Automatic Mediation between Incompatible Component Interaction Styles

Klaus-Peter Löhr
Institut für Informatik
Freie Universität Berlin
lohr@inf.fu-berlin.de

Abstract

Incompatibility of component interaction styles is identified as a major obstacle to interoperability when using off-the-shelf components or dealing with legacy software in compositional development. It is argued that a language for defining abstract interfaces – AID – can serve as a basis for accommodating heterogeneous interaction styles. AID is independent of any concrete style, such as invocation, pipe-and-filter, event-based or others. An AID text just specifies elementary input and output events which happen at the boundary of a component. Code that mediates between different styles can then be generated automatically from an abstract interface description.

The focus of this paper is on mediating between dataflow and invocation interaction. The design of the mediation code for invocation-based interaction with mismatching push/pull modes is described in some detail. How to accommodate event-based interaction is shown in the context of the CORBA Notification Service. Enterprise Java Beans are taken as an example of a complex component model, and the problems of accommodating the message-driven beans of EJB 2.0 are analyzed.

1. Introduction

The notion of *component-based software development* is attractive because it suggests the end of programming: just pick some reusable components off the shelf and build a system by interconnecting them in the right way. While the idea makes sense, we also know that life is not that easy. There are several basic obstacles to the vision of *componentware*. A prominent one is the lack of a commonly accepted component model. Several competing standards exist, and we can expect more as our understanding of components matures. So the precise technical definition of *component* remains a moving target.

We may hope that the world will agree on *one* standard in due time. But this is not likely to happen, and homogeneity in a world of components may not even be desirable. At least when the bulk of legacy software is considered, heterogeneity is a fact of life.

1.1. Heterogeneity of component interfaces

As a component should basically be a reusable black box, the most important part of a component model is the nature of *component interfaces*. Object-oriented interfaces are popular with existing component models [7,9,14,11]. But not only may a component be more than a simple object (it may, e.g., feature event-based interaction in addition to invocation-based interaction), it can also behave quite different from an object. For example, a program interacting with its environment through input/output streams may certainly be useful as a component of a larger system. So we should be prepared to deal with different kinds *of interaction styles* when composing a system from components; invocation-based interaction is only one of them. Each interaction style has its own way of how an interface is described. The challenge therefore is to accommodate *interface heterogeneity*.

1.2. The need for mediation

The *environment* of a component within a system includes other components and *glue code* that ties the components together and regulates their interaction. When assembling a system from components we may encounter a component that does offer the desired semantics but exhibits an interaction style that does not meet the expectations of the environment. For example, if we stick with simple interactions, a sorting program like the Unix `sort` will not be able to interact directly with a Java object: the *interaction styles are incompatible*.

A large part of what has been termed *architectural mismatch* [2] is due to incompatibility of interaction styles. It may be possible to mediate between different styles, using code that is often called *wrapper* (or *mediator)*. But the manual construction of wrappers is a tedious and error-prone task, in particular if different type systems are involved.

1.3. Automatic mediation

Code for mediating between incompatible interaction styles should be generated automatically by suitable tools. An approach based on *Abstract Interface Definitons – AID* – has been suggested in [5]. Essentials of this approach are 1) interface descriptions that are not only independent of programming languages (like traditional IDLs) but also of concrete interaction styles; 2) separation of mediation code into two parts, a component-side *driver* for the component and an environment side *proxy* for the component. Proxy and driver communicate by using a *mediation protocol* over a channel established on some inter-process communication platform. Generators for proxies and drivers are based on mappings between AID texts and concrete interaction styles. This technique can be classified as a generalization of well-known techniques for *remote invocation* towards arbitrary *remote interaction*.

The basic language for describing abstract interfaces in the AID framework covers sequential components only: the behaviour of a component is modelled as a simple state machine with input/output events. It is possible to extend the basic model to cover complex components – *subsystems* which are assembled from components themselves: they exhibit *concurrent* behaviour, support *sessions* and expose subcomponents to their environment.

An extended version of the AID language will be presented in section 2. Generators have been built for different concrete interaction styles, and section 3 will discuss the design of proxies and drivers for those styles. Event-based interaction in CORBA and the message-driven beans of EJB 2.0 are addressed in section 4. A discussion of related work follows in section 5.

2. Abstract interface definition using AID

The *abstract interface* of a component is a set of *event types* which is described using a language *AID*. Any concrete interaction of a component with its environment involves data flow either *into* or *out of* the component and is correspondingly modelled either as an input event or as an output event. Synchronous invocation is modelled by a pair of input/output events. Note that the AID events are *abstract events* in the sense that they are not necessarily related to any concrete "event-based systems" or "messaging systems".

When constructing a system from components, we have to distinguish *component instances* from *component types*: a system may include several instances of the same type. Often just the term *component* will be used below; it should be clear from the context whether "instance" or "type" is meant.

2.1. AID syntax and semantics

AID comes in different levels of complexity. The simplest AID texts obey this syntax:

```
Interface =     interface Identifier
                { EventType }
                end Identifier
EventType =     InOut Identifier { Type }
InOut =         in | out
Type =          Identifier
```

Input event types are defined by `in` followed by the event type name, optionally followed by *parameter* type names. Output event types begin with `out` instead of `in`. *Matching event types*, i.e., pairs of synonymous input and output event types, are allowed; otherwise, there must be no name clashes among event types. The informal semantics of an AID text is as follows:

1. *Input event type* without a matching output event type: There will be events in the life of a component instance where information consisting of the name and the parameters of the event flows *from the environment into the component*.
2. *Output event type* without a matching input event type: Like in 1., but with information flowing *from the component into the environment*.
3. *Input event type* followed by a matching output event type: Like in 1. and 2., with the proviso that an input event a) establishes a transient *connection* between the environment and the component and b) will be followed by an output event that is associated with, and terminates, the connection.
4. *Output event type* followed by a matching input event type: Like in 3., but with the output event establishing the connection and the input event terminating the connection.

We consider two simple examples. The abstract interface of a program that filters a sequence of text lines, suppressing some of them and reacting to an invalid input line with an error line, is described as follows:

```
interface LineFilter
in  stdin String
out stdout String
out stderr String
end LineFilter
```

The names of the event types are borrowed from the Unix standard I/O ports. `String` is one of the built-in parameter types of AID (all of which begin with upper-case letters). Note that the operating system, when supporting the data flow through ports, is transferring just bytes; it may not know about "strings" or "lines" separated by newline characters. The typed abstract interface makes the static semantics of the program explicit in stating that newline-terminated strings – rather than single bytes – represent the event parameters. We will come back to the typing of typeless interaction in section 3.1. – Here is another interface:

```
interface PhoneBook
in  enter String Integer
in  lookup String
out lookup Integer
in  count
out count Integer
out samenum String String Integer
end PhoneBook
```

An object featuring an invocation interface may hide behind this abstract interface. The `lookup` and `count` events would correspond to the obvious operations. **in** `enter` events would be implemented by starting an asynchronous (!) execution of an `enter` operation. Alternatively, though, the component could be a program that uses I/O ports; section 3.1 will discuss how the mapping from abstract to concrete interaction would take this into account. Conversely, the `LineFilter` abstract interface can be implemented by a component featuring both exported and imported invocation interfaces, as explained in 3.2.

2.2. Parametrized interfaces

For enhanced reusability, components are usually designed so that their semantics can be *adapted* to a certain degree, e.g., through parameters to be passed at instantiation time. AID allows instantiation parameters to be specified as parameters of an input pseudo-event with the reserved name `init`. So an AID text may start

```
interface PhoneBook
in init Integer
...
```

where the integer value might specify the maximum size of a specific instance of the phone book component.

While this is fine, more power is gained from parametrizing the very interface in a macro-like fashion which allows for describing different interfaces by one AID text only:

Interface	=	**interface** Identifier { Parameter }
Parameter	=	Identifier

The interface name is followed by the formal parameter names. The actual parameters are submitted as simple strings to the proxy/driver generator when a proxy or driver is to be generated from an AID text. Any applied occurrence of a formal parameter name will then be textually replaced by the actual parameter before the proxy/driver is generated from the modified AID text. This can be used for supporting *genericity*, as in

```
interface dictionary Key Data
in enter Key Data
in lookup Key
out lookup Data
.....
end dictionary .
```

"Late naming" of event types is another application of interface parametrization. A simple example is once again found in the Unix repertoire of filters:

```
interface tee copy
in  stdin Byte
out stdout Byte
out copy Byte
end tee
```

The Unix command `tee foo` copies the standard input to the standard output and to the file `foo`. Adopting the convention that a named pipe rather than a regular file be used, the component features two output event types. More details will be given in 3.1.

2.3. Nested interfaces define sessions

The behaviour of complex components cannot always be modelled by simple state machines: 1) a component is usually able to engage in several *connections,* or *sessions,* with its environment simultaneously; 2) *more events* than just one pair of matching events (as introduced in 2.1) can usually occur during a session. A session is modelled in a natural way as a temporary instance of a component type that is managed by a component instance of another type. If we would restrict ourselves to the object-oriented world we could define – in Java parlance – a session as an instance of a non-static inner class of a certain object (of some class). As we neither have classes nor want to stick with object orientation, *nesting abstract interfaces* is the natural way to model sessions. The syntax is extended accordingly:

```
Interface =  interface Identifier { Identifier }
             { EventType | Interface }
             end Identifier
```

An Internet-style server can serve as an example: a certain port is used to ask for the time, and a certain other port is used for establishing connections; the kind of dialogue that can be held over a connection is left unspecified, but is assumed to be line-oriented:

```
interface server
in  time
out time Integer
    interface session
    in  stdin  String
    out stdout String
    end session
end server
```

It is now possible to subsume the transient connections defined by matching input/output events under the session concept: if we understand that an input/output interaction means establishing a session for just one input event and one output event, then

```
in  op ..
out op ..
```

is equivalent to

```
interface op
in  arg ...
out res ...
end op
```

with standard identifiers `arg` and `res`.

3. Mediation architecture and code generation

Let us consider a composition scenario where a decision about the nature of the glue code has been made, presumably determined by the interaction style of several readily available components. We would like to incorporate another component but find that its interaction style does not fit. It may (or may not) be possible to write some wrapper code that mediates between the two styles, as shown in Figure 1. But we want 1) to have this code generated automatically and 2) to allow this in the general case where the component has to live in a separate address space (e.g., if different kinds of component frameworks are involved).

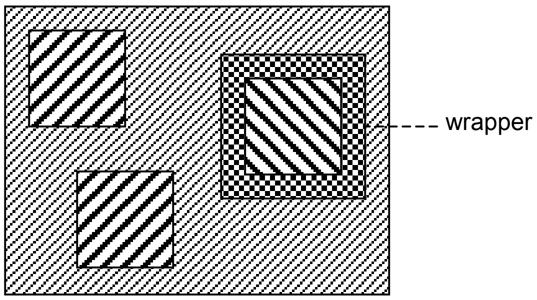

Figure 1. Wrapping an alien component

If the alien component lives in a separate address space, it will be represented by a *component proxy* on the environment side; the environment will be represented by a *component driver* on the component side. Proxy and driver are analogous to the client/server stubs known from remote invocation (but keep in mind that invocation is only one of several possible styles of interaction). Proxy and driver communicate through a *mediation channel* as shown in Figure 2. A *mediation protocol* is used on top of a reliable transport service (e.g., TCP or local pipes).

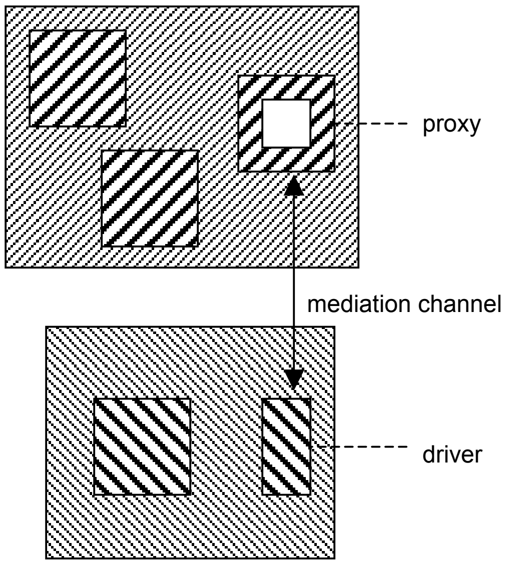

Figure 2. Component lives in a separate address space

Both the proxy and the driver are generated from an AID text. In principle, each interaction style has its own proxy/driver generator. But a generator may also work with different options, in order to account for slight variations of the respective style.

3.1. Interaction styles without invocation

The conceptual gap between abstract events and concrete interactions is small with all those interaction styles that are based on the dataflow paradigm. Typical examples are pipe-and-filter architectures (not only the pipeline version supported by the Unix Shell), stream-based interaction in non-strict functional languages (as e.g., Haskell [3]), and event/message systems (e.g., the CORBA Notification Service [10] or the Java Message Service [15]). Creating proxies and drivers from an AID text that specifies the abstract interface of a component is rather straightforward in those cases where the AID types are easily related to types in the respective type systems.

I/O-port-based communication between binaries is the prime example of *typeless* interaction. Here it is necessary to embody knowledge about the syntax of the byte streams into the AID text. In particular, a sensible decision has to be made regarding event data boundaries within a byte stream. Does input of a single byte constitute a `stdin` event with a one-byte parameter? Or does input of a sequence of bytes delimited by a newline character constitute a `stdin` event with a string parameter? Or maybe the string is meant as a textual command, i.e., a sequence of words, separated by blanks, the first word being an event name followed by event parameters. Still other possibilities may exist, and similar questions arise for output events.

The absence of a formal description of a typed concrete interface makes it inevitable to study the documentation of the component in question. Even with simple Unix filters we have a spectrum of possibilities, e.g., `gzip` is a component that operates on bytes, `grep` operates on lines, `ftp` operates on commands. The generators for filter proxies and drivers know about the particulars to be obeyed here. For example, they understand the event names `stdin` and `stdout`, which means that these names will never be passed to, or be expected from, a filter component. For an event type **`in`** `connect` of an `ftp` component, however, the name `connect` *will* be passed to the component; and for the corresponding **`out`** `connect` the name `connect` *will not* be expected from the component.

3.2. Invocation of exported and imported interfaces

For invocation-based interaction, an input event of a component can either be the start of an operation of that component or it can be the return from an operation of another (previously invoked) component. In the former case, the operation belongs to an *exported interface*, in the latter case it belongs to an *imported interface*. Output events can be classified in the same way. Here is a simple Java example which alludes to a well-known design pattern:

```
interface I {           | interface Observer {
A op(B b);              | void notify(A a);
}                       | }
```

```
class  Z implements I {
public Z(Observer o) {obs = o;}
       Observer obs;
public A op(B b) {
          ... obs.notify(a); ...
          return a;
          }
}
```

Both **`return`** `a` and `obs.notify(a)` are output events, but of different kinds, as mentioned above. The following abstract interface definition can be given for component `Z`:

```
interface comp
in  op B
out op A
out notify A
in  notify
end comp
```

Now suppose a dataflow-style environment is to employ this component, using one input channel (for data of type `B`) and two output channels (for data of type `A`). The environment would likely prefer to see an interface such as

```
interface comp'
in  input B
out out1 A
out out2 A
end comp'
```

In a similar vein, suppose we do have an invocation-style environment, but the notification is to be *pulled* from the component, using an additional exported operation, rather than *pushed* by the component, as seen above. Here the proxy should be generated from an interface such as

```
interface comp*
in  op B
out op A
in  getnote
out getnote A
end comp* ,
```

not from `comp` as given above.

There are two options for dealing with this situation:

1. a given abstract interface is extended with additional information for the purpose of proxy/driver generation;
2. the interface description is obeyed literally, and a similarity relation among interfaces is defined which allows a proxy to cooperate with a driver if both were generated from similar interfaces.

We have adopted the first approach for our experimental implementation, and we are currently exploring the second approach.

Note that incompatible push/pull modes among component and environment imply non-trivial mediation even if both the environment and the component are invocation-based. If the push/pull mode of an event as featured by the component is different from what is expected by the environment, proxy and driver will accommodate this heterogeneity. Push/pull usage modes are well-known from the event channels of the CORBA Notification Service [10]. Here the notion is used in a generalized way for arbitrary components rather than for channels. It specifies whether an information flow between component and environment is instigated by the source or by the destination.

Two mismatching behaviours have to be accommodated for inward flow, and two for outward flow:

	behaviour expected by environment	*behaviour of component*
inward:	a) push in	pull in
	b) pull in	push in
outward:	c) pull out	push out
	d) push out	pull out

Cases a) and c) have in common that both sides are active and want to trigger the flow. Cases b) and d) have in common that each side is inactive and relies on the other side to trigger the flow. Cases a) and c) are handled easily: proxy and driver perform event buffering. Cases b) and d) require that proxy and driver play the necessary active roles (using threads).

3.3. Status of implementation

Proxy and driver generators are available for port-based (Unix-style) binary programs, for invocation-based Java classes and for stream-based Haskell functions, all for a Unix (Solaris) environment. The generators for Java recognize options for push/pull modes. Generators for EJB components are preliminary versions that work for special cases, as explained in section 4 below.

When a component is to be instantiated, the environment instantiates a proxy. The proxy creates a new process, gets hold of an input/output channel for that process and causes the process to instantiate the component and its driver. The input/output channel then serves as the *mediation channel* mentioned above. Process creation is local rather than remote in our prototypical version of the mediation platform. So the mediation channel is just a pair of simple pipes (or a duplex pipe).

Several comments are in order. First, *public* component instances (like public server processes) are not considered here. A component is always instantiated as a *private* entity of an instantiated program. In general, if there is a mismatch among the interaction styles, a component has to be instantiated in a separate address space. This will usually, although not necessarily, happen on the same machine. Placing the component and its environment in *one* address space is possible in those cases where the languages can coexist safely (e.g., on the .NET platform [8], or if the languages are identical); but this is not yet supported by our generators.

Secondly, component *deployment* over the network is not addressed by the AID system. The issues of mediation and deployment are considered orthogonal. Obviously, it would be nice to generate the proxy (and possibly the driver) for a downloaded component on the fly. But this requires an accompanying interface description and a non-trivial infrastructure.

The current version of the *mediation protocol* which is used on top of the transport service supports simple types only. Its design is ad-hoc. We refrained from using XML (or even SOAP) because local, efficient mediation were considered more important than worldwide interoperability among generators from different sources.

4. Issues in accommodating component standards: CORBA and EJB

Typical examples of complex component models are the Component Object Model, COM+ [7], Enterprise Java Beans, EJB [14], and the CORBA Component Model, CCM [9]. Interaction in all these models is based on invocation. Even the event-based interaction features of EJB 2.0 and CCM have a distinct invocation flavour. This suggests that interaction style heterogeneity should not be a serious impediment to interoperability with, or among, those components. Indeed, the big problems result from the semantic richness of those models. But the AID approach can still be helpful to a certain degree.

4.1. Event-based interaction in CORBA

CORBA components use the CORBA *Notification Service* for event-based interaction [10]. This service is available through several interfaces (specified in IDL) ; it allows clients to interact indirectly, by invoking operations on *Event Channels* (which should not be confused with our mediation channel). Dataflow between a client and a channel object can be either in push mode or in pull mode; mixed modes between suppliers and consumers of events are possible.

Basic event supplying and consumption is generic: the event notifications flowing through an Event Channel are not statically typed (i.e., their IDL type is `any`); generic `push` and `pull` operations are used for supplying and consuming events of type `any`[1].

An event-style component proxy presents itself as a regular event-supplying/consuming component to its environment while clandestinely communicating with some driver for the real component which may not know anything about CORBA components and may use a completely different interaction style.

The abstract event types to be specified in AID are derived from four kinds of invocation events (shown with their IDL signatures):

(input events:) `push(in any data)`
on consumer, called by an Event Channel;
`any pull()`
on supplier proxy, called by consumer;

(output events:) `push(in any data)`
on consumer proxy, called by supplier;
`any pull()`
on supplier, called by an Event Channel.

Each invocation involves a certain Event Channel. Connections between channels and their clients can be set up in different ways. For example, a client may connect to a well-known, existing channel, using the Naming Service. Let us assume that the connections are established during an initialization phase and are not changed afterwards. For this kind of behaviour, an abstract event type is associated with input events from a named channel (or output events to a certain channel). For example, the abstract interface `Threshold` of an event filter that listens on a channel `"sensor"` and generates new events on a channel `"alarm"` would be specified as

```
interface Threshold
in  sensor Any
out alarm Any
```

This abstracts from the modes – push or pull – that the implementation actually uses for supplying and consuming events. If indeed a proxy or driver for a CORBA environment is to be generated from the interface, the generator has to be instructed accordingly (as explained in 3.2).

It is important to keep in mind that the interpretation of AID event names such as `sensor` or `alarm` depends on the style of setting up a configuration of channels and their clients. Different styles require different stub generators – or a generator that uses a parameter indicating the style.

CORBA Components are examples of Notification Service clients whose event-based interconnections are set up by associating event sources and event sinks, as specified by a *component assembly descriptor.* Specifying event sources and event sinks as special *ports* of a component enhances the flexibility of component usage. It also allows for a more direct correspondence with an AID specification because event channels do not have to be dealt with explicitly. So the IDL specification

```
component Threshold {
consumes  SensorEvent temperature;
publishes AlarmEvent warning;  };
```

would correspond to the AID specification

```
interface Threshold
in  temperature SensorEvent
out warning AlarmEvent
```

We do not pursue the subject further; a complete treatment of accommodating CORBA Components in heterogeneous architectures is beyond the scope of this paper.

4.2. JMS and message-driven beans

While EJB is compatible with CCM to a certain degree (delineated by CORBA's *Basic Components*), EJB 2.0 has incorporated event-based interaction along the

[1] The Notification Service also supports Typed Events and Structured Events; these will not be considered here.

lines of the *Java Message Service*, JMS [15] which is different from the CORBA Notification Service.

The Java 2 Enterprise Edition, J2EE, supports EJB and JMS. JMS is a *messaging middleware*: reliable inter-process communication between *producers* and *consumers* of messages via *message queues,* or *channels*, is supported in a platform-independent manner. Different channels, identifiable through the Name Service, can be established; they are managed by the *JMS Server* which is usually included in the EJB Application Server.

Two types of channels are supported, *Queue* and *Topic*. They are distinguished by the style of message delivery: *Queue* implements *point-to-point* messaging, i.e., a message will be delivered to exactly one consumer (of possibly several consumers listening on the channel). *Topic* implements *publish-and-subscribe* messaging, i.e., a message will be delivered to *all* consumers that have subscribed to the channel. A *filtering* mechanism can be used to ignore messages whose *headers* and *properties* do not match certain SQL-like queries.

A *message-driven bean* is an instance of a class that implements (at least) two interfaces, `Message-DrivenBean` and `Message-Listener` (from `javax.ejb, .jms`). `MessageListener` defines the public operation

```
void onMessage(Message message);
```

this will be the message handler that is activated by the container when a message is available on the channel.

The *deployment descriptor* of a message-driven bean tells the container which channel the bean should listen on and – for a Queue channel – how many instances of the bean should be created. When a message is available the container tries to find an instance that is not busy and makes a thread invoke that instance.

4.3. Accommodating message-driven beans

Message-driven beans are an example of components with a well-defined interaction style. So it should be possible, using a proxy and a driver, to accommodate a component featuring a different style in lieu of a "real" bean. This should also work the other way around, i.e., using a message-driven bean in an environment that does not know anything about beans and bean interaction.

Using an alien component instead of a bean makes sense if that component has the desired message-processing semantics: there should be an in-event type and zero or more out-event types. The proxy generator for the message-driven-bean style must generate proxy code that correctly represents a message-driven bean. In particular, it must implement the operation `onMessage`, irrespective of the component's in-event name. It also has to perform a type conversion from the prescribed argument type `Message` to the input type of the component. While the name problem is easily solved, the type conversion problem cannot be solved in a satisfying, general way for all cases. We settle for a solution that ignores the *head* and *properties* of the message and works for those types of the message's *payload* that are easily converted to the component's input type.

Note that the proxy's environment does not directly address (invoke) the proxy. Its functionality is used by sending a message to a certain channel. The proxy has to listen on that channel; so the proxy generator will construct the deployment descriptor accordingly.

The output generated by the component must be output by the proxy bean "in bean style". For instance, the bean could *produce* messages, in addition to consuming messages. Code for this can also be generated automatically.

While generating a proxy bean for an alien component is feasible, generating a driver for a given bean is a problem with no satisfactory solution yet. Imagine this situation: when composing a system using a certain non-EJB style you discover the need for a certain functionality and remember "that powerful message-driven bean" that would do the job. The bean must live in a container and should be accompanied by the driver. Now an EJB container has a rich functionality and many conventions – and restrictions – the programmer has to obey. It turns out that these restrictions make it hard to come up with a general solution for the driver generation problem.

5. Related work

Accommodating mismatching push/pull modes in imperative code is a subject with a long history, dating back to Jackson's *program inversion* [4]. But there is also rather recent work on the problem: *automatic program transformation* has been suggested as a technique for coping with mismatches [3]. This approach is attractive from a performance point of view: it avoids the execution overhead of mediation code. On the other hand, it is limited to rather restricted scenarios: it applies to a specific language, and the program sources have to be available. The mediation approach, being independent of language (and even of source code), has the advantage of being applicable to any kind of legacy software.

Techniques for accommodating incompatible *interaction protocols* have been studied in [16]. Input/output events of components are modelled in a way similar to AID, but the interaction behaviour (event sequences) is the object of study. *Adapters* are used for mediation between mismatching behaviours. In simple cases, these adapters can be generated automatically, but manual intervention is often required. The approach is not suited for automatic generation of mediation code for different interaction styles. Similar kinds of adapters have recently been suggested for concurrent components with incompatible synchronization properties and have been applied to imperative components [13,1].

It is important to demarcate the AID terminology from similar but only weakly related terminology in the literature. Extending object-oriented languages with a construct called *pluggable composite adapters* is suggested in [6]. Here a component resembles an encapsulated ensemble of classes. *Component adaptation* extends a given component in such a way that it will be adapted to a certain *collaboration* with other components. Composite adapters are object-oriented language constructs, intended to support behavioral adaptation. They are different from the adapters of Yellin and Strom, and quite different from automatically generated wrappers or proxies and drivers.

6. Conclusion and future work

Specifying the interface of a component by an AID text hides the actual interaction style of the component behind a façade of abstract input/output events. If a component's style does not meet the expectations of the environment, a proxy for the component will be used. The proxy communicates with its counterpart, the driver, which interacts with the component according to the component's style. The messages exchanged over the mediation channel are the real-world representations of the abstract event data.

The mediation protocol is of course the basis for the mutual understanding between proxy and driver. It can be viewed as a common denominator of different styles. If adopted by a certain group of people, it would serve as a "standard". Now our motivation as stated in the introduction was accommodation of different standards. Does it make sense to postulate another standard for the purpose of accommodating different standards? No answer is given here. But it is helpful to compare the issue with what Microsoft has done for its .NET platform [8]. Different programming language standards have been accommodated by the introduction of the Common Language Runtime: CLR *is* a new standard – but on a different level than the other ones.

Much further work is needed to develop AID and its generators into a production system. Extending the type system beyond simple types is an important concern. In addition, we would like to support a more flexible mapping between abstract and concrete interfaces. For example, several abstract event types should be mappable to one concrete operation with an argument typed as a variant record, or the other way around. Of paramount importance are further studies on the feasibility of accommodating complex component models such as COM, EJB, CCM and OSGi.

Acknowledgement

This paper has benefited in many ways from discussions with Karsten Otto and Volker Siegel. Volker has recently started to work on extensions to the implementation described in section 3.3. Their contributions are gratefully acknowledged.

References

[1] A. Bracciali, A. Brogi, C. Canal: Dynamically adapting the behaviour of software components. Proc. 5. Int. Conf. on Coordination Models and Languages, COORDINATION 2002. Springer LNCS 2315, 2002, 88-95

[2] D. Garlan, R. Allen, J. Ockerbloom: Architectural mismatch: why reuse is so hard. IEEE Software 12.6, November 1995, 17-26.

[3] D. Heuzeroth, W. Löwe, A. Ludwig, U. Assmann: Aspect-oriented configuration and adaptation of component communication. Proc. 3. Int. Conf. on Generative and Component-Based Software Engineering (GCSE), Springer LNCS 2186, 2001, 58-69

[4] M.A. Jackson: Principles of Program Design. Academic Press 1975.

[5] K.-P. Löhr: Towards automatic mediation between heterogeneous software components. Proc. SC 2002 – Workshop on Software Composition, Grenoble, April 2002. Electronic Notes in Theoretical Computer Science 65.4.
http://www.elsevier.nl/locate/entcs/volume65.html

[6] M. Mezini, L. Seiter, K. Lieberherr: Component integration with pluggable composite adapters. In M. Aksit (ed.): Software Architectures and Component Technology. Kluwer 2002, 325-356.

[7] Microsoft Corporation: COM Development.
http://www.microsoft.com/com

[8] Microsoft Corporation: .NET Development.
http://www.microsoft.com/net

[9] Object Management Group: CORBA Components. March 1999.
ftp://ftp.omg.org/pub/docs/orbos/99-02-05.pdf

[10] Object Management Group: CORBA Notification Service. June 2000.
http://www.omg.org/technology/documents/formal/notification_service.htm

[11] Open Services Gateway Initiative: OSGi Services Specification.
http://www.osgi.org

[12] S. Peyton Jones et al.: Haskell 98: a non-strict, purely functional language.
http://www.haskell.org/onlinereport

[13] H.W. Schmidt, R.H. Reussner: Generating adapters for concurrent component protocol synchronisation. Proc. 5. IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems. Kluwer 2002.

[14] Sun Microsystems: Enterprise JavaBeans 2.0 Specification.
http://java.sun.com/products/ejb

[15] Sun Microsystems: Java Message Service.
http://java.sun.com/products/jms

[16] D. Yellin, R. Strom: Protocol specifications and component adaptors. ACM TOPLAS 19.2, March 1997, 292-333.