

Alfa: A framework for composing software architectures 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, neno}@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 architecture-based 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 and, indeed, software architectures 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 software architectures and are expressive enough to compose elements of architectural styles used in modern, distributed systems. Such an approach enables systematic construction of style-based architectures. In that direction, this paper evaluates the suitability of Alfa as an assembly language for software architectures and its suitability for composing architectural styles for network-based systems.

Keywords: Architectural style; Style characterization and composition; Architectural primitive and architectural assembly language; Channel-based communication; Network-based systems; Alfa

1. Introduction

Software architectures [27,31] have been proposed to address the challenges of growing complexity and size in 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 [11]. This additional level of abstraction, while aiding comprehension and construction of software systems, also requires additional effort for its use. This additional effort can, however, be reduced 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 [5,10,13,18].

A complementary and more powerful approach for bringing economy to the design of software architecture is the use of architectural styles [25]. 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 [32], implementation [16], and evolution [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, such that no guarantees can be made about the consistency of style models and the implementation of their elements, 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 composition-

ality of systems. It is widely believed that compositional approaches to software development (e.g., analogously to how this is done in computer hardware architecture [6]) are key to constructing large, complex systems [26,27]. 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 implementing 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* (*a*ssembly *l*anguage *f*or software *a*rchitectures), 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 [21]. We have focused on channel-based communication because it has been shown to be easily mappable to other communication models such as message passing, shared spaces, communication buses, and remote calls [4].

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

- (1) a novel approach for orthogonal characterization of architectural styles;
- (2) an expressive set of architectural primitives for composing elements of architectural styles used in modern, distributed software systems; and
- (3) an assembly language for style-based software architectures.

The rest of this paper consists of five sections. After presenting the background research that has motivated our work on a composition framework for software architectures in Section 2, Section 3 presents details of our approach, Alfa. Section 4 illustrates the composition of primitives using styles for network-based systems. Section 5 evaluates the reusability of Alfa's primitives for composing styles and assesses Alfa as an architectural assembly language. Section 6 concludes along with pointers to future work.

2. Background and motivation

Over the last decade software architecture has emerged as a research discipline [31] and several approaches have begun to introduce this level of abstraction in the development of modern, distributed software systems. Moreover, research in software architectures has revealed the need for an explicit focus on first-class software connectors [29] to avoid several problems arising from their neglect. An improved understanding of software connectors is possible due to extensive research on coordination models and languages. In this section, we summarize existing research in the three areas—architectural styles, software connectors, and coordination models—and distinguish our work from this research.

Architectural styles. 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 exchanged in these interactions [18]. There are three kinds of architectural elements namely, *processing*, *connecting*, and *data* [27]. Architectural styles are named sets of constraints on these elements and their inter-relationships [11]. Appropriate analytical models can be used to predict properties of style-based architectures [32]. Various architectural frameworks and middleware can help partially automate and economize the implementation of style-based architectures [16]. Finally, styles influence architectural evolution by restricting the possible changes an architect is allowed to make [12]. Therefore, when designing a software system, selection of an appropriate architectural style becomes a key determinant of the system's success. Some common examples of styles are layered system, client/server, pipe-and-filter, and C2 [31,33].

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 [9,31] or classify styles based on their features [30]. Other comparative studies of architectural styles evaluate properties resulting from the use of styles [11,28]. Collectively, these approaches can help an architect to choose a style for her problem, but do not identify the shared building blocks of these styles.

The use of architectural styles has mostly been supported through the use of specialized ADLs. For example, architectures in the C2 style [33] are specified using C2SADEL [17]. Formal techniques such as Z, PI-calculus, and graph grammars have been used for formalizing an ADL's semantics, providing rigor to the definitions of style-based architectures and allowing systematic analysis thereupon

[2,7,14]]. Software architectures in arbitrary styles can be specified using Acme, a generic ADL [13] and the Armani constraint specification language [24].

However, none of these approaches considers the synthesis of architectural styles from shared building blocks. Further, our approach goes beyond modeling style-based software architectures: Alfa is used to *construct* style-based architectures from a small number of architectural *primitives*. The availability of architectural primitives further simplifies and reduces the need for specification of stylistic constraints, which are otherwise specified using general purpose expression languages (e.g., [24]). Our approach is aimed at enabling better understanding of different styles and greater effectiveness in modeling the dynamics of architectural assemblies [22].

Software connectors. Software connectors mediate interactions among components: that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required [31]. Comparative studies of styles such as [30] indicate that software connectors, i.e., interactions among components, are key determinants of properties of a style. Several problems are found to occur when software connectors are not treated as first-class, suggesting the need for an explicit focus on connectors [29].

Our own previous work on classifying software connectors [21] proposes that every connector maintains one or more *ducts*, which are used to link the interacting components and support the flow of data and control between them. Connectors can also have an internal architecture that includes computation and information storage. A recent study by Arbab [4] confirms this belief by supporting the composition of connectors from a handful of primitive ducts. Simple connectors (e.g., procedure calls) are typically implemented in programming languages. On the other hand, composite connectors (e.g., asynchronous event multicast) are achieved through composition of several connectors (and possibly components), and are usually provided as libraries and frameworks. Simple connectors only provide one type of interaction service, whereas composite connectors may combine many kinds of interactions. Complex connectors can help us in overcoming the limitations of modern programming languages and in constructing modern, distributed systems.

Coordination models. 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. In one of the most comprehensive techniques for comparing coordination models for their expressiveness, Bonsangue et al. [8] introduce the notion of coordination architecture as a common language for different coordination models involving one of the following kinds of delays: undelayed (i.e. synchronous), globally delayed, and locally delayed (i.e. unordered). These coordination architectures are used to prove equivalence or difference between different coordination models. Our own work focuses on a higher level of abstraction—architectural styles, rather than coordination styles—and is aimed at developing a common assembly language for a variety of architectural styles.

Likewise, various calculi such as process algebras [23] and coalgebras of timed data streams [4] have been developed to provide rigor to coordination 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*. 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-based software architecture. In this section, we present details of the Alfa framework, comprising a technique for characterizing styles and a set of architectural primitives for composing style elements. We also illustrate our approach using the familiar client-server 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 [18]. 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 interaction services. In order to delineate orthogonal aspects of style elements, instead of this four-way description, we favor a five-way characterization of styles using the following dimensions:

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

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 [27], but did 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 [14] 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. As a result, our characterization of styles lends itself more naturally to the goal of synthesizing style elements.

In the Client-Server style, a server component provides services to clients on demand, and these components interact via rendezvous connectors [2]. The Client-Server style is orthogonally characterized along the five dimensions discussed earlier as shown in Table 1.

Table 1: Client-server style characteristics

Data	<i>Requests and responses are exchanged between clients and servers.</i>
Structure	<i>A client has interfaces to request services and receiving responses. A client has a response interface for each of its request interface. A server has an interface to accept service requests and provide responses. A router has interfaces to accept service requests and provide responses, and an interface to make request for service and accept responses.</i>
Interaction	<i>Clients block for responses to their service requests. Servers and routers do not block when accepting service requests. Routers do not block when making requests for service.</i>
Behavior	<i>Servers reply to a request by generating a response. Routers reply to a request by generating a response. They forward all requests to a server through their make service request interface, and the responses received on accept responses interface to the response interface for the corresponding client. Clients immediately switch to receiving a response after sending service requests.</i>
Topology	<i>A client's request interface is connectable to a router's request interface, and its response interface is connectable to a router's response interface. A router's make request interface is connectable to a server's request interface and its accept response interface is connectable to a server's response interface. Each server should be connected to one transport, and vice versa. A client cannot be connected to a transport via more than one request/response interfaces.</i>

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.

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 decom-

posed 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*. In order to create an assembly language for style-based software architectures, its architectural primitives should satisfy three conditions that characterize any assembly language. Unlike an assembly language for programming machine, architectural primitives describe system characteristics at a much higher level of abstraction. Correspondingly, these primitives may require more complex implementations than simple collections of machine-level instructions. The three conditions are as follows:

- (1) Primitives should address all characteristics of an architectural style, i.e., they should form a *complete* set.
- (2) Primitives should be *implementable* in a computer program.
- (3) Primitives should be *reusable* across architectural styles.

Alfa's primitives are fine-grained, low-level abstractions, each of which focuses on a *single* dimension of architectural styles identified above. Collectively, Alfa's primitives are able to express all five orthogonal dimensions of architectural styles. In order to completely model a style, Alfa's primitive set is coupled with a constraint expression language as explained at the end of this section. Alfa's primitives consist of eight nouns, capturing the *form* of architectural style elements, and nine verbs capturing the elements' *function*¹ as shown below:

- (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

The implementability of Alfa's primitives is discussed throughout this section along with their definitions. We use a modular programming paradigm such as that provided in the C programming language to demonstrate implementability of Alfa's primitives, although an equivalent implementation in other paradigms (e.g., object-oriented programming) is also possible. Moreover, the fine-grained semantics of primitives enable their wider reusability as discussed later in Section 4. We also argue for the completeness of the set of Alfa's primitives in Section 5.

Our approach employs point-to-point communication channels, called *ducts*, for the exchange of data and control to tie together computational elements of a style. Each duct provides input/output behaviors to communicate data elements and synchronize the communicating elements as a result of the communication. In Alfa software components are treated as black boxes of application-specific functionality that relate inputs and outputs, whereas software connectors have a visible style-specific but application-independent structure made of Alfa's primitives.

DATUM is the type of data items used in a style, e.g., the use of data streams in a pipe-and-filter system or lightweight events in an event-based interaction system [12]. Any typed programming language can support the implementation of the DATUM primitive as a data type.

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. Both INPUT and OUTPUT portals define patterns of DATUMS that can be received from or written to that portal. The realization of a PARTICLE varies for different programming paradigms and implementation platforms. For example, in modular programming a module could be considered a single PARTICLE, while, INPUT and OUTPUT portals correspond to a function. Moreover, patterns of allowed DATUMS can be implemented as the signature of such a function.

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 CREATE primitive can be treated as a library function that can be used to produce an executing program structure that represents primitive instances.

The SEND function is used to write a data item at an OUTPUT portal and the HANDLE function is used to dispatch a data item read from an INPUT portal to the PARTICLE containing that INPUT. In a modular program, the SEND function can be implemented as an invocation of a function that corresponds to an OUTPUT portal. On the other hand, the HANDLE function can be implemented as a function declaration

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.

that is exported to and can be invoked by other modules. The RECEIVE function can be used to temporarily suspend processing until a data item is available for being read from an INPUT portal. This primitive is highly useful in lock-step programming models where active processes synchronize and communicate via explicit input/output operations. The RECEIVE function can be implemented as an function that retrieve a data item from a buffer identified by the INPUT portal. When this buffer is empty, the execution of receive blocks until data items become available. However, if the buffer contains a data item, receive unblocks after removing that item from the buffer.

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, whereas RELAYS and BIRELAYS may contain multiple INPUT and/or OUTPUT portals. A DUCT is a FIFO queue that provides two functions—HOLDS and LOSES—to determine its communication characteristics. A DUCT with heterogenous portals on either end is called a *flow* DUCT. Behaviors on INPUT and OUTPUT portals of a flow DUCT are synchronized on the basis of the DUCT’s HOLDS and LOSES functions. A flow DUCT can hold DATUM instances 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 DATUM instances 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 DATUM instance. When the DUCT is full, *last* LOSES characteristic allows a DUCT to lose the incoming DATUM instance, and *first* LOSES characteristic results in a loss of the oldest DATUM instance. A flow DUCT can thus be implemented as a queue with a certain buffer size and lossiness. A DUCT that terminates in both OUTPUT portals, called a *sink* DUCT, can only consume DATUM instances. A sink DUCT can be implemented as a *memoriless* sink of DATUM instances. A DUCT with both INPUT portals, called a *spout* DUCT, spontaneously produces DATUM instances on demand. A spout DUCT, therefore, can be implemented as a random generator of DATUM instances based on the commonly allowed DATUMS of the DUCT’s INPUT portals.

A RELAY contains multiple INPUT and OUTPUT portals. DATUM instances from each INPUT portal are forwarded to every OUTPUT portal of the RELAY. Moreover, a DATUM instance appears simultaneously at all the OUTPUT portals. A RELAY is memoriless and cannot buffer any DATUM instances. It chooses non-deterministically from any number of INPUTS that have DATUM instances ready to be HANDLED. A RELAY can thus be implemented as an unbuffered multiple producer-consumer algorithm.

When performing bidirectional communication among multiple initiating and responding (i.e. terminating) PARTICLES, a RELAY is insufficient for performing the correct routing of responses. Hence, a slightly more powerful primitive, called BIRELAY, is used for performing reciprocal communication. A BIRELAY contains one or more initiator TWOWAY ports (where the *bidirectional* communication originates), and one or more terminator TWOWAY ports (where the *reciprocal* communication originates). Each TWOWAY port consists of a single INPUT and a single OUTPUT portal, and produces a common identity for its INPUT and OUTPUT portals. A BIRELAY performs routing of reciprocal communication back to the TWOWAY port that initiated the communication. A BIRELAY can be implemented as a routing function that SENDS DATUM instances along one of many different DUCTS based on the portal on which the DATUM instance is received. In a PARTICLE, the REPLY function is used to respond to a DATUM instance that has previously arrived at its INPUT. It can be implemented as an invocation of a pass-through function for the OUTPUT portal that adds routing information to the DATUM instance being sent so that it reaches the TWOWAY port where the communication originated. The TWOWAY primitive is implemented as an identifier for use in DATUM instances as part of a routing path carried in the DATUM instance.

An aspect of the completeness of Alfa’s primitive set is the *expressiveness* of its interaction primitives, which determines the domain in which the Alfa framework can be used. The expressiveness of Alfa’s primitives can be studied by comparison with the channel-based coordination model Reo [4]. 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 [4], Arbab shows that the five primitive channels shown in Table 2 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.

At the same time, Alfa's choice of interaction primitives impacts its applicability. An important class of styles that cannot be composed using Alfa's primitives are styles that require the notion of absolute time, such as real-time styles. This limitation arises because Alfa's interaction primitives treat time as a discrete ordering between simultaneous events. We believe such a limitation to be reasonable since even assembly programming languages are domain-driven. Despite this limitation, Alfa is still applicable to a large design space employed for developing modern, distributed systems.

Finally, two topological function primitives are available in Alfa—CONNECT and DISCONNECT—to establish and remove, respectively, a DUCT between corresponding portals of two PARTICLES. The CONNECT primitive can be implemented in a modular program as the compile-time or run-time binding between function invocation and function declaration, and the creation and insertion of an appropriate FIFO buffer for the DUCT. The DISCONNECT primitive can only be implemented as a run-time facility to remove such a binding and its related buffer.

3.3. Graphical metaphor for Alfa's primitives

We use a graphical metaphor, shown in Figure 1, to render compositions of architectural styles from Alfa's primitives. Each element of the graphical metaphor corresponds to an Alfa primitive. This metaphor does not provide support for the run-time function primitives CREATE, CONNECT, and DISCONNECT. However, the function primitives SEND, RECEIVE, HANDLE, REPLY, HOLDS and LOSES can be represented as attributes of their corresponding form primitives. When a PARTICLE contains further PARTICLES, RELAYS, or BIRELAYS, such composition is rendered as geometric containment of corresponding shapes. An additional notational construct, which is not a primitive, is portal mapping used for relating a portal of an outer PARTICLE to portals of its internal elements. Finally, allowed DATUMS are shown as undirected lines between INPUT and OUTPUT portals and DATUMS. This metaphor is used as the legend for style composition diagrams given in the rest of this paper. Note that in the composition of a style's elements, the use of a portal on a RELAY or a BIRELAY, in fact, indicates that as many portals of that kind may be present as required to CONNECT to other PARTICLES, RELAYS and BIRELAYS. This is required as a style defines *types* of elements allowed in architectures based on the style, and types may be instantiated differently in terms of their connectivity.

We now illustrate Alfa's primitives and their composition into architectural styles using our example style—client-server. The Alfa composition of the client-server (CS) style is shown in Figure 2. The DATUMS of this style are *request* and *response*. It also contains constituents *client*, *router* and *server*, and DUCTS *client-request*, *client-response*, *server-request*, and *server-response*. Both the client and the server are PARTICLES and are treated as black-box components that are not decomposed into RELAYS and DUCTS. Moreover, the behavior of the CS client's *request* OUTPUT is SEND, that of the CS server's *request* INPUT is HANDLE, that of the CS server's *response* OUTPUT is REPLY, and that of a CS client's *response* INPUT is RECEIVE.

A CS *router* is a single BIRELAY that contains two TWOWAY ports: an *initiator* that combines *input-in* INPUT and *input-out* OUTPUT portals, and a *terminator* that combines *output-in* INPUT and *output-out* OUTPUT portals. This BIRELAY is responsible for ensuring that a response generated by the server is routed back to the appropriate client that made the original request. In addition, all DUCTS other than *server-request* are synchronous flow DUCTS that LOSE *none*, while the asynchronous flow DUCT *server-request* has an *infinite* HOLDS capacity and LOSES *none*.

This composition of the client-server style needs additional constraints on the arrangements of portals of client and server, and that of input/output operations at these portals. For example, a client cycles between making requests and waiting for a response; a server waits for a request and generates a response every time it gets one. Both the CS client and CS server use request and response portals, called the *request interface*, in lock-step. Similarly, the CS router also uses client-request and client-response portals, called the *input interface*, and the server-request and server-response portals, called the *output interface*, in lock-step. Every CS server and router can have at most one request interface. Such stylistic constraints need to be expressed using constraint expression languages as discussed in [20].

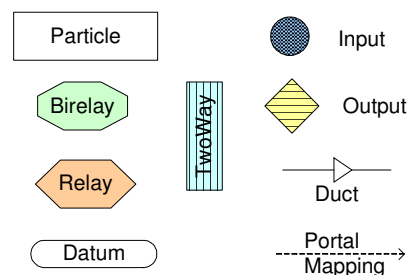


Figure 1 Graphical metaphor for Alfa primitives

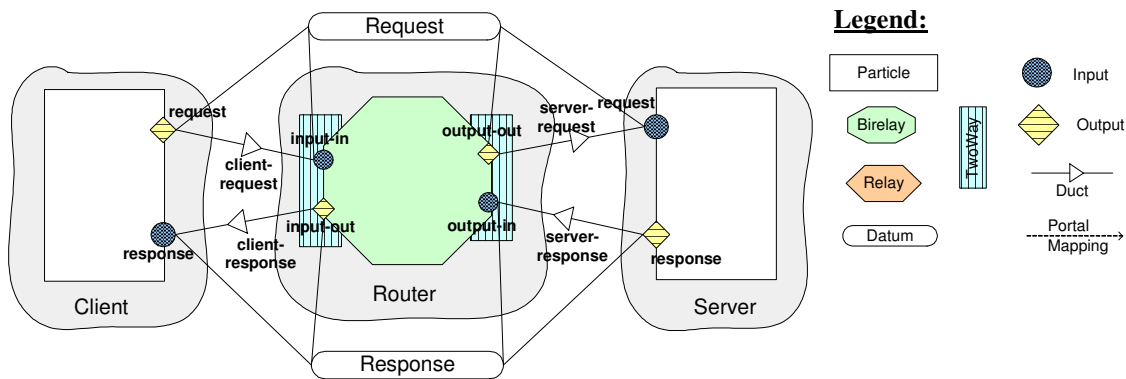


Figure 2 Client-Server style (CS) composition from Alfa's primitives

Without the use of constraint expressions, each portal can be used independently of another portal. Moreover, input/output behaviors would be executed independently on each portal. Although this might be suitable for certain styles, many styles allow only certain static or dynamic arrangements. The use of constraint expressions bounds the valid instances of a style's elements. When used together, constraint expressions make the set of primitives *complete* because the two collectively address all the characteristics of an architectural style.

4. Alfa in action

The above-described Alfa framework for characterizing and composing architectural styles has been applied to various network-based styles identified in [11]. A detailed presentation of the composition of all network-based styles is not possible here due to space constraints. However, the same is available at the Alfa Web site (<http://cse.usc.edu/~softarch/Alfa>). In order to illustrate the reusability of Alfa's primitives, we describe here the composition of two more network-based styles from [11]: C2 and pipe-and-filter (PF).

4.1. Pipe-and-filter style

The pipe-and-filter style has been previously used to illustrate approaches for formalizing and analyzing architectural styles [2]. Table 3 describes the orthogonal characterization of the pipe-and-filter (PF) style based on our approach described above.

Table 3: Orthogonal pipe-and-filter style characteristics

Data	Records are exchanged between pipes and filters. A special <i>end-of-pipe</i> signal is used to indicate that the pipe does not have any further records.
Structure	<p>A <i>filter</i> has interfaces for <i>writing</i> to, <i>reading</i> from, <i>closing</i> pipes, and being <i>notified</i> 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.</p> <p>A <i>pipe</i> has interfaces for <i>source</i> and <i>sink</i> filters, to allow a source to <i>close</i> the pipe, and to <i>notify</i> 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.</p>
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	<p>A filter cannot write to a pipe, once it is notified that the pipe has been closed. A filter cannot close a pipe that has already been closed.</p> <p>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.</p>
Topology	A filter's write interface is connectable to a pipe's source interface, its read interface is connectable to a pipe's sink interface, its close interface is connectable to a pipe's close interface, and a pipe's notify interface is connectable to a filter's notified interface. A filter cannot be connected to a pipe through more than one read/write interface.

The Alfa composition of the PF style is shown in Figure 3. The DATUMS of this style are *record* and *end-of-pipe*, and its composition contains constituents *pipe* and *filter*, and four DUCTS *fwriter*, *freader*, *pcloser*, and *pnotifier*. The PF pipe is a PARTICLE whose composition contains an *Inhibitor* PARTICLE and two RELAYS. Further, the inhibitor is decomposed into interconnected RELAY and DUCT primitives. The PF *filter*, on the other hand, is a PARTICLE that is treated as a black-box component and not decomposed into RELAYS and DUCTS.

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. The *write* and *close* OUTPUT portals employ SEND behavior, while the *read* and *notified* INPUT portals employ RECEIVE and HANDLE behavior, respectively. 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 its being closed. As all the portals of a pipe are in the interfaces of RELAYS, and, hence, automatically HANDLE all incoming and SEND all outgoing DATUM instances.

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 DUCT that has 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 one 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. The last two DUCTS are asynchronous and hence a pipe does not block when SENDING data to them.

The graphical composition of the PF style also shows the internal composition of a pipe where an inhibitor is used to constrain arrival of data on the pipe's source INPUTS to be before any data appears on the pipe's close INPUTS. Further, the record DATUM instances appearing on the source INPUTS and the end-of-pipe DATUM instances appearing on the close INPUTS are propagated through to the sink OUTPUTS of the pipe. Moreover, end-of-pipe DATUM instances are also sent from the inhibitor to all source filters of a pipe to ensure that they do not send any further records to their write OUTPUT.

This behavior of a pipe depends on the behavior of the inhibitor as it correlates a pipe's source INPUTS to its close INPUTS. An inhibitor is itself composed from RELAYS and DUCTS (shown at the bottom of Figure 3). The RELAY *a* propagates the inhibitor's *data* INPUT as the inhibitor's *data-out* OUTPUT under the regulation of the *inhibit* INPUT. It is connected to Relay *b* using a sink DUCT that has zero HOLDS and *none* LOSES. Relay *e*, likewise, propagates DATUM instances at the *inhibit* INPUT as the inhibitor's *inhibit-out* OUTPUT while regulating the propagation of *data*. It is connected to RELAY *c* using a synchronous flow DUCT. RELAYS *b*, *c*, and *d* collectively perform the regulation by forming a memory cell with a capacity to hold two DATUM instances, of which one is initially present. Whenever data is input to the inhibitor, this initial DATUM instance is shifted out and back in to the memory cell. The memory cell is filled with the second DATUM instance as it arrives from the inhibitor's inhibition INPUT, and this causes any further data inputs to the inhibitor to be propagated, as the shifting process inside the mem-

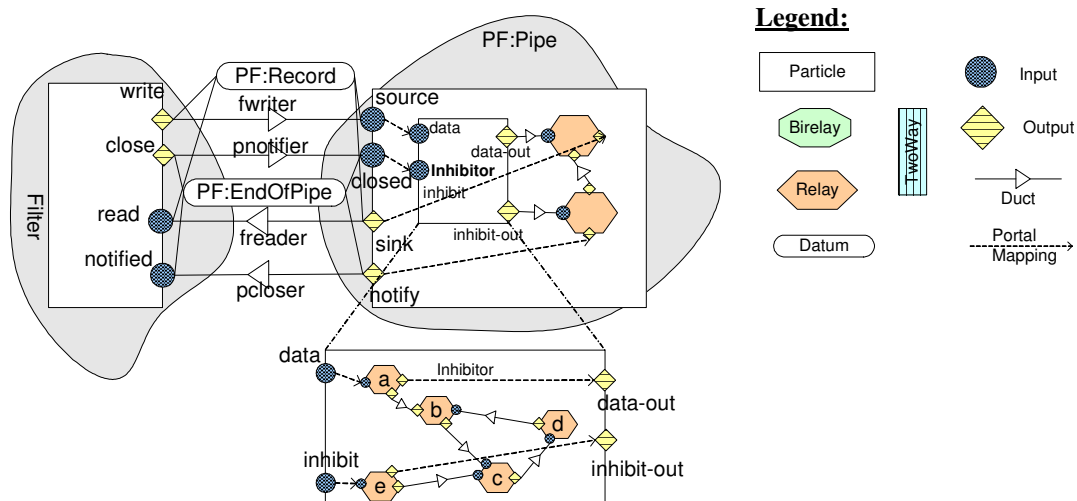


Figure 3 Pipe-and-Filter (PF) style composition from Alfa's primitives

ory cell ceases. To achieve this, the DUCT connecting RELAY b to RELAY c is a synchronous flow DUCT that LOSES *none*, the DUCT connecting RELAY c to RELAY d is an asynchronous flow DUCT that HOLDS one and LOSES *none*, and the DUCT connecting RELAY d to RELAY b is an asynchronous flow DUCT that HOLDS one and LOSES *initial*. Such a compact representation of the complex inhibition behavior speaks a lot about the expressive power of Alfa’s primitives. Our inspiration for this model of interaction is Reo [4], where similar circuits can be constructed from arbitrary primitive channels.

When a filter is CREATED, all its portals are also implicitly instantiated. Similarly, when a pipe is CREATED, its internal elements, including its two RELAYS, three DUCTS, and the inhibitor along with its five internal RELAYS and DUCTS, are automatically instantiated. 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.

This composition of the pipe-and-filter style also needs additional constraints on the arrangement of portals on instances of pipes and filters. For example, three portals of a filter—write, close, and notified—should always be used in tandem. Moreover, every filter should have either at least one read portal or at least one set of the three portals above. Such constraints involve the use of a suitable expression language for restraining arrangements of Alfa’s primitive forms or for the arrangement of input/output behaviors of a style’s valid constituents. No constraint expressions are required on the arrangements of input/output behaviors in the PF style’s elements as these are completely modeled by Alfa’s primitives.

4.2. C2 style

The C2 style is a layered network of concurrent components that communicate via neighboring connectors [33]. In order to compose this style from Alfa’s primitives, we characterize the style along five orthogonal dimensions as shown in Table 4.

Table 4: C2 style characteristics

Data	<i>Notifications and requests are exchanged.</i>
Structure	<i>Components contain an interface for producing and consuming notifications and requests. 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> <i>Connectors contain interfaces for producing and consuming notifications and requests. For every produce notification interface, a component must have a consume request interface, and vice versa.</i>
Interaction	<i>Components and consumers do not block for consuming requests and notifications.</i>
Behavior	<i>Connectors broadcast notifications received on each of their consume notifications interfaces to their produce notifications interfaces, and vice versa.</i>
Topology	<i>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.</i> <i>Connected components and connectors cannot form a loop.</i>

The Alfa 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). Each C2 *component* is a PARTICLE, which is split across the diagram into two PARTICLE shapes. It is treated as a black-box component and is not decomposed into RELAYS and DUCTS. Also, note that the constituent C2 *connector* is a PARTICLE whose composition contains two asynchronous flow DUCTS that each HOLDS *infinite* data items, and RELAYS ti (top-incoming), to (top-outgoing), bi (bottom-incoming), and bo (bottom-outgoing). This composition collectively determines the behavior of a C2 connector. The internal DUCTS of a connector are designed to be asynchronous so that components do not block while producing notifications or requests. In the interest of space, we omit the discussion about internal portals of a C2 connector. Portals in the external interface of the C2 connector are mapped to portals of the internal RELAYS of a C2 connector. Moreover, the INPUT portals of C2 component and connector are defined to have the HANDLE behavior, which is required to ensure that they do not block for consuming requests and notifications, as required by the interaction characteristics in Table 4. The OUTPUT portals of C2 component and connector are defined to have the SEND behavior.

When a C2 component is CREATED, all its portals are also implicitly instantiated. Similarly, when a C2 connector is CREATED, its internal elements including its four RELAYS and two DUCTS are automatically instantiated. The CONNECT primitive is used for establishing a DUCT between corresponding por-

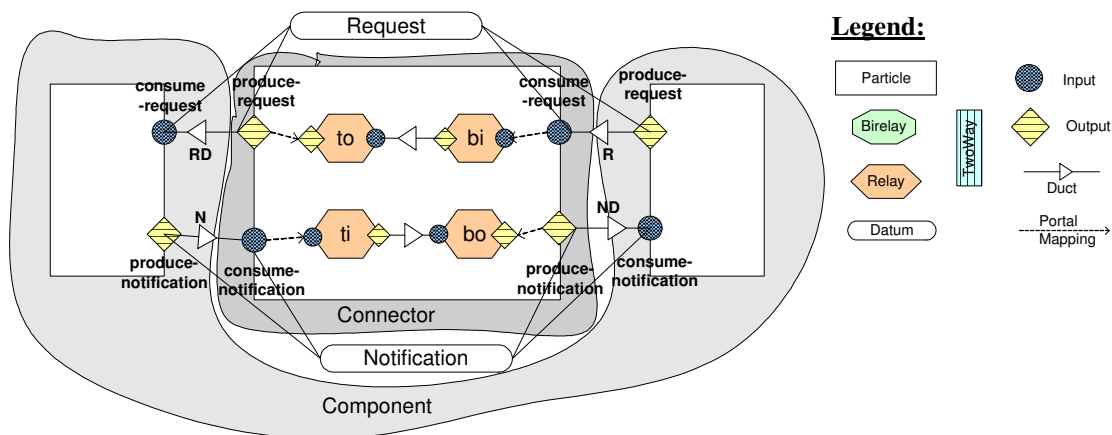


Figure 4 C2 style composition from Alfa's primitives

tals of a C2 component and connector. The DISCONNECT primitive is used in a similar manner for removing any DUCT thus created.

Additional constraint expressions are not required on the dynamic arrangements of input/output behaviors in the C2 style's elements. However, constraint expressions are required for ensuring correct structural arrangements of C2 components and connectors. Both the C2 component and connector use produce-notification and consume-request portals, called the *bottom interface*, in lock-step. Similarly, they also use produce-request and consume-notification portals, called the *top interface*, in lock-step. Moreover, every C2 component can have at most one top interface and at most one bottom interface. Further a topological relationship is required to track the indirect connectivity of a component to other components through a connector. The transitive closure of this relationship of a component cannot contain the component itself in order to prevent loops. The topological relationship and this transitive closure constraint also require additional constraint expressions.

An important implication of the composition of the C2 style in this manner is that the long sought "higher order" C2 components and connectors [33] can be composed from a combination of simpler C2 components and connectors, where the outward facing portals of the composite elements can be mapped to the portals of internal elements.

5. Discussion

The previous sections have discussed a framework for composing elements of an architectural style from architectural primitives and illustrated its use in the composition of several network-based styles [11]. In this section we discuss the strengths and limitations of the Alfa framework by assessing the reusability of its primitives over the entire set of network-based styles and arguing that Alfa is an architectural assembly language.

The composition of eighteen network-based styles identified by Fielding [11] has allowed us to assess the use of primitives across a diverse set of architectural styles. As shown in Table 5, we find that Alfa's primitives are widely *reusable* across these styles, satisfying the third condition for an architectural assembly language. The table shows a shaded cell when a primitive is used in the style corresponding to the cell and an empty cell otherwise. We find that five patterns of primitive usage can be observed in the table corresponding to five basic styles¹: pipe-and-filter (PF), client-server (CS), event-based interaction (EBI), C2, and virtual machine (VM), of which the first three styles have been discussed in this paper.

Although Alfa's primitives are sufficient to compose elements of styles, constraint expressions are often required to limit the static and/or dynamic arrangements of these elements in style-based architectures. These expressions are also necessary for black-box components whose behavior cannot be determined as a simple composition of its primitives. Hence, there is a need to supplement Alfa's primitives with additional notations for recording such constraints of architectural styles. When combined with these expression languages, Alfa's primitives indeed form a *complete* set, the first condition for

1. A basic style is defined by Fielding [11] as a style that is not obtained by the addition of constraints to another existing style.

Table 5: Reuse of Alfa's primitives across network-based styles. Basic styles are highlighted

Style	Pipe & Filter	Uniform Pipe & Filter	Client-Server	Replicated Repository	Cache	Layered System	Layered Client-Server	Client-Cache-Server	Layer-Client-Cache-Server	Remote Evaluation	Code-On-Demand	La-Cl.-Cod.-Dem-Ca.-Ser.	Mobile Agent	Distributed Objects	Brokered Dist. Objects	Virtual Machine	Event-based Interaction	C2
DATUM																		
PARTICLE																		
INPUT																		
OUTPUT																		
TWOWAY																		
DUCT																		
RELAY																		
BIRELAY																		
HOLDS																		
LOSES																		
CREATE																		
SEND																		
HANDLE																		
RECEIVE																		
REPLY																		
CONNECT																		
DISCONNECT																		

an assembly language. A detailed discussion of these expressions is beyond the scope of this paper, but can be found in [20].

The most common constraint expression we found necessary for composing elements of styles was the lock-step use of a group of portals, i.e., all the portals in this group are instantiated in every architectural element of that type of style constituent, or none of the portals is instantiated. Such a constraint is often captured as an interface in programming languages, where a group of functions are combined to form an interface, and any concrete implementation of the interface should implement each of its functions. Therefore, we plan to model the lock-step use of a number of portals as a special type of constraint expression called interface. This should simplify constraint expressions and, in turn, analysis of style-based architectures, while improving the understanding of style compositions. Another common constraint expression needed was the support for topological relations among style elements. Such relations should also be modeled as a special type of constraint expression as it simplifies the topological constraints on configurations of style elements, i.e., interaction paths that are necessary or disallowed in the style.

A systematic notation to compose architectural styles from Alfa's primitives is needed in order to streamline the use of the Alfa framework. This notation should provide constructs for Alfa's primitives as well as for the additional constraint expressions. Since Alfa's primitives satisfy all three conditions for an architectural assembly language, such a notation can be used to systematically assemble style-based software architectures. We have begun to design a notation for composing style constituents from Alfa's primitives, called xAlfa [20]. xAlfa also 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.

6. Conclusion and future work

Compositional approaches towards software architectures remain an area of active interest. This paper describes one such bottom-up approach, Alfa, that recognizes architectural primitives. Alfa is a framework for the characterization and composition of architectural styles from such primitives. Alfa characterizes architectural styles along five orthogonal dimensions to enable their systematic composition from primitive building blocks. This characterization approach has been applied to a diverse set of architectural styles for network-based systems.

We should emphasize that we do not assume Alfa's set of primitives to be the only possible, or even best such set. In fact, our previous work had suggested a different such set of primitives that proved to be applicable to a narrower class of architectural styles [19]. We recognize that further research in this area is likely to lead to better architectural assembly languages, or languages applicable to a wider class of styles. At the same time, the Alfa framework presented in this paper forms the first step along that trajectory.

We have evaluated Alfa based on three necessary conditions for a software architectural assembly language: completeness, implementability, and reusability. Collectively, Alfa's primitives are able to address all the five orthogonal dimensions of architectural styles, and hence form a complete set when augmented with a constraint expression language. Moreover, each of the primitive is implementable as per the discussion in Section 3. Finally, we have seen that the primitives of Alfa are reused across network-based styles. As a result, we feel confident in terming Alfa an assembly language for style-based software architectures. Although their use has only been evaluated in the domain of network-based systems, Alfa's primitives are found to be expressive enough to compose any architectural style that employs interactions involving a regular expression of input/output operations on point-to-point channels, sufficient for a large class of modern, distributed systems.

Much work remains to be done for supporting the creation and use of style compositions. We are currently working on a notation for Alfa, called xAlfa, for systematically composing Alfa's primitives into architectural styles and elements of style-based architectures. xAlfa will also provide support for the use of expression languages to specify static and dynamic constraint expressions. Additionally, we plan to provide visualization tools to render Alfa style compositions using a graphical metaphor. Using xAlfa style compositions, we also plan to support conformance checking of such style-based architectures to their styles. Also, in a related study [22], we have shown that constraint automata effectively model architectural assemblies of Alfa primitives. We plan to leverage this and combine the dynamic constraints on style compositions with the semantics of Alfa's primitives to produce effective dynamic models of style-based architectural assemblies.

Last, but not 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 Alfa compositions.

References

- [1] G. D. Abowd, R. J. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology* 4 (4) (1995) 319-364.
- [2] R. J. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology* 6 (3) (1997) 213-249.
- [3] F. Arbab. Abstract behavior types: A foundation model for components and their composition. Proc. 1st Symposium on Formal Methods for Components and Objects, Leiden, The Netherlands, LNCS vol. 2852, Springer, 2002.
- [4] F. Arbab. Reo: A channel-based coordination model for component composition. To appear in *Mathematical Structures in Computer Science*, Cambridge University Press, 2003.
- [5] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. Proc. 24th International Conference on Software Engineering, May 2002.
- [6] C. G. Bell and A. Newell. *Computer structures: Reading and examples*, McGraw-Hill, New York, 1996.
- [7] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebra. *ACM Transactions on Software Engineering and Methodology* 11 (2002) 386-426.
- [8] M. M. Bonsangue, J. N. Kok, and G. Zavatarro. Comparing coordination models and architectures using embeddings. *Journal of Science of Computer Programming* 46 (2003) 31-69. Elsevier Science.
- [9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, Chichester (1996).
- [10] E. Dashofy, A. van der Hoek, and R. N. Taylor. An Infrastructure for the Rapid Development of XML-Based Architectural Description Languages. Proc. 24th International Conference on Software Engineering, May 2002.
- [11] R. Fielding. *Architectural styles and the design of network-based software architectures*. Ph. D. Dissertation, University of California at Irvine, 2000.
- [12] D. Garlan, S. Cheng, and B. Schmerl. Increasing System Dependability through Architecture-Based Self-Repair. *Architecting Dependable Systems*, R. de Lemos, C. Gacek, and A. Romanovsky. (eds). Lecture Notes in Computer Science, vol. 2677, Springer, 2003.
- [13] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*. G. T. Leavens, and M. Sitaraman. (eds). Cambridge University Press, 2000.
- [14] D. le Metayer. Describing architectural styles using graph grammars. *IEEE Transactions on Software Engineering* 24 (1998) 521-533.

- [15] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 21 (4) (1995).
- [16] N. Medvidovic, M. Mikic-Rakic, N. R. Mehta, and S. Malek. Software Architectural Support for Handheld Computing. *IEEE Computer Special Issue on Handheld Computing*, 36 (9) (2003) 66-73.
- [17] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. Proc. 21st International Conference on Software Engineering, May 1999.
- [18] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26 (1) (2000) 70-93.
- [19] N. R. Mehta and N. Medvidovic. Distilling software architectural primitives from architectural styles. University of Southern California Center for Software Engineering Technical Report USC-CSE-2002-509.
- [20] N. R. Mehta and N. Medvidovic. Concise composition of architectural styles from architectural primitives. University of Southern California Center for Software Engineering Technical Report USC-CSE-2003-510.
- [21] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
- [22] N. R. Mehta, M. Sirjani, and F. Arbab. Effective Modeling of Software Architectural Assemblies Using Constraint Automata. University of Southern California Center for Software Engineering Technical Report USC-CSE-2003-509.
- [23] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N. J, 1989.
- [24] R. T. Monroe. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163.
- [25] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *Proc. 4th International Conference on Software Reuse*, Orlando, FL, USA, 1996.
- [26] D. L. Parnas. On the criteria to be used for decomposing systems into modules. *Communications of the ACM* 15 (1972) 1053-1058.
- [27] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes* 17 (1992) 40-52.
- [28] M. Shaw. Comparing architectural styles. *IEEE Software*. 12 (6) (1995) 27-41.
- [29] M. Shaw. Procedure calls are the assembly language of software interconnections: Connectors deserve first-class status. Proc. Workshop on Studies of Software Design, 1993.
- [30] M. Shaw and P. Clements. A field guide to boxology: preliminary classification of architectural styles for software systems. Proc. 21st Computer Software and Applications Conference, Washington, DC, USA, 1997.
- [31] M. Shaw and D. Garlan. *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.
- [32] B. Spitznagel and D. Garlan. Architecture-Based Performance Analysis. Proc. 10th International Conference on Software Engineering and Knowledge Engineering, June 1998.
- [33] R. N. Taylor et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22 (6) (1996) 390-406.