n_1, n_3), (n_3, n_4)\})$.

5.4. Modelling with architectural composition

The composition relationship replaces the previous *hasPart* predicate. As *hasPart* was only informally defined, this formalisation through $\triangleright$ provides a more sound and rigorous definition of the style ontology. These definitions prescribe properties of the respective elements to provide enhanced built-in support in the architecture ontology for architecture-specific modelling tasks. The formal definition allows to check for instance the consistency of compositions in terms of the types of the constituent parts.

We can now replace the previous definition of *Configuration* in the based style ontology

$$Configuration \equiv \exists hasPart.(Component \sqcup Connector \sqcup Role \sqcup Port)$$

by the equivalent, formally defined

$$Configuration \triangleright \{Component, Connector, Role, Port\}$$

Similar to previous definitions, disjointness or completeness properties are not entailed.

Structural compositions allow multiple occurrences of instances of each component element. This can, however, be restricted using the predicate restrictions as discussed earlier. Predicate restrictions can be combined with composition. For instance,

$$SimpleStyle \triangleright \leqslant 5\{Component\}$$

would limit the number of component types in a simple style to 5 – other types are not constrained.

Behavioural composition can express simple interaction protocols that connects implement:

$$InteractProcess \triangleright [LogIn; !(Activity_1 + Activity_2); LogOut]$$

which describes an interaction process along a connector between two components consisting of a login and iteratively selecting one of two possible activities, before logging out.

6. Integration with architecture description languages

Our objective is not to define yet another ADL. Instead, we aim 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 about style refinement, style instantiation and composition 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,

We have summarised some possible application scenarios in Fig. 1. We discuss the architectural description language ACME in this section to illustrate the benefits. We look at ACME (and ACME Studio as its supporting development environment) in more detail to demonstrate the applicability of our formal framework in this important scenario. We use the architectural style ontology to formally define the ACME style language. Later on, in the discussion section, we also look at UML and at service ontologies like WSMO.

6.1. ACME and architectural styles

ACME is an ADL that supports the component and connector view on architectures [10]. For that purpose, a basic set of architecture elements is introduced. These include the same five terms that we have defined as the core vocabulary of our style ontology. ACME provides specific support to define architectural styles. The basic architecture elements such as component or connector are supported by a type language that introduces these a basic 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 and invariants. Our style ontology provides a formal reasoning framework through its underlying description logic.
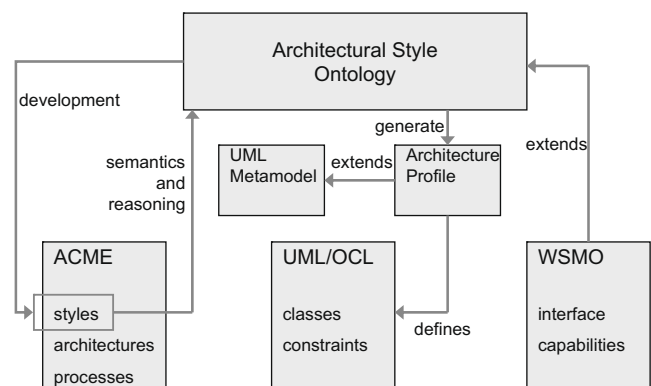

Fig. 1. Application of the architectural style ontology to ACME, UML and WSMO.

Our architectural style ontology can provide a standard semantics for ACME styles. Due to the syntactic equality of the elementary types in the style vocabulary, 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 for ACME and its support environment.

The architectural styles can be defined using an ontology editing tool such as Protégé. The styles can be exported into ADML, the Architecture Description Markup Language, which a standard XML-based mark-up language for describing software and system architectures that can be imported by ACME Studio. Both tools are open platforms and allow the integration of plugins.

6.2. Ontological definition of ACME architectural style elements

An architectural style (family) in ACME consists of component type definitions containing ports and related properties and connector type definitions containing roles and related properties plus invariants. A system in ACME is an architecture specification, which instantiates a family.

ACME families can be directly defined in terms of our architectural style ontology. The basic vocabulary elements that we have introduced in the ontology – configuration, component, connector, port, role – are motivated by ACME and directly reflect the intended ACME semantics. Thus, ACME elements can be formally defined in terms of their architectural style ontology (ASO) counterparts:

$$configuration_{ACME} ::= configuration_{ASO}$$
$$component_{ACME} ::= component_{ASO}$$
$$connector_{ACME} ::= connector_{ASO}$$
$$part_{ACME} ::= part_{ASO}$$
$$role_{ACME} ::= role_{ASO}$$

ACME Studio imports a style ontology as an architectural type using ADML as the interchange format.

6.3. Ontological style-based ACME architecture specification

The following ACME specification describes an integration architecture for application services *AS* in a heterogeneous environment, which process data from different sources (data source providers *DS*) and which require the data consumed by them to be mediated by a separate, central mediation engine *ME*. The mediator *ME* is an intermediary between application services as data consumers and data servers as data providers whose aim is the integration of data formats. The specification consists of components, connectors and attachments. The attachments associate component ports with the respective connectors and the roles they play.

```
System IntegrationArchitecture : HubSpokeStyle = {
    Component AS : Spoke = {
        Ports {In, Out}   };
    Component ME : Hub = {
        Ports {AS-In, AS-Out, DS-In, DS-Out}   };
    Component DS : Spoke = {
```

```
        Ports {In, Out}   };
    Connector AS–ME : Hub–Spoke = {
        Roles {requestIntegr, provideIntegr}   };
    Connector ME–DS : Hub–Spoke = {
        Roles {requestData, provideData}   };
    Attachments = {
        AS.Out to AS–ME.requestIntegr; ME.AS-In to AS–ME.requestIntegr;
        ME.AS-Out to AS–ME.provideIntegr; AS.In to AS–ME.provideIntegr;
        ME.DS-Out to ME–DS.requestData; DS.In to ME–DS.requestData;
        DS.Out to ME–DS.provideData; ME.DS-In to ME–DS.provideData}
}
```

This specification instantiates the *HubSpokeStyle* defined earlier on in Section 3.2.2. The mediator engine *ME* is defined as the central hub and the application services and the data servers are the spokes that all communicate with the mediation hub. As a consequence of applying the style, for instance the uniqueness property of the mediator (there can only be one in an architecture implementation) is automatically inherited through instantiation.

Hub-and-spoke architectures are often implemented in the form of services. As part of our development environment, we have implemented a transformation tool that converts ACME architectures into Web service implementations by creating WSDL service descriptions and executable WS-BPEL service processes.

It should be noted that an architecture is often not uniquely associated to a particular style, such as the association of the *IntegrationArchitecture* system above to the hub-and-spoke style. For instance, we can also identify the pipe-and-filter style in the architecture. Data server *DS*, mediation engine *ME* and application service *AS* can act as source, filter and sink, respectively. While we feel the hub-and-spoke structure is the primary architectural characteristic of the integration architecture, other characteristics such as pipe-and-filter aspects could be modelled by refining the hub-and-spoke style. We will illustrate this principle in Section 6.5.

6.4. Ontological reasoning for ACME architectural styles

As ACME does not provide any native support for its property and invariant sublanguages, the primary practical benefit of the formal definition of ACME styles in Section 6.2 is that reasoning about properties is now defined and enabled within the language.

In general, architectures inherit properties from the style specifications they are derived from:

- Structural aspects such as disjointness and completeness properties can be inherited and do not need to be specified explicitly.
- Logical properties described in styles can be verified in order for benefits to be guaranteed. The uniqueness of the hub is an example.

An ACME invariant

$$Forall\ r : Role | Type(r, Provider)$$

that constrains a connector role to be of a particular type, *Provider*, is defined by its equivalent description logic formula in the architectural style ontology

$$Role \equiv \forall Type.Provider$$

that uses a *Type* property to associate a *Provider* concept. The role variable *r* is implicit in the ontology formulation.

While this only formalises the definition of ACME invariants and enables automated reasoning, more advanced forms of reason-

ing are also possible if our style ontology is extended. The consistency of data processing can be verified, if connectors for instance provide data type information for source and provider. This can be used to verify the correctness of connections in terms of data that is provided and required. The notion of connectors and ports would need to be extended to capture data aspects in a type concept – predefined data property roles in the ontology would serve this purpose. Protégé plugins such as JessTab and SWRLTab allow rules to be defined and executed in the respective languages (Jess, SWRL) that implement the style constraints.

The potential of reasoning could even be extended, if, as discussed later in Section 7.2, process-like behavioural connector specifications is possible. Then, reasoning about connector behaviour in a modal logic style would be possible [27,25].

6.5. ACME architectural style development

We have used the hub-and-spoke style to provide a type language for the specification of the integration architecture system above. This architecture specification itself describes architectural properties common to a large number of systems, in this case mediator architectures [31]. This would merit a representation of the integration architecture as a separate architectural style. We develop this style by refining the hub-and-spoke style.

$$IntegrationArchitectureStyle \equiv HubSpokeStyle \oplus (\Sigma, \Phi)$$

where signature Σ and semantic properties in Φ have to be determined by the software architect. This exercise shall illustrate the benefit of formally supported architectural style development using our calculus for an ADL such as ACME.

Overall, the hub-and-spoke style shall be refined into an integration architecture style (using the refinement operator) as follows:

- Renaming of some of the hub-and-spoke elements to reflect the specific context of information integration: the hub is replaced by the mediation engine and the application servers are spokes.
- Addition of two new components, both of which are spokes and therefore do not violate the uniqueness constraint for the hub that is inherited using the refinement operator:
  - *DS* as a spoke, which represent the data servers already used in the ACME architecture specification in Section 6.3,
  - *IE* as a spoke, which is the integration engine that separates the actual execution of the integration from the coordination of the mediation in *ME* – this component is an extension of the previous ACME specification.
- Addition of new connectors and connectivity (the latter in the form of attachments), in particular to connect the newly added components using the connectors *ME–DS* and *ME–IE*, whereby the integration engine *IE* should be an extension of the hub functionality, here expressed using a uniqueness property on the connector between *ME* and *IE*.
- Addition of new (invariant) disjointness and completeness properties.

In the first step, renaming would be carried out using

$$\Sigma' = \Sigma[Hub \mapsto ME, Spoke \mapsto AS]$$

The extension element (Σ', Φ) for the refinement operator, after renaming hub and spoke, would comprise the signature Σ':

$$\{ \quad Components = \{DS, IE\},$$
$$Connectors = \{ME - DS, ME - IE\} \quad \}$$

and the concept descriptions Φ:

$$\{ \quad instances(ME - -IE) \leqslant 1$$
$$AS \sqcup ME \sqcup DS \sqcup IE = IntegrationArchitecture,$$
$$AS \sqcap ME \sqcap DS \sqcap IE = \emptyset \quad \}$$

We assume here a built-in operator *instances* to formulate the uniqueness property for the *ME–IE* connector. Consistency is here preserved and we would get $IntegrationArchitectureStyle \sqsubseteq HubSpokeStyle$.

Using the refinement operator $\oplus$, the properties of the original hub-and-spoke style are preserved in the extension. The result of the extension – here expressed as a family in terms of the ACME notation – is the following:

> Family IntegrationArchitectureStyle = {
>   Component Type AS = {
>   Ports {In, Out} };
>   Component ME = {
>   Ports {AS-In, AS-Out, DS-In, DS-Out, IE-In, IE-Out} };
>   Component DS = {
>   Ports {In, Out} };
>   Component Type IE = {
>   Ports {ME-In, ME-Out} };
>   Connector AS–ME = {
>   Roles {requestInt, provideInt} };
>   Connector ME–DS = {
>   Roles {requestData, provideData} };
>   Connector Type ME–IE = {
>   Roles {requestTrans, provideTrans};
>   Invariant {instances(ME–IE) ≤ 1} };
>   Invariant {
>   AS ⊔ ME ⊔ DS ⊔ IE = IntegrationArchitecture
>   AS ⊔ ME ⊔ DS ⊔ IE = ∅}
> }

Clearly, this formulation resembles the earlier ACME architecture specification as a system, only at the style level with further constructs and semantical constraints added.

We also demonstrate now how the composition relationship ▷ can be used in this context. In order to detail the architectural style even further, one of the components, the integration engine *IE* can be presented as a composed component consisting of a connector generator *CG*, which handles the communication with the mediation engine, and an execution engine *XE*:

$$IE \rhd \{CG, XE\}$$

The new components *CG* and *XE* would be defined as follows

> Component Type CG = {
>   Ports {ME-In, ME-Out, XE-In, XE-Out} };
> Component Type XE = {
>   Ports {In, Out} };
> Connector Type ME–CG = {
>   Roles {requestTrans, provideTrans};
>   Invariant {instances(ME–DS) ≤ 1} };
> Connector Type CG–XE = {
>   Roles {requestExec, provideExec};
>   Invariant {instances(CG–XE) ≤ 1} }

6.6. Summary

In this section, we have applied our style ontology to ACME to demonstrate the benefits such as

- giving formal semantics to previously only informally defined style languages,

- using enhanced reasoning capabilities arising from the formal ontology framework, and
- developing a rich style catalog for architecture modelling.

ACME acts here as prototypical example of an ADL. Based on a review of ACME families (the ACME term for styles), we have formalised ACME's family sublanguage using our style ontology. Then, we demonstrated the application of the ontology-based style language in ACME. We illustrated the benefits of formally defined property and invariant sublanguages for (ontology-based) reasoning. Finally, we showed how a range of predefined styles can be developed using the style combinators we introduced. Behavioural composition will be addressed in the next section in the context of UML as the application language.

7. Discussion – language extensions and applicability

The architectural style framework we introduced consists of a core ontological style description language, a style development operator calculus and a composition technique. The central property an evaluation needs to establish about our framework is its suitability to enhance existing ADLs in terms of the benefits outlined in Section 6.6. We have demonstrated the suitability of the style description and development language by applying it to ACME as a formally defined style sublanguage in the previous section.

In this section, we discuss some other important aspects of our style language – namely extensions in terms of explicit quality links and advanced behavioural composition – in more detail. ACME is only one possible application language. We will briefly look at UML and WSMO as other, non-ADL application languages of our approach in the context of these extensions.

7.1. 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 [12,13]. Non-functional quality aspects ranging from availability, performance, and maintainability guarantees to costs are equally important functional aspects of components and need to be captured explicitly to clearly state the quality requirements. The reliability of a system, the availability of services, and the individual component and overall system performance are often crucial. Links between the styles of architectures and quality properties of these systems have been observed [28,9].

A catalogue of *architectural styles* or *patterns* [6], 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 quality 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 [16,15].

We illustrate this using an architectural style. Some of the advantages of the hub-and-spoke architectural style in terms of quality aspects are [6]:

- 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 suboptimal.

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

$$HubSpokeStyle \equiv \exists hasAdvQual.(Maintainable \sqcup LowOverhead$$
$$\sqcup Reusable) \sqcap \exists hasDisadvQual.(\neg Reliable$$
$$\sqcup \neg Scalable \sqcup \neg Performant)$$

Further formalised descriptions such as the association of metrics, for instance in the format $Performant \equiv \exists hasMetric.ResponseTime$, are possible.

WSMO [19] is, like OWL-S [29], an ontology-based approach to describing services. In the traditional understanding, these two are not ADLs [22]. 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 postconditions or quality attributes. Nonetheless, looking at service ontologies helps us to understand how quality attributes can be integrated into an architectural style-driven ADL. 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. Capability descriptions are similar to our proposal for quality description above.

7.2. Advanced behavioural composition and application to UML

UML is often used to describe software architectures [4]. Class diagrams define components and connections between components through classes and associations. Additional constraints can be added using the Object Constraint Language (OCL).

Architectural styles can be mapped to MOF meta-level models, i.e. architectural style definitions correspond to the M2 level. The elementary architectural types map directly to classes and their associations in UML. Description logic can be translated to MOF easily, thanks to the Ontology Definition Metamodel (ODM) [24], which defines a number of MOF-based metamodels for a range of modelling languages including description logics and UML and a number of 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 necessarily 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.

UML activity diagrams provide a modelling framework to which our behavioural composition can be applied. Sequence, iteration and choice can be represented diagrammatically to express interaction processes between components. However, while modelling behavioural composition as introduced here is often sufficient as our application to ACME demonstrates, full process specifications with interaction and data flow elements, however, cannot be ex-

pressed in the notational format introduced here. Ontological support for the process combinators exists in description logics [3]. While this aspect of composition cannot be investigated here in detail due to the complexity of a comprehensive process composition solution in ontology languages [25], it is important to discuss the benefits and also the potential of ontologies and description logics to provide adequate language support for architectural behaviour modelling.

Connectors can be *processes* that consist of ordered process elements, expressed using process composition operators such as sequence ";", iteration "!", and choice "+". An example is to define connector C as $B_1; B_2$, meaning that connector C is actually a process sequence of connectors B_1 and B_2. While this example can be expressed using our current composition notation $C \rhd [B_1, B_2]$, data flow elements such as parameters are currently not introduced. A different semantic model from the set-theoretic interpretation we have used so far would allow the required semantical support for complex process expressions.

An adequate solution to this problem lies in a different interpretation of behaviour in architectural style definitions. If we consider connectors as behavioural elements in terms of the architecture – components also exhibit behaviour, but are considered here as black boxes – then these connectors can and need to be defined differently in the ontology language. Some proposals exist to interpret computational elements through accessibility relations [3]. This would mean

- to introduce a special form of roles for connectors that model accessibility relations between static constructs with some notion of state [25],
- to provide a rich role expression sublanguage for this new role type consisting of operators such as sequence, iteration or choice together with names to represent data [3].

These behavioural roles would complement the existing roles, which are more descriptional and static in nature. The benefit of this interpretation of behaviour is compatibility of behaviour reasoning with subsumption reasoning, as for instance refinement of behaviour can be expressed through role subsumption.

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 [1]. 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 a highly suitable candidate 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. ODM with its predefined transformations can be used to integrate our style ontology into other modelling languages defined within ODM (such as UML).

Architectural styles have been integrated in some ADLs, such as ACME. Styles can also be considered in managing architectural evo-

lution. In [23], a graph grammar approach is used to capture architectural evolution. The suitability of ontological frameworks here would need to be investigated further. The operator algebra for style development we introduced, however, provides a starting point to control change. Operators such as restriction and union can be used to define elementary changes, on which more complex changes can be described through operator composition.

Around the notion of an architectural style, similar abstractions have emerged. In [17], 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 Hirsch et al. [14]. 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. The benefit is here a higher degree of automation.

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 influence the quality of architectures and their implementations. Architectural styles are often also linked to platforms; middleware platforms often support only specific styles by constraining interaction to synchronous or asynchronous communication or by enforcing a client–server type of architecture. In this context, architectural styles help to determine essential aspects of software systems.

Our approach ties in with current attempts to utilise Semantic Web and ontology technology for software engineering – most prominently within the Ontology Driven Architecture initiative by the W3C. We use ontologies as a mechanism describe and formally defined architectural styles. In terms of model-driven approaches, ontology-based architectural styles can be viewed as abstracted architecture models, which complements previous work [20,30]. The core of the contribution, however, is a foundational framework that carries the specification approach further into a comprehensive development calculus for styles.

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 [1], an extensible type language based on subsumption, composition and constraints [3], 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 [3], i.e. provides the basis for termination and reliable tool support.

In this paper, we have carried past work, such as [26], further to incorporate composition into the development framework. We also added a discussion of quality considerations in the context of architectural styles. A significant extension is also the application of the architectural style ontology presented here to architecture specification and modelling approaches. Our discussion of ACME, which is a recognised and widely used ADL, demonstrates that the style ontology can provide a number of essential benefits, which in the ACME case comprise a formal semantics with the reasoning support that is entailed, but also brings an important dimension to ACME architecture modelling. The development of

styles themselves based on a formalised operator calculus enhances reuse through abstraction. Behavioural composition adds behaviour, which is often neglected in architectural description.

Overall, ontology mechanisms provide a suitable conceptual modelling support, using a classical ontology approach. The notation is adequate, as the examples have demonstrated, to model architectural styles. An ontology approach is also suitable as it provides two intrinsic benefits over other conceptual modelling approaches based on the subsumption relationship:

- firstly, easy extensibility and configurability of the style ontology based on the operator calculus and the subsumption and composition relations,
- secondly, modelling of meta-level style ontology vocabulary and style-specific terms within one modelling layer.

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 advanced process modelling capabilities in the notation introduced here. Concepts are not meant to model the details of data and control flow 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] G. Abowd, R. Allen, D. Garlan, Formalizing style to understand descriptions of software architecture, ACM Transactions on Software Engineering and Methodology 4 (4) (1995) 319–364.
[2] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–249.
[3] F. Baader, D. McGuiness, D. Nardi, P.P. Schneider (Eds.), The Description Logic Handbook, Cambridge University Press, 2003.
[4] F. Bachmann, L. Bass, P. Clements, D. Garlan, J. Ivers, J. Little, R. Nord, J. Stafford, Documenting Software Architecture: Documenting Behavior, Technical Report CMU/SEI-2002-TN-001, SEI, Carnegie Mellon University, 2002.
[5] L. Baresi, R. Heckel, S. Thöne, D. Varro, Style-based refinement of dynamic software architectures, in: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture WICSA4, IEEE, 2004, pp. 155–164.
[6] R. Barrett, L.M. Patcas, J. Murphy, 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.
[7] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., SEI Series in Software Engineering, Addison-Wesley, 2003.
[8] C. Canal, E. Pimentel, J.M. Troya, Compatibility and inheritance in software architectures, Science of Computer Programming 41 (2001) 105–138.
[9] V. Cortellessa, A. Di Marco, P. Inverardi, Software performance model-driven architecture, in: SAC'06: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM Press, 2006, pp. 1218–1223.
[10] D. Garlan, B. Schmerl, Architecture-driven modelling and analysis, in: Tony Cant (Ed.), Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06), Conferences in Research and Practice in Information Technology, vol. 69, 2006.
[11] S. Giesecke, A method for integrating enterprise information systems based on middleware styles, in: International Conference on Enterprise Information Systems (ICEIS'06), Doctoral Symposium, INSTICC Press, 2006, pp. 24–37.
[12] S. Giesecke, W. Hasselbring, M. Riebisch, Classifying architectural constraints as a basis for software quality assessment, Advanced Engineering Informatics 21 (2) (2007) 169–179.
[13] S. Giesecke, J. Bornhold, W. Hasselbring, Middleware-induced architectural style modelling for architecture exploration, in: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society Press, 2007.
[14] D. Hirsch, J. Kramer, J. Magee, S. Uchitel, Modes for software architectures, in: Third European Workshop on Software Architecture EWSA 2006, LNCS Series, Springer-Verlag, 2006.
[15] ISO/IEC, Software engineering – product quality – Part 1: Quality model, June 2001, Published standard.
[16] H.-W. Jung, S.-G. Kim, C.-S. Chung, Measuring software product quality: a survey of ISO/IEC 9126, IEEE Software 21 (5) (2004) 88–92.
[17] R. Kazman, S.J. Carriere, S.G. Woods, Toward a discipline of scenario-based architectural evolution, Annals of Software Engineering 9 (1–4) (2000) 5–33.
[18] D. Kozen, J. Tiuryn, Logics of programs, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, 1990, pp. 789–840.
[19] R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, D. Fensel, Web service modeling ontology, Applied Ontology 1 (1) (2005) 77–106.
[20] C. Pahl, A conceptual architecture for semantic web services development and deployment, International Journal of Web and Grid Services 1 (3/4) (2005) 287–304.
[21] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: W. Schäfer, P. Botella (Eds.), Proceedings of the 5th European Software Engineering Conference (ESEC 95), vol. 989, Springer-Verlag, Berlin, Sitges, Spain, 1995, pp. 137–153.
[22] N. Medvidovic, 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, Springer-Verlag, 1997, pp. 60–76.
[23] D.L. Metayer, Describing software architecture styles using graph grammars, IEEE Transactions on Software Engineering 24 (7) (1998) 521–553.
[24] Object Management Group, Ontology Definition Metamodel – Submission (OMG Document: ad/2006-05-01), OMG, 2006.
[25] C. Pahl, An ontology for software component matching, International Journal on Software Tools for Technology Transfer (STTT), Special Edition on Foundations of Software Engineering 9(2) (2007) 169–178.
[26] C. Pahl, S. Giesecke, W. Hasselbring, An ontology-based approach for modelling architectural styles, in: First European Conference on Software Architecture ECSA 2007, Lecture Notes in Computer Science, vol. 4758, Springer, 2007, pp. 60–75.
[27] K. Schild, A correspondence theory for terminological logics: preliminary report, in: Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney, Australia. 1991.
[28] B. Spitznagel, D. Garlan, Architecture-based performance analysis, in: Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering (SEKE'98), June 1998.
[29] K. Sycara, M. Paolucci, A. Ankolekar, N. Srinivasan, Automated discovery, interaction and composition of semantic web services, Journal of Web Semantics 1 (1) (2003) 27–46.
[30] C. Pahl, Layered ontological modelling for web service-oriented model-driven architecture, in: European Conference on Model-Driven Architecture ECMDA 2005, LNCS, vol. 3748, Springer, 2005, pp. 88–102.
[31] G. Wiederhold, Mediators in the architecture of future information systems, IEEE Computer 25 (1992) 38–49.
[32] R.N. Taylor, N. Medvidovic, E.M. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley, 2009.