Component Coordination in Middleware Systems

(full paper)

Matthias Radestock and *Susan Eisenbach*
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom
Phone: +44 171 594 8264, Fax: +44 171 581 8024
Email: {M.Radestock,S.Eisenbach}@doc.ic.ac.uk

March 15, 1998

Abstract

Configuration and *coordination* are central issues in the design and implementation of middleware systems and are one of the reasons why building such systems is more difficult and complex than constructing stand-alone sequential programs. Through configuration, the structure of the system is established — which elements it contains, where they are located and how they are interconnected. Coordination is concerned with the interaction of the various components — when an interaction takes place, which parties are involved, what protocols are followed. Its purpose is to coordinate the behaviour of the various components in a way that meets the overall system specification. The open and adaptive nature of middleware systems makes the task of configuration and coordination particularly challenging. We propose a model that can operate in such an environment and enables the dynamic integration and coordination of components through observation of their behaviour.

Keywords: Coordination, Adaptive Open Distributed Systems, Novel Paradigms

1 Introduction

We can view a distributed system as a collection of distributed components that interact with each other. The concerns of any distributed system, including middleware systems, can be separated into four parts:

- The *communication part* defines *how* components communicate with each other.
- The *computation part* defines the implementation of the *behaviour* of individual components. It thus determines *what* is being communicated.
- The *configuration part* defines the *interaction structure*, or *configuration*. It states which components exist in the system and which components can communicate with each other, as well as the method of communication. Basically it is a description of *where* information comes from and *where* it is sent to.
- The *coordination part* defines patterns of interaction, ie. it determines *when* certain communications take place.

Inter-part dependencies yield a layered structure (cf. Fig. 1). From a software engineering viewpoint lower layers need not, and should not, know about the higher layers. As far as the lower layers are concerned the upper layers need not even exist. Each of the layers could have its own model, language and implementation (ie. support in a distributed system platform). This clear separation of concerns is extremely beneficial, enabling a high degree of reuse and easier maintenance.

1.1 Dynamic Configuration and Coordination

The interaction structure in middleware systems often changes dynamically: new components are created, existing components are destroyed, connections between components are established and broken up. Such dynamic configuration activities are derived from the functional specification of the system which may state, for instance, that a new member can join a video conference after receiving an invitation. These activities thus need to be triggered by the components in the system themselves, and so the configuration layer needs to be supported by the distributed system platform during the entire life-time of the system in order to enable dynamic access to its functionality.

Coordination specifies patterns of interaction. Such a pattern may, for instance, be that component A can only send message X to component B after component C has sent message Y to component D. Coordination requires configuration — The patterns of interaction need to be specified *before* the parties of interaction; which is precisely the task performed by configuration. We can make a distinction between static and dynamic coordination. In the former case, the interaction patterns are fixed throughout the life-time of a system. In the latter case, interaction patterns are altered dynamically as part of satisfying the application requirements, ie. the changes to the interaction structure and patterns are ultimately triggered by computational components. The coordination layer must exist during the entire life-time of the system. A mechanism is required that enables the interaction with the computation layer.

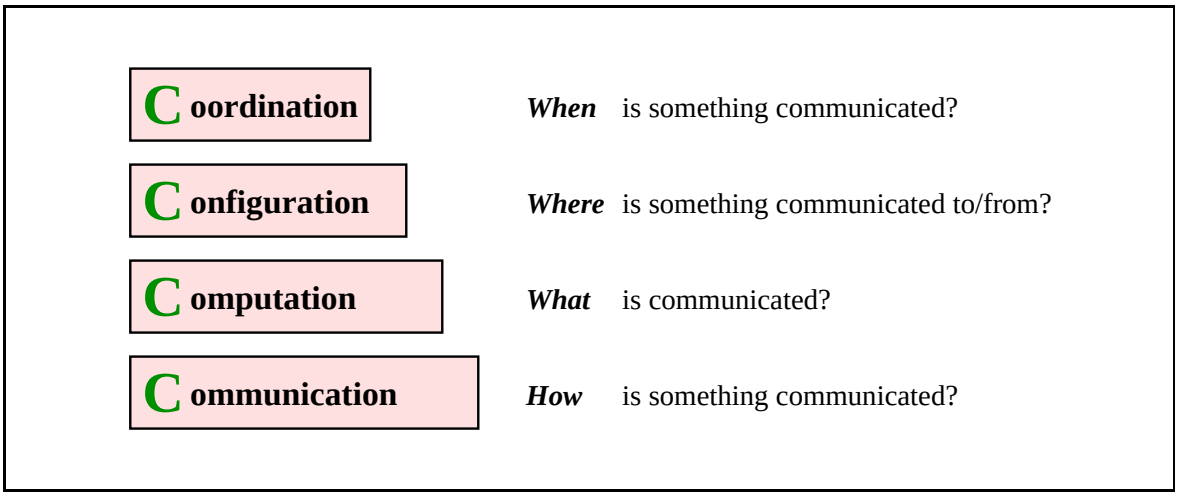

Figure 1: The Four Concerns in Distributed Systems

1.2 Adaptive Systems

A dynamic coordination model allows us to specify systems where all possible dynamic changes to the interaction structure and patterns are known at compile time and are triggered by application components. However, this is insufficient in many large distributed systems, especially middleware systems. Such systems are typically long-lived, often running for days and in some cases even years. They require interactive management; both human and automated agents need to be able to reconfigure the system while it is running. Furthermore they need to be able to alter the specification of the coordination, configuration and computation layers in order to make permanent changes to the overall system behaviour. An example would be a video-conferencing system where some new hardware, say a projection screen, is added to the system during a conference. The components representing the screen need to be added to the system's computation layer. Then the configuration layer needs to be modified to forward all data of the conferencing communication to that component. Finally we need to alter the coordination layer to ensure that the new component interacts with the rest of the system in the desired manner.

These so-called *adaptive systems* or *evolving systems* are capable of accommodating changes that were not anticipated during the original system development. This is in contrast to static and dynamic systems. Both of these can contain interactive user interfaces or can interact with external components, but such interaction and the resulting changes need to be implemented as *part* of the system functionality; the system functionality itself cannot be altered. Adaptive systems create considerable demands on the capabilities of a middleware architectures and the use of *reflection*[MWY91, GK91] as a means of supporting these advanced requirements has been advocated in recent research[RE96a, BP97].

1.3 Open Systems

A universal model for configuration and coordination has to be suitable for operating within the context of open systems. This means that it has to be easy to integrate it into existing

distributed system platforms and it needs to enable configuration and coordination of existing components without requiring any alterations to them. The model needs to function across heterogeneous systems that may be based on a variety of programming paradigms, languages and platforms. Not only should it be possible to control the configuration and coordination of components in a heterogeneous system, but it must also be possible to control it from the *inside* of the various platforms that make up the system — if configuration and coordination are part of application requirements, then they need to be controllable from potentially any part of the application.

In an open system little is known of the components' implementations and, furthermore, it may be impossible to alter them. Thus for configuration and coordination to operate in a truly open setting and enable the dynamic integration of components, they must not depend on any knowledge of component behaviour. They certainly should not *rely* on any behaviour specifications, because in general it is impossible to ascertain whether components actually meet them, and hence system safety and security could be compromised. Coordinating components without relying on any explicit behaviour specification is crucial when it comes to middleware systems. In such systems it is important to perform integration with a minimum impact on existing components. Typically, integration is achieved by embedding calls to some special communications API that enable interaction with other system components via the middleware infrastructure. The impact on the existing application in terms of code changes is usually minimal and introducing coordination should not increase this.

1.4 Related Research

The issues of configuration and coordination have received growing attention from the research community, and, as a result, several models, languages and implementations have been proposed and executed. Distributed System standards such as CORBA[MZ95, Pop97], DCE[Sch93] and RM-ODP[BS97], and their implementations, address the issue of configuration by introducing a *brokering* mechanism which matches requests by components for particular services with components providing these services. With this basic building block in place, most configuration issues can be addressed. However, coordination is not addressed at all and left entirely to the programmer of the components.

Formalisms, such as Gamma[BM90] and languages such as Linda[Gel85, Ban96] have emerged. However, they are not aimed at integration with existing systems or operation in an open environment. Furthermore, only limited facilities exist for re-using coordination patterns, and coordination is typically embedded in application code rather than being separated. Research in software architecture[GS93, PW92, GP94, RE96b], by contrast, has placed considerable emphasis on layer separation. However, the distinct role of coordination has only been recognised recently. Consequently several systems have emerged that address coordination issues, usually as extensions to existing systems. Examples of this are ToolBus[BK96] (an extension to the PolyLith software bus[Pur94]), ConCoord[Hol96] and Midas[Pry98] are extension of Darwin[MEK95]. ActorSpace[CA94] is an extension of an actor language and Manifold[Arb96] is based on a model where processes communicate anonymously via streams. Common to all approaches is the lack of openness — coordination in these systems relies on particular features that are unique to the specific system. Dynamic integration of existing components is usually possible, but only for components that have been designed, implemented and compiled for the particular system used. Dynamic change is supported, but systems cannot adapt to changes in the requirements that go beyond the scope of the original

specification. Furthermore, the above coordination mechanisms only provide limited means of abstraction, ie. the construction of patterns of coordination and their reuse. This is mainly due to the use of separate coordination languages that lack expressiveness.

1.5 Outline of Our Approach

Our aim is to enable coordination in adaptive and open distributed systems, such as middleware systems. Further to that, we want to be able to integrate components on the level of source code, object code and running code, including existing and running legacy applications. The key element in our solution is a mechanism that enables the observation and coercion of dynamic component behaviour through the interception of messages. The first part of this paper is devoted to the description of this so-called *traps* model. Traps employ a sophisticated type system for specifying message patterns and rules for defining actions to be taken when messages have been intercepted. The patterns and rules can be altered dynamically and thus traps represent a dynamic configuration and coordination layer. Since traps operate without having any knowledge of the behaviour of the components, they do not depend on any component interface/behaviour specification. Traps integrate the configuration and coordination layers into the computation layer without jeopardising the benefits of clear separation. Thus coordination can be designed and implemented using the same techniques deployed in the design and implementation of the application components. As a result, coordination code can be reused in the same way as application code. The approach also enables *meta coordination*; the coordination of coordination itself. In the final part of this paper we use the well-known example of the Dining Philosophers to illustrate how our model can deal with various, increasingly complex, coordination tasks.

2 Traps — A New Model For Configuration and Coordination

In order to facilitate configuration and coordination in an evolving heterogeneous distributed environment, we need to devise a suitable model that has very few demands on the system architecture and is thus easily incorporated into both existing and new systems. The first step in devising our coordination model is to take a slightly different view of the message-based communication model. This new view is illustrated in Figure 2. When a component A sends a message to another component B, the message gets stored in a location of the so-called *message space*, based on its *type*. From that location it is then forwarded to the receiving component. It should be noted that this transformation of our view of the communication model happens on the *conceptual* level, unlike, for instance, in Linda where the tuple-space model is exposed to the programmer. The new view is transparent to the components involved; as far as component A is concerned it is still sending a message to component B, and as far as component B is concerned it is still receiving a message from component A. Conceptually though we can view things differently. Component B is *notified* of an 'interesting' activity: a message that component A is trying to send to component B.

2.1 Message Types

A message between two components consists of

- the *originator*, ie. the component that sent the message,

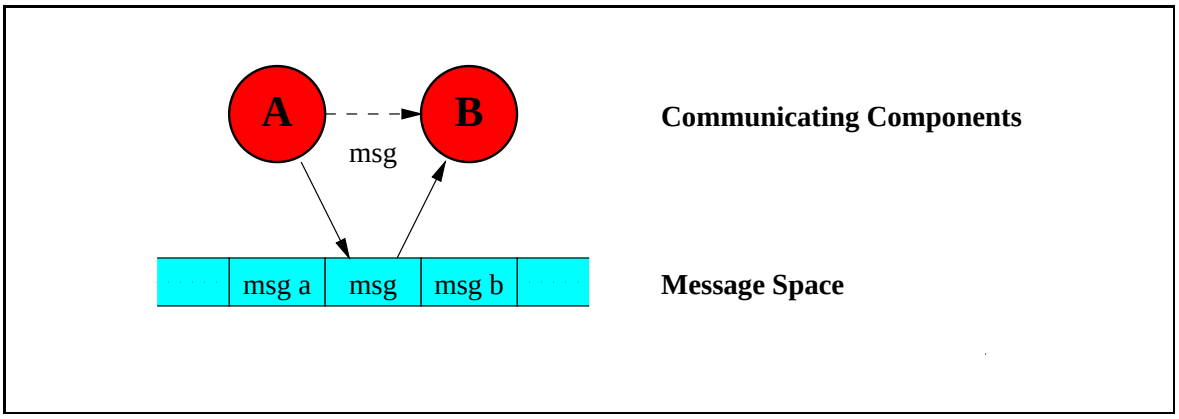

Figure 2: Communication via Message Spaces

- the *recipient*, ie. the component that is the intended recipient of the message,
- the *content*, ie. the data elements, and
- the *context*, ie. additional information required by the communication and coordination layers, such as time stamps and request ids.

The *type* of a message encompasses those elements that are visible to the programmer, ie. everything apart from the context information. It can therefore be defined as

$$\mathsf{MessageType} = \mathsf{Component} \times \mathsf{Component} \times \mathsf{Component}^*$$

ie. the product type of components (originator) , components (recipients) and sequences of components (message content). Locations in a message space correspond to message types, hence messages sharing the same originator, recipient and content are stored in the same location. Some examples of message types (in pseudo-code) are:

`device=>handler()`	an empty message from *device* to *handler*
`device=>handler(handle,data,12)`	a message from *device* to *handler* with three components as content: *handle, data* and *12*.

Note that we do not attach any special significance to the first element of the message content. In many object-oriented systems this will be the name of a method to be invoked, however, our model operates on a more abstract level and can therefore be oblivious to this special semantics.[1]

It is important that message types are defined in terms of components and not component types, because coordination operates on the level of individual components rather than their types. Consequently the domain of message types can be very large (even infinite) and thus, plainly, locations in a message space cannot be real entities requiring system resources. They, along with the notion of message spaces, are just concepts.

[1] The `=>` in our notation should not be confused with the `->` method invocation construct found in languages like *C++*. In our notation the element to the left of the arrow is the *sender*, the element to the right is the *recipient* and the arguments follow.

2.2 Message Patterns

A *message pattern* defines a subset of the domain of message types. Its domain can therefore be defined as the power-set of message types, ie.

$$\mathsf{MessagePattern} = \mathrm{P}(\mathsf{MessageType})$$

Message patterns are used by the programmer to identify interesting messages, ie. messages requiring special treatment by the coordination layer. They typically use the type system of the underlying programming language. However, it should be noted that the type system ought to be sophisticated, with the ability to dynamically construct types from instances and not just other types. If these capabilities are not present then a separate type system must be introduced to complement the existing one. Examples of some more sophisticated message patterns are:

```
device=>handler(handle,data,12)
Device=>Handler,'special(handle,Any)+Ŝtring
Device=>handler()+Any,Device=>Device(transfer)
Any=>Any()+Any
```

The first pattern covers exactly one message. The second pattern covers all messages from components of type *Device* to components of type *Handler* or the symbolic component *special*, with at least two arguments, the first of which must be the component *handle*, the second of which can be of any type, and the remaining arguments being of a type other than *String*.[2] The third pattern covers all messages from components of type *Device* to the component *handler* with any number of arguments of any type, and messages between components of type *Device* with *transfer* as an argument. The fourth pattern covers all messages.

As can be seen from these examples, a sophisticated type system enables the concise specification of very complex patterns. Traps do not inherently depend on such type systems though, as there are other places in the trap system where such complex decisions can be made. However, the more expressive the type system is, the less computationally expensive the introduction of traps becomes.

2.3 Translation Rules

The new way of viewing the message-based communication model is obviously of not much use if all that happens is essentially the same as before; locations in the message space just serve as 'trampolines' that bounce messages to their target components. What we require for coordination is some means of altering the flow of messages. We achieve this by installing *translation rules* at locations in the message space. These translate the messages at the location into other messages, thus relocating them to different places in the message space and effectively intercepting the message.

The translation rule in Fig. 3 translates the original message (from component A to component B) into a message that has component C as the target. The new message could be seen as having precisely the meaning presented earlier as an alternative perspective, ie. component C is notified of the attempt by component A to send a particular message to component B. Component C thus conceptually resides in the coordination layer. It could coordinate all the

[2] Upper case identifiers in our pseudo-code denote *types*, lower case identifiers denote *variables* holding component references and identifiers prefixed with a single quote denote symbols.

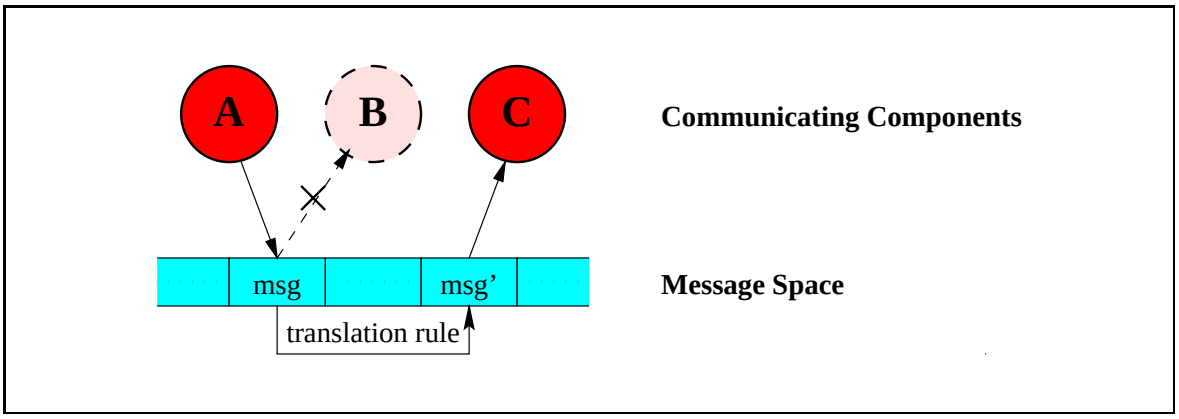

Figure 3: Message Relocation by Translation Rules

activities between component A and B if translation rules were specified that relocate any messages exchanged between the two components. Thereby C could act as a simple forwarder, or could accomplish arbitrarily complex coordination tasks, such as protocol translation and enforcement, interaction with other coordinators etc.

In our model, translation rules always translate messages into new messages where the originator is a so-called *message wrapper* of the original message, ie. an encapsulating component for the original message.[3] Further, the recipient and content of the new message does not depend on the original message. A translation rule thus simply specifies a new recipient and content:

$$\mathsf{TranslationRule} = \mathsf{Component} \times \mathsf{Component}^*$$

Such translation rules can be defined completely independently of the underlying programming language since they do not perform any computation whatsoever. This keeps the semantics simple, offers opportunities for easy and efficient implementation, and enables deployment in a heterogeneous language/platform setting.

When a translation rule is applied to a message the resulting message contains the message wrapper of the original message as the originator. The recipient and content are supplied by the translation rule. For instance, the translation rule

```
'logger('io-event)
```
[4]

applied to the message

```
device=>handler(handle,data,12)
```

will result in the message

```
[device=>handler(handle,data,12)]=>logger('io-event)
```

[3] The purpose of message wrappers is to expose messages as components in the programming language, even though messages themselves may not be components.

[4] In our pseudo-code we define translation rules in the same way as message types, except that the originator and following `=>` are not present.

where the square brackets denote the message wrapper of the original message.

Messages resulting from the application of translation rules get stored at their appropriate locations in the message space. Hence they can be subject to further translation. Eventually the messages cannot be translated any further and are sent to their intended recipient. Since translation rules always generate messages containing an encapsulation of the original message, the elements of the original message, such as the original recipient, can all be used in the further decision process by coordination components.

2.4 Defining Traps

Placing a translation rule on a location in the message space is the equivalent of 'setting a trap', hence the name of this model. Instead of being bounced back and delivered to the intended recipient, a trapped message undergoes translation. The same translation rule often applies to many locations in the message space. As we noted before, the number of locations in the message space can be very large or even infinite. It is therefore impossible to install translation rules individually at locations in the message space. Hence a trap definition consists of two components:

- a *message pattern*— using the described type system for messages, this defines a subset of the domain for messages, ie. locations in the message space. Messages in the subset are caught by the trap.
- a *translation rule*— using a new recipient and content, this translates it into a new message.

Thus, traps can be formally characterised as

$$\begin{aligned} \mathsf{Trap} &= \mathsf{MessagePattern} \times \mathsf{TranslationRule} \\ &= \mathsf{MessagePattern} \times \mathsf{Component} \times \mathsf{Component}^* \end{aligned}$$

In our pseudo-code we define traps using a `>>` operator. For instance, the trap

```
Device=>Handler(handle)+Any >> logger('io-event)
```

will trap all messages sent from devices to handlers with *handle* as the first argument plus any number of further arguments of any type. It will translate these messages to messages to the component *logger*, with the symbolic component *'io-event* as the first argument and the encapsulated original message as the originator. Note that message patterns are part of the type system and message wrappers can be matched against them. This enables the specification of traps that further translate a message that has already undergone some translation. For instance, messages generated by the above trap would match a pattern

```
[Device=>Handler(handle)+Any]=>logger('io-event).
```

2.5 Matching Policies

When a message is matched against the message patterns of the currently installed traps, it is possible that it matches more than one pattern. In dealing with this situation, we have a choice between two matching policies:

1. Message translation is performed by all traps whose message pattern matches a message.

2. Message translation is performed by the traps whose message pattern matches the message most specifically, compared to the other patterns.

Both policies are useful in certain contexts. The first policy would be employed in cases where several independent coordinators are interested in a message and therefore install traps to intercept it. For instance the two traps

```
Device=>Handler(handle)+Any >> logger('io-event)
Device=>Handler()+Any >> forwarder('io-event)
```

could be installed completely independently; one in order to log the message, one in order to forward it. In that case, we would actually want all coordinators to deal with the message in this case, instead of a selection being performed based on the most specific message pattern (which in the above case would select the first trap in preference to the second). The second policy is typically employed in cases where a coordinator installs several traps; more general traps for dealing with 'normal' messages and specific traps for dealing with 'exceptional' messages requiring special coordination, e.g.

```
Device=>Handler()+Any >> forwarder('io-event)
Device=>Handler(handle)+Any >> forwarder('handle-io-event)
```

In order to deal with these two cases we therefore implement the following policy:

> Traps with the same new recipient form a *trap group*. When a message matches the message patterns of several traps in the group, then only the translation rule of the trap with the most specific matching message pattern is invoked. Message translation is performed by all trap groups that contain traps with patterns matching the message.

Trap groups define the boundaries of pattern-based selection and ensure that a message is not translated into two messages with the same recipient. Thus in the above example a message

```
device=>handler(handle,data,12)
```

would be translated into two messages,

```
[device=>handler(handle,data,12)]=>logger('io-event)
[device=>handler(handle,data,12)]=>forwarder('handle-io-event)
```

which is exactly what we would expect.

There is a special case involving the pattern-based selection — when several patterns match a message but neither of them is more specific than any of the others. In the simplest case this will occur when two patterns are identical. The policy we employ in this case is to select the most recently installed trap, thus ensuring a deterministic outcome of the selection process.

2.6 Coordination Protocol

In our model, coordination is accomplished by encoding the coordination logic in components that receive intercepted messages as an input. The principal decision to be made by the components is when (if at all) a message should be dispatched, ie. delivered to its final destination. In order to make that decision, the coordination components need to interact

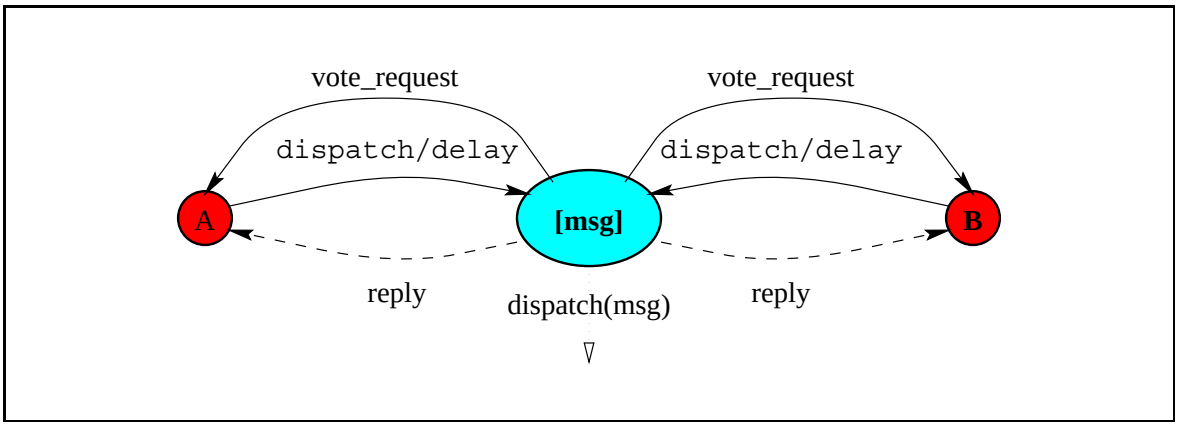

Figure 4: Coordination Protocol

with each other. This causes a software engineering problem because the coordination logic is often a composite entity whose elements are unaware of each other and hence cannot engage in any interaction. The composite nature of the coordination logic is a result of the composite nature of applications — they are built out of components which each have their own coordination logic and are 'glued together' by yet more coordination logic. To overcome this problem we have to add a *coordination protocol* to our model.

When a message is intercepted by traps, it is eventually translated into a set of messages that cannot be further translated. Then the following happens (cf. Fig. 4):

1. The message wrapper of the original message initiates a *round of voting*. Messages in the message set are interpreted as *requests for votes* and dispatched.

2. The message wrapper waits until the same number of votes as requested have been received. Participants in a vote submit their vote through sending a `dispatch` or `delay` message. They expect to receive a reply to that message, containing the outcome of the round of voting.

3. If all participants voted for `dispatch` then the outcome is dispatch. The original message is dispatched, ie. sent on its way to its destination.

4. If one participant voted for `delay` then the outcome is delay. Nothing is done.

5. The reply messages with the vote result are sent to the participants.

This two-phase protocol ensures that the original message is only dispatched if consensus has been reached by all participants, even though participants do not know about each other. The protocol makes no assumptions on how components deal with messages, e.g. components could process messages in parallel or in an order different to the order of arrival. Introducing the coordination protocol does not require any changes to the semantics. The first step in the protocol is captured by the semantics already since the application of translation rules always results in messages that have the message wrapper of the original message as an originator. Thus they always appear to be sent by the message wrapper anyway. The rest

of the protocol is implemented by the message wrapper and the participating components in terms of ordinary component-to-component interactions.

A new round of voting can be initiated at any time by delivering a `vote` message to the message wrapper. The message will be *ignored* by the message wrapper if the message has been dispatched already, and it will be *delayed* if the current round of voting hasn't been completed yet, thus ensuring that rounds of voting on a message occur in a sequence. A new round of voting is initiated by participants when they 'change their minds' regarding the vote for a message. Such a change of mind is only significant if it was from `delay` to `dispatch`, since that is the only case where it may cause a previously delayed message to be dispatched or discarded. The message wrapper expects as many `vote` messages as there were `delay` votes in the previous round of voting. Only then will it actually trigger a new round of voting by dispatching all the messages in the message set (which it needs to remember) again. Participants need to remember those messages that they voted to delay in order to be able to initiate new rounds of voting. However, they are allowed to forget messages they voted to dispatch since they will be requested to vote on them again anyway, if necessary.

2.7 Protocol Extensions

The requests for votes sent to participants are just ordinary messages and hence they can be subject to delays. Participants receiving the same requests may receive them in a different order. This can lead to deadlock if participants wait for the results of a vote before proceeding. Other scenarios can lead to livelock or lack of fairness. In order to free the programmer from having to deal with these issues we need to extend the above protocol. The protocol also requires extension in order to enable the replacing of intercepted messages in the context of configuration, ensure ordered message delivery and provide easy coordination of request-reply-style interactions. The details of these extensions are beyond the scope of this paper.

The complexities of the coordination protocol can be hidden from the programmer by splitting coordination components into two separate components; a *protocol wrapper* and a *logic wrapper*. Programmers need only to be concerned with the logic wrapper which, typically, implements some kind of state machine. All that the component has to do is vote on a message and perform a state transition if the vote succeeds. The protocol wrapper, which can be automatically created by the system, controls the message flow to and from the logic wrapper and implements the coordination protocol. It takes care of message ordering and re-voting and isolates the programmer from any changes that may be made to the protocol over time.

3 An Example

We shall now demonstrate how configuration and coordination in applications can be accomplished using traps, using the well-known example of the 'Dining Philosophers'. Philosophers sit around a table with food. There is a chopstick between every two philosophers. Philosophers require both their left and right chopstick in order to eat. A chopstick can only be held by one philosopher at a time. It needs to be ensured that philosophers don't starve — we need to prevent situations of deadlock and livelock and ensure fairness. Philosophers and chopsticks are to be treated as 'given' types of components, ie. we do not have access to their source code and hence cannot modify it. Neither do we have any detailed knowledge of the components' behaviour. Thus coordinating philosophers and chopsticks in the context of

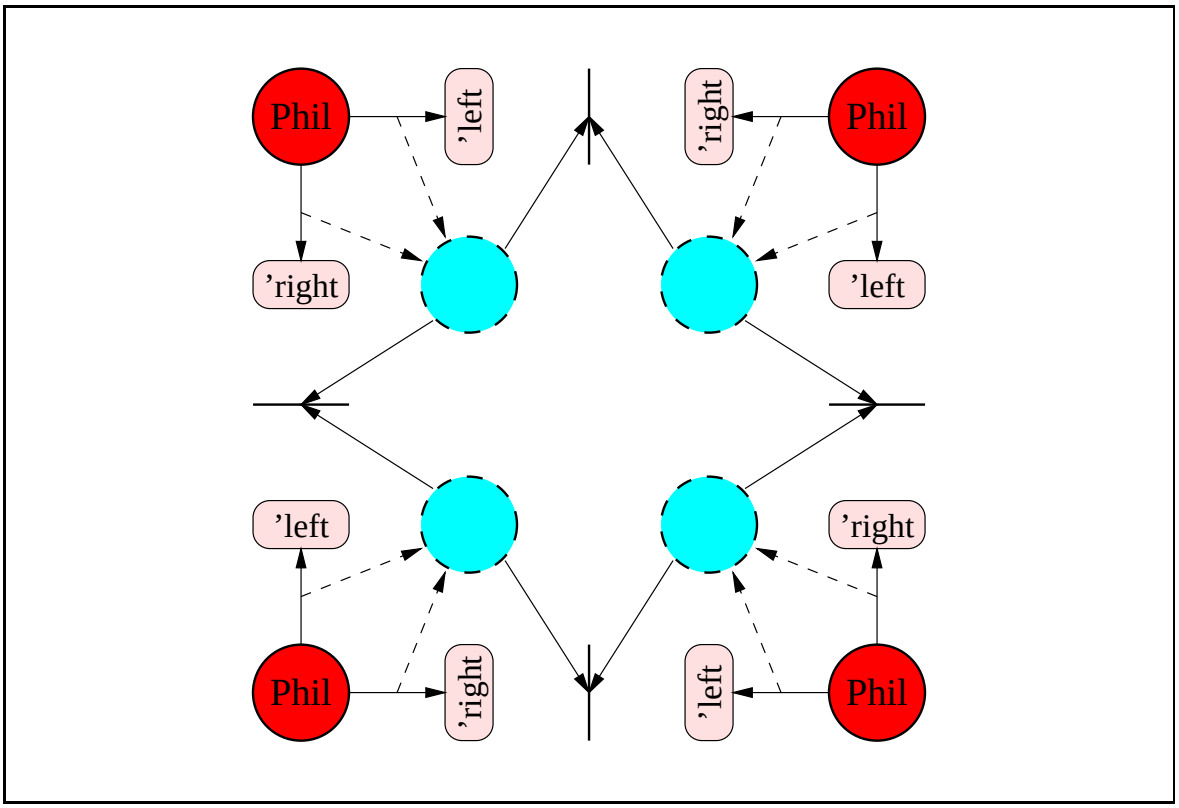

Figure 5: Configuring the Dining Philosophers

Dining Philosophers is very similar to the integration/reuse of 'legacy' components in a heterogeneous distributed system, such as a middleware system. We investigate three variations of the example, with increasing degree of complexity and show how the additional complexity is primarily accommodated in an incremental fashion without the need for rewriting existing code.

3.1 Configuration

Initially, we deal with static configuration only, ie. we create a certain number of philosophers and chopsticks and establish a configuration where philosophers are assigned chopsticks in the manner specified. We are assuming that coordination is performed by the philosophers and chopsticks themselves, which therefore have to be aware of the context they are being used in. When philosophers and chopsticks are created, the required coordinators are set up as well. After all coordination logic is in place, the philosophers are told to start eating. A philosopher attempts to pick up a chopstick by sending a `pick` message to the symbolic components `'left` or `'right`. It drops a chopstick by sending a `drop` message. We assign a coordinator to each philosopher, which installs traps that intercept the `pick` and `drop` messages of a philosopher and translate them into messages to the coordinator (cf. Fig. 5). Upon receipt of one of these messages the coordinator replaces the original `pick` or `drop` message with a `get` or `put` message to the actual left or right chopstick.

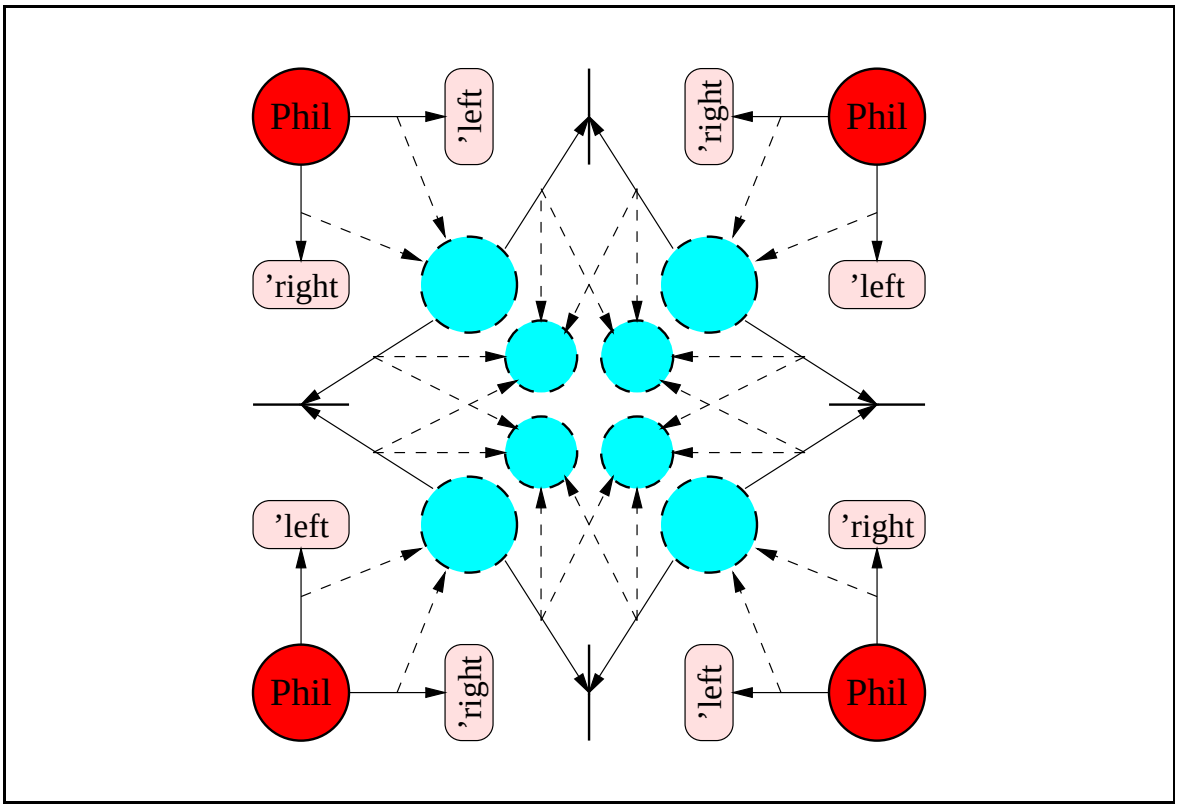

Figure 6: Coordinating the Dining Philosophers

3.2 Coordination

To facilitate reuse, philosophers and chopsticks should be completely unaware of the context in which they are being used, specifically philosophers should not require knowledge that picking up a chopstick requires coordination with other philosophers. Philosophers should be able to attempt picking up a chopstick at any time and dropping a previously picked up chopstick at any time. They can attempt to pick up both chopsticks at the same time or one after the other or pick one up and then drop it again. The only assumptions we make, for simplicity's sake, is that philosophers will not pick up a chopstick they currently hold, and will not drop a chopstick they do not currently hold. The result of this liberal approach is that a great variety of component implementations can fulfil the roles of philosophers or chopsticks. Thus there is a wide scope for reusing existing components in that role without the need for modification.

We coordinate the dining philosophers by ensuring that when a philosopher attempts to pick up the first chopstick, the request is delayed until *both* chopsticks are available. When the first chopstick is being picked up by a philosopher all requests by other philosophers to pick up the complimentary chopstick are delayed. This policy ensures both freedom of deadlock and livelock as well as guaranteeing that chopsticks are only picked up by one philosopher at a time.

In order to implement the coordination policy, coordinators create an auxiliary component.

Traps are installed that intercept messages to the philosopher's chopsticks (cf. Fig. 6). One set of traps intercepts the messages sent by the philosopher's coordinator, another set has a message pattern which matches messages to the chopsticks from *any* component. Since this pattern is less specific than the one of the first set of traps, these traps will intercept all messages sent to the chopsticks by other coordinators. The intercepted messages are submitted for voting to the auxiliary components. The auxiliary components implement a state machine encapsulating the coordination policy by delaying `get` messages.

3.3 Dynamic Change

In the previous two variations of the Dining Philosophers, both philosopher and chopstick components are created by the coordination code itself. So while the coordination logic has no knowledge of the component implementations, it *does* have knowledge of component's existence. However, coordination is often required in settings where components are created dynamically by existing code. We shall now demonstrate how this can be accomplished in a variation of the Dining Philosophers example where the creation of philosophers happens outside the coordination logic and is not controlled by it. To keep things simple, we assume that philosophers will not attempt to pick up the second chopstick until they've successfully picked up the first.

We modify the coordination logic to dynamically create a chopstick and the required coordinators whenever a philosopher accesses a chopstick for the first time. We do this by installing traps that intercept all `pick` or `drop` messages sent by philosopher components to the symbolic `'left` and `'right` components, and forward them to a `Table` component, which serves as a coordinator. Since the message pattern is less specific than the ones of the traps installed by the individual coordinators, messages from philosophers will not be caught in these traps once the philosophers' individual coordinators have been installed. Messages caught by the generic traps are resubmitted once the necessary coordinators have been created, thus ensuring that they get processed in the same way as if the coordinators had been in place all along. We have to deal with a special case: Since coordinators require two chopsticks, we need to wait until we have got two philosophers before creating any coordinators. We do this by implementing a simple state machine on our `Table` coordinator.

New philosophers are placed at the 'end' of the table, ie. next to the last philosopher. If the new philosopher first requested the left chopstick then he is placed to the right of the last philosopher and the new chopstick will be shared between himself and the last philosopher. If the new philosopher first requested the right chopstick then he is placed to the left of the last philosopher and the new chopstick is shared in the same way.[5] The coordinator of the last philosopher needs to be notified of the changed configuration so that it can amend its existing traps. We do this by sending it a `setL` or `setR` message, containing a reference to the new chopstick, which causes all existing traps of the coordinator to be removed and new traps to be installed. The resulting changes in the system configuration are illustrated in Figure 7.

Replacing a chopstick is not always safe. We cannot replace it while it is held by the philosopher in question. We therefore amend the coordinator state machine to defer the `setL` and `setR` message in certain states — whenever the chopstick in question is held by *any* philosopher.[6] We further need to enforce *atomicity* of all changes in the trap configuration

[5] In principle we could always seat the philosopher to the left (or right) of the last (or, for that matter, any other) philosopher. We have chosen the described policy for reasons of symmetry.

[6] This simplification allows us to leave the state machine essentially unchanged. We would have to extend

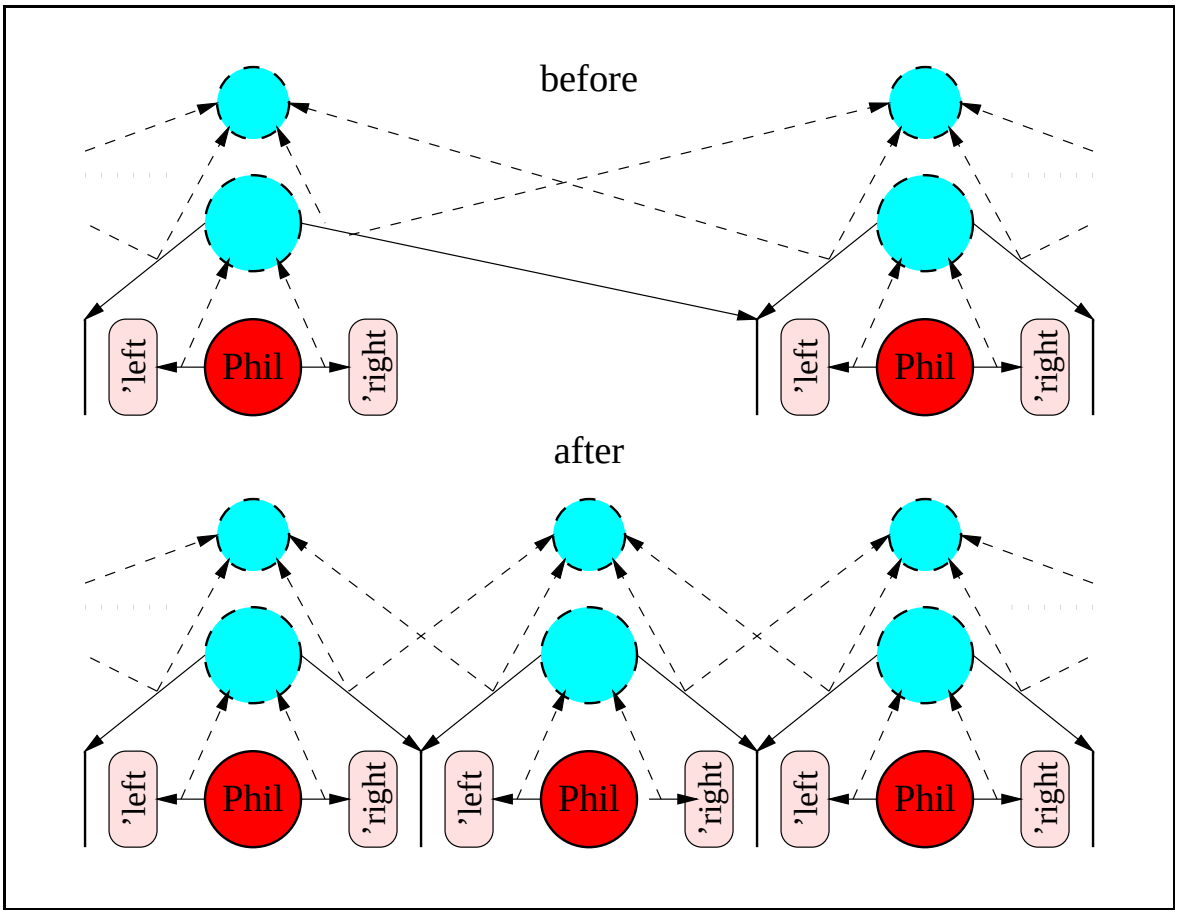

Figure 7: Introducing a new Philosopher

relating to the introduction of a new philosopher. This is important in order to provide the new and old coordinators with a consistent view of the system state (ie. which chopsticks are held by which philosophers). Atomicity is ensured by making all activities part of a *transaction* which is committed once all changes to the traps have been scheduled, thus causing all the changes to take place at the same time. The trap transaction logic also causes messages caught by 'old' traps to be resubmitted (provided voting on them hasn't been completed yet) and thus being subjected to the new traps.

4 Conclusions

The model of coordination presented in this paper enables the coordination of components in open adaptive systems, independently from the underlying distributed system platforms. Some recent advances in this direction can be found in the notions of synchronizers[FA93, Fro96], regulated coordination[MU97] and programmable coordination media[DNO97]. Our approach shares many of the initial motivations and there are also some similarities between

it if the constraint turned out to be too strict.

the concepts. For instance, the notion of a programmable coordination media is based on intercepting messages in a very similar way to our traps, and underlying synchronizers is a coordination protocol that is in many ways similar to ours. However, in our opinion these models are still not sufficiently open and only have very limited support for system evolution and the abstraction and reuse of coordination patterns in a truly open setting.

We can make several important observations on the model presented in this paper. Firstly, the presence of coordinating components is transparent to the components in the computation layer that are being coordinated. Coordination can be imposed without changes to these components by observing and coercing their visible behaviour. All that is required is the ability to observe (and intercept) the messages emitted from components. Secondly, coordination will only take place where it is needed. Components are free to interact with each other without the involvement of the coordination layer if the coordination layer hasn't specified that such an involvement is required, by defining suitable traps. The safety and liveness requirements of a system can be met using whatever information is available about the messages sent/expected by a component, it's internal behaviour, protocols etc. This, somewhat pragmatic, approach enables the integration of components implemented in a multitude of languages and running on a multitude of systems.

Finally, we can observe that coordination in our model is performed by ordinary components residing in the computation layer. This is a result of the reflective nature of traps. The only difference is in the *role* played by the components. Their specification, design and implementation can utilise the same tools, paradigms and languages. Hence the means of abstraction and reuse apply to our coordination logic in the same way as they are applicable to the application logic. For instance it doesn't take much effort to abstract generic resource allocation coordination patterns from our Dining Philosopher example. In addition to the obvious software engineering advantages of implementing coordination logic in the same way as application logic, we gain the ability to perform meta coordination, ie. coordinator components themselves can be subject to coordination by other components. Thus, instead of statically categorising components into those dealing with configuration/coordination and those dealing with computation, we have a dynamic relationship between components that is a result of the role they play with respect to each other at a particular point in time. This dynamic categorisation provides the means for implementing systems where coordination is an integral part of the functionality, and hence complex interactions take place between the coordination and computation layers.

Traps can easily be integrated into existing systems by modifying the communication layer. Thus all existing application code remains unaffected and the model functions in a heterogeneous setting, enabling the coordination of existing components across system boundaries. We have successfully implemented trap-based coordination in a heterogeneous setting consisting of a distributed actor-based system and a CORBA system. Traps are distributed and sometimes replicated between nodes in order to improve performance. The model has been successfully used commercially by TECC Ltd in the design and implementation of several middleware applications[TR98]. Our current research concentrates on establishing a formal semantics for traps and their application in the definition of reusable coordination patterns.

5 Acknowledgements

We gratefully acknowledge the advice and help provided by the Distributed Software Engineering Research Section at the Imperial College Dept. of Computing and the members of the Middleware Systems Group at TECC Ltd, as well as the financial support from the EPSRC under grant ref: GR/K73282.

References

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[Ban96] M. Banville. Sonia: an adaption of Linda for coordination of activities in organizations. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[BK96] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[BM90] J.-P. Banatre and D. Le Metayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[BP97] G. Blair and M. Papathomas. The case for reflective middleware. Proc. of the 3rd Cabernet Plenary Workshop `http://www.newcastle.research.ec.org/cabernet/workshops/3rd-plenary.html` , April 1997.

[BS97] G. Blair and J.B. Stefani. *Open Distributed Processing and Multimedia*. Addison Wesley Longman, 1997.

[CA94] C.J. Callsen and G. Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[DNO97] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. LeMetayer, editors, *Coordination Languages and Models, 2nd Int. Conference*. Springer Verlag, September 1997.

[FA93] S. Frolund and G. Agha. A language framework for multi-object coordination. In *ECOOP'93 Proceedings*, volume 707 of *LNCS*. Springer Verlag, 1993.

[Fro96] S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, 1985.

[GK91] J. Des Rivieres G. Kiczales, D. Gureasko. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[GP94] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proc. of the 16th Int. Conf. on Software Engineering*, May 1994.

[GS93] D. Garlan and M Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Co., 1993.

[Hol96] A.A. Holzbacher. A software environment for concurrent coordinated programming. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[MEK95] J. Magee, S. Eisenbach, and J. Kramer. System structuring: A convergence of theory and practice? In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, Proc. of the Dagstuhl Workshop*, volume 938 of *LNCS*. Springer Verlag, 1995.

[MU97] N. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In D. Garlan and D. LeMetayer, editors, *Coordination Languages and Models, 2nd Int. Conference*. Springer Verlag, September 1997.

[MWY91] S. Matsuoka, T. Wanatabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In O. Nierstrasz, editor, *ECOOP'91 Proceedings*, LNCS. Springer-Verlag, 1991.

[MZ95] T. Mowbray and R. Zahavi. *The Essential CORBA: Using Systems Integration, Using Distributed Objects*. John Wiley & Sons, 1995.

[Pop97] A. Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1997.

[Pry98] N. Pryce. A model of interaction in concurrent and distributed systems. In *Second International Workshop on Development and Evolution of Software Architectures for Product Families*, February 1998.

[Pur94] J.M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages*, 16(1):151–174, January 1994.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RE96a] M. Radestock and S. Eisenbach. Agent-based configuration management. In *Proc. of the 7th IFIP/IEEE Int. Workshop on Distributed Systems: Operation and Management*, 1996.

[RE96b] M. Radestock and S. Eisenbach. Formalizing system structure. In *Proc. of the 8th Int. Workshop on Software Specification and Design*, pages 95–104. IEEE Computer Society Press, 1996.

[Sch93] A. Schill, editor. *DCE — The OSF Distributed Computing Environment*. Springer-Verlag, October 1993.

[TR98] F. Taylor and M. Radestock. TECCware product definition. Technical report, Trans Enterprise Computer Communications Ltd, 1998. `http://www.tecc.co.uk`.