

Composing Architectural Styles From Architectural Primitives

Nikunj R. Mehta

Department of Computer Science
University of Southern California
Los Angeles, CA 90048, USA
+1-213-740-6504

mehta@usc.edu

Nenad Medvidovic

Department of Computer Science
University of Southern California
Los Angeles, CA 90048, USA
+1-213-740-5579

nenom@usc.edu

ABSTRACT

Architectural styles are named collections of constraints on configurations of architectural elements, and are believed to bring economies of scale in applying software architecture techniques to software development. Existing research on architectural styles provides little guidance for the systematic design and construction of architectural style elements. This paper proposes a framework, Alfa, for systematically and constructively composing “architectural primitives” to obtain elements of architectural styles. This is based on our observation that architectural styles share many underlying concepts that lead to architectural primitives. We have identified eight forms and nine functions as architectural primitives that reflect the syntactic and semantic characteristics of architectural styles and are expressive enough to compose their elements. Our approach is also illustrated using a familiar style – pipe-and-filter.

Categories and Subject Descriptors

D.2.11 [Software architectures]: Languages and patterns.

General Terms

Design, Languages.

Keywords

architectural style, style characterization, style composition, architectural primitive, Alfa, channel-based communication.

1. INTRODUCTION

Software architectures [14,16] provide high-level abstractions in the form of coarse-grained processing, connecting, and data elements, their interfaces, and their configurations [7]. The system composition patterns and constraints on architectural elements comprise *architectural styles*, which are targeted at families of systems with shared characteristics [1]. Architectural styles codify the recurring architectural design practices, and successful system organizations. Styles are therefore reusable software architectural idioms and have the potential to economize software development [12]. 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.

Existing software architecture approaches support abstractions “layered” on top of those provided by programming languages, thus ensuring continuity and reuse of past investments as newer abstraction techniques are mapped to existing ones. However, such approaches do little to improve compositionality of systems. It is widely believed that compositional approaches to software development (e.g., analogously to how this is done in computer hardware architecture [4]) are key to constructing large, complex systems [13,14]. Delivering the full value of architectural design would, therefore, require that we identify the primitives underlying software architectural elements. Although low-level communication and programming primitives are used for building software architectures, there is an almost complete lack of understanding of software architectural primitives.

This paper proposes a framework for characterization and composition of architectural styles, called *Alfa* (*assembly language for software architectures*), based on a small set of architectural primitives. Alfa’s characterization of styles identifies their orthogonal aspects and, in turn, naturally supports composition of style elements. Our approach is based on the use of a point-to-point communication channel, called *duct*, for interaction among communicating style elements. A duct is a path of interaction that is used for the transfer of both data and control [10]. Channel-based communication can be easily mapped to other communication models such as message passing, shared spaces, communication buses, or remote calls [3].

The main contributions of our work presented in this paper are:

1. a novel approach for characterization, and composition of architectural styles; and
2. a small and expressive set of architectural primitives for composing elements of architectural styles.

The remainder of this paper is organized as follows. After summarizing related work and concepts in Section 2, in Section 3 we present details of our approach and illustrate it using the pipe-and-filter style. Finally, Section 4 provides a discussion of the expressiveness of Alfa’s primitives and hints at future work.

2. RELATED WORK

Our work has been directly influenced by advances in software architecture, coordination models and languages, distributed systems, and middleware. For brevity, we focus on the first two topics here.

Software architecture: A software architecture allocates system function across its elements, determines the configuration

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEC/FSE’03, September 1-9, 2003, Helsinki, Finland.

Copyright 2003 ACM 1-58113-743-5/03/0009...\$5.00.

whereby these elements are organized into the system, fixes the nature and protocols of interaction required between these elements, and specifies the data exchanged in these interactions [9]. There are three kinds of architectural elements, namely, *processing*, *connecting*, and *data* [14]. Architectural styles are named sets of constraints on these elements and their inter-relationships [7].

Formal techniques such as Z [1], PI-calculus [5], and graph grammars [8] have been used with styles to provide rigor to their definitions and perform systematic analysis thereupon. However, such approaches do not consider the synthesis of architectural styles from shared building blocks. On the other hand, empirical studies of styles including classifications such as [7,15] and catalogs such as [6] aid the practitioner in comparing styles for their properties, but do not identify the shared building blocks of these styles. Comparative studies of styles indicate that software connectors [16], i.e. interactions among components, are key determinants of properties of a style. Our previous work [10] has provided a taxonomy of software connectors, and identifies ducts, transfer of data, and transfer of control as the building blocks of software connectors. Recent studies (e.g., Reo [3]) confirm this belief and support the composition of software connectors from a handful of primitives.

Coordination models and languages: Coordination is the process of managing dependencies among software components in a software system. Various coordination models have emerged, each with its own set of primitives, including message-passing, shared spaces, communication buses, and point-to-point communication channels. Point-to-point communication channels are shown to provide more architectural expressiveness and efficiency than other existing coordination models [3]. Likewise various calculi such as process algebras [11] and coalgebras of timed data streams [3] have been developed to provide rigor to these models and support their analysis. Our work is the first of its kind in recognizing that architectural styles can be composed from architectural primitives and unique in relating architectural primitives to a coordination model.

3. ALFA

The primary goal of our research is to create a framework, called *Alfa* (*assembly language for architectures*), for understanding, and as a result, composing architectural styles using a set of architectural *primitives*. The framework is meant to support concise composition of style elements from shared, reusable architectural primitives. Our research is motivated by the potential benefits of architectural styles in software development as well as the lack of *constructive* techniques for understanding and *composing* elements of a style. In this section, we present details of the Alfa framework including a technique for characterizing styles, and architectural primitives for composing style elements. We also illustrate our approach using the familiar pipe-and-filter style.

3.1 Characterizing Architectural Styles

Traditionally, architectural styles have been described in terms of constraints on (1) processing and (2) data components, (3) connectors, and (4) their configurations [9]. However, both processing components and connectors embody multiple concerns, making it difficult to synthesize these style elements from discrete primitives: they both possess structure and exhibit behavior; moreover, connectors also provide interac-

tion services. In order to delineate orthogonal aspects of style elements, instead of this four-way description, we favor a five-way characterization of styles. Our characterization lends itself more naturally to the goal of synthesizing style elements:

1. Structure – the form of elements from which a system is composed, including the input and output interfaces of the elements;
2. Interaction – the means by which data and control are transferred among different elements of a system;
3. Behavior – the processing logic of elements by which input data is consumed and output data produced;
4. Topology – the setup and tear-down of paths of interaction among different elements and the rules constraining allowed interaction paths; and
5. Data – the types of data exchanged during interactions among style elements.

Existing research has variously combined the above five aspects of architectural styles, but never unified them all under a single systematic characterization approach. Instead, different subsets of these five aspects have been used to support the traditional four-way characterization of styles. Perry and Wolf set the stage for studying architectural styles in [14], but do not distinguish between structure and behavior. Abowd et al. [1] formalized styles as the syntax of components and connectors, behavior as their semantics, and topology as the syntax of configurations, but did not separate interaction and data aspects. Finally, le Metayer [8] highlighted communication topology in terms of a graph grammar, and came closest to our approach, but did not identify each of the five style aspects as first-class dimensions. Our five-way characterization has the distinct advantage of explicating the orthogonal aspects of styles *constructively*; as described next, each aspect is supported by its own set of architectural primitives in the Alfa framework, enabling style composition from these primitives.

3.2 Alfa's Primitives

Software architectures of large systems often involve layers of architectural abstractions. A single component or connector may have its own internal architecture, which, further, can be made of several components and connectors. What this means is that a software architecture can be recursively decomposed to reveal more primitive architectures, until the elements obtained can no longer be decomposed further into architectural elements. At this level, architectural elements are made up of *architectural primitives*. These primitives are fine-grained, low-level abstractions, each of which focuses on a *single* aspect of architectural styles identified above.

Our approach 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, similarly to Reo [2], 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 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.

1. Data - DATUM
2. Structure - PARTICLE, OUTPUT, INPUT, TWOWAY
3. Interaction - DUCT, RELAY, BIRELAY, HOLDS, LOSES
4. Behavior - CREATE, SEND, RECEIVE, HANDLE, REPLY
5. Topology - CONNECT, DISCONNECT

DATUM is the data type of data items used in a style as data types used in a style, rather than individual data items, are important contributors to the performance of a style-based software system. PARTICLE is the locus of computing in a style, and by extension 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 patterns of DATUMS that can be received from or written 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 a duct with a non-zero capacity is asynchronous. A DUCT can lose data items written to it based on its LOSES function, which can take either of three values: *none*, *first*, and *last*. When the 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 an indefinite number of INPUT and OUTPUT portals. Data items from every 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 an indefinite number of initiator TWOWAY (where the bidirectional communication originates) and terminator TWOWAY ports (where the bidirectional communication ends).

Behavioral primitives enable the use of interaction and structural primitives. The CREATE function is used to create instances of a PARTICLE form, 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. The SEND function is used to write a data item at the OUTPUT end of a DUCT. The RECEIVE function blocks the calling thread and is used to synchronously read a data item at the INPUT end of a DUCT. The HANDLE function allows the PARTICLE to continue its processing while data items are asynchronously removed from the DUCT whenever they become available, which are 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.

Two topological function primitives are available—CONNECT and DISCONNECT—to establish and remove, respectively, a DUCT between corresponding portals of two PARTICLES.

3.3 Alfa in Action

The Alfa framework for characterizing and composing architectural styles has been applied to 10 different styles identified in [7]: layered system, pipe-and-filter, client-server, event-based interaction, replicated repository, virtual machine, code-on-demand, C2, and distributed objects. Due to space constraints, here we discuss the details of only one style – pipe-and-filter.

The pipe-and-filter style has been previously used to illustrate approaches for formalizing and analyzing architectural styles [1]. Table 1 describes the orthogonal characterization of the pipe-and-filter style based on the approach described in Section 3.1. It can be noted that this characterization naturally identifies the needed primitives to compose various style elements.

Table 1. Orthogonal characteristics of pipe-and-filter style

Data	Records are exchanged between pipes and filters.
Structure	A filter has interfaces for writing to, reading from, closing pipes, and being notified of the closing of pipes. Every filter should have interfaces for either reading from or writing to pipes or both. It should also have an interface for closing every pipe to which it has an interface for writing, and being notified about its closing by any filter. A pipe has interfaces for source and sink filters, to allow a source to close the pipe, and to notify source filters about its closing. Every pipe should have interfaces for both source and sink filters. It should also have interfaces for being closed by every source filter, and notifying all sources when it is closed.
Interaction	A filter blocks when it writes to a pipe, when it closes a pipe, and when it reads from a pipe but no records are available for reading. A pipe does not block when it notifies sources of its closing.
Behavior	A filter cannot write to a pipe, once any of the pipe's sources close it. A pipe should notify all its sources once it is closed by any of its sources. A pipe should relay all data received from its source filters to every one of its sink filter after combining the data uniformly.
Topology	A filter's write port is connectable to a pipe's source port, a filter's read port is connectable to a pipe's sink port, a filter's close port is connectable to a pipe's close port, and a pipe's notify port is connectable to a filter's notified port.

The DATUM in this style is a *record*. Two PARTICLE primitives are used in this style – a *filter* and a *pipe*, as shown in Figure 1. Two OUTPUT portals are required for a filter – one for *writing* to a pipe and another to *close* the pipe to which it is writing. Two INPUT portals are also required on a filter – one for *reading* from a pipe, and another for being *notified* about the closing of a pipe to which it is writing. Similarly, a pipe also requires two INPUT portals – one for a *source* filter, and another for being *closed* by a source. Moreover, a pipe contains two OUTPUT portals – one for a *sink* filter, and another for *notifying* source filters about being closed. The kind of DUCT primitives required for connecting a pipe to a filter are determined by the style's interaction characteristics. The write OUTPUT of a filter is connected to the read INPUT of a pipe using a

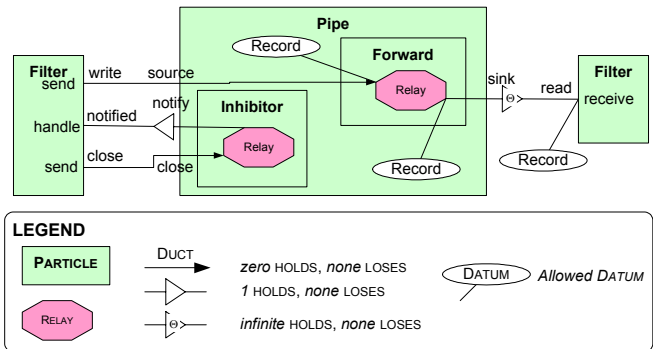


Figure 1. Composition of pipe-and-filter style's elements

DUCT with functions zero HOLDS and *none* LOSES. The close OUTPUT of a filter is connected to the close INPUT of a pipe in a similar way. The pipe's *notify* OUTPUT is connected to the filter's *notified* INPUT using a DUCT with 1 HOLDS and *none* LOSES. The pipe's sink OUTPUT is connected to a filter's read INPUT using a DUCT with *infinite* HOLDS and *none* LOSES. In Figure 1, the presence of an arrow at a DUCT end indicates an INPUT, and its absence an OUTPUT. A behavior primitive is shown next to the portal to indicate the kind of input/output behavior used with the portal.

The graphical composition of the pipe-and-filter style also shows the internal behavior of a pipe where a RELAY primitive is used to pass records from the pipe's sources to its sinks. Another RELAY is used for notifying a pipe's sources about the pipe being closed by any source filter. The filter's behavior is not shown in the graphical composition, since a filter is treated as a component with a black box behavior. Instead, a timing constraint is added to the composition to indicate the filter's behavior using the notion of timed data streams introduced in [2], details of which are beyond the scope of this paper. Finally, structural and topological constraints, elided for brevity, are expressed using first-order logic as relations among the ports of pipes and filters. The CONNECT primitive is used for establishing a DUCT between corresponding portals of a pipe and a filter. The DISCONNECT primitive is used in a similar manner for removing any DUCT thus created.

4. DISCUSSION AND FUTURE WORK

This paper has described Alfa, an integrated framework for characterization and composition of architectural styles. Through the example of the pipe-and-filter style, we have illustrated the conceptual power of Alfa's primitives. However, the expressiveness of Alfa's primitives can be better expressed by comparison with a channel-based coordination model Reo [3]. Alfa's interaction primitives can be mapped to Reo's primitive channels as shown in Table 2.

Table 2. Relating Alfa's primitives to Reo's primitive channels

Reo primitive	DUCT HOLDS	DUCT LOSES	DUCT portals
Synchronous	0	none	INPUT and OUTPUT
Synchronous drain	0	none	OUTPUT and OUTPUT
FIFO1	1	none	INPUT and OUTPUT
Asynchronous drain	1	none	OUTPUT and OUTPUT
Lossy synchronous	0	last	INPUT and OUTPUT

Moreover, Alfa's RELAY is a combination of Reo's *merge* and *replicate* operators and the *synchronous* channel. In [3], Arbab shows that the above five 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. By analogy, it can be said that Alfa's set of primitives is just as expressive.

Our goal remains the development of the Alfa *language* that combines a graphical notation for composition of style elements with structural and topological constraints, and timing behaviors. The result will be a framework for concise, analyzable, and implementable composition of architectural styles.

5. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback, which enabled us to strengthen the paper. The final version of this paper was prepared after applying our modified approach to a smaller, though not any less variegated, set of styles than used in the original submission.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

6. REFERENCES

- [1] Abowd, G. D., Allen, R. J., and Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Trans. Software Engg. and Methodology*, 4, 319-364, 1995.
- [2] Arbab F. Abstract behavior types: A foundation model for components and their composition. In *Proceedings of the First Symposium on Formal Methods for Components and Objects*, Leiden, The Netherlands, 2002.
- [3] Arbab, F. Reo: A channel-based coordination model for component composition. To appear in *Mathematical Structures in Computer Science*, 2003.
- [4] Bell, C. G. and Newell, A. *Computer structures: Reading and examples*, McGraw-Hill, New York, 1996.
- [5] Bernardo, M., Ciancarini, P., and Donatiello, L. Architecting families of software systems with process algebra. *ACM Trans. Software Engg. and Methodology*, 11, 386-426, 2002.
- [6] Buschmann, F. et al. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons Ltd., England, 1996.
- [7] Fielding, R. *Architectural styles and the design of network-based software architectures*. Ph. D. Dissertation, University of California at Irvine, 2000.
- [8] le Metayer, D. Describing architectural styles using graph grammars. *IEEE Trans. Software Engg.*, 24, 521-533, 1998.
- [9] Medvidovic, N. and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engg.*, 26, 70-93, 2000.
- [10] Mehta, N. R., Medvidovic, N., and Phadke, S. Towards a taxonomy of software connectors. In *Proceedings of 22nd Intl. Conf. on Software Engg.*, Limerick, Ireland, 2000.
- [11] Milner, R. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N. J., 1989.
- [12] Monroe, R. T. and Garlan, D. Style-based reuse for software architectures. In *Proc. of the 4th Intl. Conf. on Software Reuse*, Orlando, FL, USA, 1996.
- [13] Parnas, D. L. On the criteria to be used for decomposing systems into modules. *Comm. of the ACM*. 15, 1053-1058, 1972.
- [14] Perry, D. E. and Wolf, A. L. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engg. Notes*, 17, 40-52, 1992.
- [15] Shaw, M. and Clements, P. A field guide to boxology: preliminary classification of architectural styles for software systems. In *Proc. of the 21st Computer Software and Applications Conf.*, Washington, DC, USA, 1997.
- [16] Shaw, M. and Garlan, D. *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.