Rethinking Software Connectors

Stephen Kell
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
Stephen.Kell@cl.cam.ac.uk

ABSTRACT

Existing work on software connectors shows significant disagreement on both their definition and their relationships with *components*, *coordinators* and *adaptors*. We propose a precise characterisation of connectors, discuss how they relate to the other three classes, and contradict the suggestion that connectors and components are disjoint. We discuss the relationship between connectors and coupling, and argue the inseparability of connection models from component programming models. Finally we identify the class of *configuration languages*, show how it relates to *primitive connectors* and outline relevant areas for future work.

1. INTRODUCTION

As software systems become more complex, it becomes preferable to build systems by *evolution*, *extension* and *reuse* rather than from scratch. Separation of interface from implementation aids evolution within a component by localising the code changes necessary when altering its implementation. However, this isn't sufficient for our desired levels of extension and re-use: we might need to change some of the interfaces themselves, or to combine components which have mismatched interfaces.

Software architecture research has identified another useful separation, namely that between *computation* and *communication*, or rather the distinction between *components* and *connectors* [31]. Separating a component's computational code from artifacts concerned with communicating with (or *connecting to*) other components in the system might improve the potential for extension and re-use.

For example, by separating out connection logic, we might be able to change it independently of the core component logic. By encouraging consolidation and sharing of connection logic, we can introduce more sophisticated and re-usable connection abstractions. By confining the problem of interoperability to the connection domain, we can eliminate coupling across computational code, hence easing re-use. Finally, by developing languages and tools which specialise in connectors, we can better optimise this aspect of the development process.

Such a separation is not adopted by today's development practices. However, there has been much research work under headings such as linking languages [28, 22], coordination languages [7, 4], composition languages [1], packaging techniques [9, 6] and interface adaptation [27, 35, 8].

All this work helps tackle some aspect of connection, but we still lack a coherent sense of exactly which problems are more fundamental. For example, Knit [28] addresses mismatches of symbol names, but not mismatches of arguments, protocol or concurrency. Meanwhile Reo [4] addresses protocol and concurrency, but cannot transform the contents of messages sent between components.

The converse problem often goes unconsidered: given a particular connection model, what are the consequences for the component programming model? These two converses are inextricably linked, yet often only considered separately.

More fundamentally, there is little agreement on exactly what connectors *are*. Even in Shaw's seminal paper [31], there is no precise definition of a connector. Authors have since used the term to mean several different things.

A taxonomy of connectors [23] is useful for identifying many differences and similarities between connectors, but introduces a huge number of non-orthogonal dimensions. This means it can't identify fundamental similarities, nor tell us how to build connectors compositionally.

In this position paper, we make the following contributions.

- We provide a more precise characterisation of connectors than in previous work, contradicting the idea that connectors are disjoint from components and discussing relationships with *coordinators* and *adaptors*.
- We describe the relationship between coupling and connectors, arguing the inseparability of component programming models from connection, and that connectors should be capable of *adaptation* in order to maximise component reusability.
- We identify the class of *configuration languages* and argue their importance to software composition. Noting that a shared notion of *primitive connectors* joins configuration languages with component programming, we discuss directions for future work in this area.

We begin with the most fundamental question: what are connectors?

SYANCO 2007 September 3-4, 2007, Dubrovnik, Croatia

2. WHAT ARE CONNECTORS?

To answer this question, we will start from the existing literature motivating and conceptualising connectors. Mary Shaw's paper [31] is the most complete piece on the subject, and to our knowledge has not been superseded by any subsequent work.

Shaw tells us intuitively what connectors *do*: they "mediate interactions between components". However, she stops short of pinning down just what kind or kinds of *thing* they *are*.[1] As Mehta et al note, for some time the architecture community had "maintained a studied silence on the exact nature of connectors" [23].

This difficulty probably arises because it appears that connectors can be several different kinds of thing. They may or may not have independent identity at run-time. They need not be implemented in user code, and may instead be implemented within the programming language or operating system. They can, it is claimed, simply be conventions, or they can be complex pieces of implementation. They can connect between statically determined components, or they can dynamically alter their associations at run time.

If connectors really are a sufficiently coherent class of entities to be modelled by domain-specific languages and tools, as Shaw claims, then we should be able to capture their essence more succinctly than by these apparently disparate observations. In the following sections, we will attempt to provide a precise characterisation of connectors. At first we will take "connector" to mean "connector instance", and then consider by generalisation what "connector types" might be.

2.1 Connectors provide mechanisms

Mehta et al refine Shaw's intuitions by observing that all connectors provide what they call *ducts* – channels along which data or control can be passed between components. It is the number, cardinality, behaviour and configuration of these ducts which differentiate connectors.

We can refine the above analysis further by using a more familiar word: *mechanisms*. Connectors provide mechanisms enabling communication between components.

Mechanisms may be *primitive* or *complex*, and we can identify primitive mechanisms by reference to some underlying computation abstraction. This might be a programming language, an instruction set architecture, or a symbolic calculus such as the pi-calculus [24]. Under any such abstraction where the program can be partitioned into *components*, there are only a finite number of primitive connection mechanisms – transitions which allow different elements of the partition to causally influence each other's state. Consideration of these primitives will be the basis of our work in Section 5.

For now, we will simply say that a connector instance provides one or more mechanisms by which its attached components may communicate. We also note that non-primitive connectors necessarily depend on simpler connectors in order for their mechanisms to be invoked.

[1]The paper's only attempt is the statement that "connectors are the locus of relations among components", while by contrast "components are the locus of computation and state". As will become clear, these are not useful as definitions, not least since even simple connectors may also contain both computation and state.

2.2 Connectors demand agreements

Mechanisms are not sufficient for *meaningful* communication. Communication theory tells us that meaning can only be transferred between parties in the presence of some shared *code* or *context*.[2] In software, as in any system, the need for such contextual *agreement* is precisely the source of inter-component coupling. Two parts of a system *agree* if (and only if) any assumptions they make about each other are simultaneously satisfiable. More complex assumptions demand more complex agreements and therefore lead to greater coupling.

The most basic form of agreement is with a connection abstraction, i.e. a connector type (see Section 2.3). Any component wanting to communicate with the outside world must incorporate assumptions about some sort of connector into its code. For example, if a component communicates with others through procedure calls, its code embodies behavioural properties of these (such as the fact that the caller always waits until the callee returns). Consequently, the component is coupled in some way to a connection abstraction, and indeed, such coupling is unavoidable to some extent. This is a main constituent of "packaging mismatch" [32, 9].[3]

Similarly, if a connector instance is ever to transfer *meaning* from one component to another, those two components must share some common assumptions about the syntax and semantics of the messages they exchange. These assumptions encompass the meaning of individual messages, and also the semantic relations between sequences of messages over time. Disagreements regarding the former result in simple naming and operational mismatches [28, 27], while the latter cause protocol mismatches [35, 8].

In general, coupling need not stop at the immediately connected components: one component may make assumptions which only hold when it is instantiated amid some very particular complex arrangement of components and connectors. This is the root of the more complex instances of "architectural mismatch" [13]. We will discuss coupling and agreement further in Section 4.

By considering a connector instance as both providing mechanism and demanding agreement, we can eliminate one source of confusion in Shaw's original paper. It is claimed that simple *agreements*, such as data encoding conventions, are connectors. According to our definition, they are but one aspect of connection. In order to connect two components using a shared data representation, one must also provide a communication mechanism. Accordingly, although these shared agreements are noteworthy (being the source of coupling), they are not themselves connectors.

The same analysis applies to several of the connector species listed in the taxonomy of Mehta et al [23]. Some

[2]This distinction between *information* and *meaning* was famously highlighted by Weaver in the introduction to his and Shannon's groundbreaking book [30], saying that "two messages, one of which is heavily loaded with meaning and the other of which is pure nonsense, can be exactly equivalent, from the present viewpoint, as regards information". The importance of *code* in the model was later highlighted by Schramm and other communication theorists.

[3]Packaging mismatch spans the entire range of potentially mismatched agreements, and has more to do with the bundling of agreements than with a particular kind. However, the cited work places considerable emphasis on mismatches of communication abstraction.

are merely *agreements* devoid of mechanism (for example X.400, SQL); others are cross-cutting properties shared by broad sets of connectors (such as inter-thread versus inter-process, or FCFS and LRU).

2.3 Connector types, instances and state

We have argued that mechanism and agreement are necessary in order to define a connector instance. However, these are both abstract entities. For a connector instance to actually exist in a running system, it must be something more concrete: we must be able to observe data or control passing between whatever we identify as the components of that system. In other words, the connector must exist in the *state* which constitutes that running system. We paraphrase this by saying that any connector instance has run-time state – even though that state might not be mutable, or might be difficult to identify.

2.3.1 Identifying state

Depending on its implementation, the state which constitutes a connector instance might be less than obvious. For example, in the case of a direct procedure call, the connector's state is contained in instructions within the program text. These are the instructions which push the arguments onto the stack (or into registers), call the procedure, and pop the return value from the stack; they also include the operands of instructions in the procedure definition which reference the arguments on the stack, and the instructions which push the return value and jump back to the caller. The procedure binding, i.e. the connector's configuration, is manifested in the address argument to the branch instructions. All of this state is immutable.

By contrast, the state of a complex connector might be large and dynamically-changing, encapsulating both code and data, and resembling a component instance. We will discuss this resemblance further in Section 3.

2.3.2 Types, instances and other distinctions

This notion of run-time state allows us to clarify the distinction between connector *type* and connector *instance*. Clearly a connector *type* has no run-time state, but is some (perhaps partial) expression of the *mechanisms* and *agreements* (including behavioural properties) of a connector. We can say that a connector implementation satisfies, or doesn't satisfy, a given connector type. Just as with types in programming languages, connector types range from the highly specific to the highly general: they may specify a lot about the mechanisms and agreements, or may specify only a little.

The distinction between connector type and connector instance are important, as is that between the instance and the run-time state of that instance. This state is in turn distinct from that of individual dynamic invocations of a connector (e.g. the activation records created during a procedure call). Note also how the run-time state of the connector instance can change while preserving the instance's identity – for example in a dynamic language such as Smalltalk where an object can dynamically change its virtual function bindings. In this sense, software architectures are frequently dynamic, although higher levels of abstraction might hide this dynamism.

2.3.3 Examples and revelations

Table 1 summarises the concepts and distinctions outlined in this section, using some familiar examples of connector types. For each type, we describe its mechanisms and their specified dynamic behaviour, give example agreements which are typically layered above the connector, and reference a typical implementation. For the implementation, we specify the connectors with which it is invoked, describe a typical instance, list the minimal run-time state of such an instance, and describe the transient invocation state maintained by the implementation.

These distinctions also help to clarify some more of Shaw's observations. It is true that connectors may "manifest themselves as table entries, instructions to a linker, dynamic data structures, system calls, initialisation parameters, servers that support multiple independent connections...". Most of these descriptions refer to the *state* of connector instances rather than their *type*, *identity* or *implementation*. One exception is the system call, which is a type of connector similar to the procedure call.

The other exception is that of servers. Intuitively these appear to be components, but might be considered a type of connector for abstraction purposes. In this example, and in preceding ones, we have noticed that the distinction between connectors and components is not completely clear-cut. The next section discusses this issue in more detail.

3. WHAT AREN'T CONNECTORS?

Just as there is noticeable lack of agreement on exactly what connectors are, there is noticeable disagreement on the relationship between *connectors* and other kinds of thing: *components*, *coordinators* and *adaptors*. This section looks at each of these in turn.

3.1 Are connectors and components disjoint?

Most software architecture research advocates the view that connectors and components are "distinct" classes of entity [2, 23, 19]. Models which treat connectors as a special kind of component, such as those of Rapide [20] or Darwin [21], are criticised for, in the words of Mehta et al, "obscuring [the] distinct nature" of connectors. In other words, many researchers advocate the view that connectors and components should be modelled as *disjoint* classes of entity: that a thing cannot be both.

3.1.1 Continuum, not disjunction

In reality, connectors and components form not a disjunction but a *continuum*, differentiated by the relevance of communication over computation to role of a given piece of implementation. Contrary to the implications of some work [31], notions of both *interface* and *protocol* are useful right across the continuum, albeit with respective biases.

To demonstrate this, we first assert that clearly not all components are connectors: many coherent pieces of implementation do not themselves provide any communication mechanism. Exactly how we define "component", then, would appear to answer the question of whether it is possible to be both a component and connector. Unfortunately, defining "component" is comparably difficult to defining "connector".

If we define "component" to be the instantiation of any unit of implementation, then clearly at least some connectors are also components – for example, all three of the example connectors discussed by Mehta et al consist of clear units of implementation within the Linux kernel. Of course,

Connector type	Roles and configuration variables (static, *dynamic*)	Mechanisms (behaviour defined elsewhere)	Typical layered agreements	Example implementation technique(s)	Typical connector dependencies (immediate only)	Example instance(s)	Run-time state	Transient invocation state
procedure call	caller component S callee component R entry point P in R	S: invoke	signature pre-/post-conditions approx duration wait/block behaviour	stack-supported branch and return	program counter increment shared registers shared memory (stack) branch instruction	(any local early-bound non-inlined procedure call)	caller/callee instruction sequences	stack frame saved register values
system call	caller component S callee R (implicit) call index C	S: invoke	(as above)	stack-supported software interrupt	program counter increment shared registers shared memory software interrupt instruction	`read()`, `write()` on Unix pipe	caller/callee instruction sequences, system call table	stack data processor mode hidden registers
inline procedure call	caller component S callee component R entry point P in R	S: invoke	(as above)	compile-time control flow subgraph instantiation	program counter increment shared registers shared memory	(any inlined procedure call)	expanded instruction sequence at call site	register state during instruction sequence execution
virtual procedure call	caller S candidate callees R_i target signature N	S: invoke	(as above)	vtable-dispatched indirect branch and return	program counter increment shared registers shared memory indirect branch instruction	(any late-bound call)	caller/callee instruction sequences, vtables and vtable pointers	stack frame saved register values
remote procedure call	caller S callee *R* entry point P in R	S: invoke	(as above)	stub/skeleton, marshalling to/from agreed network encoding, transmission by network datagram service	local procedure call network datagram service	Web Service calls e.g. in software update services	stub/skeleton code, file handle table, socket descriptor	(in dependencies only, i.e. in implementations of datagram service and procedure call)
pipe	writer W reader R	W: write R: read	data meaning additional behaviour	kernel-managed ring buffer	system call	Unix pipe e.g. between `tar` and compression filter	kernel ring buffer, file descriptors	(in dependencies only)
n-way network load balancer	source S sinks R_i	S: forward	signature	round-robin forwarding with session management	network service (stream or datagram)	Web Service load balancer	configured server addresses, dispatch-next address, active session map	(in dependencies only)
shared associative store	participants P_i	P_i: in, out	tuple structures tuple meanings referential/semantic constraints sequencing conditions	distributed hash table	procedure call network datagram service	Chord network	node storage, node forwarding tables, node proximity data	(in dependencies only)
shared linear store	participants P_i	P_i: read word, write word	per-location meanings and representations sequencing conditions	shared virtual memory object	shared physical memory virtual memory manager	windowing system shared framebuffer	framebuffer contents	(in dependencies only)
publish-subscribe network	publishers P_i subscribers S_i	P_i: publish E S_i: subscribe	message structure meanings for standard headers/metadata e.g. topic	network of store-and-forward broker nodes	network stream service	Usenet NNTP network (including user agents)	stored messages stored subscriptions overlay network topology	(in dependencies only)
2-way synchronisation barrier	participants P_1, P_2	P_1, P_2: enter	entry conditions	lightweight threading and condition variables	shared semaphores thread scheduler	(e.g. any Barrier instance in a Java program)	semaphore state	(in dependencies only)

Table 1: Example connector types and instances

any communication mechanism, however apparently primitive, is implemented somewhere, even if we have to descend to a lower level of abstraction to see that implementation concretely. For example, we might not see the implementation of procedure calls at the source level, but when we look at the compiler output we see the sequences of instructions which perform the calls.[4]

3.1.2 Levels of abstraction

This latter all-inclusive definition of "component" may not be a useful one at the architectural level. An alternative is to define components to be any piece of implementation which does *not* provide a communication mechanism. We would then, however, rule out many entities which are classically considered to be components.

For example, in the canonical pipe-and-filter example, filters are considered components. However, since they forward data at the same time as transforming it, they clearly do perform communication. It is simply because we see their abstract role as *primarily* computation that they are considered components. From these observations, it seems that whether something is a component or connector might depend on how we prefer to picture it at our chosen level of abstraction.

Using the box-and-lines metaphor, our system is actually composed entirely of boxes of various sizes and thicknesses. A connector in such a diagram is simply a box which joins two or more others, and is *longer* and *thinner* than most. Comprehensibility, however, demands that we can abstract away from this into a simpler diagram. Some boxes must therefore disappear or be coalesced together (because their details are not sufficiently interesting), and some very thin boxes are adequately approximated as lines. It is these simplifications which disjointly classify a box as a connector or component. Exactly where the classification thresholds lie clearly depends on the chosen level of abstraction.

In Figure 1 we have illustrated this by borrowing an example used by Allen and Garlan [2]. They describe a toy system called *Capitalize* which transforms an arbitrary alphabetic character stream by outputting it in alternately lower- and upper-case characters. We show the system at three levels of abstraction, the middle one corresponding to their "architectural description". The less detailed version shows the entire system as a single box, as it might be seen within the architecture of a larger system (where it might well be considered a connector). The most detailed shows the pipes as components, and emphasises how the pipes are themselves connected to the other components using even simpler connectors.

We have now established that whether a piece of a system is a component or a connector depends entirely on the considered level of abstraction; it is not an intrinsic property of that piece. It is therefore still possible to make such distinctions, but it is not helpful to do so when *implementing* connectors. During implementation, we do not have a particular level of abstraction in mind: we are simply implementing some given communicational functionality, and its architectural importance depends entirely on how it is composed with other pieces.[5]

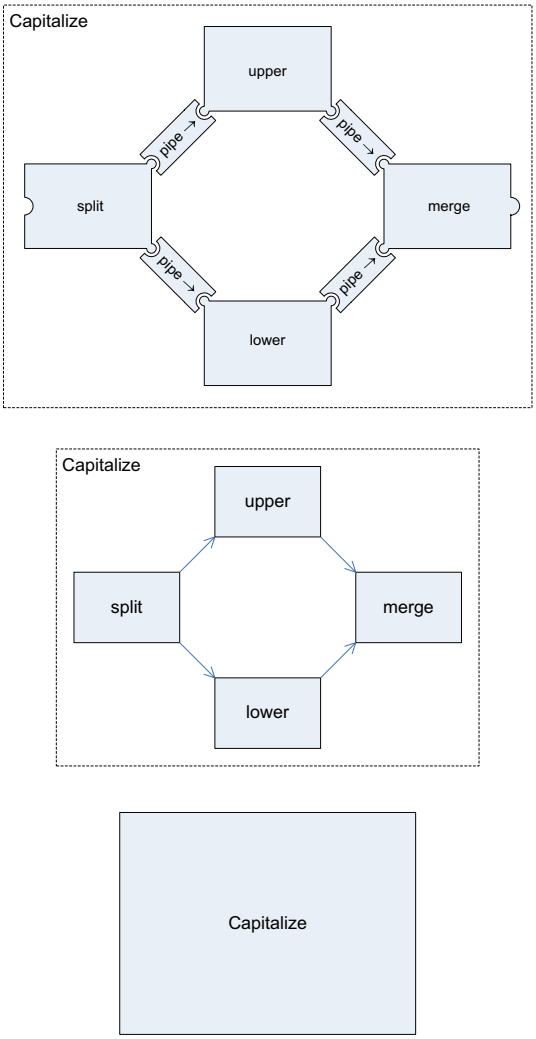

Figure 1: *Capitalize* at three different levels of abstraction

We therefore cannot argue that we should pick from alternative sets of tools and languages based on whether something is a component or a connector. Rather, we simply have a range of implementation options, and we should pick whatever mix is best tailored to the individual task at hand. It is for this reason that we might prefer to see connectors as a subcategory of components – specifically, those lying some minimum distance from the purely computational end of the continuum.

3.1.3 Components as objects

In Figure 1, we considered a set of pipes as a set

[4]The fact that these instruction sequences are distributed or "woven" into the code, rather than being cleanly delineated, is inconsequential. This phenomenon stems either from efficiency requirements (as with inline functions) or from the problem of dominant decomposition (as with *aspects* in aspect-oriented programming [18]). Neither aspects nor inlined functions need have anything to do with communication, so this "woven" manifestation is clearly not a property unique to connectors.

[5]An extreme example of this continuum comes from the world of operating system security, where covert channel analysis shows us that any stateful shared resource can act as a communication channel. Tom Van Vleck gives a detailed example at `http://www.multicians.org/timing-chn.html`.

of distinct architectural elements (connectors, if you prefer). Likewise, we could imagine multiple instances of a non-communicational component (such as a hash table or database) appearing separately on an architectural diagram. This raises another interesting question about the nature of components and connectors: given that the pipes are all implemented by the same code, should we count them as one component or as many?

If we took the former position, we would end up with a diagram much like Allen and Garlan's earlier diagram labelled an "implementation description". Since calls made on the pipes are simply considered undistinguished calls to the I/O library, we lose the architectural view of the system. Indeed, although Allen and Garlan are vague about the notational differences between their two diagrams, we see that the architectural notation adds three new features: logical containment (shown by the larger *Capitalize* box enclosing the other features), abstraction by elimination of uninteresting modules (shown by the absence of a *config* component) and *first-class dynamic objects* (shown by the presence of four distinct pipes, instead of connections to the I/O library).

This raises an interesting question about the nature of components. We have established that individual architectural elements (be they components or connectors) can be created dynamically, and that some of these distinct elements may in fact be implemented by the same code. In our example, these are the different pipes; more generally, they are things which in an object-oriented setting we'd describe as objects of the same class. How do we know which really are components, and which are merely internal state of some enclosing component?

The only possible answer is that, as with the distinction between components and connectors, it all depends on our chosen level of abstraction. At some level, the smallest pieces of state are "architecturally" important, just as *design patterns* [12] are the miniature analogues of software architectures. Usually such low levels of analysis are not necessary at the architectural design stage, but these levels of abstraction, again, form a continuum rather than showing any fundamental divisions.

3.1.4 Protocol versus interface

Given that components and connectors form a continuum, we must now revisit the claimed distinctions and resolve them with this viewpoint. One such distinction surrounds the notions of *protocol* and *interface*. Shaw states that where components have *interfaces* specifying *operations*, connectors have *protocols* specifying *dynamic interaction behaviour*.

Again, we argue that this should be seen not as a disjunction but as a continuum. All pieces of implementation have both operations and dynamic behaviour, but some show more of their complexity in one or other of these aspects. We have already seen how connectors provide one or more mechanisms. These may be thought of as operations, such as the procedure call's *invoke* or the pipe's *read* and *write*. Shaw herself notes that "connectors are often implemented as sets of procedures".

Another illusory distinction between connectors and components is the suggestion that connectors do not provide operations with *types*. This idea arises because connectors are frequently *polymorphic* with respect to the data they convey. (There are exceptions, such as a typed interface to a shared database.) Traditionally these operations might have been rendered as procedures passing effectively untyped messages (such as pipes reading and writing byte strings). More recently, with the popularisation of parametric polymorphism in languages such as Java and C#, this erroneous division is less tempting.

Conversely, components often feature protocols not related to communication. These appear as ordering constraints on their operations. For example, a stack may not be popped more times than it has been pushed, or an error will occur. Just as it is useful to formally describe the dynamic behaviour of a connector, such as a procedure call or pipe, so it is useful to describe the protocol constraints on data structures and other non-communicational components. Indeed, there is considerable work on such specifications [35, 8, 29].

3.1.5 Remarks

This section has provided a notion of component consistent with the notion of connector described in the previous section. We have also shown why the consideration of any software entity as a connector or a (non-communicational) component can only be made relative to some given level of abstraction, and argued that this makes it inappropriate to design tools and languages exclusively targeting one or other kind of implementation.

It has been argued that the distinction between components and connectors is not sufficiently important to merit careful definition [10]. We disagree. Such a lack of consistent logical definitions beneath any model of software risks substantial confusion among developers who might wish to adopt that model. This confusion might be sufficient to prevent its adoption. Alternatively, it might lead to a poor understanding and consequent poor development choices, for example regarding tools and languages.

If connectors are just a particular kind of component, why are we giving them special consideration at all? The answer is the same as it was at the beginning of this paper: in order to improve re-use. We want to reduce and mitigate *coupling* by separating out the concern of *communication* from that of *computation*. We will continue this discussion in Sections 4 and 5.

Note that we will continue to use the terms "component" and "connector" as a shorthand for the approximate notions of "computational component" and "communicational component", without implying any contradiction of our belief these are more accurately considered as regions of a continuum.

3.2 Are connectors the same as coordinators?

Within the coordination community, authors occasionally use the terms "coordinators" and "connectors" in confusing ways. Some make no clear distinction between the two [5], whereas others define "connectors" as a particular type of "coordinator" [4].

Clearly the latter definition, at least, takes too narrow a view of connectors to be consistent with that used by the software architecture community. Is it reasonable, however, to argue that coordination is equivalent to connection?

3.2.1 Definitions of coordination

Mehta et al [23] define coordination as only one of four

"service categories" provided by connectors. However, this view is clearly narrower than the one intended in coordination literature: the coordinators in systems such as Linda [14] and Reo [4] support at least *communication*, and also some kinds of *conversion*. We must therefore turn to another definition.

Carriero and Gelernter define coordination as "the process of building programs by gluing together *active* pieces" [14]. The emphasis in that quotation is our own, and highlights the fact that coordination inherently involves connection of *concurrently executing* entities. As evidenced by the simple procedure call, two components joined by a connector need not be concurrently active. However, we may *model* non-concurrent execution as a special case of concurrent execution. Coordination therefore appears to lose no generality with respect to connection, above a certain level of abstraction.

3.2.2 Modelling emphasis and capabilities

One thing which clearly does distinguish coordination models from connection models in practice is the specific *emphasis* on concurrency and parallelism. There is also a corresponding de-emphasis on two things: adaptation at the level of messages rather than behaviour, and the component programming model.

To illustrate the former, we may return to the taxonomy: many coordination models usually don't support at least some aspects of the "conversion" service. Linda [14] and Reo [4], for example, contain no features to inspect or change the contents of messages from within the connection domain. This focus on behavioural considerations is appropriate within the domain of highly parallel systems, which is the intended target of both models. However, it does mean that the "loose coupling" often cited as an advantage of these models is in need of further qualification.

In particular, these models cannot address coupling associated with the meaning of individual messages (as opposed to assumptions about their relative timing). This is a very basic problem associated with re-use of independently developed code, albeit orthogonal to other issues. For example, coordinators cannot resolve mismatched procedure names or signatures, in contrast to adaptation systems such as Nimble [27] or that of Yellin & Strom [35].

3.2.3 Coupling to a coordination abstraction

There is also another kind of coupling: that between the connectors they provide and their client components. Coordination models generally assume that all components will be programmed to a standard interface exported by the coordination abstraction – for example Linda's *in*, *out*, *rdp* and the like, or Reo's *write*, *read* and *take*. As well as being a source of coupling, we argue that these interfaces are insufficiently abstract.

This latter point is essentially similar to the arguments of Gorlatch [15] within the domain of parallel programming.[6] It seems indisputable that in practice we would like to write components to a more abstract interface than these primitive operations. Are such abstractions re-usable across components? Is it reasonable to retain the same lower-level connection model when composing components programmed against such abstractions? The answers are not clear without considering what such abstractions might be, which is outside the scope of this paper. However, this fact only reinforces the view that we should consider the design of the two models together.

3.3 Are connectors different from adaptors?

As the taxonomy notes, connectors frequently do *conversion* or *adaptation*, either of data representations or of behaviour. This service category is particularly interesting when re-using multiple independently developed components, because it allows us to overcome mismatched assumptions. By interposing an adaptor instead of a non-adapting connector, we provide an intermediary which satisfies the assumptions of all parties.

Intuitively, therefore, it seems useful to include expressions of *adaptation* in the same domain as connection. We note that adaptation of behaviour may require a connector to be *stateful*. This is because invocations made by one component may need to be delayed, or to show some other effect later on, in order to interact with the other connected components in the desired way. Similarly, adaptation of representation requires the connector to do some data processing or *computation* in order to transform a message between different representations.

Clearly, then, our model of connection must be powerful enough to express both of these things. Our view will be that adaptors are a *most general* form of connectors, in that all connectors may be considered as adaptors. A non-adapting connector can be interpreted as performing a no-op or *null* adaptation. Since adaptors provide not only communication but also ancillary computation and state[7], and since that computation might in general need to be Turing-powerful, they are equal to components in expressiveness. They must, however, clearly have at least two points of architectural attachment. More significantly, their abstract role shows a firm emphasis on communication.

As we will see in the next section, adaptation is essential when combining independently developed components, because it allows us to conveniently re-use software in the presence of mismatched expectations.

4. CONNECTORS AND COUPLING

As mentioned earlier, connectors are invariably coincident with *agreements* between components, since it is only with agreement that communication can have meaning. These agreements are precisely the cause of coupling. A primary motivation for study of connectors, therefore, is to examine how different connectors, and their consequent impact on component code, affect the coupling between components and hence determine their re-usability.

[6]Interestingly, parallel programming libraries may provide connectors which also have a primary role in computation, such as MPI's "collective operations". This further blurs the distinction between connectors and computational components. Note that the coupling across these connectors is not necessarily a problem, because these components may be intrinsically non-reusable – their division is for the sake of distribution, rather than delimiting cohesive pieces of functionality.

[7]DeLine's discussion of interfaces between components and adapting connectors, which he calls packagers [9], has an interesting flaw in that it neglects the possibility of the packager containing a finite state machine. Such a technique would cleanly solve the stated shortcoming of a procedural interface (p.100, figure 3).

4.1 Coupling minimisation

In general we may deal with coupling both by *minimising* it and *mitigating* it. We minimise coupling by eliminating agreement unnecessary to the meaning being conveyed. This is the principle underlying Parnas's information hiding [25], where the extent of agreement is confined to the interface definitions and their implied semantics. Another technique which avoids making agreements in code is to provide logic for run-time resolution of agreements, such as with late-binding techniques (e.g. virtual functions) or negotiation in network protocols (e.g. character encoding negotiation in HTTP).

However, minimisation techniques will only go so far. As already described, any component must make some assumptions about the components which lie outside of it, and about how it is to communicate with them. Accordingly, we must have ways of *mitigating* this unavoidable coupling.

4.2 Coupling mitigation

Mitigation of coupling seeks to ensure that agreements, although present in code, do not impede re-usability more than necessary. We have identified three classes of technique: localisation, standardisation and adaptation.

4.2.1 Localisation

Localisation techniques are supported by most programming languages, and intend to allow agreements to be conveniently changed. Symbolic constants, macros, functions, aspects and other features all serve to abstract and localise definitions which would otherwise be repeated inline. They are useful, but only if we are able to change the source code.

4.2.2 Standardisation

Standardisation is a popular technique within the software engineering world. One can try to eliminate the problems of coupling by dictating some standardised *global* agreement. There are countless successful applications of standardisation in computer systems – the ASCII character set, Unix tools, the C standard library, the Internet protocol, HTML and so on.

Unfortunately, all these standards also have widely documented foibles. Some suffer obvious functional limitations – for example, the inability of ASCII to encode characters not found in American English. Some are impractical to implement, as with some of the OMG's CORBA standards [17] or C++ export templates [33]. Sometimes, non-conformant implementations are widely deployed, either unwittingly or maliciously, causing workaround maintenance overheads (amounting effectively to *adaptation*) [34, 36]. In all cases, standards suffer from an inherent overhead in agreeing on a standard and guiding its adoption, making the process infeasible for any but the most widely used of agreements. More fundamentally, innovation will eventually demand the abandonment or breakage of any standard.

Since standards are so inherently expensive and fragile, they are suited only to technologies which demand wide deployment but expect only infrequent innovation. There are many such domains, such as network protocols, data encodings and programming languages. However, there are innumerably more which do not fit these criteria. Most software components are continually evolving, adding new features and improve existing ones, and some of these changes invariably necessitate changed interface agreements. If we are to maximise the potential for re-use in these scenarios, adaptation is a necessary technique.

4.2.3 Adaptation

Adaptation can be performed by introducing extra code which mediates between mismatched agreements. This has the clear bonus of being non-invasive – we can perform adaptation from the outside, enabling *black-box re-use* and possibly reducing maintenance overhead. However, there is inevitably some performance cost owing to the added indirection.

We also observe that coupling increases as we compile down our code into lower-level representations, because this compilation process essentially replaces abstract *specification* with concrete *implementation details* which inherently demand more complex agreements. For example, an object code representation of a component will assume a particular calling convention, binary data encodings and so on, whereas the source-level representation assumes only certain abstract properties of these. This shows us that adaptors will be more complex when specified relative to components lower down this stack of representations: black-box binary components may require connectors of greater complexity than are required to perform equivalent adaptation at the source level.

We also distinguish three kinds of adaptation logic, in roughly increasing order of implementation cost: *generated*, *reusable* and *ad-hoc*. Given machine-readable specifications with sufficient semantic content, we can write programs to generate any necessary adaptation code [8, 16]. However, currently such specifications are rare. Adaptation logic can itself be re-used – obvious examples include translation tables and network gateways. By contrast, ad-hoc adaptation is specific to the components it connects, and by definition cannot be systematically re-used.

4.3 Complexity trade-offs

Simple connectors may have no knowledge of the many agreements between the components they connect. Rather, the agreements are *layered* over the simple connector. This layering allows connectors to stay simple, but it extends the reach of coupling into surrounding components. This is part of a more general phenomenon: we can *trade off* complexity between components and connectors. Different positions in this trade-off show different distributions of coupling, and may be correspondingly better or worse at mitigating that coupling. This adds further support to the suggestion that connection models and component programming models should be considered together.

We can vary the complexity of connectors along two dimensions. One is the complexity of the communication abstraction which they provide: for example, compare a simple pipe with a complex publish-subscribe network. The other is the extent of *adaptation* which the connector performs across component interactions, and is observed in *differences* and *asymmetries* between the abstractions exported to each connected component. (Note that while a pipe exposes substantially different interfaces to its reader and writer ends, their symmetry naturally permits a simple implementation.)

This latter trade-off also appears to trade re-usability: pushing more adaptation into a connector couples it more tightly to the components it is designed to adapt. However,

like all components, such connectors have internal structure; a complex connector is likely to contain several simpler components, many of which could well be general in nature and therefore re-usable. The development of a novel connector is therefore, as with developing a novel application, amenable to simplification by re-use. We will explore this idea in the final section.

5. SEPARATING CONCERNS

We have so far used the term "connection domain" in a rather confusing manner. We have argued that separating out the communicational concerns of components into some such domain is an approach which mitigates coupling, especially if the domain supports adaptation. However, we have also argued that connectors are simply a kind of component. The latter might seem to imply that there is no such thing as the connection domain.

To recover the notion of a connection domain, we must fix our view at some level of abstraction. When we choose a computational abstraction, such as a programming language, instruction set or calculus, we fix on a set of *primitive* connectors. The connection domain is the domain in which bindings of these connectors are specified.

We have also been arguing that connection models should be considered inseparable from component programming models. Since primitive connectors are also somehow represented in the component code, it is sets of primitive connectors which tie connection models and component programming models together.

To illustrate how these ideas fit together, we will first return to a high-level view of software and the principle of *re-use*, the realisation of which remains our goal.

5.1 Re-use, recursion and configuration

The re-use paradigm is inherently recursive in applicability: a task which requires a novel output can often be realised by a small amount of new implementation gluing together re-used artifacts in novel configurations. We have already noted in Sections 4.3 and 3.1.3 that components are recursive – they have an internal structure which itself may contain components.

Such composite components are frequently specified using programming languages, as modules which depend on other modules. Sometimes a module will do essentially nothing other than wiring together a set of existing modules – the `main` module in Allen and Garlan's example is an instance of this. Such modules are also frequently written in interpreted languages, like the Unix shell. However, there are even simpler languages which can sometimes fill the same role. One of the simplest is the language of command-line arguments to a linker: it contains a list of object files, which are implicitly wired by symbol matching.

We note that this latter language is clearly not Turing-complete, and hence that this class of *configuration languages* is strictly larger than that of programming languages. It includes architecture description languages (ADLs), composition languages, linkage languages and the like. Indeed, such languages are often better at specifying composite components, because their simplicity admits more automated reasoning and doesn't introduce unnecessary dynamism [11].

Logically, each component bound within a configuration description may be implemented by another configuration description (recursively) or else by code in an implementation language.[8] Just as with implementation languages, as we recurse deeper into lower levels of abstraction, the particular suitability of each configuration language may vary. ADLs may therefore be thought of as higher-level configuration languages, relative to lower-level linkage languages such as Knit [28].

When we have been referring to the "connection domain", we have therefore been referring to the domain of a configuration language. This language need not be a programming language, and specifying certain aspects of communication within it may offer better mitigation of coupling. Where many authors have argued for "first-class connectors", we argue for explicit configuration and suitable configuration languages. Questions remain of what features these languages should have, and of how those features relate to the features of programming languages.

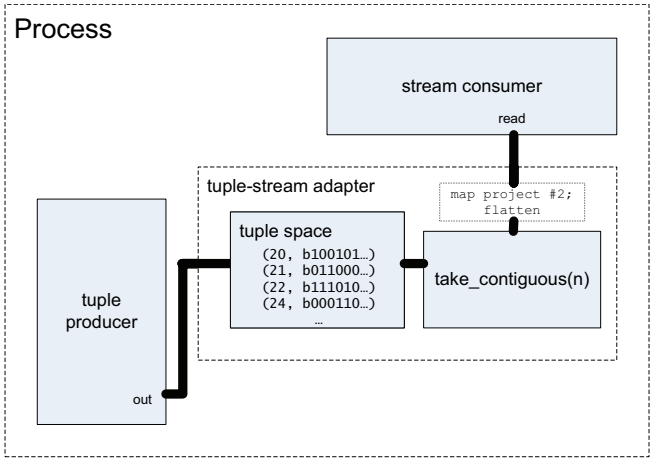

Figure 2: A configuration connecting tuple output and stream input

Figure 2 shows an example configuration with four key features: connector binding, nested configurations, ad-hoc adaptation and re-usable adaptation. The top-level configuration is a combination of two components, one outputting tuples to a tuple space and the other wishing to read a bit-stream, as might be used to handle out-of-order packet delivery in a network. The tuple producer outputs tuples of the form $(sequence\_no., bit\_string)$. The two components are connected by a third component, specified as a configuration of two smaller components and some ad-hoc adaptation. The smaller components are a tuple store and a re-usable adaptation component providing a `take_contiguous(n)` procedure. The latter retrieves from the space a list of two-tuples, each of whose first parts are sequentially numbered, and whose second parts are bit-strings not exceeding n bytes in total length. Some ad-hoc adaptation code projects out the bit-strings and flattens the resulting structure into a single string, which is handed to the stream reader as the output of a *read* call.

[8]Note that not all configuration languages may support such recursion. Indeed, many architecture description languages intended solely for *modelling* cannot express references to concrete component implementation. This is a cause of *drift* [26].

5.2 Design of configuration languages

The design and implementation of configuration languages remains a highly active topic of research, despite the many existing efforts [21, 1, 4, 28]. It is arguably the major research topic in the areas of "software composition" or "software connectors". We now survey the goals and challenges in this area, as viewed from the position we have stated thus far.

5.2.1 Notion of component

A configuration language must have some notion of component. Since we have argued that components have recursive structure, it should also have the property that configured collections of components are also components. We will call this the algebraic property, as have Achermann et al [1].

5.2.2 Primitive connectors

In order to compose components at all, a configuration language must have a notion of primitive connectors which matches or subsumes the set of connectors found within component code. Each component fills one or more roles of some primitive connectors within its own code, but leaves some roles free to be associated with corresponding roles in other components. Since these bindings are expressed in the configuration language, there must therefore be some mapping from the primitive connectors in one or more component programming models onto the notion of connectors present in the configuration language.

This is what we have meant in saying that the component programming model *joins* or *meets* the connection model. Table 2 shows the primitive connectors found in various computational abstractions. We will shortly return to the problem of choosing a suitable set of primitive connectors.

5.2.3 Support for adaptation

We have argued that configuration languages should support adaptation, of which we earlier identified three types (Section 4.2.3). Of these, re-usable adaptation is naturally supported by any configuration language with the algebraic property – units of adaptation logic appear like any other subcomponent. However, configuration languages often lack support for the two other kinds of adaptation: ad-hoc adaptation and generated adaptation.

Ad-hoc adaptation is useful because it relaxes the need for prior agreement on superficial differences of syntax and behaviour. These differences are inevitable between independent designs, but resolving them is sufficiently simple that there is little value in re-using the adaptation logic. We argue that ad-hoc adaptation of arguments [27] and protocol [35] would make a useful addition to linkage languages such as Knit (which already adapts between mismatched names). Without this ability, the developer's only option is to express adaptation in an implementation language, either by modifying the component source (which adds maintenance overhead), or by interposing an entirely new component (which adds unnecessary structural complexity).

Generated adaptation in the general case requires semantically interface specifications, and there are not yet common under contemporary development practices. It is consequently often overlooked, except in constrained domains: stub generation for remote procedure calls is a very common example. The generator is effectively a higher-order component instantiated during construction of software configurations, much like the higher-order connector proposed by Lopes et al [19]. Both of these techniques amount to a basic form of abstraction within the configuration domain, and are therefore potentially useful language features for simplifying the specification of complex configurations.

Computational abstraction	Primitive connectors
generic instruction set architecture (ISA)	register file access main memory access shared I/O device register access program counter increment branch instruction indirect branch instruction interrupt
ISA-level virtual machine (e.g. Xen domain)	register file shared storage local memory access shared memory access program counter increment branch instruction indirect branch instruction hypercall upcall from hypervisor software interrupt from user-space upcall to user-space
C language	global variable access heap access arbitrary memory access statement sequencing function call indirect function call longjmp
Pascal language	global variable access heap access arbitrary memory access statement sequencing function/procedure call indirect function/procedure call
Unix process	virtual processor (sequencing, jumps, branches, registers etc, as in host ISA) virtual memory access (local) virtual memory access (inter-process) virtual memory access (trap to kernel) filesystem access sockets access other system calls signal handling process replication (fork) process replacement (exec)
Java language	static field access instance field access (through heap) static method call instance (virtual) method call exception handling
Haskell language	call-by-name evaluation
pi calculus	synchronous rendezvous

Table 2: Example primitive connectors

5.2.4 Property checking

Given a configuration of components, each with certain known properties, it is desirable to be able to derive and test properties of the whole. Candidate properties span the whole range of software's extra-functional and correctness properties: safety, liveness, termination, performance, security, quality of service, extensibility, and many more. There is already considerable work on each of these in various compositional settings. The real challenge is therefore

to support these kinds of checks wherever possible, without compromising the level of component heterogeneity supported.

5.2.5 Heterogeneity and practicality

There are direct tensions between the need to support composition of *heterogeneous* components, and the desire to support adaptation and property checking. The latter both benefit from increasing levels of semantic specification and metadata in components. However, heterogeneous composition multiplies the complexity in supporting these, by demanding the ability to comprehend all the various type systems and meta-models espoused by the many languages, programming models and packaging standards of both present and future.

Many industrial middlewares (such as Enterprise JavaBeans or the CORBA Component Model) achieve interoperability by enforcing some level of homogeneity: components must all conform to the same model. As discussed in Section 4.2, this, like any standard in a domain of high innovation, is severely limiting.

The usual solution is to push standardisation to the meta-level: we pick a unifying model, and map each component model into it. In the context of a configuration language, this means unifying the component models' primitive connectors.

The practical benefit of configuration languages hinges on the convenience and comprehensibility with which they permit developers to express the required mappings, bindings and adaptation. Choosing a unifying abstraction which is too low-level will make the configuration language unwieldy; one which is too high-level will not be able to unify the primitive connectors of some component models. Variants of the pi-calculus [24] have proved popular unifying abstractions, as evidenced by Darwin [21] and Piccola [1]. It remains to be seen whether this approach will satisfy the demands of adaptation and property checking for highly heterogeneous composition, or whether some other unifying abstraction will prove necessary.

5.3 Directions for future work

Here we outline some practical avenues for demonstrating the ideas we have developed, some of which we will be pursuing in future work.

Role of the operating system The operating system has an inherent role in composition of software, typically providing services of linking, loading and dynamic loading. Its privilege, low level, and language-agnostic nature, together with its pervasiveness, suggest that it could be an effective place to add support for configuration languages. It particularly suits a bias towards heterogeneous, secure, dynamic and high-performance composition.

Extensible checking The difficulties in supporting property checking at the same time as highly heterogeneous composition suggest that such checks might be better implemented in an *optional* or *pluggable* fashion, analogously to pluggable type systems [3] and supporting user extension.

Adaptation as the default The inclusion of adaptation in configuration languages means that programmers need not target any pre-existing interfaces when writing new components. Coding components against idealised interfaces, with the expectation of adapting to real ones nearer deployment time, might reduce the overall incidental complexity of component and configuration code, resulting in more maintainable and reliable systems.

Refactoring legacy code Whatever new component programming techniques are introduced, there is a wealth of existing code which it remains desirable to re-use. There is a possibility of developing automatic or semi-automatic refactoring techniques to separate out communicational concerns, hence enabling more effective re-use of this code.

Extensibility by interposition We have so far spoken little about extensibility as distinct from re-use. The discussed techniques of hierarchical composition and configuration may hold useful approaches. Given sufficiently fine granularity of decomposition, extension can generally be implemented at the configuration level as interposition. A worthwhile challenge would therefore be to quantify and minimise the possible performance or maintenance disadvantages of such fine-grained decomposition.

6. CONCLUSIONS

We have precisely characterised connectors, resolving many ambiguities and inconsistencies in the literature and contradicting the popular assumption that components and connectors are disjoint. We have overviewed the relationship between connectors and coupling, described the techniques for overcoming coupling and achieving re-use, and argued the relevance of adaptation. We have identified the class of configuration languages and stated their relevance to connection, proposing *explicit configuration* and suitable configuration languages as a more meaningful manifesto than "first-class connectors". Finally, we motivated some directions for future work.

7. ACKNOWLEDGMENTS

I am grateful for feedback and encouragement I have received from my supervisor, Dr David Greaves. The final version of this paper has greatly benefited from proofreading by Derek Murray, Henry Robinson and Tope Omitola. I would also like to thank the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] F. Achermann and O. Nierstrasz. Applications = components + scripts. In *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[3] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *ACM SIGPLAN Notices*, 41(10):57–74, 2006.

[4] F. Arbab and F. Mavaddat. Coordination through channel composition. *Coordination Languages and Models: Proc. Coordination*, 2315:21–38, 2002.

[5] M. Barbosa and L. Barbosa. Specifying software connectors. *1st International Colloquium on Theorectical Aspects of Computing (ICTAC '04)*, pages 53–68, 2004.

[6] J. Callahan and J. Purtilo. A packaging system for heterogeneous execution environments. *Software Engineering, IEEE Transactions on*, 17(6):626–635, 1991.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[8] L. de Alfaro and T. Henzinger. Interface automata. *Proceedings of the 8th European software engineering conference*, pages 109–120, 2001.

[9] R. DeLine. Avoiding packaging mismatch with flexible packaging. *Software Engineering, IEEE Transactions on*, 27(2):124–143, 2001.

[10] S. Edwards and B. Weide. WISR8: 8th annual workshop on software reuse: summary and working group reports. *ACM SIGSOFT Software Engineering Notes*, 22(5):17–32, 1997.

[11] E. Eide, A. Reid, J. Regehr, and J. Lepreau. Static and dynamic structure in design patterns. *Proceedings of the 24th international conference on Software engineering*, pages 208–218, 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[13] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th international conference on Software engineering*, pages 179–185, 1995.

[14] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[15] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, 2004.

[16] C. Haack, B. Howard, A. Stoughton, and J. Wells. Fully automatic adaptation of software components based on semantic specifications. *Algebraic Methodology & Softw. Tech., 9th Int'l Conf., AMAST*, 2002.

[17] M. Henning. The rise and fall of CORBA. *ACM Queue*, 4(5):28–34, 2006.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th European Conference in Object Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[19] A. Lopes, M. Wermelinger, and J. Fiadeiro. Higher-order architectural connectors. *ACM Transactions on Software Engineering and Methodology*, 12(1):64–104, 2003.

[20] 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):336–354, 1995.

[21] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, 1995.

[22] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: new-age components for old-fasioned Java. *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–222, 2001.

[23] N. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. *Proceedings of the 22nd international conference on Software engineering*, pages 178–187, 2000.

[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

[25] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[26] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[27] J. Purtilo and J. Atlee. Module reuse by interface adaptation. *Software - Practice and Experience*, 21(6):539–556, 1991.

[28] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.

[29] R. Reussner. Automatic component protocol adaptation with the CoConut/J tool suite. *Future Generation Computer Systems*, 19(5):627–639, 2003.

[30] C. Shannon and W. Weaver. A mathematical theory of communication. *University of Illinois Press*, 1949.

[31] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *ICSE Workshop on Studies of Software Design*, pages 17–32, 1993.

[32] M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *Proceedings of the IEEE Symosium on Software Reusability*, 1995.

[33] H. Sutter and T. Plum. Why we can't afford export. *ISO C++ committee paper ISO/IEC JTC1/SC22/WG21 N1426*, Mar. 2003.

[34] T. Ts'o. Microsoft "embraces and extends" Kerberos v5. *Usenix ;login: Windows NT Special Issue*, Nov. 1997.

[35] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[36] N. Zelnick. Nifty technology and nonconformance: the web in crisis. *Computer*, 31(10):115–116, 1998.