Towards Automatic Mediation between Heterogeneous Software Components

Klaus-Peter Löhr

Insitut für Informatik
Freie Universität Berlin*
lohr@inf.fu-berlin.de

Abstract. An approach to software composition from heterogeneous components is presented. The focus is on heterogeneity of interaction styles. The interface of a component is described in an abstract manner, hiding the component's actual interaction style. This allows for automatic generation of code that mediates between incompatible styles, thus obviating the need for manual construction of wrappers.

1 Introduction

Compositional software development still suffers from the apparent lack of a commonly accepted definition of "component". We could take the view that standardization will solve this problem in due time. The premise of this paper, however, is that different kinds of components and competing standards are here to stay, raising the question of how to alleviate the ensuing heterogeneity problems.

1.1 Incompatibility of interaction styles

Probably the least common denominator of all component definitions is "a black box with a well-defined interface". The nature of the interface is the crucial point in this definition. There are many different ways a component can interact with its environment. For example, consider a Java object (featuring data abstraction) and a Unix filter program. Both represent black boxes which present well-defined interfaces to the outside world; but the object's typed invocation interface is very different from the filter's untyped dataflow interface based on input/output channels. These are examples of rather simple components. Typical component models of contemporary component technology comprise more elaborate component interactions, featuring event-based communication and often providing means for customized configuration; in addition, component platforms usually offer a set of standardized services through *containers* [OMG 99] [OMG 00] [Sun 00].

In software architecture, any given kind of component interaction is said to establish a certain architectural style [Shaw/Garlan 96]. This style may or may not correspond to some paradigm for programming-in-the-small. Interaction with a Unix filter, or event-based interaction, is certainly not close to popular object-oriented programming. On the other hand, any dataflow style is strongly related to stream processing as found in lazy functional languages. The proponents of *composition* or *coordination languages* maintain that components should be composed using special languages which reflect the needs of programming-in-the-large [Achermann/Nierstrasz 01].

In developing software from existing components, we may have to accommodate *heterogeneous components* featuring incompatible interaction styles, e.g., because legacy software is to be exploited or because off-the-shelf components are to be employed. Small-scale examples again suffice to illustrate the situation. Suppose, for instance, we are using Java to implement a system that has to do some text suppressing. The Java libraries do not offer anything appropriate, but we find that the Unix *awk* utility would do the job. Unfortunately, *awk* has no object-oriented interface; so we would try to design some wrapping scheme, perhaps using threads, inter-process communication etc. This is the kind

* Part of this work was done while the author was on sabbatical leave at UC Santa Barbara.

of troublesome situation that has been termed *architectural mismatch* [Garlan et al. 95]: there is no simple plug-and-play at all; laborious and error-prone mediation between incompatible interaction styles is required.

Note that the source of the problem is not just programming language heterogeneity (actually, *awk* might have been written in Java as well) – it is *interaction heterogeneity*. If a component and its environment agree on an interaction style, they can use whatever implementation language is seen fit. The differences are smoothed out either by the compilers or by some suitable middleware: for the prevalent style, object invocation, this is exemplified by Microsoft's binary *.NET* platform [Microsoft 01] or by the OMG's *CORBA* standard which uses a canonic interface description language (IDL) [OMG 01]. This approach breaks down, though, if a component and its environment *cannot* agree on a common interaction style, object invocation or else.

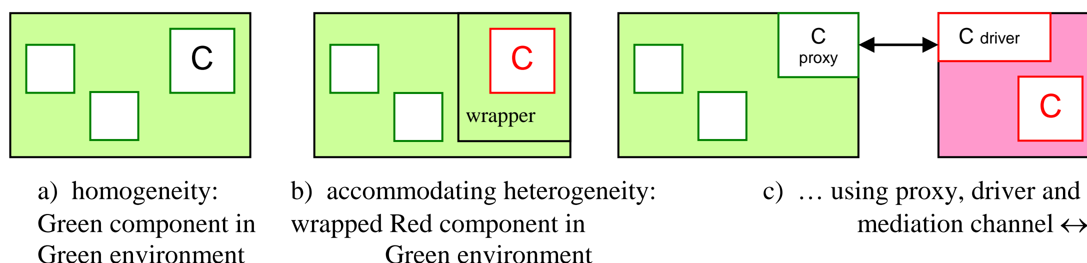
1.2 Multi-paradigm languages?

It can be argued that the best way to support the coexistence of different interaction styles in a software system is to provide multi-paradigm languages. Those languages do have their merits, e.g, in reconciling prejudices against certain paradigms (say, functional programming). But there are also drawbacks: multi-paradigm programming is harder than sticking to a single paradigm; it may also detract from the very beauty of clean uni-paradigm programming. Even a technique like monadic programming, although cleanly integrated into the functional programming paradigm, might meet with resistance. Most importantly, legacy systems and off-the-shelf components just come "as is", with limited possibilities for configuration - and certainly not for adjustment of their interaction style.

1.3 Using abstract interfaces

Obviously, some "dirty" mixture of styles/paradigms must occur somewhere in the code that mediates between different interaction styles. It may be possible, however, to generate that code automatically, as known from stub generation for remote invocation. Using this approach, humans would never be concerned with writing multi-paradigm wrapper code.

If mediation between different kinds of components is to be automated by using tools, a solution must be based on formal interface descriptions – only these have to abstract from the actual interaction style of a given component. Now, on the most abstract level, a simplified model of a component is just a *state machine*, and the most elementary kind of interaction is an *input or output event*. Thus an interface description has to list those events, including their (typed) parameters. Based on a given style-and-language mapping, a generator tool will process an interface description and produce a *proxy* (environment-side stub) for the component. Accordingly, a *driver* (component-side stub) for the component will be generated. Proxy and driver communicate through a *mediation channel*, using a certain protocol for message exchange. Fig. 1b) gives a conceptual view of how an alien component is accommodated through the use of some wrapper code. Fig. 1c) gives a more technical description: the alien component will usually live in a separate address space; the wrapper is implemented as proxy and driver, plus inter-process communication software. Whether inter-process communication is local or involves network messages is a secondary issue.



a) homogeneity: Green component in Green environment

b) accommodating heterogeneity: wrapped Red component in Green environment

c) … using proxy, driver and mediation channel ↔

Fig. 1: Homogenenous vs. heterogeneous ("Red/Green") interaction

Note that our view of system composition focuses on one component at a time; the term "environment" is meant to denote all other components plus the *composition code* (sometimes called *glue code* or *script* [Achermann/Nierstrasz 01]) which describes how the system is composed from its components, obeying a certain interaction style ("Green" in Fig. 1). *Connectors*, often considered essential for composition, are not made explicit. We take the view that connectors are part of the composition code, whether textual or graphical. No a-priori restrictions are imposed on the language used for the composition code. For example, Shell code, special scripting code, object-oriented code etc. are equally acceptable.

It is important not to confuse mediation with remote invocation over a network: we do have separate address spaces; but 1) invocation is only one special case of interaction and 2) usually there will be no need for physical distribution.

Section 2 will present the simple experimental language *AID* for describing abstract interfaces and will relate abstract events to concrete interactions. The behaviour of proxies and drivers will be exemplified for several interaction styles in section 3. A sketch of an experimental implementation is given in section 4, and section 5 will discuss related work.

2 Abstract interfaces and concrete interaction styles

We start with *sequential components;* concurrency will be addressed shortly. We will often use "component" to mean either "component type" or "component instance" (as in "class" vs. "object"). If not explicitly specified, the intended meaning should be obvious from the context.

2.1 Component interface descriptions

A sequential component can be viewed as a state machine. Its interactions with the environment are modelled as input and output events. An observed behaviour is a sequence of such events. So if S denotes the set of states, the behaviour of a component is determined by two mappings,

$$\begin{array}{ll} \text{in:} & I \times S \rightarrow S\,, \\ \text{out:} & S \rightarrow S \times O\,, \end{array}$$

where I and O are the sets of input/output values. The user of a component must of course know both its interface and its behaviour. But for the purpose of automatic mediation, we are only concerned with the static semantics of components, as given by a typed interface. Here is a simple concrete syntax of a language called *AID* that is to be used for **a**bstract **i**nterface **d**escriptions:

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

Interfaces are named, and so are the event types and the given primitive types. The input event types (**`in`**) of an interface constitute a simple algebraic type, the event identifiers serving as constructors; I is the set of values of this type. Similarly, O is the set of values of the algebraic type defined by the output event types (**`out`**). An *event* is then a triple (i,s,s') from I×S×S such that in(i,s)=s' (for input) or (s,s',o) from S×S×O such that out(s)=(s',o) (for output).

This captures the semantics of an interface *as long as* the sets of input identifiers and output identifiers are disjoint. If an input event type and an output event type have identical names, this indicates a correspondence between input and output events of the respective type: an input event establishes a *connection* between the component and its environment; there will be a corresponding output event which is associated with, and terminates, the connection. Synchronous invocation is the typical – though not the only possible – implementation of this. Long-term connections allowing for multi-event interactions according to certain protocols are not supported in the initial version of AID.

A *concurrent* component may exhibit concurrent input/output events and a behaviour that cannot be modelled by a deterministic state machine. The interface, however, is not affected by this.

2.2 Push versus pull behaviour

Component interface descriptions are *abstract* in the sense that they are not concerned with interaction styles. Consider the following example:

```
interface Directory
in  initialize
in  enter String Integer
in  lookup String
out lookup Integer
out count Integer
out synonyms String String Integer
```

We may suspect that the component behind this interface is just an object (or class) implementing an object-oriented interface such as (in Java)

```
interface Directory {
void initialize();
void enter(String name, int number);
int  lookup(String name);
int  count();
}
```

Note that an implementation faithful to the abstract interface will provide asynchronous operations `initialize` and `enter` whereas `lookup` and `count` would be synchronous. But how would `synonyms` be implemented if it is to signal, say, the detection of two names being mapped to the same number? This might occur directly following an `enter` or at an arbitrary later time. In an invocation-based context, it must be implemented as a call to some object that has been passed to the component before, e.g., as a parameter at instantiation time.

This demonstrates that information flow into and out of a component may be triggered either by the environment or by the component itself. If information flow is triggered by the information source this is known as *push style* (or data-driven style); if it is triggered by the destination, it is known as *pull style* (or demand-driven style). In the Java example above, **`in`** `enter`, **`in`** `lookup` and **`out`** `synonyms` are implemented in push style whereas **`out`** `lookup` and **`out`** `count` are implemented in pull style.

Notice that the abstract interface could also be implemented in a very different fashion, say, by a process communicating with its environment through I/O channels, using read operations for in-events and write operations for out-events. In this case, **`in`** `enter` and **`in`** `lookup` are implemented in pull style while **`out`** `lookup`, **`out`** `count` and **`out`** `synonyms` are implemented in push style. Still other push/pull combinations are possible. The point is that *any* push/pull combination may hide behind the abstract interface; the interface does not tell us anything about this.

2.3 Typing issues

Middleware IDL compilers are based on language mappings between the IDL and a given implementation language, the bulk of which is concerned with mediating between the different type systems. All issues involved resurface with abstract interfaces. Additional complications are caused by the abstraction from interaction styles, but they have little to do with the typing issues. This is because we insist that there be a one-to-one correspondence between each event type in an abstract interface and an interaction endpoint in the implementation (procedure, method, port, channel, event handler, etc.). It is not possible, for instance, to implement the event type **`in`** `switch Boolean` by two operations `void switchOn()` and `void switchOff()`.

It should be noted that packaging the parameters of an event into some composite data object of the implementation language is a frequent necessity, simply because not every event type will have an

implementation allowing for an arbitrary number of parameters. But then, this is a well-known technique from IDL compilation and does not raise new issues.

3 Mapping events to interactions

We address the problem of how to map abstract events to concrete interactions by studying four different examples of interaction styles: typed object invocation, untyped byte streams, functional streams, and event systems. To avoid confusion, it should be noted that "event systems" are concerned with sending and receiving *event notifications*; "event" is an undefined term in this context, and is not to be confused with the abstract events introduced above. An abstract event will be mapped to the interaction of either sending or receiving an "event notification".

3.1 Typed object invocation

Object-oriented programs are written in an imperative style; the interaction mechanism is *invocation*. Imperative programming, working on a rather low level of abstraction, allows the programmer to choose among push and pull versions of interaction. An AID specification as shown above does not tell the proxy/driver generator anything about push and pull. Additional information has to be provided in the form of *push/pull annotations*: the event types in the interface description may be preceded by either **push** or **pull**, as in

```
interface Directory
push in enter String Integer
...
```

for the purpose of proxy or driver generation. The default is **push** for **in** events and **pull** for **out** events, which corresponds to the "normal" interaction with an object through its interface.

For a **push out** interaction, the environment provides a *listener object* for invocation by the component. The generated proxy will have a constructor that is parametrized with a listener for each **push out** event; its type is (in Java parlance) an interface type featuring one method. The roles of proxy and driver are reversed in this case: a thread within the proxy is responsible for invoking the listener on behalf of the component. For a **pull in** interaction, the proxy will use a *provider object* whose method has to return the data requested by the component. On the component side, the driver has to be built in an analogous fashion: if the component features a **push out** or **pull in** interaction, the driver will implement the appropriate interfaces.

Non-trivial mediation will take place if the push/pull behaviours of a component and its environment do not match. Proxy and driver will be generated from *two* interface descriptions which differ in their push/pull annotations. Four combinations are possible for each event:

proxy	driver
push in	**pull in**
pull out	**push out**
pull in	**push in**
push out	**pull out**

The first two of these occur when both sides actively engage in the information flow into/out of the component. As a consequence, the stubs must have *buffering* capabilities: data that has been pushed either by the component or by the environment has to be buffered by the proxy or driver, respectively.

The last two of the above combinations are trickier. The proxy invokes parameter objects, as mentioned above; the driver invokes the component. For actually causing information to flow, a thread is set aside in the proxy which pulls data from a provider (**pull in**) and sends it to the driver, which in turn invokes the component (**push in**). Similarly, a thread is set aside in the driver which pulls data out of the component (**pull out**) and sends it to the proxy, which in turn invokes a listener (**push out**). CORBA experts will be reminded of the mechanisms employed for push/pull usage of CORBA Event Channels [OMG 00].

3.2 Untyped byte streams

Inter-task communication via message channels is found in many variations and in different contexts, from high-level programming languages down to binary mechanisms supported by the operating system. Choosing a simple example which still allows to make the important points, we consider the binary Unix standard: a component is a process which interacts with its environment using read/write operations on channels. (Recall that the simple linear configuration known as a *pipeline* has given rise to the term *pipe-and-filter architecture*.)

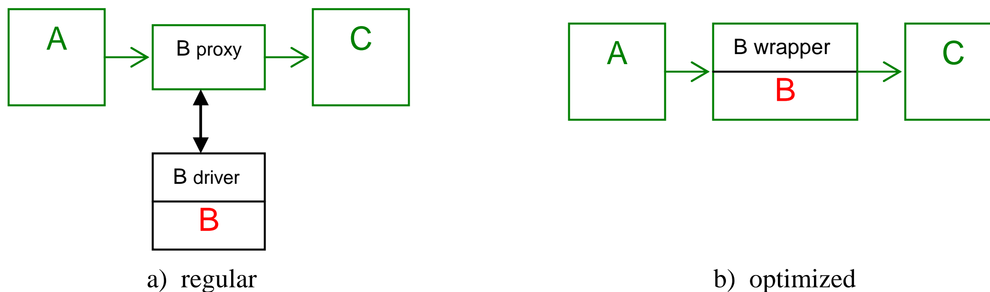
A Unix program (if it is to be started through the Shell) can rely on the availability of three standard channels, `stdin` for input and `stdout`, `stderr` for output. Designing a suitable interface, say

```
interface UnixProcess
in   stdin  String
out  stdout String
out  stderr String ,
```

raises a question about the parameter types. While the I/O mechanism deals with typeless byte streams only (and some applications, e.g., compression programs, may do just the same) many applications deal with printable characters or even character strings. A typical interactive program would interact with its environment on a line-by-line basis, recognizing strings terminated by line separators. This would justify the typing chosen above; but other typings, e.g., choosing `Byte` rather than `String`, are possible.

Many interactive programs are of the *command interpreter* type: an input line starts with a command name which may be followed by parameters; responding to a command, the program may or may not produce some output. In this case, the environment may want to view the different commands as different event types. The stub generators support this by accepting arbitrary interface definitions, not just the generic one shown above. The standard identifiers `stdin`, `stdout`, `stderr` do keep their meaning, though, and are *not* delivered to/from the component.

Note that different push/pull options are *not available* here. A read operation is always a **`pull in`** interaction, a write operation is always a **`push out`** interaction, and that is it. So the stub generators process a pure interface description as shown above; any **`push`**/**`pull`**s are ignored. The generated proxy communicates with its environment (e.g., a pipeline) in Unix style and with its peer, the driver, through the mediation channel. This is shown in Fig. 2a where the driver is assumed to communicate with the component in some other style. The fact that the proxy resides in a separate address space anyway suggests an optimization: coalescing proxy and driver into a single wrapper attached to the component in its address space (Fig. 2b).



a) regular   b) optimized

Fig. 2: Alien component in pipe-and-filter environment
($\rightarrow$ = dataflow interaction through pipe, $\leftrightarrow$ = mediation channel)

3.3 Functional streams

The notion of *state* does not exist in pure functional programming. Lazy evaluation of lists, however, allows for the well-known device of *stream-processing* functions which recursively compute partial output lists from partial input lists (“streams”). If furnished with a “state” parameter, such a function

represents a stateful component; consumption and production of list items represent the I/O events. Lazy evaluation is demand-driven and thus amounts to a **`pull in/pull out`** mode of interaction. Similar to 3.2, other modes are not available, and any **`push`**/**`pull`**s in the AID are ignored.

Both functional proxies and functional drivers have to be able to communicate through the mediation channel. This can be achieved by monadic input/output code, either explicit or implicit. Implicit I/O is supported, e.g., by the *Haskell* library function `interact :: (String->String) -> IO()` which reads/writes the characters processed by its argument from/to the standard I/O channels [Haskell 98].

If `interact` is used in conjunction with a simple, string-based external data representation on the mediation channel, an event would be represented by its name, followed by the parameter values (as strings). Considering a simplified version of the example from section 2.2, supporting only `enter` and `lookup`, let module `Directory` contain the stream-processing function `directory`. The driver generated from the abstract component interface will look like this:

```
module Main where import Directory
main  = interact inter
inter = unlines. map output. directory. map input. map words. lines
input["lookup",p1]     = Lookup p1
input["enter" ,p1,p2] = Enter  p1 (read p2)
output(Lookup_ p1)     = "lookup" ++ " " ++ show p1
```

Parameter un/marshalling is performed using the functions `input` and `output`, based on an algebraic type with constructors `Enter` and `Lookup` for input to `directory` and a type with constructor `Lookup_` for output from `directory`.

The program comprising the modules `Main` and `Directory`, while meant to serve as a component of some larger system, could of course be used interactively by humans, thanks to the human-readable external data representation. And if we forget about its implementation and just consider its interaction behaviour, we see a component in Unix style. So if the program is to be used in a Unix environment, no proxy will be required. Or, alternatively, we can view the module `Main` as proxy and driver coalesced, as shown in Fig. 2b.

Combining Haskell components with imperative (i.e., invocation-based) components is an approach to mediation between imperative and functional programming. Haskell aficionados may want to compare this approach to the monad-based “calling hell from heaven and heaven from hell” [Finne et al. 99].

3.4 Event systems: the CORBA Notification Service

Programming language support for event-based interaction is rare; limited forms are found in real-time languages. Event-based communication infrastructures, however, are of increasing importance. A prime example is the CORBA *Notification Service* [OMG 00] [Brose et al. 01], an extension of the earlier *Event Service*. 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 the mediation channel introduced in 1.3). 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 consuming 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 `any` events*.

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):

---

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

(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, annotations **`push`**/**`pull`** have to be added in the same way as shown in 3.1.

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 [OMG 99] 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 Experimental implementations

A very simple implementation of the mediation channel between proxy and driver has been chosen for an experimental evaluation of the AID approach: a pair of Unix pipes provides an elementary transport service; an event notification is sent as a message consisting of the event name and its parameters in textual representation.

Experimental stub generators, both for proxies and for drivers, are available for the interaction styles described in sections 3.1 – 3.3. For Java object invocation, the generators recognize the push/pull annotations and generate appropriate code.

Component instantiation involves the creation of a process that acts as a carrier for the component instance and its driver. The environment, being responsible for component instantiations, first instantiates the component's proxy. The proxy then creates a new process, establishing pipes for input and output to/from that process and causing the process to load the component and its driver.

5 Related work

The imperative paradigm is the prevailing style in software development, both for programming-in-the-small (based on statements) and for component interaction (based on invocation). So most of the existing research is confined to the imperative paradigm. Different push/pull modes in a purely imperative setting can be accommodated either by mediation, as described in 3.1, or by source code modification – *if* the source code is available. The latter approach amounts to program inversion as introduced by M. Jackson [Jackson 75]. An automatic procedure for program transformation in order to eliminate push/pull mismatches is described in [Heuzeroth et al. 01]. Code transformation has the advantage of runtime efficiency. But the mediation approach, being independent of any source code (availability, language,…), has the advantage of being applicable to any kind of legacy software.

A semi-automatic method for mediating between incompatible abstract interaction *protocols* has been suggested in [Yellin/Strom 97]: *adaptors* are inserted between the components. Like AID, this approach is not tied to the imperative paradigm. The AID way of treating in/out events is related to the protocol specification method of that paper, but does not support the notion of protocol yet. On the other hand, AID provides for fully automatic stub generation in the presence of vastly different interaction *styles*. Adaptors similar to those of Yellin and Strom have been suggested for concurrent components with incompatible synchronization properties [Schmidt/Reussner 02] and have been applied to imperative components.

6 Conclusion

Real-world components are often complex and ill-specified. Thus, establishing the appropriate AID specification may not be easy. It won't even make sense if there is no stub generator, e.g., because the component exhibits a rare or one-of-a-kind interaction style that does not warrant the effort of constructing generators. There is also a lot more to a component than just its interaction interface: customization, management interfaces, containers, deployment etc.

This paper does not claim to solve the general interoperability problems for all the world's component models. The AID technique represents a lightweight, pragmatic approach to coping with one important aspect of component heterogeneity – incompatible interaction styles. It should be applicable to any style – beyond the given examples – that is based on black-box components and information flow through a well-defined interface.

The deficiencies of the technique as introduced above and of the experimental implementations are obvious: the rudimentary type system, the primitive communication infrastructure, the lack of configuration support, the insufficient treatment of standard component models as found in CORBA, EJB, COM, .NET and others. So there is no shortage of issues to be investigated before a safe statement about the practical viability of abstract interfaces can be made. AID was deliberately kept simple; this allowed for a rapid construction of several generators as a proof of concept.

Acknowledgement

Comments by Karsten Otto and Volker Siegel on a previous draft have helped improving the presentation and are gratefully acknowledged.

References

[Achermann/Nierstrasz 01] F. Achermann, O. Nierstrasz: Applications = Components + Scripts. In: M. Aksit (ed.): Software Architectures and Component Technology. Kluwer 2001.

[Brose et al. 01] G. Brose, A. Vogel, K. Duddy : Java Programming with CORBA (3. ed.). Wiley 2001.

[Finne et al. 99] S. Finne, D. Leijen, E. Meijer, S. Peyton Jones: Calling hell from heaven and heaven from hell. Proc. ICFP '99, ACM 1999.

[Garlan et al. 95] D. Garlan, R. Allen, J. Ockerbloom: Architectural mismatch: why reuse is so hard. IEEE Software 12.6, November 1995, 17-26.

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

[Jackson 75] M.A. Jackson: Principles of Program Design. Academic Press 1975.

[Microsoft 01] Microsoft Corporation: .NET Development. 2001. http://msdn.microsoft.com/net

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

[OMG 00] Object Management Group: CORBA Notification Service. June 2000. http://www.omg.org/technology/documents/formal/notification_service.htm

[OMG 01] Object Management Group: CORBA 2.4 Specification. http://www.omg.org/technology/documents/formal/corba_iiop

[Schmidt/Reussner 02] 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.

[Shaw/Garlan 96] M. Shaw, D. Garlan: Software Architecture. Prentice-Hall 1996.

[Sun 00] Sun Microsystems: Enterprise JavaBeans 2.0 Specification. http://java.sun.com/products/ejb

[Yellin/Strom 97] D. Yellin, R. Strom: Protocol specifications and component adaptors. ACM TOPLAS 19.2, March 1997, 292-333.