

Concise Composition of Architectural Styles From Architectural Primitives

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, medvidovic}@usc.edu

Abstract. Architectural styles represent composition patterns and constraints at the software architectural level and are targeted at families of systems with shared characteristics. They enable architectural reuse and hence can bring economy to the design of software architecture. Existing approaches support systematic description of style-based software architectures. Our approach, Alfa, focuses on the construction, instead of description, of style-based software architectures using architectural primitives. This is based on our observation that architectural styles and, indeed, software architectures share many underlying concepts that lead to architectural primitives. Previously, Alfa's primitives were shown to be sufficient for modeling architectural styles. In this paper, we present the composition of a diverse set of styles for network-based systems using xAlfa, a systematic notation for composing styles from Alfa's primitives. We then show that two reuse mechanisms in xAlfa, inheritance and composition, enable concise style compositions and unambiguously bring out similarities among architectural styles.

1 Introduction

Software architectures [20,23] have been proposed to address challenges of growing complexity and size of modern, distributed software systems. Software architectures provide high-level abstractions in the form of coarse-grained processing, connecting, and data elements, their interfaces, and their configurations [8]. Architectures provide an additional level of abstraction that, while aiding comprehension and construction of the software system, also requires additional effort for its use. This additional effort can, however, be reduced by a great extent through the use of appropriate mechanisms for specifying software architecture, which is the goal of previous and continuing research on architectural description languages (ADLs) and associated tools [2,7,15].

A complementary and more powerful approach for bringing economy to the design of software architecture is the use of architectural styles [19]. Architectural styles codify the recurring architectural design practices and successful system organizations. Architectural styles are the composition patterns and constraints on architectural elements targeted at families of systems with shared characteristics [1]. The use of styles in software architectures simplifies their analysis [24], implementation [13], and evolution [9]. Although there are many techniques for *analyzing* and *describing* styles, there is insufficient support for systematically *constructing* elements of architectural styles. This often leads to haphazard realizations of style elements in the construction of software

systems, which eventually results in the loss of benefits of using a given style in the first place.

Systematic construction of style-based software architectures would require that we identify the primitives underlying elements of architectural styles and, indeed, software architectures. Our previous work [16] has proposed Alfa, a framework for understanding and constructing style-based software architectures from architectural primitives, to bridge this gap between architectural specification and construction. Alfa recognizes five orthogonal characteristics of architectural styles to assist in the construction of style elements. Alfa also provides a small set of expressive architectural primitives, each addressing one of the five orthogonal characteristics, for being composed into stylized software architectural elements.

In this paper we present an approach, called xAlfa, for systematically composing Alfa's primitives into elements of architectural styles. xAlfa provides two object-oriented reuse mechanisms, inheritance and composition, to reduce the effort required to compose architectural styles from Alfa's primitives and, as a result, produce concise style compositions. An important benefit of this approach is that it enables one to unambiguously and constructively codify similarities among a variety of architectural styles in terms of well-understood object-oriented concepts.

In this paper, we evaluate the suitability of xAlfa for composing styles from Alfa's primitives using a large and diverse set of styles for network-based systems [8], which are typically employed in modern, distributed software systems. We present the similarities among these *network-based styles* in the form of an inheritance i.e., similarity graph where common ancestors signify similarity between styles. We compare our findings to the analytical study of the same set of styles in [8] and interpret the differences in the results of the two studies.

We evaluate the *concision* of style compositions resulting from the use of xAlfa reuse mechanisms using network-based styles. Concision is considered as the reduction in sizes of style compositions in terms of xAlfa constructs required resulting from the reuse of a style (or its constituents). Our results indicate that xAlfa indeed produces concise compositions of network-based styles: creating a new composition typically results in reuse of over 50% of the constructs from an existing composition, and incurs up to 10% overhead in reusing existing compositions.

The rest of this paper presents details of the approach focusing primarily on the design of xAlfa for composing elements of architectural styles from architectural primitives. We evaluate the suitability of xAlfa for composing architectural styles and the concision of xAlfa compositions in the context of network-based styles. We also provide some pointers to considerable existing research and explain the novelty of our work in its context.

2 Related Work and Motivation

Over the last decade software architecture has emerged as a research discipline [23] and several approaches have been developed to introduce this level of abstraction in the development of modern, distributed software systems. In this section, we summarize existing research on architectural styles and distinguish our research from this work.

A software architecture allocates system function across its elements, determines the configuration whereby these elements are organized into the system, fixes the nature and protocols of interaction required between these elements, and specifies the data ex-

changed in these interactions [15]. There are three kinds of architectural elements namely, *processing*, *connecting*, and *data* [20]. Architectural styles are named sets of constraints on these elements and their inter-relationships [8]. When designing a software system, selection of an appropriate architectural style becomes a key determinant of the system's success. Styles also influence architectural evolution by restricting the possible changes an architect is allowed to make [9]. Appropriate analytical models can also be used to predict properties of style-based architectures [24]. Finally, various architectural frameworks and middleware can help partially automate and economize the implementation of style-based architectures [13]. Some common examples of styles are layered, client/server, pipe-and-filter, and C2 [23,26].

Architectural styles originate in diverse applied computing fields such as networking, artificial intelligence, signal processing, and so on. Many styles share similar characteristics. Several catalogs of architectural styles provide a context for the use of specific styles [6,23] or classify styles based on their features [22]. Other comparative studies of architectural styles evaluate properties resulting from the use of styles [8,21]. Collectively, these approaches can help an architect to choose a style for her problem.

In one of these approaches, Fielding [8] proposes a similarity-based approach for organizing styles: *basic* styles are the simplest styles, followed by *derived* styles that add new constraints to an existing basic style, and finally *hybrid* styles that combine the constraints of two or more styles. A large number of network-based styles, used later in this paper for evaluating our approach, have been empirically organized using this three-tiered classification resulting in the similarity graph as shown in Figure 1. The graphical notation used in this similarity graph is similar to a UML Class Diagram, where styles are represented as classes. For example, the uniform pipe-and-filter (UPF) style inherits from the pipe-and-filter (PF) style, and the layered client-server (LCS) style inherits from both the layered system (LS) style and the client-server (CS) style. Such genealogy of styles, when available, can be leveraged to reduce the effort involved in defining new styles with significant downstream benefits in analyzing, implementing, and evolving systems using each new style. More fundamentally, a better understanding of styles can help better understand software architectures in general. Our approach supports the formalization of such empirical knowledge of style similarities using two basic object-oriented composition mechanisms: inheritance and composition.

Architectural styles have mostly been supported by specialized ADLs. For example architectures in the C2 [26] style are specified using C2SADEL [14]. Software architectures in arbitrary styles can be specified using Acme, a generic ADL [10] and the Armani constraint specification language [18]. Our approach goes beyond modeling style-based software architectures; Alfa is used to *construct* style-based architectures from a small number of architectural *primitives*. Moreover, the availability of architectural primitives further simplifies specification of stylistic constraints, which are otherwise specified using general purpose expression languages. Our approach is aimed at enabling better understanding of different styles and greater effectiveness in modeling the dynamics of architectural assemblies [17]. As style compositions from primitives can

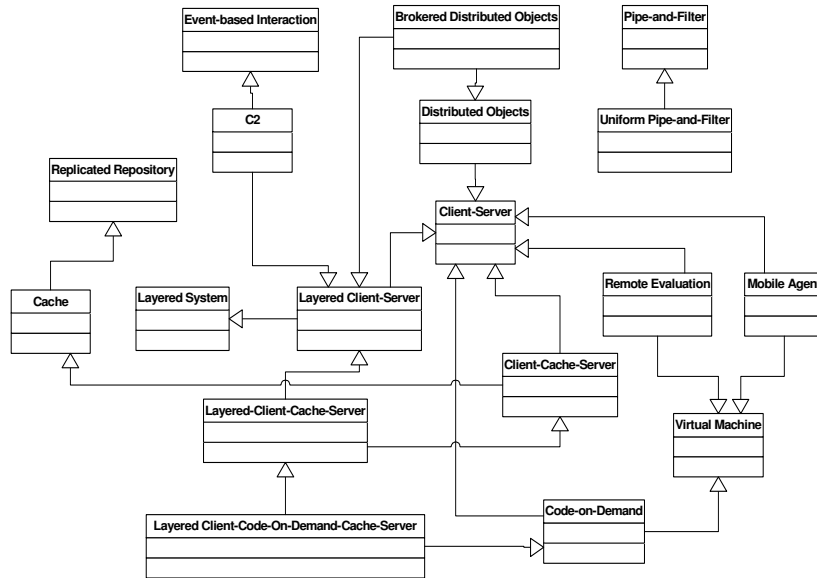


Figure 1 Similarities among network-based styles empirically described in [8]

potentially become quite large, an important challenge faced in Alfa is the greater need for *concision*, which we address via inheritance and composition.

3 Overview of Alfa

As research efforts focus on the process of constructing software architectures, various compositional approaches have started to emerge [3,12,16,25]. In this section, we briefly describe one such compositional approach—Alfa [16], an *assembly language* for software architectures—used in this paper to compose elements of architectural styles.

Alfa is a framework for understanding and constructing style-based software architectures from a small set of architectural primitives. Alfa’s seventeen architectural primitives are classified based on five orthogonal characteristics of architectural styles: structure, behavior, interaction, topology, and data [16].

Alfa employs point-to-point communication channels, called *ducts*, to tie together computational elements of a style. Each duct provides input/output behaviors to communicate data elements and, as a result, synchronize the communicating elements. In Alfa software components are treated as black boxes of functionality that relate inputs and outputs, whereas software connectors have a visible structure made of Alfa’s primitives. Alfa’s primitives consist of eight nouns, capturing the *form* of architectural style elements, and nine verbs capturing the elements’ *function*:¹

1. Data - DATUM
2. Structure - PARTICLE, OUTPUT, INPUT, TOWAY

1. SMALL CAPS are used to identify Alfa primitives. Moreover, FORM primitives are written as nouns with the initial letter capitalized, whereas FUNCTION primitives are written as verbs.

3. Interaction - DUCT, RELAY, BIRELAY, HOLDS, LOSES
4. Behavior - CREATE, SEND, RECEIVE, HANDLE, REPLY
5. Topology - CONNECT, DISCONNECT

In a given style, DATUM is the type of data items exchanged. PARTICLE is the locus of computing in software architectures. It is a container for the behavior of a processing element, and provides INPUT and OUTPUT portals for interacting with its environment. When required a single INPUT and a single OUTPUT portal can be combined into a TWOWAY port for bidirectional communication, which results in a single identity to be assigned to both portals. Both INPUT and OUTPUT portals define DATUMS that can be received from or sent to that portal.

The means of interaction between PARTICLE primitives are DUCT, RELAY, and BIRELAY primitives. A DUCT contains two ends, which are either INPUT or OUTPUT portals. Every DUCT is a FIFO queue that provides two functions—HOLDS and LOSES—to determine its communication characteristics. A DUCT can hold data items up to its HOLDS capacity, which is a whole number. A DUCT with zero capacity is synchronous, while one with non-zero capacity is asynchronous. A DUCT can lose data items written to it based on its LOSES function, which can take either of four values: *none*, *initial*, *first*, and *last*. The *initial* LOSES characteristic produces a DUCT that is initialized with a single data item. When a DUCT is full, *last* LOSES characteristic allows a DUCT to lose the incoming data item, and *first* LOSES characteristic results in a loss of the oldest data item.

A RELAY contains multiple INPUT and OUTPUT portals. Data items from each INPUT portal are forwarded to every OUTPUT portal of the RELAY. A slightly more powerful primitive is required for bidirectional communication, called a BIRELAY. This primitive performs routing of reciprocal communication back to the TWOWAY port that initiated the communication. A BIRELAY contains multiple initiator TWOWAY (where the bidirectional communication originates) and terminator TWOWAY ports (where the bidirectional communication ends).

Behavioral primitives are used to enact interaction and instantiate primitives. The CREATE function is used to create instances of a PARTICLE, which in turn may result in the instances of the contained forms—any of PARTICLE, INPUT, OUTPUT, TWOWAY, DUCT, RELAY, and BIRELAY—to be created automatically and recursively. The SEND function is used to synchronously write a data item at the OUTPUT end of a DUCT. The RECEIVE function is used to synchronously read a data item at the INPUT end of a DUCT. The HANDLE function allows a PARTICLE to continue its processing while data items are asynchronously removed from the DUCT whenever they become available and then handed off to the PARTICLE for processing. Finally, the REPLY function is used to respond to a previously received data item, and results in routing information to be added to the data item being sent in the reply so that it reaches the TWOWAY port where the communication originated.

Alfa's topological primitives—CONNECT and DISCONNECT—are used to establish and remove, respectively, a DUCT between corresponding portals of two PARTICLES. Dynamic semantics of Alfa's primitives have been formalized in [17] using constraint automata.

Alfa's primitives have been compared with Reo channels [3] to evaluate their expressiveness [16]. Arbab has shown that Reo's primitive channels together with the merge and replicate operators are expressive enough to model any interactions involving a regular expression of input/output operations on point-to-point channels [3]. Alfa's primitives INPUT, OUTPUT, DUCT, RELAY, HOLDS, and LOSES can be used to model all

primitive Reo channels and its merge and replicate operators [16]. By analogy, it can be said that Alfa’s set of primitives is just as expressive.

4 xAlfa

The systematic composition of Alfa’s primitives into style elements requires a precise notation. As previously discussed, such a notation must facilitate conciseness, but not at the cost of reducing expressiveness of the concepts being formalized. To solve this problem, we have designed a notation for composing style constituents from Alfa’s primitives, called xAlfa. xAlfa also enables the use of interchangeable expression languages for specifying constraints on enumerations of static arrangements of style constituents and the dynamic arrangements of their behaviors. We illustrate the xAlfa constructs through the composition of a network-based style—C2 [26].

4.1 Example style—C2

The C2 style is a layered network of concurrent components that communicate via neighboring connectors [26]. In order to create an xAlfa composition for this style, we characterize the style along five orthogonal dimensions as shown in Table 1. Such characterization makes it possible to manually verify the completeness of the composition when no definitive formal model is available to define a style, a case with many architectural styles that have originated in practice.

4.2 Design of xAlfa

In order to streamline the implementation of the xAlfa notation and design of tools that manipulate xAlfa compositions, we created an object-oriented design of its constructs. xAlfa constructs can be divided into two areas: *primitive* and *composition*. The *primitive* xAlfa constructs, shown in Figure 2, are counterparts of Alfa primitives and share

Table 1: C2 style characteristics

Characteristic	C2 style
Data	<i>Notifications</i> and <i>requests</i> are exchanged.
Structure	<i>Components</i> contain an interface for <i>producing</i> and <i>consuming notifications</i> and <i>requests</i> . For every produce notification interface, a component must have a consume request interface, and vice versa. All components should either have produce notification and consume request interfaces, or produce request and consume notification interfaces, or both. <i>Connectors</i> contain interfaces for <i>producing</i> and <i>consuming notifications</i> and <i>requests</i> . For every produce notification interface, a component must have a consume request interface, and vice versa.
Interaction	Components and consumers do not block for consuming requests and notifications.
Behavior	Connectors broadcast notifications received on each of their consume notifications interfaces to their produce notifications interfaces, and vice versa.
Topology	A component’s (connector’s) produce notification interface is connectable to a connector’s (component’s) consume notification interface, while its produce request interface is connectable to a connector’s (component’s) consume request interface. Connected components and connectors cannot form a loop.

their semantics described in Section 3. We have introduced two additional abstract constructs, *Abstract Behavior Type* (ABT) and *Portal*, to represent the commonalities of certain Alfa primitives. *ABT* forms a base class for the xAlfa constructs *Relay*, *Birelay*, and *Particle*, and *Portal* for *Output* and *Input*. Additionally, we have introduced the *Interface* construct to represent a group of *Portals* that are always used atomically, i.e., all *Portals* in the *Interface* at once or none at all. Note that the *Portal Mapping* construct specialized from a *Portal*, as shown in Figure 2, is, in fact, a *composition* construct (discussed below), but is shown here in the context of the other *Portal* specializations.

The *composition* constructs of xAlfa, as shown in Figure 3, are used for composing Alfa primitives into architectural styles and their constituents. The main composition construct is a *Style*, which serves to organize and constrain Alfa's primitives used in an architectural assembly. A *Style* is a uniquely identified object, and contains any number of *Datums* as well as a *Composition* of *Ducts* and *Constituents*. A *Style* may contain a *Topology* and/or a *Data Constraint*, both of which are constraint expressions on the static arrangements of style elements. We have designed flexibility in the choice of a constraint *expression* notation by defining *Topology* and *Data Constraint* as abstract base classes. This allows us to plug in any suitable constraint expression language without changing the rest of the xAlfa constructs. A *Style* might also be a specialization of one or more *base Styles*, thus formalizing similarities between different styles. Specializing a *Style* automatically adds all the *Datums* and *Constituents* of the base *Style* and constraints thereupon.

A *Constituent*, a uniquely identified construct, contains one *ABT*, an optional *Behavior Constraint*, and an optional *Structure Constraint* thereupon. Both *Behavior Constraint* and *Structure Constraint* are also defined as abstract base classes similar to *Topology* and *Data Constraint* above. An important inclusion in this design is the construct *Composite Particle* which forms the crux of the concision mechanisms of xAlfa. A *Composite Particle* may contain its own internal *Composition*, *inherit* from another *Constituent* and/or may *compose* other *Constituents* defined in the given *Style* or its base *Styles*. Each *composed Constituent* resides in a namespace, as identified by its *name*, different from the other *composed Constituents* as well as from the composition itself.

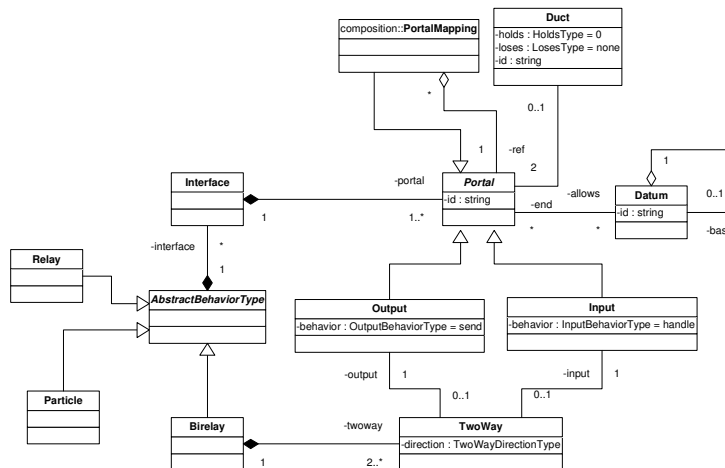


Figure 2 UML class diagram of xAlfa primitive constructs

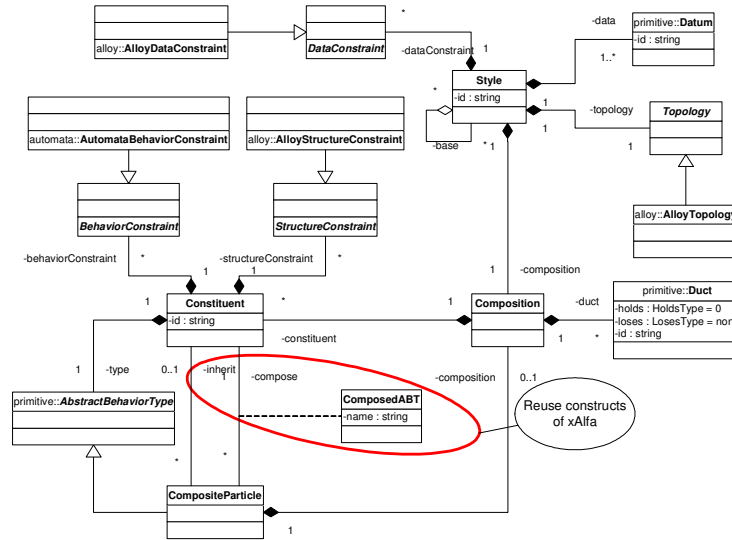


Figure 3 UML class diagram of xAlfa *composition* constructs

Each *Constituent* that is inherited lives in the same namespace as the *Composite Particle* itself. This design prevents the collision of identifiers across the sub-components of a *Constituent*. In order to aid understanding, it is often useful, although not necessary, to create a mapping from portals of the composite to portals of the components using the *Portal Mapping* construct described earlier.

4.3 Graphical metaphor for xAlfa

We use a graphical metaphor, shown in Figure 4, to render xAlfa style compositions. The graphical metaphor corresponds to xAlfa constructs previously discussed. Since this metaphor does not provide support for constraint expressions, such as those for *Topology*, or for attributes of xAlfa constructs, we provide such information textually. Also, *Style* is not shown separately, but its identifier is used to prefix the identity of every *Constituent* and *Datum*. The *Composition* of a *Composite Particle* is rendered as a containment of its *Constituents* and *Ducts*.

The xAlfa composition of the C2 style is shown in Figure 5. The datums of this style are *Notification* and *Request*, and its composition contains constituents *Component* and *Connector*, and ducts *N* (notification), *ND* (notification-delivery), *R* (request), and *RD* (request-delivery). The ABT used in the constituent C2 *Component* is a particle, which is split across the diagram into two particle shapes. Also, note that the constituent C2 *Connector* is a composite particle whose composition contains two ducts *Conn.req* and *Conn.notif*, and four constituents *Conn.ti* (top-incoming), *Conn.to* (top-outgoing), *Conn.bi* (bottom-incoming), and *Conn.bo* (bottom-outgoing), each made up of a relay. This composition

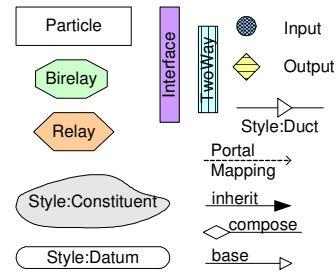


Figure 4 Graphical metaphor for xAlfa constructs

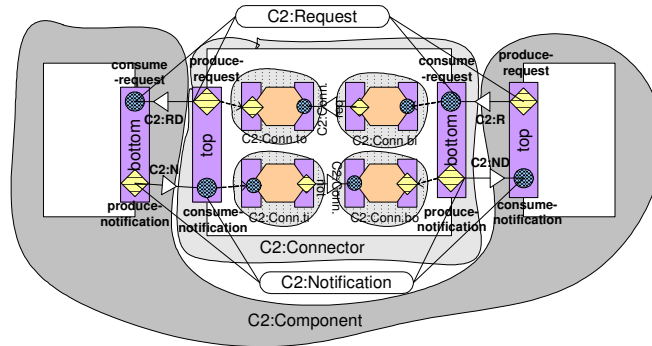


Figure 5 C2 style composition using xAlfa

collectively determines the behavior of a C2 *Connector*. The ducts *Conn.req* and *Conn.notif* are provided with an *infinite* holds capacity so that *Components* do not block while producing *Notifications* or *Requests*. In the interest of space, we omit the discussion about internal interfaces and portals of C2 *Connector*. Portals in the interface of the C2 *Connector* are mapped to portals of the constituents in the composition of C2 *Connector*. Moreover, the inputs on both the *top* and *bottom* interfaces of a C2 *Component* are defined to have a *handle* behavior, which is required to ensure that it does not block for consuming requests and notifications as required by the interaction characteristics in Table 1.

4.4 Constraint expression languages in xAlfa

Our design of xAlfa allows the choice of arbitrary expression languages for *Topology*, *Data Constraint*, *Structure Constraint*, and *Behavior Constraint*. Currently, we use Alloy [11] for the first three and Event Synchrony Language (ESL) based on Constraint Automata [5] for the last one.

Alloy is a lightweight first-order relational logic notation that is naturally suited to model structural organization of types in terms of sets and relations among sets [11]. For specifying constraint expressions in xAlfa, we treat all constituents and datums as sets and interfaces of constituents as relations to ducts. The notation provides quantifiers—*sole*, *one*, *all*, *some*, and *no*—for creating relational expressions in terms of elements of a set, which are used for constraining interfaces of constituents and ducts between the portals of these interfaces. In Alloy, the *dot* (‘.’) operator is used for dereferencing relations of a set, and the *converse* (‘~’) operator is used to refer to the converse of a relation.

Based on this, we now revisit our running example of the C2 style composition to illustrate Alloy constraint expressions. A structure constraint is required to ensure that every C2 *Component* has at least one of its two interfaces expressed using Alloy, where ‘||’ is the disjunction operator, as:

```
all c: C2_Component |
  one c.C2_Component_bottom || one c.C2_Component_top
```

In order to define topological constraints, sometimes it becomes useful to define an Alloy relation between xAlfa constituents. Here, we define a relation *connected*, which identifies those *Components* to which a *Component* is connected via *Connector* through its *top* or *bottom* interfaces as:

```

C2_Component_bottom.~C2_Connector_top.C2_Connector_bottom.
~C2_Component_top + C2_Component_top.~C2_Connector_bottom.
C2_Connector_top.~C2_Component_bottom

```

This relation can, in turn, be applied in an Alloy expression, where 'in' is the set inclusion operator, '*' represents transitive closure, and '!' is the negation operator, to constrain C2 topology to exclude loops of connected components:

```

all c: Component | c !in c.*connected

```

ESL can be used to specify a *Behavior Constraint* in xAlfa style compositions. Alfa considers behavior as the processing logic of elements by which input data is consumed and output data [16]. Therefore, behavior constraint expressions constrain the ordering of input and output behaviors of style constituents. Therefore, in xAlfa behavior constraints on constituents are specified in terms of their portal names as the portal names symbolize input and output behaviors of constituents.

ESL, a plain text notation for constraint automata [5], is defined to specify synchrony and asynchrony relationships between events of a process. For example, in the following constraint on process *Proc*, an infinite process of finite states. Events *a* and *b* are asynchronous and *b* cannot occur before *a*, and parenthesized events *c* and *d* are synchronous. The '|' choice operator allows a non-deterministic selection of events *a* or *{c, d}*, and '->' orders asynchronous events.

```

Proc = a -> b -> Proc | {c, d} -> Proc

```

The C2 style composition does not require behavior constraints since it captures all the behavioral characteristics of this style without requiring any constraints on the dynamic arrangement of its events. Moreover, only three constraint expressions are required in the entire composition of the C2 style, made up of 69 xAlfa constructs.

5 Hierarchical Styles Composed Using xAlfa

In order to illustrate the concision mechanisms provided in xAlfa, we briefly describe the composition of two hierarchical network-based styles from [8]: Client-Server (CS) and Layered Systems (LS).

The CS style is composed using xAlfa as shown in Figure 6. A CS *Router* is a composite particle that contains only one birelay. Moreover, the behavior of a CS *Client*'s *response* input is *receive*, that of the CS *Server*'s *request* input is *handle*, and that of the CS *Server*'s *response* output is *reply*. In addition, the duct *server-request* has an *infinite* holds capacity.

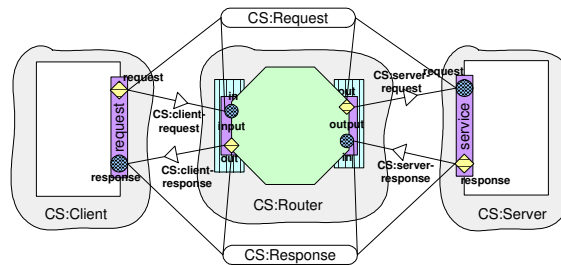


Figure 6 Client-Server style (CS) composition using xAlfa

A CS *Server (Router)* carries a structure constraint that requires it to have exactly one *service (request)* interface, which is specified in Alloy as:

```

all s: CS_Server | one s.CS_Server_response
all t: CS_Router | one t.CS_Router_output

```

Behavior constraints on both the *Client* and *Server* ensure that progress is made. These constraints as expressed on the portals of CS *Client* and CS *Server*, respectively, using ESL as:

```
CS_Client = CS_Client_request_request ->
  CS_Client_request_response -> CS_Client
CS_Server = CS_Server_response_request ->
  CS_Server_response_response -> CS_Server
```

Finally, a topology constraint on the style ensures uniqueness of interaction paths between any given *Client* and *Server* expressed in Alloy as:

```
all c: CS_Client | all disj c1, c2: CS_Client_request |
  c1.~CS_Router_input != c2.~CS_Router_input
```

In all, five constraint expressions are required in the xAlfa composition of the Client-Server style.

As shown in Figure 7, the LS style is derived from the CS style, as its *Layer* constituent can be composed from a combination of CS *Clients* and a CS *Server*. This allows an LS *Layer* to invoke any other LS *Layers*, and itself process invocations from any other LS *Layers*. Moreover, the CS *Request* is defined as the base for the LS *Parameter*, and the CS *Response* as the base for the LS *Result*. Due to the availability of these reuse mechanisms, it is possible to concisely compose the LS style using the composition of the CS style including the constraints on its constituents. Thus a *Layer* behaves exactly as expected by virtue of being composed from CS *Clients* and a CS *Server*. An LS *Layer* waits for the *Result* in response to invoking another *Layer* with a *Parameter*, and generates a *Result* for every *Parameter* it receives, but no additional constraint expressions are required to include this in the LS style composition, as it is automatically derived in the LS style from the CS style.

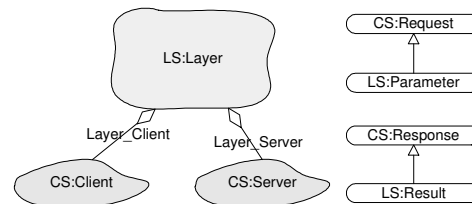


Figure 7 Layered System style (LS) composition using xAlfa

A relation for connected *layers* is needed in order to specify topological constraints on them, and is specified in Alloy as:

```
Layer_Client.CS_Client_request.~CS_Router_input.CS_Router_output.
~CS_Server_response.~Layer_Server
```

A topology constraint is, in turn, defined that disallows any loops among layers, and is expressed in Alloy as:

```
all l: LS_Layer | l !in l.*layers
```

Finally, by constraining all CS Servers and CS Clients to be components of LS Layers, we can prevent them from being incorrectly combined in a non-LS architecture. This is achieved by the following Alloy expression:

```
all c: CS_Client | one c.~Layer_Client
all s: CS_Server | one s.~Layer_Server
```

Thus LS style only requires four additional constraint expressions in addition to the five derived automatically from the CS style.

6 Evaluation

A detailed presentation of the composition of all network-based styles is not possible here due to space constraints. Detailed compositions can, however, be found on the Alfa Web site (<http://cse.usc.edu/~softarch/Alfa>). In this section, we evaluate xAlfa in terms of two properties – style understanding and conciseness of compositions – over the entire spectrum of network-based styles [8].

Style interrelationships. An analytical study of the interrelationships among architectural styles for network-based systems by Fielding [8] identified similarities among them as shown in Figure 1. The composition of the same styles using xAlfa reveals a different picture as shown in Figure 8. The main differences between the two results are summarized below:

- Contrary to Fielding’s study, our study found that the C2 style was found not to be related to Layered Client-Server (LCS) and Event-Based Interaction (EBI). An important requirement of all CS-derived styles is that a response is generated after processing a request. As we saw in Section 4, no such constraint exists in the C2 style. Moreover, EBI requires subscription before events can be delivered to a subscriber, while in C2 subscriptions do not exist.
- LS and CS styles are treated as basic in Fielding’s study and conventionally considered unrelated. In our experiments, however, we have found that the LS style is derived from the CS style as explained in Section 5.
- The Cache (\$) style was considered by Fielding to be derived from the Replicated Repository (RR) style, but we found the two styles to be independently derived from an abstract *Sharing* style.

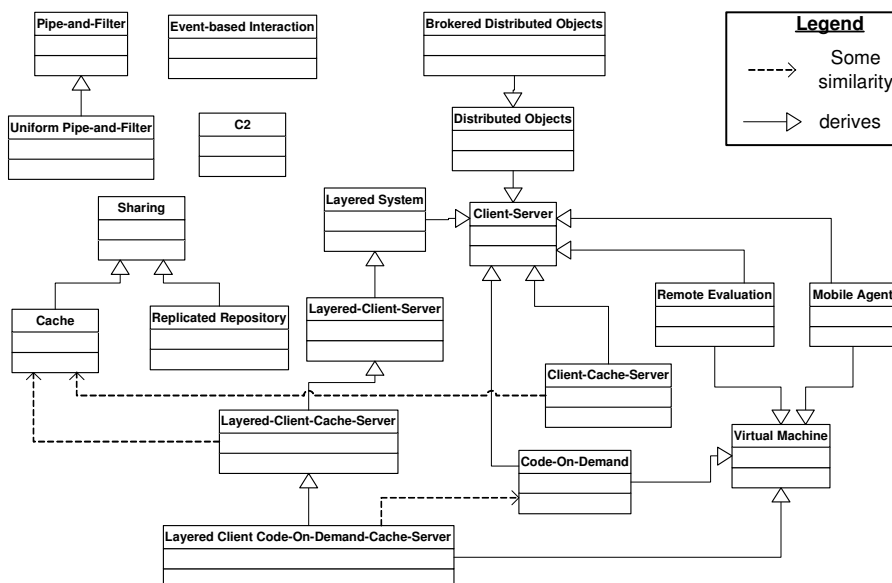


Figure 8 Similarities among network-based styles composed using xAlfa

- Fielding treats the \$ style as the basis for the Client-Cache-Server style, which, in turn, is considered as the basis for the Layered-Client-Cache-Server (LCSS\$) style. In our finding, the \$ style could not be directly used in either of these two derived styles, although parts of them were similar to the Cache style.
- We derived the Brokered Distributed Objects (BDO) style from the Distributed Objects (DO) style alone, whereas Fielding treats BDO as a hybrid of both the LCS and DO styles.
- Fielding treats the Code-on-Demand style to be one of the bases of Layered-Client-Code-On-Demand-Cache-Server (LCCOD\$\$) style, while we found the latter style to derive from Virtual Machine and LC\$\$ styles. Some parts of LCCOD\$\$ are, however, similar to COD.

These differences between the two results can be attributed to the fact that xAlfa requires more rigorous decomposition of styles and their constituents than Fielding's analytical study. Moreover, xAlfa provides systematic mechanisms for reusing existing styles in new styles, precisely bringing out similarities among different styles.

Concision of style compositions. While an improved understanding of architectural styles can be obtained using xAlfa, an important focus of our work is the ability to create concise compositions of architectural styles. We have already seen that xAlfa composition of styles requires few, if any, constraint expressions.

To assess the impact of two object-oriented reuse mechanisms in xAlfa for composing new styles from existing ones: inheritance and composition, we have measured the concision of network-based style compositions. Table 2 summarizes the benefits of the two reuse mechanisms on the concision of style composition. We measure *concision* as the reduction in the total number of constructs used due to the availability of reuse mechanisms. Further, we measure the *concision overhead* as the constructs required to reuse existing compositions as a fraction of reused constructs. A reuse *figure of merit*, an indicator of effort savings attributed to reuse, can be obtained by comparing the concision measure to the concision overhead.

To give a true picture of the impact of reuse, and produce a meaningful concision measure, we have counted every instance of reuse in a new style while measuring the number of constructs reused. As can be noted from the results, basic styles do not benefit from concision mechanisms. However, if one can work under the assumption that more derived and hybrid styles would be composed than basic styles, this is acceptable. Further, across all the derived and hybrid styles, concision overhead is under 10%, which serves as a major encouragement for reuse. In the same styles, concision varies between 59% and 95%. Derived styles that add little to a basic styles experience less concision, and those that are further down the similarity chain usually obtain greater benefits of concision.

Table 2: Concision of network-based styles compositions from xAlfa constructs

	Pipe & Filter	Uniform Pipe & Filter	Replicated Repository	Cache	Layered System	Client-Server	Layered Client-Server	Client-Cache-Server	Layered-Client-Cache-Server	Virtual Machine	Remote Evaluation	Code-On-Demand	La-Ci-Cod-Dem-Ca-Ser.	Mobile Agent	Event-based Interaction	C2	Distributed Objects	Brokered Dist. Objects
xAlfa constructs used (A)	136	28	146	43	40	61	39	64	32	33	13	18	13	42	6	3	5	3
Reuse constructs used (B)	0	5	15	4	5	0	7	2	0	4	4	4	4	6	0	0	5	6
Constructs reused (C)	0	188	228	56	58	0	215	58	239	0	89	88	281	89	0	0	58	200
Concision $\chi = C/(A+C-B)$	0%	89%	64%	59%	62%	0%	87%	64%	89%	0%	86%	86%	95%	71%	0%	0%	66%	88%
Concision overhead $\sigma = B/C$	-	2.7%	6.6%	7.1%	8.6%	-	3.3%	3.4%	0.8%	-	4.5%	4.5%	1.4%	6.8%	-	-	8.6%	3.0%

7 Conclusion and Future Work

In this paper we have presented a systematic approach, xAlfa, for composing architectural styles and their constituents from Alfa's architectural primitives. This approach has the dual benefit of improving our understanding of the similarities and differences between styles, as well as producing compositions that are concise and reducing the effort involved in creating them. Our experiments on a large number of network-based styles have found that less than 10% overhead occurs while reusing existing xAlfa compositions and typically more than 50% compositions were reused in new style compositions. However, basic styles do not benefit from the concision mechanisms of xAlfa.

The object-oriented design of xAlfa constructs can be implemented using several different grammars; we have chosen XML schemas [27] due to their suitability for systematically recording structured information such as that in xAlfa style compositions, and the support available for rapid development of associated tools. While an automated tool has been developed to compose styles using xAlfa, much work remains to be done in the automated use of style compositions. We plan to focus next on the conformance checking of style-based architectures to their styles using xAlfa style compositions. This will benefit from the use of Alloy in xAlfa compositions for specifying static constraint expressions. Also, in our previous work [17], we have shown that constraint automata effectively model architectural assemblies of Alfa primitives. We plan to tap into this and combine the behavior constraints on style compositions with the semantics of Alfa's primitives to produce effective dynamic models of style-based architectural assemblies. Additionally, we plan to provide visualization tools to render xAlfa style compositions using a graphical metaphor.

Last, but not the least, our goal remains the construction of style-based architectures from primitives. This requires support for consistent and efficient implementation of Alfa architectural assemblies. Therefore, we plan to develop code generation techniques from xAlfa compositions.

8 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] Aldrich, J., Chambers, C. and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. Proc. 24th International Conference on Software Engineering, May 2002.
- [3] Arbab, F. Abstract behavior types: A foundation model for components and their composition. Proc. 2nd International Symposium on Formal Methods for Components and Objects, Lecture Notes in Computer Science, vol. 2852, Springer, Nov. 2003.
- [4] Arbab, F. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, Cambridge University Press, 2003.
- [5] Arbab, F., Baier, C., Rutten, J. J. M. M., and Sirjani, M. Modeling component connectors in Reo by constraint automata. Proc. 2nd Workshop on Foundations of Coordination Languages and Software Architectures, Electronic Notes in Theoretical Computer Science, Elsevier Science, Sep. 2003.
- [6] Buschmann, F. et al. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons Ltd., England, 1996.
- [7] Dashofy, E., van der Hoek, A. and Taylor, R. N. An Infrastructure for the Rapid Development of XML-Based Architectural Description Languages. Proc. 24th International Conference on Software Engineering, May 2002.

- [8] Fielding, R. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph. D. Dissertation, University of California at Irvine, 2000.
- [9] Garlan, D., Cheng, S., and Schmerl, B. Increasing System Dependability through Architecture-Based Self-Repair. *Architecting Dependable Systems*, de Lemos, R., Gacek, C., and Romanovsky A. (eds). Lecture Notes in Computer Science, vol. 2677, Springer, 2003.
- [10] Garlan, D., Monroe, R. T., Wile, D. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*. Leavens, G. T., and Sitaraman, M. (eds). Cambridge University Press, 2000.
- [11] Jackson, D., Shlyakhter, I. and Shridharan, M. A micromodularity mechanism. Proc. Joint 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering ACM SIGSOFT Symposium on Foundations of Software Engineering, Sep. 2001.
- [12] Lopes, A., Fiadeiro, J. L., and Wermelinger, M. Architectural primitives for distribution and mobility. Proc. 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Nov. 2002.
- [13] Medvidovic, N., Mikic-Rakic, M, Mehta, N. R., and Malek, S. Software Architectural Support for Handheld Computing. *IEEE Computer Special Issue on Handheld Computing*, **36**(9):66-73, 2003.
- [14] Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. A Language and Environment for Architecture-Based Software Development and Evolution. Proc. 21st International Conference on Software Engineering, May 1999.
- [15] Medvidovic, N. and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, **26**(1):70-93, 2000.
- [16] Mehta, N. R., and Medvidovic, N. Composing architectural styles from architectural primitives, Proc. Joint 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Sep. 2003.
- [17] Mehta, N. R., Sirjani, M., and Arbab, F. Effective Modeling of Software Architectural Assemblies Using Constraint Automata. University of Southern California Center for Software Engineering Technical Report USC-CSE-2003-509, 2003.
- [18] Monroe, R. T. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163.
- [19] Monroe, R. T. and Garlan, D. Style-Based Reuse for Software Architectures. Proc. 4th International Conference on Software Reuse, Apr. 1996.
- [20] Perry, D. E. and Wolf, A. L. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, **17**(4): 40-52, 1992.
- [21] Shaw, M. Comparing architectural styles. *IEEE Software*. **12**(6): 27-41. November 1995.
- [22] Shaw, M. and Clements, P. A field guide to boxology: preliminary classification of architectural styles for software systems. Proc. 21st International Computer Software and Applications Conference, Aug. 1997.
- [23] Shaw, M. and Garlan, D. *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.
- [24] Spitznagel, B. and Garlan, D. Architecture-Based Performance Analysis. Proc. 10th International Conference on Software Engineering and Knowledge Engineering, June 1998.
- [25] Spitznagel, B. and Garlan, D. A compositional approach for constructing connectors. Proc. Working IEEE/IFIP International Conference on Software Architectures. Oct. 2001.
- [26] R. N. Taylor et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, **22**(6): 390-406, 1996.
- [27] Thompson, H., Beech, D., Maloney, M. and Mendelsohn, N. (eds). XML Schema Part 1: Structures. URL: <http://www.w3.org/TR/xmlschema-1/>.