Toward Composition Of Style-Conformant Software Architectures

Nikunj R. Mehta and Nenad Medvidovic

Department of Computer Science
University of Southern California
941 West 37th Place, SAL 327, Los Angeles, CA 90089, USA
{mehta,neno}@cse.usc.edu

Abstract. The codification of software architectural decisions made to address recurring software development challenges results in architectural styles. The primary benefit of architectural styles is that properties demonstrated at the level of a style are carried over into the software system architectures constructed using that style. However, in the absence of comprehensive techniques for checking conformance of a software architecture to its style(s), the expected stylistic properties are not always present in the architecture. This paper argues for a need to look beyond the existing formalizations of styles and architectures to construct style-conformant software architectures. The paper proposes a compositional formalization of styles and style-based architectures aimed at ensuring an architecture's conformance to its style(s).

1 Introduction

Architectural styles codify the recurring design practices and successful organizations of software systems. Styles are the composition patterns of and constraints on the computing, interaction, and data elements of the architectures of similar software systems [8]. Styles are considered useful for instituting high-level reuse and bringing economy to the design of software architectures [7]. A number of styles have emerged in research and industry, such as publish-subscribe, client-server, peer-to-peer, and so on. Software development environments that take advantage of stylistic constraints have been studied in the past (e.g., DRADEL [5]), whereas recent research has focused on general-purpose environments for supporting user-defined styles (e.g., AcmeStudio [9]).

A prominent reason for the increasing dependence on styles in the design of software architectures is the observation that a property proven about a style holds true in the architectures based on that style [3]. However, in order to ensure that stylistic properties are preserved in architectures, one must verify the conformance of the architectures to their style(s). As with syntactic and semantic checking in the compilation of a program, style conformance verifies the satisfaction of stylistic constraints in a style-based architecture. Although the formalization of styles enables such conformance checking, little attention has been paid to it thus far. Current approaches and tools, such as AcmeStudio [9], check conformance of an architecture to its style in terms of its topology and structure. However, checking the conformance at the level of data and dynamic behavior is not supported. Neither is the need to compose architectures *hierarchically*, or to use *multiple* styles in an architecture as required by modern software systems.

We feel that the increasing use of styles in the design of modern, large-scale software systems creates a pressing need for supporting style conformance checking of software architectures. In order to be practically useful this checking has to be both incremental and compositional. Existing formalizations of styles (e.g., using Z [1], graph grammar [4], and process algebra [2]) illustrate different means of analyzing them but do not delve into the nature of style conformance. It is in this context that we propose the need to look beyond existing formalizations of styles. Our Alfa project deals with the hierarchical composition of style-based software architectures using architectural primitives [6]. Our experience in this project illuminates a potentially suitable formalization of styles and style-based architectures to comprehensively and compositionally check style conformance.

In the rest of this paper, we summarize the existing research on styles and their formalization, propose a new compositional formalization of styles and style-based architectures, and discuss future work based on this novel formalization.

2 Related Work

Perry and Wolf [8] describe software architectures in terms of three kinds of architectural elements: *processing*, *connecting*, and *data*. As defined by them, an architectural style comprises important constraints on the architectural elements and their relationships. The rest of this section focuses on existing formalizations of styles and their lack of support for checking style conformance comprehensively and compositionally.

In the first formal treatment of architectural styles, Abowd et al. [1] treat a style as a specialized vocabulary with associated semantics. The resulting formal framework allows uniform definition of styles for the purpose of analyzing properties within and among styles. Three syntactic classes are supported for defining software architectures—components, connectors and configuration. Every component and connector is mapped to its stylistic type, but their behaviors are fixed at the level of their style. Further, components and connectors may be composed hierarchically although such composition can only occur within styles and not at the level of architectures. Finally, data is not treated as a first-class architectural element.

Le Metayer has formalized software architectures as graphs and styles as graph grammars [4]. The main focus of this approach is to emphasize the geometry of architectures as an independent object. A coordinator is used to specify valid steps of evolution of an architecture in a manner that is consistent with its style. However, in this approach, the verification of style conformance is an undecidable problem. Furthermore, the resulting algorithm is hard to extend to support an arbitrary degree of hierarchy.

Finally, Bernardo et al. have introduced architectural types as an intermediate formalization between styles and architectures [2]. An architectural type (AT) is a specific instance of a style which allows variation at the level of internal behaviors of components and connectors. ATs are used to bridge the gap between architecture description languages (ADLs) and style formalizations. However, as ATs have a fixed topology, their use restricts the allowed topology of components and connectors over and above that allowed by the AT's style. Finally, neither can ATs be hierarchically composed nor do they support data conformance checks similar to the approach of [1].

3 Formalizing Architectural Styles

In this section, we propose a novel formalization of styles to help remedy the shortcomings of existing approaches. The formalization provides first-class support for data, computing, and interaction elements in an architectural style. A *datum* represents a meta-type of data elements used in a style. A style *constituent,* representing a computing element, contains well defined entry (*input*) and exit (*output*) points, called *portals,* for data and control flow. A *duct* forms the interaction element between two portals. An *interface* groups together related portals whose instances occur atomically in instances of the constituent containing them. We use the *dom* function for the domain of a function, dot (.) operator to perform relational navigation, star (*) as the transitive closure operator, and plus (+) as the reflexive transitive closure operator.

Definition 1. A datum is defined as a node in a meta-type hierarchy. A hierarchical organization is used for the substitution of datum *d* with its successor node datums written as *succ*(d). Moreover, ϕ (the empty set) is the root of this hierarchy and, hence, can be substituted with any datum in the datum hierarchy.

Definition 2. A portal is defined as the triple (λ, δ, behavior) where λ is its identifying label, δ is a finite (possibly empty) set of datums it allows, and $behavior \in \{\text{input}, \text{output}\}$.

Definition 3. A duct is defined as the triple (u, v, B) where u and v are its endpoints, and B is its behavior expressed using labels (λ) of u and v.

Definition 4. A style constituent is the 7-tuple $(\alpha, \iota, \chi, \rho, \mu, BC, SC)$, where:

- α is a finite set of portals
- ι is a finite set of interfaces
- χ is a finite (possibly empty) set of internal constituents
- ρ is a finite (possibly empty) set of internal ducts
- $\mu : A \rightarrow U$ is a bijective function,
  $A \subseteq \alpha$ and $U = \bigcup_{\forall c \in \chi} c \cdot \alpha - \bigcup_{\forall d \in \rho} (d \cdot u + d \cdot v)$
- BC is a finite (possibly empty) set of behavioral constraints expressed using labels (λ) of α and $\chi.\alpha$
- SC is a finite (possibly empty) set of structural constraints expressed using elements of ι and χ

This definition of style constituents allows one to compose a constituent hierarchically from more constituents and ducts. Further, the mapping function μ causes *unsatisfied* portals (U) of internal constituents (χ) to be exposed as portals (α) from a constituent, necessary to enable interaction at those entry/exit points. Additionally, Rule SC1 through Rule SC7 from Figure 1 ensure well-formedness of a style constituent: Rule SC1 disallows repeating labels among distinct portals; Rule SC2 states that an interface is a subset of constituent portals; Rule SC3 disallows overlapping interfaces; Rule SC4 necessitates an interface for each portal; Rule SC5 requires that only portals of internal constituents be the end points of an internal duct; Rule SC6 requires identical behavior for mapped portals; and Rule SC7 ensures compatibility of datums at the ends of a duct.

Definition 5. An architectural style is the 5-tuple $(\delta, \chi, \rho, DC, TC)$, where:

- δ is a finite set of datum (organized as a ϕ-rooted meta-type hierarchy)
- χ is a finite set of constituents
- ρ is a finite set of ducts

- DC is a finite (possibly empty) set of data constraints expressed using elements of δ
- TC is a finite (possibly empty) set of topological constraints expressed using elements of χ+ and χ+.ι

For well-formedness of styles, Rule AS1 and Rule AS2 from Figure 1 require that all datums used in a style be its elements and the datum hierarchy be limited to datums defined in the style, respectively. Additionally, Rule SC5 and Rule SC7 are repeated as Rule AS3 and Rule AS4 respectively (not shown), using the style's constituents and ducts.

4 Formalizing Style-based Architectures

Based on the formalization of stylistic terms, in this section we propose a compositional formalization of software architectural instances. In addition to instances of stylistic elements, we introduce the notion of a *link* as a binary relation between interface instances such that their portal instances are correspondingly connected by a set of duct instances. Links allow reduction of model complexity as individual duct instances are not required for discerning the architectural topology.

Definition 6. A data type, i.e., datum instance, is the pair (n, δ) where n is its name and δ its type, i.e., a style datum.

Definition 7. A portal instance is the 4-tuple (λ_I, δ_I, behavior$_I$, α) where λ_I is the identifying label, δ_I is a finite (possibly empty) set of data types it allows, $behavior_I \in$ {input, output}, and α is its type, i.e., a portal from a style constituent.

Rules for style constituents

Rule SC1 $\forall a_1, a_2 \in \alpha, a_1 \neq a_2 \Rightarrow a_1 \cdot \lambda \neq a_2 \cdot \lambda$

Rule SC2 $\forall i \in \iota, i \subseteq \alpha$

Rule SC3 $\forall i_1, i_2 \in \iota, i_1 \neq i_2 \Rightarrow i_1 \cap i_2 = \phi$

Rule SC4 $\forall a \in \alpha, \exists i \in \iota, a \in i$

Rule SC5 $\forall d \in \rho, \exists \chi_i \in \chi$ such that $d \cdot u \in \chi_i \cdot \alpha, \exists \chi_j \in \chi$ such that $d \cdot v \in \chi_j \cdot \alpha$

Rule SC6 $\forall a \in \alpha, a \in dom(\mu) \Rightarrow a \cdot behavior = \mu(a) \cdot behavior$

Rule SC7 $\forall d \in \rho, \forall a \in d \cdot u, \forall b \in d \cdot v, a \cdot behavior \neq b \cdot behavior$

$a \cdot behavior = input \Rightarrow a \cdot \delta_I \supseteq b \cdot \delta, b \cdot behavior = input \Rightarrow b \cdot \delta \supseteq a \cdot \delta$

Rules for architectural styles

Rule AS1 $\delta \supseteq \bigcup_{\forall a \in \chi^* \cdot \alpha} a \cdot \delta$

Rule AS2 $\forall d \in \delta, succ(d) \in \delta$

Rules for constituent instances

Rule CI1 $C_I \notin \chi_I^*$

Rule CI2 $\forall (i_1, i_2) \in \kappa, i_1 = \{u_I | \forall (u_I, v_I) \in \rho_I, u_I \in i_1\} \cup \{v_I | \forall (u_I, v_I) \in \rho_I, v_I \in i_1\}$

$i_2 = \{u_I | \forall (u_I, v_I) \in \rho_I, u_I \in i_1\} \cup \{v_I | \forall (u_I, v_I) \in \rho_I, v_I \in i_1\}$

Rules for software architectures

Rule SA1 $\forall d_1, d_2 \in \delta_I, d_1 \neq d_2 \Rightarrow d_1 \cdot n \neq d_2 \cdot n$

Figure 1. Well-formedness rules on styles and style-based software architectures

Definition 8. A duct instance is the 4-tuple (u_I, v_I, B_I, ρ) where u_I and v_I are its end-point portal instances, B_I is its behavior in terms of actions performed at u_I and v_I, and ρ is its type, i.e., a duct from a style or its constituents.

Definition 9. A constituent instance, i.e., component or connector, C_I is the 8-tuple $(\alpha_I, \iota_I, \chi_I, \rho_I, \mu_I, \kappa, CB, \chi)$, where

- α_I is a finite set of portal instances
- ι_I is a finite set of interface instances
- χ_I is a finite (possibly empty) set of internal constituent instances
- ρ_I is a finite (possibly empty) set of internal duct instances
- $\mu_I: A \rightarrow U$ is a bijective function, $A \subseteq \alpha_I$ and $U = \bigcup_{\forall c \in \chi_I} c \cdot \alpha_I - \bigcup_{\forall d \in \rho_I} (d \cdot u_I + d \cdot v_I)$
- κ: $I \rightarrow I$ is a bijective function for internal links between *satisfied* interface instances, where $I = \bigcup_{\forall c \in \chi, \forall i \in c \cdot \iota, i \cdot \alpha \notin ran(\mu_I)} i$
- CB is a finite (possibly empty) set of concrete behavior expressions using labels (λ_I) of α_I and $\chi_I.\alpha_I$
- χ is its type, i.e., a constituent from a style or its constituents

For well-formedness of constituent instances, Rule CI1 and Rule CI2 from Figure 1 prohibits cyclical composition and establishes an interface link between appropriate interfaces, respectively. Further, Rule SC1 through Rule SC5 are repeated as instance-level Rule CI3 through Rule CI7 (not shown) by adding a suffix (I) to the style terms. For example, Rule CI3 is written as $\forall a_1, a_2 \in \alpha_I, a_1 \neq a_2 \Rightarrow a_1 \cdot \lambda_I \neq a_2 \cdot \lambda_I$.

Definition 10. A software architecture is the 4-tuple $(\delta_I, \chi_I, \rho_I, \kappa)$, where

- δ_I is a finite set of data types
- χ_I is a finite set of constituent instances
- ρ_I is a finite set of duct instances
- κ: $\zeta_I \cdot \iota_I \rightarrow \chi_I \cdot \iota_I$ is a bijective function for links

For well-formedness of an architecture, Rule SA1 from Figure 1 requires that data type names be not repeated. Moreover, Rule CI2 is repeated as Rule SA2 (not shown) using the architecture's constituents and ducts. Furthermore, Rule SC5 and Rule AS1 are repeated as architecture-level Rule SA3 and Rule SA4 (not shown) by adding a suffix (I) to the style terms.

5 Discussion and Future Work

In our formalization each architectural (data type, portal, duct, and constituent) instance has a stylistic type, as can be noted from definitions 6 through 10. As a result, style conformance of software architectures can be checked by verifying that its architectural instances satisfy the constraints of their individual style element types. Note, however, that our formalization allows the use of multiple styles in a single architecture, and for architectural instances to be composed hierarchically. This allows considerable freedom for the choice of types within a software architectural composition. For example, a client-server system could be designed with the server itself composed of pipes and filters. Some filters, in turn, could be designed to contain virtual machines. Such design choices are not known when defining an architectural style, but are useful in constructing architectures of modern software systems. Our formalization enables composition of components and connectors of a single software architecture from elements of multiple different styles and, therefore, enables checking their conformance to multiple styles.

Furthermore, the stylistic terms in our formalization can be *divided* into five orthogonal dimensions identified in our previous work: structure, behavior, interaction, data, and topology [6]. Therefore, conformance checking can be performed independently on these five aspects. It is our hypothesis that comprehensive style conformance is achieved by the collective conformance against these five dimensions.

Finally, our formalization allows arbitrary notations for expressing dynamic and static constraint expressions for the terms *B*, *SC*, *BC*, *TC*, *DC*, and *CB*. However, informed choices can be made that build upon existing research. Jackson uses first-order logic for constraint satisfaction-based check [3] of static style conformance, suitable for terms SC, TC, and DC. Moreover, Bernardo et al. use process algebra [4], suitable for terms B, BC and CB, which permits language inclusion as a check for dynamic style conformance.

In summary, this paper has discussed the shortcomings of existing approaches for style-based software architectures in checking style conformance. In addition to analyzing styles, the systematic use of styles in constructing software architectures is important and greatly needed. We have proposed a new formalization of architectural styles and software architectures that takes into account compositionality of architectural elements and supports construction of mixed style-based architectures. We believe the formalization also enables comprehensive style conformance checking. Our current efforts are directed towards the identification of such rules of style conformance and their operationalization using existing notations, and verification of style conformance using automated analysis tools.

6 References

[1] Abowd, G. D., Allen, R. J. and Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4): 319-364, 1995.

[2] Bernardo, M., Ciancarini, P., and Donatiello, L. On The Formalization of Architectural Types With Process Algebras. Proc. *ACM Transactions on Software Engineering and Methodology,* 11(4):386-426, 2002.

[3] Jackson, D. Automatic Analysis of Architectural Style. Unpublished Manuscript, MIT Laboratory for Computer Sciences, Software Design Group.

[4] le Metayer, D. Describing architectural styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521-533, 1998.

[5] Medvidovic, N., Rosenblum, D., and Taylor, R. N. A Language and Environment for Architecture-Based Software Development and Evolution. *Proc. ICSE-21*, Los Angeles, California, USA, May 1999.

[6] Mehta, N. R. and Medvidovic, N. Composing architectural styles from architectural primitives. *Proc. ESEC-10/FSE-11*, Helsinki, Finland, September 2003.

[7] Monroe, R. T. and Garlan, D. Style-Based Reuse for Software Architectures. *Proc. ICSR-4*, Orlando, Florida, USA, April 1996.

[8] Perry, D. E. and Wolf, A. L. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 17, 40-52, 1992.

[9] Schemrl, B. and Garlan D. AcmeStudio: Supporting Style-Centered Architecture Development. Unpublished Manuscript, CMU School of Computer Science.