Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures

Elisabetta Di Nitto
CEFRIEL - Politecnico di Milano
Via Fucini, 2
20133 Milano, Italy
+39 2 23954 272
dinitto@elet.polimi.it

David Rosenblum
University of California, Irvine
Dept. of Information and Computer Science
Irvine, CA 92697-3425 USA
+1 949 824 6534
dsr@ics.uci.edu

ABSTRACT
Architecture Definition Languages (ADLs) enable the formalization of the architecture of software systems and the execution of preliminary analyses on them. These analyses aim at supporting the identification and solution of design problems in the early stages of software development. We have used ADLs to describe *middleware-induced architectural styles*. These styles describe the assumptions and constraints that middleware infrastructures impose on the architecture of systems. Our work originates from the belief that the explicit representation of these styles at the architectural level can guide designers in the definition of an architecture compliant with a pre-selected middleware infrastructure, or, conversely can support designers in the identification of the most suitable middleware infrastructure for a specific architecture.

In this paper we provide an evaluation of ADLs as to their suitability for defining middleware-induced architectural styles. We identify new requirements for ADLs, and we highlight the importance of existing capabilities. Although our experimentation starts from an attempt to solve a specific problem, the results we have obtained provide general lessons about ADLs, learned from defining the architecture of existing, complex, distributed, running systems.

Keywords
Architectural styles, architecture definition languages, event-based interaction, middleware infrastructures, software architectures

1 INTRODUCTION

The development of complex systems demands well established approaches that facilitate robustness of products, economy of the development process, and rapid time to market. This need has led, in the last few years, to the establishment of a research area called *software architecture* [20, 7]. In this area, researchers have demonstrated the usefulness of formalizing the definition of the high-level structure of software systems and allowing designers to perform preliminary analysis on the system under development. Such analysis aims at discovering and solving design problems in the early stages of development. To support the definition of software architectures, a number of *Architecture Definition Languages* (ADLs) [12] have been proposed. Also, a number of *architectural styles* are being identified [21]. A style defines a set of general rules that describe or constrain the structure of architectures and the way their components interact. Styles are a mechanism for categorizing architectures and for defining their common characteristics. New architectures can be defined as instances of specific styles.

While these contributions hold the promise of setting up a formal foundation for the definition of software architectures, others are taking a more pragmatic approach to the development of distributed systems and are focusing on the definition of *middleware infrastructures* (or *middleware* for short), such as ActiveX/DCOM, CORBA, and Enterprise JavaBeans [19]. These infrastructures support the development of applications composed of several, possibly distributed, components and provide mechanisms to enable communication among components and to hide their distribution. Also, they offer a number of *predefined components* that provide well-defined classes of operations. For instance, CORBA defines a set of service components that support transactional communication, event-based interaction, security, etc. [17].

We argue that, despite the fact that architectures and middleware address different phases of software development, the usage of middleware and predefined components can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware used in the implementation phase. A similar observation has been presented in [5], in which the *architectural mismatches* generated by the assumptions reusable parts make about the architecture of an application are identified.

For a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture has to be compliant with the architectural constraints imposed by the middleware. Sullivan et al. in [23] corroborate this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '99 Los Angeles CA
Copyright ACM 1999 1-58113-074-0/99/05...$5.00

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

claim by demonstrating that a style, that in principle seems to be easily implementable using the COM middleware, is actually incompatible with it. This view has been stated also in [18], in which the authors discuss the importance of complementing *component interoperability models* with explicit architectural models. Regis [9] and C2 [24] are middlewares for which ADLs have been specifically defined (Darwin [10] and C2SADEL [13], respectively). These ADLs support the definition of architectures compliant with the corresponding middleware. From a different viewpoint, the ADL UniCon [22] proposes a conceptually similar approach. It predefines in the language a set of connectors that have an associated implementation. These connectors support the definition of an architecture and are part of the implementation of the corresponding system.

While Regis, C2, and UniCon define an architectural definition environment strictly tied to the implementation environment, we aim at developing a more general approach. In particular, we aim to capture the architectural assumptions induced by middlewares in terms of *middleware-induced styles*. In essence, we say that a class of related forms of middleware induces the definition of an architectural style, with each specific middleware of the class defining a *variation* of that style. We use a number of general-purpose ADLs to describe these styles and variations. Our attempt aims at demonstrating that the explicit availability of middleware-induced styles is extremely useful in guiding the architect in the definition of the architecture of an application and in selecting the most suitable middleware, independently of any special purpose development environment.

Unfortunately, our experience in using ADLs has not been fully satisfying. In particular, many available ADLs themselves introduce specific architectural assumptions, which can conflict with the ones embodied in existing middleware. In this paper we discuss our experience in using ADLs to define middleware-induced styles. We provide an evaluation of the ADLs we used, identify new requirements, and highlight the importance of existing capabilities. Although our experimentation starts from an attempt to solve a specific problem, the results we have obtained provide general lessons about ADLs, learned from defining the architecture of existing, complex, distributed, running systems.

The rest of the paper is structured as follows: Section 2 provides a brief introduction to ADLs. Section 3 presents two *event-based middlewares* that we have selected for our case studies. Section 4 describes our experience in using existing ADLs to specify the styles implemented by these middlewares. Section 5 provides an evaluation of our experience and summarizes the requirements we expect from ADLs. Section 6 provides some conclusions and discusses future work.

2 MAIN CHARACTERISTICS OF ADLs

ADLs have been thoroughly surveyed and classified in [12]. In this section we present their salient features.

ADLs usually define three main entities as primitive concepts: *components*, *connectors*, and *configurations*. A component represents a unit of computation and interacts with other components through connectors. A configuration represents the way components and connectors are composed to define a specific architecture. These basic concepts are interpreted by different ADLs in different ways. This variety in the expressive power of ADLs can make descriptions of the same architecture in different languages substantially different. In general, these differences are symptoms of a more serious disagreement on what architectural descriptions should express and what the right level of abstraction is for them.

For the purposes of our work, a middleware-induced style can be specified by describing the "characteristics" that components, connectors, and configurations of instances of the style must have to be compliant with the corresponding middleware. The characteristics to be represented have been selected according to our experience with different middleware. As we will discuss in the following sections, these characteristics encompass dynamic behavioral properties, constraints on single components or connectors or on the way they interact, and topological constraints on the way components and connectors are attached to compose a system.

3 CASE STUDIES

We chose as case studies two representative middlewares of the event-based paradigm: JEDI [3] and C2 [24]. In general, event-based middlewares support the implementation of systems in which components communicate by generating and receiving *events*. Events *published* by a source are *notified* to all components that have declared an interest in receiving them. The middlewares themselves take care of event observation and notification, thus guaranteeing a complete decoupling between sources and recipients of events.

The two case studies present interesting and complementary characteristics. JEDI is a typical representative of the event-based middleware category (including technologies such as the CORBA Event Service). The style it implies is easily generalizable and tailorable for other event-based middlewares. C2 imposes several constraints on the topology of architectures. In the following, we informally describe these two middlewares by highlighting the capabilities they offer to application developers and the rules they impose.

The C2 Runtime Infrastructure

The C2 middleware provides an object-oriented class framework and implements a specific architectural style that constrains the design of applications [24]. C2 was initially created to support the flexible development of GUI-based systems, but it has proven to be suitable as a general-purpose middleware.

As we have mentioned in the introduction, the C2 develop-

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

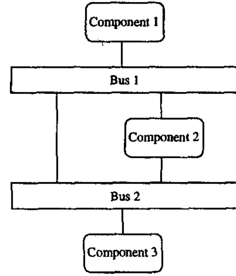

Figure 1: An example of system structure in C2.

ment environment is currently being enriched with a special-purpose ADL that supports the definition of architectures compliant with the C2 style [13]. However, it is still possible to implement a C2 system by using the framework provided by the middleware independently of the architectural definition environment. In the context of this paper we disregard this architecture definition environment. Our goal, in fact, is to assess the ability of ADLs developed independently of C2 to represent the style C2 endorses.

In the C2 style an application is composed of components that communicate through *buses* (see Figure 1). The communication is event-based. In particular, a component can send (receive) events to (from) the buses to which it is attached. Buses are in charge of delivering these events to components according to a policy that will be explained later on. Components and buses can be composed in several ways, provided that the following *topological rules and constraints* hold:[1]

1. Each component and bus has two connection points, one called *top* and the other called *bottom.*
2. The top (bottom) of a component can be attached to the bottom (top) of only one bus.
3. It is not possible for components to be attached directly to each other; buses always have to act as intermediaries between them.
4. Conversely, buses can be attached together. In this case each bus considers the other as a component as far as the publication and forwarding of events is concerned.
5. It is not possible to attach the top (bottom) of a component (or of a bus) to the top (bottom) of a bus.
6. It is not possible to have cycles. That is, a component can never receive a notification generated by itself. The meaning of this constraint will be clarified by the definition of the behavior of a C2 architecture given below.

The *behavior* of components and buses can be summarized as follows: A component can send *request* events to the bus attached to its top (if any). Also, it can send *notification* events to the bus attached to its bottom (if any). When a bus receives a request from its bottom, it forwards this request to all the components (and buses) attached to its top that can handle this request.[2] When a bus receives a notification from its top, it forwards this notification to all the components (and buses) that are attached to its bottom.

[1] The rationale for these rules is presented in [24].

JEDI

JEDI [3] is an event-based middleware developed to support lightweight and decoupled communication among distributed components called *active objects*. An active object can dynamically *subscribe/unsubscribe* to events and *publish* them. An *event dispatcher* stores the subscriptions in its internal state. Moreover, it forwards published event notifications to all the active objects that have previously issued a subscription matching these notifications. When the event dispatcher receives an unsubscription, it deletes the corresponding subscription. An active object can temporarily disconnect (*move-out*) from the event dispatcher and reconnect (*move-in*), either from the same location or from another location. The event dispatcher keeps track of all the notifications directed to a temporarily disconnected active object and delivers them when the active object reconnects again. The only constraint imposed on the topology of architectures is that active objects have to be connected to the event dispatcher in order to generate and receive notifications.

JEDI can be considered as an evolution of the CORBA Event Service, in which notifications are delivered not to all the receivers, as in CORBA, but only to the components that have explicitly subscribed them. A similar approach is also endorsed by commercial products like TIB/Rendezvous. For these reasons we decided to use JEDI as a case study instead of CORBA. Moreover, as we will discuss in Section 5, the usage of JEDI allows us to discuss some issues related with the refinement of architectures. In particular, since at a lower level of abstraction the JEDI event dispatcher is implemented as a set of event servers, we want to specify the attachments between active objects and these lower-level servers.

4 SPECIFYING THE CASE STUDIES IN SOME EXISTING ADLs

In this section we show how the two case studies presented in Section 3 can be specified using the ADLs ARMANI, Rapide, Darwin, Wright, and Aesop. For space reasons we focus mainly on ARMANI and Rapide, while we discuss only the most interesting features of the other ADLs. We selected ARMANI and Rapide from among the others for two main reasons: first, they endorse two opposite approaches, with ARMANI supporting the definition of topological properties and Rapide providing mechanisms for defining behaviors; second, both of them provide a relatively stable toolset for defining and analyzing architectures.

ARMANI

ARMANI [15] is an extension of ACME [6]. It is still under definition, but its basic principles seem well established, and the main functionality of its toolset appears to be robust and

[2] At instantiation time, each component communicates to its bottom bus the requests that it is able to handle.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

stable. It focuses on the definition of the structural properties of an architecture (e.g., how components and connectors are attached together) and disregards the definition of behavioral properties. The ARMANI toolset provides a checker that verifies the consistency of an architectural specification.

In ARMANI a component is defined in terms of a set of *ports*. A port represents a point of interaction with the other elements of the architecture. Similarly, a connector is defined in terms of its *roles*. Each role identifies a participant in the interaction represented by the connector itself. Components and connectors can be grouped in a *system*. In the system, components and connectors form a bipartite graph, with ports and roles *attached* in a one-to-one relationship.

ARMANI supports the definition of component and connector types and provides specialization and instantiation mechanisms for these types. Also, it supports the definition of architectural constraints through the *invariant* construct. Component types, connector types, and invariants can all be collected in a *style*. A system can be defined either as an instantiation of a specific style or as an independent architecture. The ARMANI checker checks whether an architecture is compliant with the corresponding style declaration.

Definition of the C2 Style in ARMANI

The natural way of describing the C2 style in ARMANI is to define a component type representing C2 components, a connector type representing C2 buses, and a number of invariants that represent the topological constraints imposed on C2 architectures. Behavioral rules must be disregarded since they cannot be expressed by the ADL.

We define a C2 component as follows:

```
Component Type C2_compT = {
  port topPort : portTopT;
  port bottomPort : portBottomT;
  invariant size(self.ports) == 2; };
```

The two ports represent the ways C2 components interact with their top and bottom buses.[3] The invariant indicates that a C2 component can have only two ports.

C2 buses cannot be represented as ARMANI connectors, since otherwise architectures like the one in Figure 1 could not be represented (because of the connector-connector links allowed by C2). Therefore, we are forced to define a C2 bus as an ARMANI component. Also, we must define a new connector type whose instances are used to connect C2 components and C2 buses. This is due to the fact that ARMANI does not allow direct attachments between components. This results in the following two definitions:

```
Component Type C2_busT = {
invariant size(self.ports) >= 1;
invariant forall p : port in self.ports |
declaresType(p,portTopT) OR declaresType(p,portBottomT);};

Connector Type intermediary = {
role top: roleTopT; role bottom: roleBottomT;
invariant size(self.roles) == 2; };
```

[3] Port and role types are not shown for space reasons.

In the `C2_busT` definition, no ports are specified, so that architectures where buses are connected to different numbers of top and bottom components can be created. The invariant ensures that the ports declared in any instance of type `C2_busT` are either of type `portTopT` (the type of the top ports) or of type `portBottomT` (the type of bottom ports).

The ARMANI definition of C2 buses and components describes the topological constraints identified as number 1 and 2 in Section 3. The other topological constraints can be defined as global invariants. For instance, constraint number 3 can be expressed as follows:

```
invariant forall comp1: C2_compT in self.components |
  forall comp2: C2_compT in self.components |
  !connected(comp1, comp2);
```

`self.components` is the set of all the components belonging to any instantiation of the C2 style in which the invariant is defined. The predicate `connected` is provided by the ARMANI constraint language. It is true when the components it receives as parameters are attached to the same connector. The character "`!`" denotes logical negation while the character "`|`" stands for "such that".

ARMANI provides limited support for the definition of constraints whose checking requires the entire graph representing an architecture to be traversed. For instance, consider the definition of constraint number 6. It can be expressed by the following formula:

```
Invariant
 forall comp: C2_comp in self.components |
  !ReachableFromTop(comp, comp, self);
```

where `ReachableFromTop` is a C2-specific predicate we defined that accepts as parameters two components and the system that contains them. This predicate returns true if the second component can be reached from the first component, by traversing the C2 architecture in the upward direction. In ARMANI, the *design analysis* construct supports the definition of new predicates and can be used to define `ReachableFromTop` as follows:

```
Design Analysis ReachableFromTop (comp1: C2_compT,
  comp2: C2_compT, sys: C2Style): boolean =
 (Exists bus: C2_busT in sys.components |
  Exists comp: C2_compT in sys.components |
  Exists med1, med2: intermediary in sys.connectors |
  Exists port1: portBottomT in bus.ports |
  Exists port2: portTopT in bus.ports |
   attached(comp1.topPort, med1.bottom) and
   attached(bus.port1, med1.top) and
   attached(bus.port2, med2.bottom) and
   attached(med2.top, comp.bottomPort) and
   (comp==comp2 or ReachableFromTop(comp,comp2,sys)));
```

This design analysis is supposed to return true when the two components passed as parameters are attached to opposite sides (bottom and top, respectively) either of the same bus or of a chain of C2 buses and components. Unfortunately, the recursion introduced in this definition is not currently handled by the ARMANI toolset. A way to circumvent this problem is to define the implementation of

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

ReachableFromTop as a Java function. ARMANI, in fact, allows Java functions to be made visible in the architecture definition environment as predicates. However, this solution is quite limiting and inelegant. First, an architect is forced to use a different (programming-level) language. Second, the definition of the predicate is no longer analyzable by the ARMANI toolset.

Assuming that C2Style is the name of the style that groups all the definitions above, the architecture of Figure 1 is described by the following specification:

```
System Example: C2Style = new C2Style extended with {
 Component C1, C2, C3: C2_compT;
 Component B1: C2_busT = new C2_busT extended with {
 port top1: portTopT;
 port bottom1: portBottomT;port bottom2:portBottomT;};
 // Component B2 has a similar structure
 Connector m1, m2, m3, m4, m5: intermediary;
 Attachments {
  C1.bottomPort to m1.top; B1.top1 to m1.bottom;
  C2.topPort to m2.bottom; B1.bottom1 to m2.top;
  C2.bottomPort to m3.top; B2.top1 to m3.bottom;
  B1.bottom2 to m4.top; B2.top2 to m4.bottom;
  C3.topPort to m5.bottom; B2.bottom1 to m5.top; };}
```

In the instances of C2_busT, ports must be explicitly declared. Notice that the introduction of the fictitious intermediaries makes the Attachments part quite cumbersome.

Definition of the JEDI Style in ARMANI

The ARMANI specification of the style defined by JEDI is quite simple since it does not require the definition of complex topological constraints:

```
Style JEDI_sty = {
 role type NotifyR = {};
 //other role and port type declarations
 Connector Type JEDI_ED = {
  invariant forall r : role in self.roles |
  declaresType(r, NotifyR) OR
  //enumeration of the other legal types
  //((Un)SubscribeR, Move-inR/Move-outR) };
 Component Type ActiveObj = {
  port Notify: NotifyP;
  //enumeration of all the other legal ports };
 invariant forall comp : ActiveObj in self.components |
  forall conn : JEDI_ED in self.connectors |
   forall p : port in comp.ports |
    forall r : role in conn.roles |
    attached(r,p) -> ((declaresType(r,NotifyR) AND
                    declaresType(p,NotifyP))
                OR //other legal types }
```

The connector type JEDI_ED defines the topological characteristics of the JEDI event dispatcher, while the component type ActiveObj represents the active objects. We have chosen to explicitly represent each kind of interaction between active objects and the event dispatcher as a pair of role and port types. The invariant ensures that each active object port is connected with a proper role (e.g., port of type SubscribeP with role of type SubscribeR, etc.).

Rapide

Rapide is an event-based ADL that has been specifically designed to support the prototyping of architectures [8]. Its toolset, in fact, supports the execution of architectural descriptions. The result of an execution is given in terms of the events that are generated and observed by each element of an architecture. These events are organized in a graph that defines the causal relationships among them.

In Rapide a component is defined by an *interface* that specifies: a) the functions and the data provided and required by the component, b) the events it is able to observe and generate, c) its behavior, and d) constraints on its behavior. An *architecture* specifies how the functions, data, and events defined for a component are connected to the corresponding functions, data, and events defined for other components; these connections are established dynamically during the execution of the architecture. Also, the architecture can define constraints on the interaction among components. The definition of styles is not explicitly supported in Rapide. However, as we will show later in this section, some simple styles can be defined as parametric architectures. A component can be associated with an implementation called a *module*. A module must *conform* to the corresponding interface. When an architecture is executed, its components wait for the occurrence of some events and react to them according to the behavior specified in the corresponding interface or, if it exists, the behavior implemented by the corresponding module.

Rapide does not provide a construct to explicitly define connectors as architectural elements. Thus, a connector has to be specified either as a connection between component constituents (functions provided and required or events generated and observed) or as an additional component (an interface). This choice depends on the complexity of the connector to be represented.

Rapide is a powerful but huge language. In our experimentation we have mostly limited ourselves to using the subset of the language that is supported by the toolset. This allowed us to actually execute and assess the specifications we defined.

Definition of the C2 Style in Rapide

As in the case of ARMANI, we define the C2 style in Rapide by specifying the properties of C2 components and buses and the way they are combined. In Rapide we can also define behavioral properties. Conversely, there is limited expressivity for the definition of topological constraints. In Rapide a C2 component is defined as an interface that is able to respond to the occurrence of two events, ReceiveFromBottom and ReceiveFromTop, and to generate two events, SendToTop and SendToBottom. C2 does not make any assumption on how components react to the occurrence of a notification or of a request. Therefore, the corresponding Rapide description does not contain any behavioral specification of such reaction.

A C2 bus is defined as follows:

```
type C2Bus is interface
action in  ReceiveFromBottom(R : Request),
           ReceiveFromTop(N : Notification);
       out SendToTop(R : Request),
```


Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

```
        SendToBottom(N : Notification);
behavior begin
(?R: Request) ReceiveFromBottom(?R) ||> SendToTop(?R);;
(?N:Notification)ReceiveFromTop(?N) ||>SendToBottom(?N);;
end C2Bus;
```

A C2 bus observes and generates the same events handled by a C2 component. This is defined in the action section of the specification. The `in` events are the ones that the C2 bus is able to observe, while the `out` events are the ones it can generate. The behavior part of the specification indicates that a C2 bus forwards all the requests and notifications it receives from one end (top or bottom) to the opposite end, by generating a `SendToTop` or a `SendToBottom` event. Events preceding and following the symbol `||>` are, respectively, the ones generated and observed by `C2Bus` components. For simplicity, we assume that all the components attached to the top of a C2 bus observe the requests coming from its bottom.

Notice that the events in the Rapide specification do not correspond to the events (requests and notifications) that are actually generated or received by a C2 component. Instead, requests and notifications in C2 are parameters of the Rapide events.

As in ARMANI, the definition of the attachment between components is defined separately from the definitions of components. In this case, these attachments are defined by connection rules that establish the relationships between generated and observed events. The following specification describes these connection rules for the particular architecture shown in Figure 1:

```
architecture C2Example() is
 C1, C2, C3: C2Comp; B1, B2: C2Bus;
connect
  B1.SendToTop() ||> C1.ReceiveFromBottom();
  C1.SendToBottom() ||> B1.ReceiveFromTop();
  B2.SendToTop() ||> C2.ReceiveFromBottom();
  B1.SendToBottom() ||> C2.ReceiveFromTop();
  C2.SendToTop() ||> B1.ReceiveFromBottom();
  C2.SendToBottom() ||> B2.ReceiveFromTop();
  B2.SendToTop() ||> B1.ReceiveFromBottom();
  B1.SendToBottom() ||> B2.ReceiveFromTop();
  B2.SendToBottom() ||> C3.ReceiveFromTop();
  C3.SendToTop() ||> B2.ReceiveFromBottom();
end C2Example;
```

Notice from the `connect` section that events produced by a component can be notified to multiple architectural elements. For instance, since the event `SendToBottom` generated by bus `B1` is connected to `ReceiveFromTop` of both `C2` and `B2`, both `C2` and `B2` will be notified of its occurrence.

The C2 topological constraints are not easy to specify in Rapide, since the language does not provide explicit mechanisms for checking the attachments between components. Instead, it provides a *pattern language* to define expressions over the partial order of events that occur during the execution of an architecture. The pattern language can be used to specify constraints on the behavior of an architecture. These constraints implicitly impose some restrictions on the topology of the architecture itself. For instance, constraint number 5 of Section 3 could be expressed as follows:

```
never (?C: C2Comp; ?B: C2Bus; ?M: Message)
   ?C.SendToTop(?M) |> ?B.ReceiveFromTop(?M);
never (?C: C2Comp; ?B: C2Bus; ?M: Message)
   ?B.SendToTop(?M) |> ?C.ReceiveFromTop(?M);
-- equivalent expressions for the bottom sides
```

The expression with `|>` matches if the event on its left side causally precedes the one on the right side and no other event is between the two in the causal order. This operator has been defined in Rapide but not yet implemented in the toolset.

Although the constraints above are related to the topological constraint we wanted to define, they do not model the desired constraint explicitly. On the other hand, it is interesting to note that the definition of constraint number 6 of Section 3 in Rapide is substantially simpler than in the ARMANI case:

```
never (?N: Notification; ?C: C2Comp)
   ?C.SendToTop(?N) -> ?C.ReceiveFromBottom(?N);
```

The operator `->` is used to represent a causal sequence relation between events that allows other intervening, unspecified events.

Rapide does not provide an explicit construct for defining styles. The constraints presented above must be defined in an architecture specification. This means that they must be physically copied in any other architecture that defines an instance of the C2 style.

Definition of the JEDI Style in Rapide

The possibility of defining behaviors in Rapide allows the semantics of the JEDI event dispatcher to be completely reified at the architectural level:

```
type Subscription is interface
 subExpr: var SubscrExpress; subscriber: var ActiveObj;
 provides
  MMatch: function(N: Notification) return Boolean;
end Subscription;
type SubTable is array[integer] of Subscription;

type JEDI_ED (NumActiveObjs: Integer) is interface
action
 in Publish(N : Notification),
    Subscribe(S : SubscrExpress; A: ActiveObj),
    Unsubscribe(SE : SubscrExpress; A: ActiveObj),
    MoveIn(A: ActiveObj), MoveOut(A: ActiveObj);
 out Notify(N: Notification; A: ActiveObj);
behavior i: var integer := 0; ST: SubTable is
 (1..NumActiveObjs, default is new(Subscription));
begin
(?S: SubscrExpress; ?A: ActiveObj) Subscribe(?S, ?A) ||>
 i:=$i+1; ST[$i].subExpr:= ?S; ST[$i].subscriber:=?A;;
(?N: Notification) Publish(?N) ||>
 for j: integer in 1..NumActiveObjs do
  if ST[j].subscriber.Is_Nil()=False and
     ST[j].MMatch(?N)=True then
  Notify(?N, $(ST[j].subscriber)); end if; end for;;
end JEDI_ED;
```

The `JEDI_ED` interface describes the event dispatcher. The behavior part defines the reaction of the event dispatcher to the occurrence of events `Subscribe` and `Publish`. For the sake of brevity, the reactions to `MoveIn` and `MoveOut` events are not described. `ST` represents the table containing the information about the subscriptions issued by agents. Each

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

element of the table contains a subscription expression and a reference to the active object that has issued it. Also, the elements in the table define a function called `MMatch` that checks if notifications match the corresponding subscription expression. The implementation of this function is not shown for space reasons. When a component implementing the `JEDI_ED` interface receives a `Subscribe` event, it stores the corresponding subscription in the table `ST`. When it receives a `Publish` event, it searches the `ST` table for the agents that have issued a subscription compatible with the published notification and forwards this notification to them.

By exploiting the Rapide construct for defining parametric architectures, we can provide a general definition for the JEDI style:

```
architecture JEDI_St(NumActiveObjs: Integer)
return JEDI_Style is
 A: array [Integer] of ActiveObj is
 (1 .. NumActiveObjs, default is new(ActiveObj));
 ED: JEDI_ED(NumActiveObjs);
connect
(?A: ActiveObj; ?N: Notification)
 ?A.Publish(?N) ||> ED.Publish(?N);
(?A: ActiveObj; ?N: Notification)
 ED.Notify(?N, ?A) ||> ?A.Notify(?N);
...
end JEDI_St;
```

In the definition above, the connection rules between the events generated (received) by active objects and the ones received (generated) by the event dispatcher are specified. These rules work correctly independently of the number of active objects that are actually instantiated. This number is determined by the value of the parameter of the architecture. Notice that for C2 it is not possible to specify in Rapide a mapping between events of buses and components that is independent of the instances defining a specific architecture. This is because in C2 the topology of an architecture is not a "star" as in the JEDI case (every component connected to the same event dispatcher).

The definition of a specific architecture based on the JEDI style can be accomplished by instantiating `JEDI_Style`:

```
architecture JEDI_Inst() is
 System: JEDI_Style is JEDI_St(7);
end JEDI_Inst;
```

In this case, an architecture with seven active objects is created.

Darwin

Darwin [10] is a a *configuration language* that supports the development of architectures implemented on top of Regis. Darwin supports the definition of architectures in terms of *components*, *services*, and *bindings*. Components can be either *primitive* or *composite*. Composite components are defined in terms of their internal subcomponents. A component can *require* and *provide* services. Required and provided services are associated with each other through *bindings*. In Darwin it is not possible to define the behavior of components; these are supposed to be defined directly with Regis.

An interesting characteristic of Darwin is the possibility of defining dynamic architectures, in which it is possible to describe how components are dynamically created during the execution of the system and how they are attached to the remaining architecture. Here is an example that describes a dynamic version of a JEDI architecture:

```
component JEDI_Arch {
 provide newActiveObj;
 inst ED: JEDI_ED;
 bind newActiveObj -- dyn ActiveObj;
      ED.Subscribe -- ActiveObj.Subscribe; ... }
```

The composite component `JEDI_Arch` allows a new active object to be created each time the service `newActiveObj` is invoked. This is expressed by the first binding shown in the specification. The second binding associates the `Subscribe` service provided by the event dispatcher to the corresponding service required by active objects. The binding is expressed generically for the component type since the identity of the active objects that will be created is unknown. For the same reason, the bindings involving services provided (not required) by active objects cannot be specified in the Darwin description, since the binding rules require the identity of the components that provide a service to be known. With this restriction, the push semantics adopted by JEDI for the delivery of notifications cannot be expressed in Darwin, since JEDI requires the active objects to provide a `Notify` service to the event dispatcher.

Darwin has been used also to describe CORBA-based applications [11]. The approach is based on the idea of mapping each component identified in a Darwin architecture to a corresponding CORBA object. A tool has been implemented that supports the translation from the Darwin description of a component to CORBA IDL.

Wright

Wright has been presented in [1], with some extensions proposed in [2]. Wright shares several characteristics with ARMANI. In particular, it provides the same basic constructs (components, connectors, ports, roles, attachments). Moreover, the extensions support the definition of styles. Differently from ARMANI, Wright supports the definition of the behavior of both components and connectors. To this end, a subset of the language CSP is adopted. In CSP, behaviors are expressed algebraically in terms of patterns of events.

The following specification defines the structure of C2 buses as a connector type:

$$
\begin{array}{l}
connector\ C2_bus(nt : 1 .., nb : 1 ..) = \\
role\ top_{1 \ldots nt} = receive \rightarrow top \ \Box\ \overline{send} \rightarrow top \ \Box \surd \\
role\ bottom_{1 \ldots nb} = receive \rightarrow bottom \ \Box\ \overline{send} \rightarrow bottom \ \Box \surd \\
glue = top.receive \rightarrow (; i : 1 \ldots nb \bullet bottom.\overline{send}) \rightarrow glue \ \Box \\
bottom.receive \rightarrow (; i : 1 \ldots nt \bullet top.\overline{send}) \rightarrow glue \ \Box \surd
\end{array}
$$

The connector has two role types, *top* and *bottom*. The indexes indicate that the number of roles actually created depends on the parameters of the connector. For both role types the events they can receive/produce are specified. The glue

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

describes the behavior of the connector. In the example, the glue specifies that whenever the bus receives a request from its bottom or a notification from its top, it forwards this request or notification to all the roles on the opposite side.

Thanks to its capability of describing behaviors and to the recent introduction of constructs for supporting the definition of styles, Wright seems to be suitable for our purposes (except for its connector semantics, as discussed in Section 5). However, since it currently does not provide a toolset to support the definition and analysis of styles and architectures, we could not assess its features in greater detail.

Aesop

Aesop [4] is not properly an ADL. It is an environment for defining style-dependent architecture definition environments. It allows a programmer to define a style in terms of component, connector and connection rule types. In general, these building blocks are quite similar to the ones provided by ARMANI and Wright. However, Aesop does not provide an architectural language for defining styles. Instead, a developer of styles must work at the programming language level to customize the environment on the basis of the rules defined by the style. The programming environment that is used is an extension of Tcl/Tk. Aesop also provides a tool integration environment that makes it possible to integrate analysis tools developed for specific styles. We argue that the main limitation of Aesop is that styles are not defined in an ADL. This means that it is difficult to reuse and specialize styles, or to communicate them to architects.

5 EVALUATION OF THE EXPERIENCE AND NEW REQUIREMENTS FOR ADLs

Our experimentation started with the goal of identifying the proper set of ADLs to support the definition of libraries of reusable middleware-induced styles. In this experience, we have realized that while many of our needs are addressed by different ADLs, no ADL fully addresses all of them. In this section we focus on the evaluation of our experience and on the identification of the characteristics that an ADL should have in order to be suitable for our needs.

Style Definition

ADLs should be able to define styles, that is, a coherent collection of component types, connector types and stylistic constraints. Moreover, they should provide a mechanism for exploiting a style in the definition of an architecture. ARMANI, the extended version of Wright, and Aesop explicitly support the definition of styles. In ARMANI, an architecture can extend a style by defining new architecture-specific constraints. The ARMANI constraint checker is able to check if the constraints defined in a style are satisfied by instantiated architectures. Conversely, Rapide and Darwin provide quite limited support for the definition of styles. Both languages provide a construct to define parametric architectures that, as we have shown in Section 4, can be used to describe styles in limited cases. Even in these cases, this solution introduces some restrictions. For instance, the definition of the JEDI style in Rapide (see Section 4) does not enable the corresponding architectures to be composed of different coexisting specializations of type `ActiveObj`. In fact, active objects are instantiated directly in the style definition which is only aware of the definition of type `ActiveObj`. The only way to specialize the generic definition of this type is to overwrite it.

It is also useful to specialize styles as substyles. The definition of substyles, on the one side, enables the reuse of existing architectural descriptions and, on the other side, provides an organization of styles that can guide the architect in selecting the proper paradigm for a particular application domain. For instance, it is intuitive that most event-based middlewares specialize a general event-based style, in which the components and connectors interact through the basic publish, subscribe, and notify operations. Each specialization can introduce new operations (for instance, JEDI adds mechanisms to support the temporary disconnection of a component) or can redefine the semantics of existing operations (for instance, each middleware defines its own specific semantics for matching notifications with subscriptions). ARMANI fully supports the definition of substyles. In Rapide, defining substyles could be accommodated by subtyping the components of the superstyle. However, while some subtyping conditions between interface definitions are defined, the behavioral part of interfaces cannot be inherited (see below for more on behavior). Moreover, the connection rules defined in a Rapide architecture for a supertype do not seem to apply to any specialization of this supertype.

Topological Constraints

In a style definition, we do not want to specify the attachments or connections of specific components. Instead, we would like to provide some general topological constraints that must be respected by any specific instantiation of the component and connector types defined in the style. In ARMANI and the extended Wright, invariants support the definition of such constraints in a powerful way. As we have mentioned in Section 4, a limitation of ARMANI is the fact that it does not currently support the definition of constraints that contain recursive rules. In Rapide it is possible to define constraints on the sequence in which events are generated and/or received by components. We have shown that this kind of constraint can be used to establish certain restrictions on the structure of the architecture, albeit implicitly.

Behaviors

Topological constraints are not enough to define styles. The description of the behavior of components and connectors is at least equally important. If this description is not available, architects cannot have a clear idea of how the elements of the style (and in our case the corresponding implementations in the middleware) work and how they can be used. ARMANI, Aesop, and Darwin do not address this aspect. Rapide provides powerful linguistic constructs for specifying behaviors.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

Still, it has some limitations when a component is specialized or when it is implemented in terms of a module. In both these cases, the behavior of the original interface definition is not guaranteed to be preserved. Compatibility checks are performed only as far as the static part of the definitions is concerned. An approach to checking the behavioral *conformance* between supertypes and subtypes is discussed in [14].

Connectors

The importance of explicitly describing the connectors of an architecture has been stated in several seminal papers on software architectures (see for instance [20, 7]). Surprisingly, the languages that provide an explicit construct for defining connectors associate with them a semantics that is too restrictive. For instance, in ARMANI, each connector role can be attached to only one component port. This means that to create a multicast connector, we need to define a number of roles equal to the number of components to be attached to that connector. This approach is more cumbersome than the one adopted in Rapide, in which generated events can be connected to more than one observed event. Another important limitation of explicit connector languages discussed in Section 4 is that connectors (and also components) cannot be attached together.

This model for connectors seems influenced by the characteristics of sockets, pipes, and (remote) procedure calls. More modern kinds of connectors, such as event dispatchers, ORBs (Object Request Brokers), and multicast channels, highlight the limitation of this semantics. It could be argued that these connectors can be described as components. However, we believe that architectural definitions are more readable and clear when the special purpose of these architectural elements for component interoperability is made explicit. Also, as we have discussed in Section 4, where we present the definition of the C2 style in ARMANI, the definition of these connectors as components adds a new level of indirection. In fact, we needed to define an intermediate, artificial connector type to attach the "real" connectors to the actual components of a C2 architecture.

Refinement of Components and Connectors

An important requirement for both connectors and components is the possibility of refining their internal structure in terms of the composition of other components and connectors. Such refinement supports the co-existence of different levels of abstraction in an architecture. In general, for a refinement to be valid, all the elements that belong to the public interface of a component/connector must be offered by some component/connector defined in any refinement.

As an example of refinement, consider the case of the JEDI event dispatcher. So far, we have defined the event dispatcher as a simple connector. However, its implementation is distributed. In particular, it is composed of a hierarchy of *event servers*. At the architectural level, the internal structure of the event dispatcher can be represented by a refinement in which the operations defined in the event servers are replicas of the operations belonging to the interface of the event dispatcher. Having defined this refinement, the attachments between active objects and the event dispatcher should be refined as well, by specifying to which event server each active object is attached. This refinement of attachments, in fact, would be extremely useful for evaluating the performance of an architecture as a function of the distribution of the active objects over the hierarchy of servers.

While all the languages we considered, except ARMANI, support the refinement of components and connectors, none of them supports the refinement of the corresponding attachments. We argue instead that, in conjunction with the refinement of a component (connector), it should be possible to refine the attachments of this component (connector) with the other architectural elements to which it is attached. This refinement should be *conservative* in the sense that the refined attachments should preserve the characteristics of the original ones (e.g., if a port `A` is attached to a role `B` in the original architecture, then, in the detailed architecture, `A` should be attached to some element of the refinement of `B`). Powerful linguistic constructs to express generic mappings between the elements of an architecture and their refinements are provided by SADL [16]. The limitation of such approach is that it does not provide explicit guidelines on how to perform this refinement.

6 CONCLUSION

The general lesson we learned from our experience is that the top-down approach adopted by the software architecture community in the development of languages and tools seems in many ways to ignore the results that practitioners have achieved (in a bottom up way) in the definition of middlewares. Middlewares have demonstrated their usefulness and effectiveness in a number of practical cases. The software architecture community has now the potential to formalize these achievements in expressive and usable ADLs and, more generally, to coordinate the definition of support technology for the development of middleware-based applications.

Our next step is to continue our exploration of linguistic mechanisms and modeling techniques that allow architecture models to capture middleware-induced architectural styles. We are broadening our search space by looking at UML and, in general, languages that are not strictly considered ADLs. Our longer term objective is to implement an environment that supports the definition of architectures by providing a library of styles induced by specific middlewares. Given an architecture defined according to such a style, this environment will be able to partially automate the implementation of the architecture on the corresponding middleware.

ACKNOWLEDGEMENTS

We wish to thank Alfonso Fuggetta and Debra Richardson who reviewed this paper and gave us useful suggestions on

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.

its structure and content. Also, we thank Neno Medvidovic, Peyman Oreizy, Arthur Reyes, and Alex Wolf who contributed to the clarification of the issues we presented. Finally, we thank Bob Monroe who assisted us with ARMANI.

Elisabetta Di Nitto worked on this paper when she was visiting University of California, Irvine. This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under grant number CCR-9701973. 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, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[3] G. Cugola, E. D. Nitto, and A. Fuggetta. Exploting an Event-Based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)*, Kyoto (Japan), April 1998.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, December 1994.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995.

[6] D. Garlan, R. T. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*, November 1997.

[7] D. Garlan and M. Shaw. An Introduction to Software Architectures. *Advances in Software Engineering and Knowledge Engineering*, 1993.

[8] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[9] J. Magee, N. Dulay, and J. Kramer. Regis: A Constructive Development Environment for Distributed Programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5), September 1994.

[10] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of SIGSOFT '96 Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.

[11] J. Magee, A. Tseng, and J. Kramer. Composing Distributed Objects in CORBA. In *Proceedings of ISADS'97*, Berlin, Germany, April 1997.

[12] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.

[13] N. Medvidovic, D. Rosenblum, and R. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles (CA), May 1999.

[14] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Type Theory for Software Architectures. Technical Report UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.

[15] R. Monroe. ARMANI Language Reference Manual. CMU Technical Report in preparation.

[16] M. Moriconi, X. Qian, and R. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[17] Object Management Group. CORBA Services: Common Object Services Specification. Technical report, OMG, July 1997.

[18] P. Oreizy, N. Medvidovic, R. Taylor, and D. Rosenblum. Software Architecture and Component Technologies: Bridging the Gap. In *Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures*. Monterey, CA, January 1998.

[19] OVUM. OVUM Evaluates Middleware. Technical report, OVUM Ltd., 1996.

[20] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[21] M. Shaw and P. Clements. Toward Boxology: Preliminary Classification of Architectural Styles. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2), San Francisco (CA) USA, October 1996*, San Francisco (CA) USA, October 1996.

[22] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[23] K. Sullivan, J. Socha, and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, 1997.

[24] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component-based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6), June 1996.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 22, 2009 at 05:02 from IEEE Xplore. Restrictions apply.