Open, Distributed Coordination with Finesse

Andrew Berry, Simon Kaplan
School of Information Technology
The University of Queensland
{andyb,simon}dstc.edu.au

Coodination languages, distributed systems

Abstract

Coordination languages have recently been attracting significant attention as a means of programming parallel and distributed systems. The approach of separating coordination from computation is particularly attractive in distributed systems because there are a wide range of possible interaction, quality and reliability semantics that are either hidden or ignored by traditional infrastructures based on remote procedure call. Introducing an explicit, programmable model for the distributed infrastructure makes these semantics visible and tractable, without requiring substantial changes in distributed components. This paper presents Finesse, a language for describing the interaction of components in open distributed systems, and demonstrates its power through a number of examples.

1 Introduction

Coordination languages have recently been attracting significant attention as a means of programming parallel and distributed systems. The approach of separating coordination from computation is particularly attractive in distributed systems because there are a wide range of possible interaction, quality and reliability semantics that are either hidden or ignored by traditional infrastructures based on remote procedure call (RPC). Introducing an explicit, programmable model for the distributed infrastructure makes these semantics visible and tractable, without requiring substantial changes in distributed components.

At the same time, research and standardisation in the field of Open Distributed Processing[3, 7, 13] has recognised the need to distinguish between the behaviour of distributed components and their interaction. The notions of *binding*[3] and *binding object*[7] are conceptually equivalent to the coordination primitives in languages like Manifold[1] and ConCoord[6].

This paper presents Finesse, a language for describing the interaction of components in open distributed systems. Finesse has evolved out of research in open distributed processing, beginning with the A1$\surd$ architecture model[3] and more recently, a semantic model for describing interactions in open distributed systems[12]. The development of Finesse is being driven by requirements of CSCW systems, whose implementors are demanding users and frequent critics of existing distributed systems infrastructure[8, 4]. The key distinguishing features of Finesse are that:

1. Finesse abstracts over communication, allowing transformation of data and compiler or run-time optimisation of message passing between components.
2. Finesse includes a representation of time, allowing the specification of quality of service properties;
3. Finesse is independent of the language used for programming the distributed components. It is similar in concept to CORBA IDL, for example, where the program is compiled to produce interface *stubs* for components in the chosen language(s);

In this paper, section 2 gives an overview of the underlying semantic model, section three describes the language syntax and informal semantics, and section 4 presents a number of examples. Section 5 discusses related work and ongoing issues, and section 6 concludes the paper.

2 Underlying Semantics

2.1 Fundamental Concepts

Finesse is an executable language for describing complex interaction models and distribution mechanisms. Finesse is used to describe a *binding*, which is an abstract entity that encapsulates the communication between distributed software components participating in an application. Bindings are described in terms of the following fundamental concepts:

binding: a binding is an infrastructure-provided configuration of network connections and behaviour. A binding specification in Finesse describes a configuration of components and their allowed or expected interactions.

role: a binding has a set of roles that can or must be filled by participating components. One or more components can fulfil a single role, providing a convenient abstraction for groups.

interface: components have interfaces through which they interact with their environment. Each interface is connected to one or more roles in the binding and must implement the behaviour specified by the roles it fills.

events: components participate in a binding (interact) by executing events at their interfaces. Events have parameters and direction (in or out).

event relationships: event relationships specify the behaviour and interactions of a binding by describing the relationships between events occurring at object interfaces.

A binding is instantiated by nominating a Finesse program (or some compiled form) and a set of components to fulfil the roles of the binding. The underlying distributed infrastructure is required to establish an appropriate set of network connections and supporting components to implement the Finesse program. A Finesse program can be used to generate stubs for the participating components in a similar manner to CORBA IDL, meaning that Finesse is somewhat independent of the language used to build the participating components.

2.2 Behavioural Model

Event relationships provide the basis for describing behaviour in bindings. Event relationships capture the dependencies between events at the interfaces of software components participating in a distributed application. Three distinct types of event relationship are identified:

Causal relationships which describe the causal dependencies between events;

Parameter relationships which describe the relationships between parameters of causally related events. Parameter relationships define the content of messages passed between interacting components, but in a declarative, application-oriented manner;

Timing relationships which describe any real-time relationships between events. These relationships can be used to describe, for example, timeouts or quality of service requirements of interactions.

These concepts, combined with the notions of *binding interface* and *role*, provide an extremely powerful technique for the description of distributed systems interaction. For example, it is possible to succinctly describe and easily extend remote procedure call, group communication, and stream behaviour. The ability to describe arbitrary interaction models and parameter relationships gives considerable openness and allows the integration of legacy systems. The semantic model described in [12] also includes powerful facilities for abstraction and composition of these behaviours, although only some of those capabilities are visible in Finesse.

3 Finesse Syntax

3.1 Structure of a Finesse Program

A Finesse program, also called a *binding* has an outer scope introduced by the keyword **Binding** and the name of the binding, followed by a set of **Import** statements, and two sections defining roles and interactions. **Roles** define the required behaviour of participating components, and **Interactions** define the relationship between events at different roles. Braces are used to delimit sections. Note that in the following examples, ellipses (...) are used to avoid including unnecessary detail and are not a syntactic construct. The basic structure is thus:

```
Binding Example {
  Import ...;
  Roles {
    ...
  }
  Interactions {
    ...
  }
}
```

3.2 Describing Roles and Interactions

A binding has one or more role definitions, introduced by a role name. A role definition can be prefixed by a cardinality constraint enclosed in square braces, which constrains the number of components that can fulfil a role. The place-holder `#` represents the actual cardinality. Where no cardinality constraint is given, the default cardinality is exactly one, for example:

```
  Roles {
    Client { ... }
    [#>=1] Server { ... }
  }
```

This specifies that there are two roles, *Client* and *Server* and that there is exactly one *Client* and at least one *Server* in the binding.

The **Interactions** specification defines relationships between events occurring in the roles. Events are referred to by the role name, followed by a period '.' and the event name. This reference to an event can also have a cardinality constraint to deal with situations where multiple components fill the role. For example:

```
Binding Example {
  Import ...;
  Roles {
    Client { send! }
    [#>=1] Server { receive? }
  }
  Interactions {
    Client.send -> [#=all] Server.receive
  }
}
```

The place-holder `#` in the **Interactions** specification refers to the number of components executing the event, while the place-holder `all` refers to the number of components fulfilling the role. In the above example, the client role executes a `send` event followed by all servers executing the `receive` event. In other words, this binding is a high-level description of reliable multicast. As with role cardinality, event execution cardinality defaults to exactly one.

Roles can contain named actions that group together a set of events and allow the **Interactions** section to refer to some subset of the role when defining interaction behaviour, for example:

```
Binding {
  Roles {
    Client {
```

```
      read { send -> receive } -> write { ... }
    }
    ...
  }
  Interactions {
    Client.read ...
  }
}
```

Named actions define a scope for event names, allowing the role and interaction definitions to distinguish between same-named events. Interactions can also contain named actions to support inheritance and overriding.

3.3 Events and Event Relationships

The behaviour within roles is defined by events and their relationships. An event is introduced by a name, a direction indicator, and a parameter list, for example:

```
e!(x:t1; y:t2)
```

where `e` is the event name, `!` indicates that it is an output event, `x,y` are the event parameters, and `t1,t2` are the data types of the parameters. Events are uni-directional, that is, they can be input events or output events but not both. The `?` character is used in place of the `!` to indicate an input event.

Event relationships are used to define causality, parameter and timing relationships between events in the role. Causality defines a partial order and is specified with the `->` operator, for example:

```
e1!(x:t1) -> e2?(y:t2)
```

This specifies that event `e1` must complete before event `e2` begins. The `->` operator is transitive and antisymmetric. Events not related by the causal order an occur in any order and may even overlap in time if that is physically possible.

An event can be followed by a specification of its parameter relationships. The specification places constraints on the values of the parameters. For example:

```
e1!(x:t1; y:t2) -> e2?(z:t3) {z = f(e1.x)}
```

Parameter relationship specifications can refer to any identifiable, causally preceding event. There is no requirement that all parameters of any output event must be consumed by an input event, and the parameters of an output event can be used many times. In the general case illustrated here, parameter relationships are functional, allowing for transformation of data. For all parameter relationships, the function or operator used must be well-defined for the data types of the parameters. This means, for example, that equality (`=`) can be used for parameters of different types provided it is well defined in the context of the binding. Due to its common use in RPC systems, Finesse has shorthand syntax for name equivalence of parameters, that is:

```
e1!(x:t1; y:t2) -> e2?(x:t1; y:t2) {*= e1}
```

This specifies that all parameters of `e2` are assigned the value of the same-named parameter of `e1`. Non-deterministic parameter relationships can also be specified, for example:

```
e1!(i:seqnr) -> e2!(i:seqnr) {e2.i > e1.i}
```

In order to simplify event identification, the keyword `prev` can be used to refer to the immediately preceding event in the current specification context.

Events can have guards. Guards are logical expressions that must evaluate to true for the event to occur. Timing constraints are included in Finesse programs through guards and the provision of three built-in functions: **start**, **end** and **now**. **Start** and **end** take an event name as a parameter and return the time when that event started or ended respectively. **Now** returns the current time. For example:

```
e1!() -> [now - end(e1) < 10.0] e2?()
```

This specifies that the event `e2` must start within 10 seconds of `e1` completing. Absolute time is extremely difficult to represent and measure accurately in distributed systems, so guards involving time are only permitted to compare time deltas. Literal values of time are represented as a real number indicating a number of seconds. Implementations of Finesse must allow for clock skew when evaluating time guards involving events at different locations.

3.4 Composing Event Relationships

Finesse has three primary composition operators for joining event relationships:

AND is a logical AND of two specifications, synchronising on same-named events and actions;

OR is a logical OR of two specifications, synchronising on same-named events and actions;

XOR is a logical exclusive OR of two specifications.

Synchronisation of events and actions means that they become the same occurrence of the event or action, and implies that their parameters and any ordering must be identical.

3.5 Inheritance and Subtyping

Finesse supports inheritance as a means of code reuse, and explicit specification of subtype relationships with the **implements** keyword. For example:

```
Binding Example inherits ExampleParent
                implements ExampleBehav {...}
```

The **inherits** keyword instructs Finesse to include the imports, roles, and interactions of the parent in the child. Roles and named actions defined in the child override same-named roles and actions in the parent. The remaining interaction behaviour is composed with a logical AND.

The **implements** keyword is intended to allow specific implementations of a high-level behaviour, for example, both CORBA and DCE implement remote procedure call (RPC) that would be semantically equivalent for many applications. A high-level Finesse program for RPC could potentially be replaced with either implementation. Although a complete complexity analysis has not yet been attempted, it is believed infeasible to automatically check subtyping specified with the **implements** keyword. Simple checks for compatibility will be able to detect certain types of incompatibilities, however, final responsibility for the correctness of the subtyping lies with the programmer.

3.6 Reuse and Generics

The **Import** keyword allows role and binding definitions to be re-used in the current Finesse program. It is followed by the name of a Finesse program to import. In the simple case, role and binding definitions are re-used without parameterisation, for example:

```
Binding Message {
  Roles {
    Sender {send!(x:t1)}
    Receiver {receive?(x:t1)}
  }
  Interactions {
    Sender.send -> Receiver.receive {*=Sender.send}
  }
}

Binding UseMessage {
  Import Message;
  Roles {
    Send2 {send1 {Sender} -> send2 {Sender}}
    Recv2 {recv1 {Receiver} -> recv2 {Receiver}}
  }
  Interactions {
    Message(send1, recv1) AND
    Message(send2, recv2)
  }
}
```

The roles of the *Message* binding are used to define two actions each in the Send2 and Recv2 roles respectively. The interactions section of the *UseMessage* binding simply binds those actions together using the *Message* binding. While this can be useful, the ability to parameterise roles with arbitrary parameter lists give more flexibility, for example:

```
Binding Message {
  Roles {
    Sender (MSG) {send!(MSG)}
    Receiver (MSG) {receive?(MSG)}
  }
  Interactions {
    Sender.send -> Receiver.receive {*=Sender.send}
  }
}

Binding UseMessage {
  Import Message;
  Roles {
    Send2 {send1 {Sender(x:t1)} ->
             send2 {Sender(y:t2)} }
    Recv2 {recv1 {Receiver(x:t1)} ->
             recv2 {Receiver(y:t2)} }
  }
  Interactions {
    Message(send1, recv1) AND
    Message(send2, recv2)
  }
}
```

This allows us to reuse the interaction behaviour with different event parameter lists, allowing definition of bindings such as generic RPC or multicast.

3.7 Iteration

Iteration in the presence of concurrency requires two separate iteration semantics; one for dependent (sequential) iteration and one for independent (parallel) iteration. In Finesse, both of these take the form of a postfix operator on an action or event. The `*+` operator indicates that the action or event should be repeated with a causal dependency on previous executions. The `*-` operator indicates that the action of event should be repeated with no dependency on previous executions. For example:

```
Binding Example {
  Roles {
    Consumer { consume?(x:t1) *+ }
    Producer { produce!(x:t1) *- }
  }
  Interactions {
    {Producer.produce -> Consumer.consume} *-
  }
}
```

This specifies a binding containing producer and consumer roles. The consumer can only consume one data item at a time, while the producer can produce many data items in parallel, and each produce event results in a corresponding consume event.

4 Example Programs

The following example programs, while not exercising all features of Finesse, introduce the language and demonstrate its strengths. The first four examples illustrate how Finesse succinctly handles the transition from two-party to multi-party interaction, and the final example shows the use of time constraints with stream behaviour.

4.1 Generic RPC

This Finesse program describes a generic RPC interaction with two roles, client and server. The *Roles* section defines the behaviour of the participants. The *Interactions* section defines the relationship between the roles. A set of required messages and hence appropriate network connections can be derived from the behaviour.

```
Binding RPC {
  -- generic RPC

  Roles {
    -- the client role is parameterised by a set
    -- of input and output values
    Client(IN, OUT) {
       -- the client executes a send (output)
       -- followed by a receive (input)
       send!(IN) -> receive?(OUT)
    }

    -- the server role is similarly parameterised
    Server(IN, OUT) {
       -- the server receives then sends
       receive?(IN) -> send!(OUT)
    }
  }

  Interactions {
     -- the client send causes the server to
     -- receive, with parameters matched by name
     Client.send -> Server.receive {*= prev} AND
```

```
    -- the server send causes the client to
    -- receive, with parameters matched by name
    Server.send -> Client.receive {*= prev}
  }
}
```

4.2 Using RPC

Use of the generic RPC binding is demonstrated in the following binding definition for file input/output:

```
Binding FileIO {
  -- read-only file access using RPC

  Import RPC;

  Roles {
    -- Client and Server implement open/read/close
    Client {
       open {RPC.Client ((string name),
                         (handle fh))}
       -> read {RPC.Client
                   ((handle fh, int bytes),
                    (buffer buf, int bytes))} *+
       -> close {send!(handle fh)}
    }
    Server {
       open {RPC.Server ((string name),
                         (handle fh))}
       -> read {RPC.Server
                    ((handle fh, int bytes),
                     (buffer buf, int bytes))} *+
       -> close { receive?(handle fh) }
    }
  }

  Interactions {
    -- Client operations result in corresponding
    -- server operations.
    RPC(Client.open, Server.open) ->
    RPC(Client.read, Server.read) *+ ->
    Client.close -> Server.close {*= prev}
  }
}
```

4.3 Multicast RPC

The original RPC binding can be extended to support multicast RPC. The client and server roles are unmodified, allowing the original client and server to be used:

```
Binding MultiRPC {

  Import RPC;

  Roles {
    Client { RPC.Client }
    -- the cardinality constraint specifies that
    -- there must be at least one server.
    [#>=1] Server { RPC.Server }
  }

  Interactions {
    -- a client send causes all servers to receive
    Client.send -> [#=all] Server.receive {*= prev}
    -- however, only one of the responses causes a
    -- result to be delivered to the client.
    [#=1] Server.send -> Client.receive {*= prev}
  }
}
```

This example introduces cardinality constraints associated with roles and their behaviour. All roles in a binding can potentially be filled by many participating objects. By default, a role is filled by only one participant. The addition of an appropriate cardinality constraint allows a role to be filled by multiple participants. This use of cardinality constraints provides a convenient and powerful mechanism for describing group communication.

4.4 Using Multicast RPC

A replicated file access binding shows how the multicast RPC binding can be used:

```
Binding ReplFileIO {
  -- replicated, read-only file access

  Import MultiRPC, FileIO;

  Roles {
    -- Client and Servers reuse open/read/close.
    -- Only Server cardinality has changed.
    Client { FileIO.Client }
    [#>=1] Server { FileIO.Server }
  }

  Interactions {
    -- RPCs by client are multicast to servers
    MultiRPC(Client.open, Server.open) ->
    MultiRPC(Client.read, Server.read) *+ ->
    Client.close -> [#=all] Server.close {*= prev}
  }
}
```

This set of examples demonstrates how a basic interaction mechanism can be extended to suit new requirements. Notice in particular, that clients and servers are unchanged despite the change in interaction mechanism. This suggests significant potential for reuse and legacy application integration.

4.5 Stream Communication

The following generic stream binding demonstrates how Finesse can be used to describe quality of service requirements, including time-related constraints. While this binding describes only two-party interaction, it can be extended for multi-party stream interaction in manner similar to the multi-party RPC.

```
Binding Stream {

  Roles {
    Producer(DATA) {
       -- sending with sequence number generation
       (send!(seqnr, DATA) ->
         [seqnr=prev.seqnr+1] send!(seqnr, DATA))*+
    }
```

```
    Consumer(DATA) {
       -- receiving with correct ordering but
       -- allowing for loss of a packet of data
       -- between successful transmissions and
       -- requiring a minimum frame rate of 1
       -- frame/sec
       {receive?(seqnr, DATA) ->
         [seqnr - prev.seqnr < 2;
          seqnr > prev.seqnr;
          now - end(prev) < 1.0]
         receive?(seqnr, DATA)} *+
    }
  }

  Interactions {
     -- basic streaming transmission behaviour.
     -- Note that producing the next element of the
     -- stream is not dependent on the receipt of
     -- the previous element, hence the '*-'.  Also
     -- note that not all produced events must be
     -- received, allowing lossy behaviour.
     {Producer.send -> Consumer.receive {*= prev}
      XOR Producer.send} *-
  }
}
```

5 Discussion

5.1 Novel Features

Finesse has a number of features that are novel in coordination languages. Of particular interest is the abstraction that it provides over messaging. Messaging is implied by declarative relationships between events, meaning that a compiler or interpreter can optimise the number and content of messages transferred between components. The use of explicit, but abstract, parameter relationships allow parameters to be ignored if not used. The use of causality relationships allows parameters from multiple events to be combined into a single message from a particular interface where appropriate.

Openness and flexibility is enhanced by allowing arbitrary parameter relationships. This can allow, for example, a DCE RPC client to call a CORBA server, provided the appropriate infrastructure and transformation functions are in place. The Finesse language has no structural knowledge of data types, freeing it from the confines of a specific data model. The use of functional relationships between parameters also provides good support for including legacy components and applications in a Finesse binding.

The inclusion of time constraints is both novel and very useful. Such constraints can be used to explicitly specify timeouts and associated behaviour, or to describe quality of service constraints on, for example, the delivery of multimedia streams.

5.2 Open Issues

There are a number of open issues associated with the syntax and semantics of Finesse. These are summarised as follows:

- Finesse has a terse syntax that is not especially friendly for first-time programmers. This keeps the language and its programs small, but it might be appropriate to introduce more familiar syntax, particularly for iteration and control structures, for example, **if-then-else** or **while** constructs. Such changes are being considered for future versions of the language, although the presence of concurrency complicates the semantics.
- It is often most appropriate to represent concurrency graphically to make non-linear (i.e. split and join) dependencies clearer. There is also some merit in providing a graphical "plumbing toolkit" containing commonly used generic bindings like streams and RPC. A graphical programming environment based on these ideas would allay concerns about the terseness of Finesse, allowing programmers to use this environment for most programs. There is an ongoing interest in providing such tools for Finesse.
- Finesse does not currently allow predefined event or action types. These could for example, be used in a macro fashion to define commonly used event *signatures* or patterns of behaviour. A similar effect can be achieved by importing bindings defining roles with appropriate event and action definitions, but future versions of Finesse might include explicit support for event and action types.
- Finesse does not support a structured data model. For openness and flexibility, it leaves the management of data and types to the connected components. In order to support rudimentary type checking and transformation of data, parameters are associated with a type name. Functional transformations of data must be supplied by the infrastructure and may or may not be dependent on type names. Such an unstructured approach to data typing is unpopular in some circles, and experience with Finesse might suggest a more structured approach in future versions.
- One of the difficulties associated with roles is describing behaviour for roles with a cardinality greater than one. For example, how do you describe a multicast RPC that chooses the response with the highest version number? Finesse cannot deal with such situations at present, so the component must implement these semantics. The most likely approach is to allow parameter relationships to operate on event *sets* as well as individual events.
- When composing specifications from different sources, there are sometimes situations where you want to *synchronise* on events having different names. This is not currently possible with Finesse. Two possibilities are being considered to address this issue: either renaming or a set of explicit synchronisation primitives.
- Finesse currently does not support reflection or evolution of behaviour. While theoretically possible, implementation is difficult because of the need to implement dynamically specified parameter relationships and the subsequent need to store all causally accessible parameters at all times.

A formal semantics for Finesse is partially complete, and is being used to assist in developing a prototype implementation. The prototype is being implemented over Hector[2], a distributed systems infrastructure based on the principles of the A1$\surd$ model[3]. Hector supports the majority of the Finesse semantics, including multi-party bindings, flexible interaction mechanisms, and an open data model. Compilation of Finesse programs requires the translation of Finesse code into program stubs and an executable Hector binding description.

5.3 Related Work

Coordination languages are quite varied in their strengths and features. The earliest attempts at distinguishing coordination from computation were based on Linda[5] and a number of variants are still in active use, indicating the power of the shared tuple-space approach. These systems have the advantage of a simple, yet powerful model of communication. Finesse lacks this simplicity but has a number of advantages, in particular the ability to abstract over communication in a way that can be optimised, and the ability to *capture* coordination protocols and build increasingly high-level abstractions of that coordination.

More recently, a number of coordination languages have been based on the idea of building a network of connections between ports and/or interfaces, as is done by ConCoord[6] and Manifold[1]. ConCoord in particular has powerful abstraction capabilities and language independence. The primary difference between these languages and Finesse is that Finesse does not use explicit connections between interfaces, with the causal and parameter relationships allowing the optimisation of messaging and message contents. This does, however, introduce additional complexity that might not be desirable. Neither the connection based languages or the shared tuple-space languages support real-time constraints to the extent supported by Finesse.

Darwin[11] is a language for describing the static, structural connections of a set of components without explicitly describing their behaviour. Coupled with a distinct component specification language, for example that used in [9], it can be used to describe similar behaviour to Finesse. It does not distinguish between components that perform computation and components that connect other components (i.e. bindings). In many systems this is quite appropriate, however, the failure and execution model of software components running on a single computer system is quite different from that of the network connections supporting the interconnection of those components, suggesting a distinct language like Finesse is preferable in open distributed systems.

Finesse is most similar to Rapide[10], an architecture description language based on *posets* (partially ordered sets of events). Rapide is intended as a simulation language for software engineering. It is event-based, with a true concurrency model based on causality, and uses event patterns for abstraction and synchronisation. Rapide also has extensive support for real-time constraints. Finesse differs most from Rapide in the way abstraction is handled, and in its data model, since Rapide has a fixed, structured data model.

6 Conclusion

This paper has described Finesse, a coordination language for open distributed systems. A Finesse program or binding describes the roles of components in a distributed application and the interactions between those roles. Roles and interaction are described using event relationships, in particular, causality (ordering), parameter and timing relationships. Finesse has strong support for group communication and provides abstraction through structuring and composition features. The examples presented in this paper suggest that Finesse can succinctly describe a wide variety of coordination protocols in a flexible and easily reusable manner.

The key advantages of Finesse over existing coordination languages are: its abstraction of communication; its open, unstructured data model; and its support for real-time constraints. It does suffer to some extent from the complexity and unfamiliarity of these features, but offers a powerful, alternative approach to the problem of coordination in open distributed systems.

References

[1] F. Arbab. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science. Springer, 1996.

[2] D. Arnold, A. Bond, M. Chilvers, and R. Taylor. Hector: Distributed objects in python. In *Proceedings of the 4th International Python Conference*, Livermore, California, June 1996.

[3] A. Berry and K. Raymond. The A1$\surd$ architecture model. In *Open Distributed Processing: Experiences with distributed environments*. IFIP, Chapman and Hall, February 1995.

[4] G. Blair and T. Rodden. The challenges of CSCW for Open Distributed Processing. In *Open Distributed Processing, II*. IFIP, North Holland, 1993.

[5] N. Carriero and G. Gelernter. Linda in context. *Communications of the ACM*, 32(4):128–139, April 1989.

[6] A. A. Holzbacher. A software environment for concurrent coordinated programming. In *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*. Springer, 1996.

[7] ISO/IEC 10746-1 10756-2 10746-3 10746-4 Basic Reference Model for Open Distributed Processing.

[8] Simon Kaplan, Geraldine Fitzpatrick, Tim Mansfield, and William J. Tolone. MUDdling through. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Information Systems—Collaboration Systems and Technology*, 1997.

[9] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Coordination Languages and Models*, number 1282 in Lecture Notes in Computer Science, pages 18–31. Springer, September 1997.

[10] D. C. Luckham and J. Vera. An event based architecture definition language. *IEEE Transactions on Software Engineering*, September 1995.

[11] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, September, 1995.

[12] A. Rakotonirainy, A. Berry, S. Crawley, and Z. Milosevic. Describing open distributed systems: A foundation. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Software Technology and Architecture*, 1997.

[13] K. Raymond. Reference Model of Open Distributed Processing (RM-ODP): Introduction. In *Open Distributed Processing: Experiences with distributed environments*. IFIP, Chapman and Hall, February 1995.