Distilling Software Architectural Primitives from Architectural Styles

Nikunj R. Mehta
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA.
+1 213 740 6504
mehta@usc.edu

Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
+1 213 740 5579
neno@usc.edu

Abstract

Architectural styles codify commonly used idioms in system structures, organizations, and interactions. Existing classifications of architectural styles provide empirical guidelines for style selection, but fail to systematically characterize styles or reveal their foundations. Moreover, the mapping between architectural styles and their implementations is generally poorly understood. This paper introduces an approach towards a systematic understanding of architectural styles based on a small set of recurring architectural primitives. Also proposed is an extensible composition framework for architectural styles based on these primitives, which is, in turn, used as the basis of style implementation. This framework, called Alfa, has been applied successfully to four different architectural styles to date. Lightweight compositional models of the styles have been created using the Alloy formal modeling notation, whereas implementation infrastructures for these styles have been developed in Java. While it would be premature to claim that Alfa is sufficiently expressive to represent any arbitrary architectural style, our experience to date suggests that we have converged on a set of primitives that effectively captures the characteristics of a large number *of styles. As such, we feel that Alfa already presents a unique contribution and is worthy of careful further study.*

1. Introduction

As software systems become more complex, there is an increasing need for employing higher-level abstractions to better represent the system structure, behavior, and interactions in ways that can support reasoning about the system properties. *Software architectures* [28,30] provide such high-level abstractions in the form of coarse-grained system *components* (computational building blocks), *connectors* (component interaction facilities), *configurations* (specific compositions of components and connectors), and *constraints* placed on them [19]. The system composition patterns and their constraints comprise *architectural styles*, which are targeted at families of systems with shared characteristics [1,25]. Styles are therefore reusable software architectural idioms.

There are many architectural styles currently in use: client-server, pipe-and-filter, push-based, layered, blackboard, and so forth [30]. A large number of these styles have not been formally described, and the exact characteristics of a given style are usually unclear. Even more difficult is the systematic comparison of different styles to understand their strengths and weaknesses in order to enable the selection of styles most appropriate for a given application. Attempts have been made to develop systematic techniques for dealing with architectural styles resulting in formalisms to describe styles [1,15], preliminary taxonomies [29], and informal discussions of the differences between a newly codified style and existing styles that have influenced it [11,31]. However, for the most part these studies fail to clarify the key dimensions along which one architectural style may differ from another.

Another related, critical issue is ensuring that a property that is established in the model of a system's style-based architecture holds true of the system's eventual implementation [18]. Current research has begun to address this issue by creating architectural implementation and execution frameworks for specific architectural styles [4,17]. However, it remains difficult to provide uniform and economical implementation techniques for a wide variety of architectural styles with sufficient flexibility so that they can be adapted for use in real applications.

The work presented in this paper proposes to create an understanding of and, as a result, a composition framework for architectural styles, called *Alfa*, using a small, *reusable* set of architectural primitives. In turn, these primitives are reused across styles and reified using precise transformations to produce implementation infrastructures faithful to each style. Our work attempts to answer three fundamental questions about the nature of architectural styles:

1. How can architectural styles be comprehensively characterized?

2. Are there essential similarities and differences between architectural styles?
3. Can complex, divergent architectural styles be composed from a reusable set of (simple) primitives?

In order to answer these questions, to date we have successfully modeled, analyzed, and implemented the structural and compositional aspects of four well-known architectural styles using Alfa: C2 [31], client-server [9], pipe-and-filter [6], and push-based [11]. While these styles are not a representative cross-section of all architectural styles, as a collection they embody a large number of recurring architectural concepts: distribution, concurrency, events, implicit invocation, dynamism, remote procedure calls, layering, publish-subscribe, dataflow, and so forth. Representing architectural styles using a single substrate has also allowed us to formally demonstrate, by reusing the same compositions (i.e., patterns) of Alfa elements, various informal claims and intuitions stated in literature about shared aspects of different styles. For example, we have found similarities between client-server and push-based systems in terms of the topology of architectural elements. Another example is the similarity between C2 style implicit invocation and client-server request dispatch. Such similarities are exploited to create opportunities for low-level reuse in the models and implementations of architectural styles.

The main contributions of our work are

1. a novel approach for architectural style characterization, understanding, and composition;
2. a small set of architectural primitives that are reused across styles;
3. formal models of styles that are compositional and lightweight; and
4. effective style implementation infrastructures.

The paper is organized as follows. After motivating our work in the next section, Section 3 presents related work and concepts. Section 4 introduces Alfa, its formal model, and its implementation infrastructure. In the process, this section discusses architectural composition and explains the nature of architectural primitives. Section 5 presents the details of Alfa's evaluation by applying it to four different architectural styles. Section 6 evaluates the framework in terms of reusability and discusses limitations of our work. The paper concludes with conclusions and a brief overview of our future plans.

2. Motivation

It is widely believed that compositional approaches to software development are key to developing large, complex systems [2,7,27,28]. Existing approaches, recently including software architectures, have provided support for 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. Delivering the full value of architecture-based design, however, would require that we identify the primitives underlying software architectural elements and the correspondence between various levels of software design abstractions (e.g., analogously to how this is done in computer hardware architecture [8]). However, there is an almost complete lack of understanding of such software architectural primitives, directly motivating our study.

Architectural styles in particular represent a potentially fertile area for attempting to codify architectural primitives. Styles facilitate systematic, high-level reuse in the form of recurring organizational patterns of coarse-grained architectural elements (software components and connectors). A difficulty associated with styles is that most real-world software systems cannot be built using a single or a "pure" architectural style. Instead, architects are required to select relevant aspects of various styles to fit their application needs. It is a daunting task for them to compose, model, and analyze their own hybrid architectural styles in the absence of common architectural primitives that can be used as building blocks. Another difficulty with architectural styles is the current lack of support for systematically implementing systems using a particular style. The availability of composition frameworks for architectural styles and their effective implementation based on architectural primitives will mitigate such difficulties and make principled use of styles more feasible.

Identifying primitives of architectural styles requires a systematic characterization of styles, but existing classifications [29] do not go beyond empirical comparisons of styles and provide little information about the underlying style elements. Moreover, since many architectural styles are available to an architect, a precise understanding of the relative merits and limitations of different styles in terms of comparable characteristics will enable reasoning about the choice of style(s) for designing applications. Hence there is a need for a style characterization approach that is linked to shared primitives. Identifying such primitives would not only improve our understanding of architectural styles, but possibly also further our understanding of the very nature of software architectures.

Unsatisfied with the current level of understanding of architectural primitives and their compositions into styles, we have ventured to create an architectural "assembly language". We believe that a unified approach to modeling architectures at four levels of abstraction—architectural primitives, architectural elements, architectural styles, and software system architectures—is possible and that such an approach will propel us towards systematically and economically supporting these levels of abstraction. While clearly outside the scope of our current work, these architectural "assembly language" primitives should be ultimately mapped to implementation constructs in practical programming languages, to help create systems that are provably consistent with their architectural models.

3. Related work

Our work has been influenced by a number of research areas that focus on the conceptual and implementation aspects of large-scale systems. We focus here on existing research and concepts in the domains of architectural styles and middleware platforms.

Architectural styles: Architectural styles have been proposed to reduce the complexity of software design by leveraging successful solutions to past problems [25]. Styles codify the best design practices and successful system organizations [7,16]. They have been considered as an economical way of developing architecture-based systems [24]. Architectural styles embody collections of *constraints* that define legal *configurations* of *components* and *connectors* for a given family of systems [1]. At the same time, existing styles typically share the view that software components are application-specific entities, placing few, if any, constraints on their functionality, behavior, and internal structure.

Formal techniques have been used in conjunction with styles to provide rigor to their definitions and thus enable analyses of the styles as well as of applications developed using the styles. Several approaches for formally modeling styles have been proposed. Abowd et al. have developed a Z-based approach for capturing the syntax and semantics of styles [1]. Although this method allows one to compare a style with another in terms of syntactic and semantic constraints, this approach neither identifies the fundamental elements of a style nor provides any support for implementing styles. Jackson et al. describe and analyze architectural styles in terms of first-order logic expressions using Alloy, a lightweight notation for describing structures [12]. However, their approach is unable to deal with complex configurations of architectural elements and does not model distribution and concurrency well. Achermann et al. propose a p-calculus based compositional approach to styles [2]. An interesting aspect of this approach is the composition of low-level primitives such as *agents* (to model processes), *channels* (to model communication paths) and *forms* (to model data) to create architectural elements such as components and connectors. Among major shortcomings of this approach are the inability to support distribution and its low scalability (every agent runs in a different thread). Finally, Le Metayer provides a graph-grammar based technique as a formal counterpart to the box-and-line diagrams used for describing software architectures [15]. This approach enables the visualization of interconnections of style elements, but does not extend our understanding of the internal organization of the style elements.

A preliminary classification of architectural styles highlights the nature of communication and coordination in the use of styles [29]. An interesting proposition of this classification is that the types of interactions supported in the style largely govern its characteristics. This classification distinguishes between styles on the basis of control and data interactions as well as the types of analyses relevant to these styles. Since architectural styles may differ from one another in terms of their basic interaction patterns, and require possibly conflicting configurations of components and connectors, the combined use of certain architectural styles in a single system can lead to unpredictable mismatches that can be expensive to resolve. Initial efforts have been made to study architectural mismatches [10], but there is still insufficient understanding of style incompatibilities that cause those mismatches.

Middleware: Effective use of architectural styles in development settings requires the availability of robust middleware technologies. Various middleware platforms have emerged (e.g. CORBA [26], COM [22], Enterprise Java Beans [13]) using specific forms of component interactions such as RPC, distributed transactions, and security. Middleware platforms provide a standard interface to low-level operating system and network services, and a runtime environment for deploying components based on interoperability guarantees [3]. Reflective and composable middleware enables developers to specify complex interaction patterns used in applications through high-level abstractions by carefully separating structure from behavior [14]. However, most middleware technologies adopt an inflexible approach towards the interaction aspects of systems by allowing very few forms of interactions among components. Further, these approaches are not based on an understanding of any fundamental building blocks of software or architecture styles, but rather focus on creating libraries of adaptable glue code that can be put together to meet deployment and interoperability needs.

4. Details of Alfa

In order to answer the research questions posed in the Introduction, we propose a composition framework for styles based on the use of *architectural primitives*. The framework includes a systematic characterization technique for styles based on their static and dynamic aspects. The framework also contains an initial set of primitives for composing style elements, with support for the extension of this initial set. Finally, the framework provides a precise implementation technique for the architectural primitives and the style elements composed from them.

4.1. Characterizing styles

Architectural styles constrain architectural models syntactically and semantically [1]. In general, the syntactic aspects can be described in terms of structure and topology, whereas the semantic aspects are captured in the form of behaviors and interactions. Hence we propose to characterize architectural styles along the following five dimensions.

1. The external *structure* of the architectural elements allowed in a style — the structure denotes the interfaces offered by the elements in terms of entry and exit points as well as required and provided services.

2. The allowed *topologies* of architectural elements — topological constraints determine how the elements can be legally connected to each other in a system.
3. The allowed *behavior* of a style element — the behavior captures the element's internal state, dynamics (i.e., the modes of operation and transitions among them), and function (i.e., the details of the processing algorithms implemented in the element).
4. The types of supported *interactions* between style elements and their allowed specializations — the interactions are captured in terms of four basic service categories: communication, coordination, conversion, and facilitation [21].
5. The quantity and types of *data* exchanged between style elements.

While these five dimensions do not necessarily capture all aspects of an architectural style, they form a sufficiently expressive foundation for our study. The approach, called *Alfa* ("*a*ssembly *l*anguage *f*or *a*rchitectures"), embodies the five dimensions of architectural styles in a set of basic architectural assembly primitives that recur in various styles and may be extended as well as composed to create more advanced style elements and, indeed, entire styles.

4.2. The basics of Alfa

Figure 1 shows the basic Alfa primitives using a visual metaphor (defined in the *Legend*) as a way of semi-formally representing the Alfa primitives and their compositions. Structural primitives are called *particles* and *portals.* Particles serve as containers of functionality and have autonomous or reactive behaviors depending on their ability to control threads of execution. Portals mark the entry and exit points of particles, and are the conduits of interaction, i.e., they are the loci of data and control transfer between two particles. Topological primitives, called *jumpers*, provide connectivity between particles. Jumpers are simple *ducts* [21] used only to create paths of interaction among particles, with no additional semantics.

An important separation of concerns in Alfa is between the structure and behavior of Alfa primitives. Each particle and portal provides a separate *controller* for specifying its behavior. Controllers are Alfa's behavioral primitives and are also used to handle style-specific interaction protocols, such as the client-server style requirement of connection setup and teardown. Particles (and their associated portals) provide the "form" of architectural style elements, whereas controllers provide their "substance." This allows one to compose the structure of Alfa primitives independently of their behavior, and to dynamically alter one without affecting the other. The controllers also manage the life cycle of primitives as they move between three basic states: halted, executing, and suspended. An associated Alfa primitive is *thread*, which provides multi-tasking ability.

Computation behavior is separated from the interaction behavior in style elements by providing two types of *controllers*, namely *fragments* and *transports*. Fragments provide the computational logic that varies by application and style, whereas transports provide the semantics of interaction between fragments that often repeats across applications. This separation maps nicely to the component-connector dichotomy of software architectures. Continuing with the previous client-server example, message distribution would be managed in a *transport* controller, whereas the *fragment* controller would focus on application-specific processing.

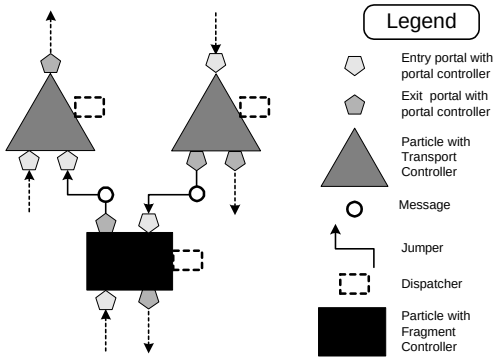

Figure 1 Alfa primitives

Data primitives organize and carry the contents of interaction over jumpers from one particle to another. Two types of data primitives are available, *messages* and *calls*. Messages carry a type identifier and payload information, whereas calls carry parameters, results, and exceptions, in addition to a call type identifier. Messages are used for one-way communication, whereas calls are used for two-way communication: sending parameters to the processing element, and retrieving results and exceptions after processing.

Interaction primitives are based on various connector dimensions, such as synchronization, notification, and delivery policies [21], and are used within the particles to provide the interaction services required by those particles. The basic interaction primitive is a *dispatcher*, which acts to schedule the processing of incoming messages. By default, all dispatching is done synchronously, i.e., instantaneously in the same execution thread as the sender of the message.

4.3. Formal Model Of Alfa

Alfa structural and topological primitives are modeled using a lightweight and compositional notation, Alloy, which has been found well suited for formal architectural modeling [12]. Alloy has a simple syntax for expressing structures and constraints using first-order logic. Alloy models may be analyzed using automated tools to discover inconsistencies in model constraints, and to simulate instances of a model. Alloy helped us to develop a measure of confidence in Alfa's structural and topological primitives, by providing an automated analysis of the formal

```
1    module Alfa/core
2    sig Portal {}
3    sig Particle {}
4    part sig Input, Output extends Portal {}
5    sig Structure {
6       particles: set Particle,
7       output: particles ?-> Output,
8       input: particles ?-> Input,
9       jumper: Output ?-> Input } {
10      some particles
11      all p: particles | all i: input[p] | one i.~jumper
12      all p: particles | all o: output[p] | one jumper[o]
13      all p: particles | all a, b : output[p] |
           input.(jumper[a]) = input.(jumper[b]) => a = b
14      all i: Input | some i.~jumper => some i.~input
15      all o: Output | some jumper[o] => some o.~output } …
```

Figure 2 Alloy model of Alfa primitives

descriptions of primitives, and by generating valid instances of systems based on their models. Analyses helped us in uncovering those constraints that were either too loose or too tight to allow correct compositions of Alfa primitives. For example, due to the vertical layering of Alfa primitives, early versions of the Alloy Alfa model considered the orientations of portals to be either from the top or from the bottom. However, analysis of Alfa models revealed that this was irrelevant, and only input or output portals were required. Analysis also helped debug the models and correct them. Of particular help was the visualization support in the Alloy analysis tool, which helps create a graphical view of the instances of a model. While generic to any Alloy model, that view may then be relatively easily compared with the "native" Alfa view (recall Figure 1). Finally, Alloy has been a moving target as much as the model of Alfa primitives. Recent additions of modularity in Alloy have also simplified the development of a compositional model of Alfa.

Figure 2 shows an excerpt of the Alloy model describing the basic Alfa structural and topological primitives. This model supports arbitrarily complex interconnections of particles and jumpers that are composed based on the simple rules of Alfa composition (e.g., an *Input* portal may only be connected to one *Output* portal via a *jumper*, a system contains a non-empty set of particles, and so on). The Alloy model provides an analyzable description of the structural and topological primitives shown in Figure 1 (namely, aspects of particles, portals, and jumpers) thus providing formal support for the visual modeling notation we have developed for Alfa. In addition to the static invariants of Alfa, structural architectural dynamism is also modeled using Alloy. The full model comprises 50 lines of Alloy code and may be found at [5].

Alfa also imposes a set of simple constraints on valid configurations of Alfa primitives. In that sense Alfa itself may be considered an architectural style. Three stylistic constraints are described in a separate Alloy model layered on top of the Alfa core model; an excerpt of this 86-line model is shown in Figure 3: (1) transports may only be linked to fragments, and vice versa; (2) a particle may not

```
1.   module Alfa/style
2.   open Alfa/core
3.   part sig Transport, Fragment extends Particle{}
4.   sig System extends Structure {
5.      link: particles -> particles } {
6.      all p: particles | input.(jumper[output[p])=link[p]
7.      all p: particles | p in Transport => link[p] in Fragment
8.      all p: particles | p in Fragment => link[p] in Transport
9.      no p: particles | p in p.link
10.     all p: particles | #particles != 1 =>
           p in link[particles – p] || some p2: particles – p | p2 in
           link[p]} …
```

Figure 3 Alloy model of Alfa style

be linked to itself; and (3) all nodes in a valid system should form a connected graph. These constraints directly ensure two important architectural concerns [30] in Alfa compositions: computation should be separate from interaction, and modeling and analysis of system architectures from a global perspective should be possible.

4.4. Implementation of Alfa

We have chosen not to model the behavioral, interaction, and data primitives of Alfa using Alloy since it is not suited to algorithmic details and distribution, and has limited scalability in the number of objects in a model instance. Instead of selecting another modeling language, we have chosen to implement all the Alfa primitives directly in Java after ensuring a one-to-one correspondence with the structural and topological aspects captured in Alloy. The complete implementations of Alfa primitives can be found at [5].

Figure 4 shows a view of the UML model of the Alfa implementation. Note that Alfa structural and topological primitives (e.g., particle and portal) are mapped to corresponding classes and associations in the UML design (*Particle* and *Portal*). The jumper primitive is represented as the association *jumper* between two Portal objects. The stylistic constraints of Alfa (recall Figure 3) are enforced through the *ArchitectureManager* class shown in Figure 4; this class is also used to control the creation and composition of Alfa primitives to form style elements discussed below. The complete design of Figure 4 has been implemented in Java to create the Alfa implementation framework.

The Alfa implementation framework has allowed us to experiment and learn about the key aspects of Alfa and, by extension, other architectural styles constructed on top of Alfa, in a manner that is completely faithful to their model. Since the Alfa behavioral, interaction, and data primitives are modeled directly in Java, the framework has also helped us refine the characteristics of these primitives and evaluate their suitability for composing architectural styles, further discussed in the next section.

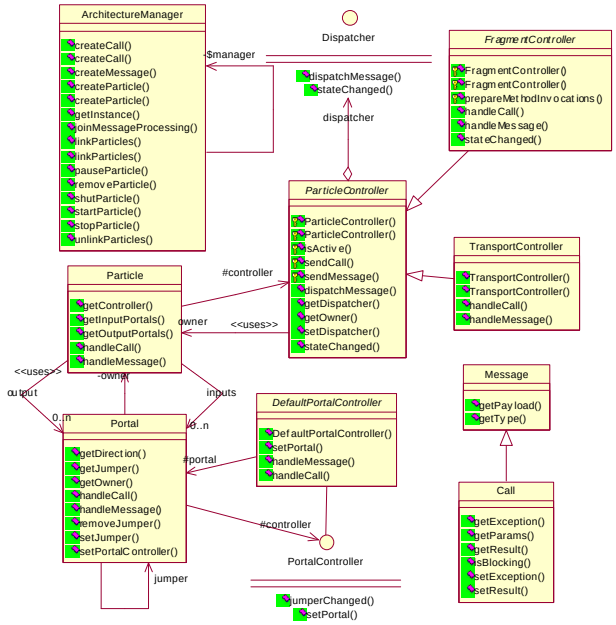

Figure 4 UML model of Alfa primitives

5. Modeling styles in Alfa

Due to the variety, heterogeneity, and a general lack of precision of architectural styles, we chose an iterative approach in developing a composition framework for styles. The first iteration involved arriving at a starting set of primitives through a "bootstrapping" step. This step consisted of a careful study of a number of existing architectural styles, in which we identified the fundamental style characteristics along each of the five dimensions identified in Section 4.1. We also leveraged our experience in designing and building implementation frameworks for a specific set of architectural styles [17,23]. This led us to a small, initial set of basic Alfa primitives, which were modeled using Alloy and implemented in Java as described in the previous section.

With the initial set of primitives identified, the next iteration involved applying Alfa to specific architectural styles. We have performed this iteration using four different styles: C2 [31], client-server [9], pipe-and-filter [6], and push-based [11]. While these styles are not a representative cross-section of *all* architectural styles, as a collection they embody a large number of recurring architectural concepts: distribution, concurrency, events, implicit invocation, dynamism, remote procedure calls, layering, publish-subscribe, dataflow, and so forth.

5.1. Style characterization

The first step in this iteration is a characterization of the candidate style in terms of the five dimensions identified in the previous section (shown in Figure 5). This characterization enables a systematic analysis of the similarities and differences among styles. An example of similarity between styles can be found in the topology of clients and

Style \ Dimensions	Structure	Topology	Behavior	Interaction	Data
C2	Separable components	Limited component dependencies	Exposed via named services only	Asynchronous coordination	Discrete events
		Partially-ordered connectivity-based "top" and "bottom" relations	Data queuing and buffering by connectors	Implicit invocation	
	Explicit connectors			Event-based interaction	Data tuples
		Dynamic creation of connections	Multi-tasking mechanisms such as threads	Direction oriented events propagated to topology-based recipients	
Client-server	Independent servers	Many-to-may connections among clients and servers	Listening server	Server location	Parameterized request
			Connection setup and teardown	Remote connection and communication protocol	
	Specialized clients	Dynamic creation of connections	Buffering and queuing of requests	Implicit server invocation	Typed response
			Multi-tasking mechanisms such as threads	Data marshalling and unmarshalling	
	Distributed protocol stacks		Exposed via named services only	Client call synchronization	
				Request-response protocol	
Pipe-and-filter	Explicit filters and pipes	Stream between a pipe and a filter	Stream transformation state machine	Synchronization between filter reads and writes	Streams of typed records
	Input and output ports on filters	No two sources or sinks connected to the same port instances	Data buffering by pipes		
	Sources and sinks on pipes			Propagation of stream contents to sinks	
Push-based	Independent producers	Producers connected only to distributors	Content filtering in distributors	Distributor location	Channel notification
	Explicit distributors		Buffering and queuing by distributors	Remote connection and communication protocol	
	Channel access/subscribers	Many-to-may channels among receivers and distributors	Subscription setup	Data marshalling and unmarshalling	Subscription request
	Receiver user interface		Content storage/expiration	Distribution policy	
				Implicit invocation	

Figure 5 Style characterization results

servers and the connection setup behavior in the client-server style, and the topology of subscribers and producers and the subscription setup behavior in the push-based style. Differences can be seen between client-server and C2: the earlier style requires a server that is listening for incoming requests, where a response is triggered by a request, whereas the latter style does not require a listening server, and involves the transmission of independent requests and notifications.

These four styles are formally modeled and composed "on top of" the Alloy model of Alfa primitives. Additionally, we have developed architectural implementation infrastructures for these styles that build on the Java implementation of Alfa. Our key concern in constructing the implementation infrastructures has been their fidelity to the particular architectural style. While other properties of such infrastructures (e.g., performance, flexibility) are frequently also desirable [17], our initial goal in this research is to understand the style "behind" the infrastructure rather than facilitate the most effective implementation of applications in that style.

5.2. Modeling and implementing styles

This section describes applications of Alfa to the four architectural styles, in the order in which we completed them. With the exception of client-server, only brief summaries of our experiences with the styles are provided due to space constraints. Additional details of the style models and their implementations can be found at [5].

Client-Server: Initially, an Alfa visual representation of client-server style elements was created as shown in Figure 6. This visual representation helped us to develop an Alloy model of the style elements using the Alfa *core* and *style* models. This *mapping* model consisted of three style elements: *block* (an abstraction over clients and servers), *protocol* (an individual element in a protocol stack), and *connection*. Figure 7 shows an excerpt of the mapping model, which extends the constraints in *Alfa/style*'s *System* (recall Figure 3), and adds new Alloy signatures for the client-server style elements. These constraints define the mappings (compositions) of Alfa primitives required for client-server style elements; the constraints shown in Figure 7 define the structure of a protocol and prescribe how blocks link to protocols to form a client-server system. An additional *style* model (not shown here) formalizes topological constraints on clients, servers, and the connections among them.

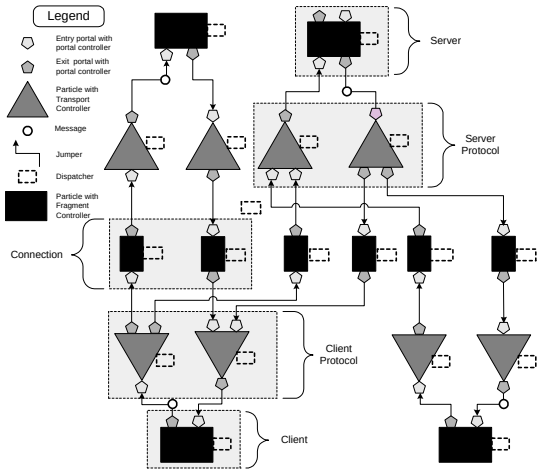

Figure 6 Client-server using Alfa

Once the style elements were identified and formalized, we created a UML design of the behavior, interaction, and data aspects of this style based on the semantics inherent in the Alfa model shown in Figure 6 and on the Alloy model. The Alfa message primitive was extended to form client-server *request* and *response* primitives. We also found that, with one exception, the characteristics of client-server were satisfied by existing Alfa primitives. The exception resulted in the addition of four interaction primitives to Alfa. We added primitives for *inter-process communication (IPC)*, *server location* behavior in order to support IPC, and data *marshalling and unmarshalling* to the *connection* style element. These interaction primitives are likely to be useful in other styles that involve distribution and have indeed been identified in our previous work as software connector dimensions [21]. These primitives have been implemented using Java's *Socket*s, DNS-based host location services, and object serialization mechanisms, respectively. In addition to these primitives, we provided a *data queuing and buffering* primitive to ensure scalability of processing client requests. This primitive was integrated with the *dispatcher* primitive by creating a *PooledAsynchronousDispatcher* that queues messages and asynchronously dispatches them to the particle controller using a thread pool.

In addition to the IPC-based client-server style, we created another extension of the style that uses *in-process* communication. This extension was achieved simply by replacing the IPC primitives by their in-process equivalents.

The implementation of client-server was structured us-

```
1.  module Alfa/cs/mapping
2.  open Alfa/core
3.  open Alfa/style
4.  sig Block {}
5.  sig Protocol {}
6.  sig CSSystem extends System {
7.     blocks: set Block,
8.     protocols: set Protocol,
9.     mainFragment: blocks ?-> Fragment,
10.    stack: blocks ?->+ protocols,
11.    outgoingTransport: protocols ?->! Transport,
12.    incomingTransport: protocols ?->! Transport,
13.    …} {
14.    all p: protocols | #(outgoingTransport[p] + incomingTrans-
          port[p]) = 2
15.    all b: blocks | some s: stack |
          link[mainFragment[b]]=outgoingTransport[stack[b]] &&
          link[incomingTransport[stack[b]] = mainFragment[b]
16.    …}
```

Figure 7 Alloy model of Alfa-Client-server

ing different Java packages: one package for the shared aspects of the style, and one package each for IPC and in-process extensions of the style. It took roughly one person-month to model (using 133 Alloy LOC) and implement the style (using 2500 Java LOC). The modeling task was completed by the paper authors, while the implementation was performed by an undergraduate student provided only with a UML design of the style.

A benchmarking application was created to assess the round-trip performance in processing 1,000 single-object requests from the client to the server, both on the same host. This application is used to compare the performance overhead created by the Alfa client-server IPC framework against a native Java socket and object serialization solution that forms the basis of many Java client-server implementations. The result of the benchmark was consistent with our expectations: the unoptimized Alfa client-server framework runs roughly half as fast and requires slightly less than twice as much bandwidth as the plain Java mechanism. One area of concern is that the initialization memory required for Alfa is more than 10 times that required in the plain Java implementation. This is not entirely surprising, however, as there are many more object instances in the running Alfa client-server infrastructure compared to a plain Java solution. Alfa is amenable to eliminating entire layers of portal and particle objects (e.g., the Connection objects in Figure 6) without affecting the fidelity to the style. We are currently investigating the extent to which such optimizations diminish the flexibility and compositionality of Alfa models, and can be made reversible.

C2: The C2 style was modeled and implemented using the same approach as the client-server style. This style is modeled using two elements, *brick* and *connection*. A brick is composed from 5 particles: one transport each for the top and bottom sides, input and output ports, and the brick's *dialog*, modeled as a fragment. This can also be seen in Figure 8, which shows a part of the Alloy model of the

```
1.  module Alfa/c2/mapping
2.  open Alfa/core
3.  open Alfa/style
4.  sig Brick {}
5.  sig Connection {}
6.  sig C2System extends System {
7.      bricks: set Brick,
8.      mainFragment: bricks ?->! Fragment,
9.      bottomIn: bricks ?->! Transport,
10.     bottomOut: bricks ?->! Transport,
11.     topIn: bricks ?->! Transport,
12.     topOut: bricks ?->! Transport,
13. …} {
14. all b: bricks |
        (bottomOut[b] + topOut[b]) = link[mainFragment[b]] &&
        link[bottomIn[b]] = mainFragment[b] &&
        link[topIn[b]] = mainFragment[b]
15. all b: bricks |
    #(topOut[b] + bottomOut[b] + topIn[b] + bottomOut[b]) = 4
16. …}
```

Figure 8 Alloy model of Alfa-C2

mapping between C2 style elements and the Alfa primitives. A *connection* is modeled as two different fragments. No additional primitives were required to model this style although controllers were extended to provide the required logic for C2 components and connectors. The PooledAsynchronousDispatcher is used for buffering as well as dispatching C2 *requests* and *notifications*, which themselves are extensions of the Alfa message primitive.

Pipe-and-filter: This style was mapped more naturally on top of Alfa. It is modeled using two elements, *block* and a stream *descriptor*. A block is composed from one particle: either a fragment (for a *filter* block) or a transport (for a *pipe* block). A descriptor identifies a block's connections so that the source and destination of information is precisely identified in terms of a block. No additional primitives were required to model this style although the fragment controller was extended to provide the required logic for *filters*. Every filter is also provided a thread for its state machine. Finally, Alfa's message primitive was extended to form a pipe-and-filter *record*. Two versions of this style were modeled and implemented, one involving an unbounded buffer, and another with a single element buffer.

Push-based: This style was mapped in a similar manner to client-server, although the style required an additional *distributor* element, and a *notification* is used in place of the request/response mechanisms of the client-server style. So far, we have only created an in-process infrastructure for the push-based style that uses a direct push approach, although a variety of other *distributors* are possible [11].

6. Evaluation

Alfa has been formally modeled, analyzed, and implemented. Furthermore, its utility as a potential architectural style has been assessed by developing applications in "plain" Alfa. For example, a moderate-sized application (approximately 10,000 Java LOC) has been developed using the Alfa implementation framework, showing performance within an order of magnitude of a reference implementation of the same application in plain Java. Four diverse architectural styles have been successfully "grafted" on top of Alfa, requiring the introduction of only four additional primitives in the process. In this section we discuss the reusability of Alfa primitives, as this is of prime importance for answering the research questions guiding this work and posed in the Introduction.

Reusability in Alfa is evaluated in three different ways: (1) reuse of Alloy models of Alfa, (2) reuse of Alfa primitives, and (3) reuse of the Alfa implementation infrastructure. The compositionality of the Alfa formal models is enabled by the support for reuse in Alloy, in turn allowing us to reuse the Alfa model as a black box in modeling an architectural style. Style models are created in a layered fashion in order to systematically reuse the primitives. The layers, in order of abstraction level, are: (1) model of Alfa

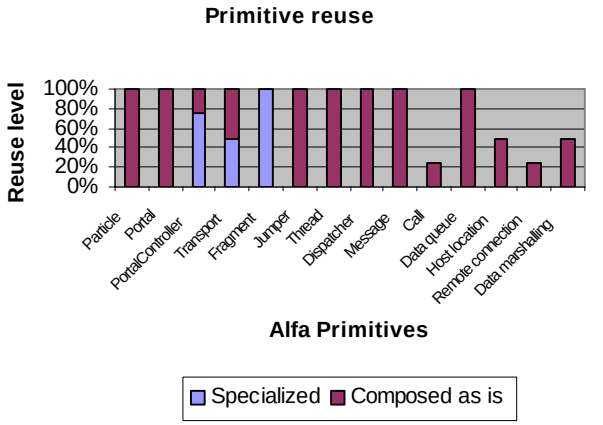

Figure 9 Reuse level of Alfa primitives. The first 10 are the basic primitives (*Particle* through *Call*).

primitives, (2) Alfa style model, (3) composition model of style elements from Alfa primitives, and (4) model of topological constraints on the style elements. It is also possible to create specialized style models on top of these four layers to impose additional constraints on the basic styles. This layered approach ensures consistency and reusability of Alfa and style models.

Reuse of Alfa primitives can also be measured in terms of their use in creating models of individual styles. Certain primitives are reused "as is", whereas others are specialized for use in a style. Additionally, the basic Alfa primitives obtained during the "bootstrapping" iteration will likely be reused more than the supplementary primitives introduced in the second iteration. Figure 9 graphically depicts the reusability of various Alfa primitives across the four styles modeled to date. Notice that the basic primitives, except *Call*, are reused 100%, whereas only *Data queue* achieves 100% reusability among supplementary Alfa primitives. The three *controller* primitives (*portal controller*, *transport* and *fragment*) are more likely to be reused through specialization rather than through composition. This is consistent with their behavioral nature, which varies according to the style.

Finally, it is worth mentioning that the Alfa style and primitive infrastructure totals about 3,000 documented source lines of code (DSLOC) spread across 21 classes. Each style implementation infrastructure involves between 900 and 2,500 DSLOC on top of the Alfa implementation. The small sizes of the style infrastructures can be attributed to the high level of composition-based reuse of Alfa primitives in styles.

7. Conclusion and future work

Architectural styles bring the benefits of software architectures to system development in an economical way and form a widely accepted industrial practice. However, for the systematic development of styles and to ensure their benefits in the development of applications, it is necessary to provide better techniques for characterizing, composing, and implementing styles. This paper presents a novel approach, called Alfa, to address these concerns using architectural primitives. Architectural primitives provide low-level reuse as well as a standard set of building blocks for designing new styles. Primitives also increase our understanding of the essential similarities and differences between existing styles and serve as a bridge between software architectures and implementations by promoting systematic style infrastructure implementations, which can be leveraged to develop style-based applications.

This paper discussed an evaluation of Alfa in the context of four different architectural styles spanning a wide array of architectural choices. The evaluations suggest that the set of Alfa primitives can be kept relatively small, and that high levels of reuse can be obtained across styles. Initial results show promise of effective style infrastructure implementations as well. Clearly, more architectural styles need to be studied before Alfa primitives can be established as applicable to most styles, i.e., before we can answer the questions posed in the Introduction with complete confidence. At the same time, we believe that there is enough support for our contention that the existing Alfa primitives are a good starting point for understanding the compositionality of architectural styles.

Our future goals are to apply this approach on additional architectural styles. We foresee three challenges in this endeavor. First, despite the level of reusability achieved so far in the Alfa approach, it remains to be seen if the same kind of results can be achieved for a broader set of architectural styles.

Another potential difficulty we foresee is the lack of standardized descriptions of styles. Since styles originate in different areas of computer science, it is difficult to perform an evaluation of Alfa across a number of currently used styles. Therefore, we have focused on the invariant aspects of styles as identified by originators of each style. The variable aspects of style are treated as extensions that can be modeled and implemented in more than one way depending on the variation chosen.

A final difficulty deals with the mechanics of our approach: the limited scalability of the automated Alloy analysis tools [12]. Since the number of object instances involved in even a small-sized system can overwhelm the tool, it becomes difficult to perform automated analyses on larger systems. Hence, we have chosen to focus on using the Alloy tools for model checking, generating instances of Alfa and style models, and generating visualizations of these instances instead of proving the correctness of the underlying models.

Acknowledgements

The authors would like to acknowledge Michael M. Gorlick for his contribution to a preliminary version of the Alfa framework and Nick Kuang for his work on implementing the architectural style infrastructures in Java.

This research is supported by the National Science Foundation under Grant Number 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.

References

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

[2] F. Achermann, S. Kneubuehl and O. Nierstrasz, "Scripting Coordination Styles", *Proc. Coord'00*, eds. Antonio Porto and Gruia-Catalin Roman, Springer Verlag, Sep. 2000, pp. 19-35.

[3] G. Agha, "Adaptive Middleware", *Comm. ACM*, 45(6), June 2000, pp. 30-32.

[4] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation", *Proc. ICSE'02*, Orlando, FL, May 2002, pp. 187-197.

[5] Alfa Web Site. http://sunset.usc.edu/~softarch/Alfa/

[6] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Trans. on Software Engineering and Methodology*, 6(3), July 1997, pp. 213-249.

[7] D. Batory, and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Trans. on Software Engineering and Methodology*, 1(4), Oct. 1992, pp. 355-398.

[8] Bell, C. G. and A. Newell, *Computer Structures: Reading and Examples*, McGraw-Hill, New York, 1971.

[9] Berson, A, *Client/Server Architecture*, McGraw-Hill, New York, 1996.

[10] D. Garlan, R. Allen and J. Ockerbloom, "Architectural Mismatch: Why Reuse is So Hard", *IEEE Software*, 12(6), Nov. 1995, pp. 17-26.

[11] M. Hauswirth and M. Jazayeri, "A Component and Communication Model for Push Systems", *ACM SIGSOFT Software Engineering Notes-Proc. ESEC-FSE'99*, Toulouse, France, Oct. 1999, pp. 20-38.

[12] D. Jackson, I. Shlyakhter, and M. Sridharan, "A Micromodularity Mechanism", *ACM SIGSOFT Software Engineering Notes-Proc.ESEC-FSE'01*, Vienna, Austria, Sep. 2001, pp. 62-73.

[13] Javasoft, Java 2 Enterprise Edition specification v1.3, Sun Microsystems, Inc., On-line at http://java.sun.com/j2ee.

[14] F. Kon, et al, "The Case for Reflective Middleware", *Comm. ACM*, 45(6), June 2002, pp. 33 – 38.

[15] D. Le Metayer, "Describing Software Architecture Styles Using Graph Grammars", *IEEE Trans. on Software Engineering*, 24(7), July 1998, pp. 521-33.

[16] A. MacDonald and D. Carrington, "Guiding Object-Oriented Design", *Proc. of TOOLS'98*, Melbourne, Australia, Nov. 1998, pp. 88-100.

[17] N. Medvidovic, N. R. Mehta and M. Mikic-Rakic, "A Family of Software Architecture Implementation Frameworks", *Proc. WICSA'02*, Montreal, Canada, Aug. 2002.

[18] N. Medvidovic and R. N. Taylor, "Separating Fact from Fiction in Software Architecture", *Proc. ISAW'98*, Orlando, FL, Nov. 1998, pp. 105-108.

[19] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Trans. on Software Engineering*, 26(1), Jan. 2000, pp. 70-93.

[20] N. Medvidovic, et. al. "Modeling Software Architectures in the Unified Modeling Language", *ACM Trans. on Software Engineering and Methodology*, 11(1), Jan. 2002, pp. 2-57.

[21] N.R. Mehta, N. Medvidovic and S. Phadke, "Towards a Taxonomy of Software Connectors", *Proc ICSE'00*, Limerick, Ireland, 2000, pp. 178-187.

[22] Microsoft Developer Network Library, *Common Object Model Specification*, Microsoft Corporation, 1996.

[23] M. Mikic-Rakic and N. Medvidovic, "Adaptable Architectural Middleware for Programming-in-the-Small-andMany", *Submitted to ICSE'03*.

[24] R.T. Monroe, and D. Garlan, "Style-based Reuse for Software Architectures", *Proc. ICSR'96*, Orlando, FL, Apr. 1996, pp. 84 –93.

[25] R. T. Monroe, et. al. "Architecture Styles, Design Patterns and Objects", *IEEE Software*, 14(1), January 1997, pp. 43-52.

[26] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Document Number 91.12.1, Revision 1.1, OMG, December 1991.

[27] D. L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules", *Comm. ACM*, 15(12), December 1972, pp. 1053-1058.

[28] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992, pp. 40-52.

[29] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", *Proc. COMPSAC'97*, Washington, DC, Aug. 1997, pp. 6-13.

[30] Shaw, M. and D. Garlan, "*Software architecture: Perspectives on an Emerging Discipline*", Prentice-Hall, 1996.

[31] Taylor, R. N., et. al., "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Trans. on Software Engineering*, 22(6), June 1996, pp. 390-406.