Language Support for Distribution in CSCW Systems

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

1 Introduction

Systems for Computer Supported Cooperative Work (CSCW) are demanding in their use of distributed systems infrastructure. CSCW systems typically require facilities such as group communication (multicast), object replication, and streamed, multi-party audio or video. The interaction mechanisms supported by mainstream tools like CORBA[19] and DCE[24] are low-level and limited to message passing or remote procedure call (RPC). While this is sufficient for information processing applications with simple client-server or three-tier communication requirements, it falls short of supporting the flexibility and complexity required by CSCW systems[5, 22, 14].

There are a number of research-oriented toolkits, for example GroupKit[23] and COAST[25], that support some of the required facilities for CSCW applications. These do not easily support extension or composition of facilities, however, because of their "toolkit" nature. In his recent PhD thesis[8], Paul Dourish suggests that correct approach to providing the necessary flexibility is through *open implementation*[15]. This technique requires an infrastructure to provide a representation of the underlying implementation that can be modified to suit application requirements, typically through reflection[16] in a programming language.

With the suggestions of Dourish in mind, we are developing an executable language for describing the distribution aspects of CSCW systems. The language, *Finesse*, is based on the principles of the A1$\surd$ architecture model for distributed systems[4]. Its underlying semantics are described informally in [20], and a draft syntax has been defined[2].

This paper gives an overview of Finesse, its motivation, and its goals. A set of simple examples is provided in appendix A to give the flavour of the language and demonstrate the power of a language-based approach.

2 Introducing Finesse

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 objects 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 describes a configuration of objects and their allowed or expected interactions.

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

interface: objects 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: objects 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 objects to fulfill the roles of the binding. The underlying distributed infrastructure is required to establish an appropriate set of network connections and supporting objects to implement the Finesse program. A Finesse program can be used to generate stubs for the participating objects in a similar manner to CORBA IDL, meaning that Finesse is somewhat independent of the language used to build the participating objects.

2.1 Behavioural Model

Event relationships provide the basis for describing behaviour in bindings. Event relationships capture the depen-

dencies between events at the interfaces of objects 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 objects, but in a declarative, application-oriented manner;

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

A draft formal specification of these concepts and their interrelationships is given in [3]. These concepts, combined with the notions of *binding*, *object*, *interface* and *role* from the A1$\surd$ model, 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 model described in [20] also includes powerful facilities for abstraction and composition of these behaviours, although only some of those capabilities are visible in Finesse.

2.2 Language Definition and Implementation

The behavioural model described in the previous subsection has been used as the basis for the draft definition of the language syntax[2]. The language has many similarities with process algebras like CSP[12] and CCS[18], but has partial ordering semantics (i.e. true concurrency) and more powerful structuring and abstraction capabilities. The informal syntax and semantics are the basis for ongoing work that will formalise and implement the language. Presently, a formal specification of the language is being developed to assist in analysis and implementation. An implementation of the language over the distributed infrastructure Hector[1] is planned.

A set of example "programs" are provided in appendix A. These are minimal examples, but serve to present the flavour of the language and demonstrate the ease with which simple interactions can be extended to suit application requirements.

3 Influences and Related Work

3.1 CSCW

The work has used examples from the CSCW research community for motivation and requirements, in particular the work of the wOrlds CSCW research group[14] at the CRC for Distributed Systems Technology. The following key requirements have been identified:

- It should be possible to incorporate existing application components and interaction mechanisms in a cohesive and flexible manner;
- The infrastructure should support dynamically configurable replication with a variety of mechanisms to deal with varying bandwidth, latency and coordination requirements in Internet-scale networks;
- It should be possible for users to dynamically vary their participation in and awareness of collaborative activities. This is necessary to support the ever-changing work focus of individuals and the varying quality of service provided by the network.

Finesse is similar in some respects to a number of existing languages for describing coordination in CSCW systems, including DWCPL[7] and Trellis[10]. These languages focus, however, on synchronous groupware and execute in a closed environment. Finesse is a more general language with applications in traditional distributed systems as well as CSCW.

3.2 Architecture and Coordination Languages

Finesse is also influenced by recent work in architecture description languages. Research into software architecture [26, 27] is strongly supporting a model of programming that distinguishes software components and their *connectors*. This model promotes reuse and reduces the coupling of software components, and a number of *architecture description languages* have been developed, for example Wright[11] and Rapide[17]. The primary difference between Finesse and these languages is that architecture description languages, with the exception of Rapide, are not executable. They provide a framework for describing interaction in an abstract manner, but no means to automatically realise that abstraction. They also tend not to deal with data type issues. Rapide[17] overcomes some of these deficiencies, but is strongly oriented towards simulation rather than programming and has a closed data model.

Recent efforts in developing coordination languages and models[6] for distributed systems have focused on the need

to distinguish components and their coupling, and incorporate strong abstraction capabilities in languages for programming distributed systems. These same principles are used in Finesse, although coordination languages tend to be closed-language environments, requiring all components and their coordination to be managed by a controlled environment. Most have minimal support for non-procedural interactions (for example, streams), and in some cases require tight coupling between software components.

3.3 Open Distributed Processing

The A1$\surd$ model upon which Finesse is based has a strong relationship with the ISO Basic Reference Model of Open Distributed Processing[13, 21]. The CRC for Distributed Systems Technology was actively involved in the ISO standardisation process, with the A1$\surd$ model used as a vehicle for their participation. The notions of binding and interface are strongly related to similar ODP concepts, although the A1$\surd$ model does not use the ODP *viewpoints*. Finesse benefits from the open systems approach, not prescribing specific data models or infrastructure.

3.4 Example Problems

To validate the work, a number of more complex application examples from distributed systems and CSCW will be written and tested using the implementation of Finesse. The following examples are planned:

Replication: one of the key requirements of CSCW systems that operate over Internet-scale networks is flexible replication. This example will provide a relatively abstract replication binding and a number of compatible refinements with different properties to suit the requirements of a particular work context. Applications written using the abstract replication binding can be instantiated with any of the compatible refinements, depending on the current requirements.

Conferencing: a binding or set of bindings for managing audio and video conferencing sessions will be produced.

Locales: the wOrlds research group is implementing a distributed framework for supporting the concept of a *locale*, which is a virtual *place* containing tools and artifacts for collaborative work[9]. A set of bindings to implement a version of the locales framework will be written in Finesse. These bindings will use the preceding examples as components.

Other examples might be defined if required. The relative success of these examples will be used to gauge the effectiveness and applicability of the Finesse. The measure of success will be based on the following criteria:

1. the ability to describe each example using Finesse (i.e. can it be done in Finesse?)
2. the complexity of the examples in Finesse compared with traditional approaches;
3. the ease with which applications using Finesse can be changed, reconfigured or extended;
4. feedback from developers involved in the creation and use of the examples.

4 Summary

This research is addressing the problem of supporting CSCW over a distributed systems infrastructure. Based on the results of research to date, a language-based approach is being pursued. A semantic model for languages that describe interactions in distributed systems has been developed, and the definition of a particular language, Finesse, is partially completed.

The remainder of this research will focus on refinement of Finesse through formalisation and validation through examples. An implementation of the language is planned, and this will be used to implement and test the examples.

References

[1] 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.

[2] A. Berry. Finesse: An event-based binding language. http://www.dstc.edu.au/AU/staff/andrew-berry/phd/syntax.html.

[3] A. Berry. A Z specification for event relationships. http://www.dstc.edu.au/AU/staff/andrew-berry/phd/spec.html.

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

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

[6] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*. Springer, 1996.

[7] M. Cortes and P. Mishra. DWCPL: A programming language for describing collaboration. In *ACM 1996 Conference on Computer Supported Cooperative Work*, Nov. 1996.

[8] P. Dourish. *Open Implementation and Flexibility in CSCW Toolkits*. PhD thesis, Department of Computer Science, University College London, 1996.

[9] G. Fitzpatrick, W. J. Tolone, and S. M. Kaplan. Work, locales and distributed social wOrlds. In *Proc. of the 4th European Conference on CSCW*. Kluwer Academic Publishers, 1995.

[10] R. Furuta and P. D. Stotts. Interpreted collaboration protocols and their use in groupware prototyping. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*, Oct. 1994.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design issues. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*. ACM Press, Dec. 1994.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

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

[14] S. Kaplan, G. Fitzpatrick, T. Mansfield, and W. J. Tolone. MUDdling through. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Information Systems—Collaboration Systems and Technology*, 1997.

[15] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, pages 8–11, Jan. 1996.

[16] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

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

[18] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[19] The common object request broker: Architecture and specification. The Object Management Group, 1995. Revision 2.0.

[20] 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.

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

[22] T. Rodden, J. A. Mariani, and G. Blair. Supporting cooperative applications. *Computer Supported Cooperative Work*, 1:41–67, 1992.

[23] M. Roseman and S. Greenberg. GroupKit: a groupware toolkit for building real-time conferencing application. In *Proc. 4rd Int. Conf. on CSCW*. ACM Press, Nov. 1992.

[24] W. Rosenbury, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly and Associates, Inc., September 1992.

[25] C. Schuckmann, L. Kirchner, J. Schummer, and J. M. Haake. Designing object-oriented synchronous groupware with COAST. In *ACM 1996 Conference on Computer Supported Cooperative Work*, Nov. 1996.

[26] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical Report CMU-CS-94-107, Software Engineering Institute, Carnegie Mellon Univerity, Jan. 1994.

[27] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice Hall, 1996.

A Example: RPC to Multicast RPC

The following examples demonstrate the basic features and structuring of Finesse. We use the language to define RPC interaction, then extend RPC to implement multicast RPC with minimal changes. Other examples, including an example of multi-party audio and video conferencing, have been developed using a similar technique, but are not included here due to length constraints.

A.1 Parameterisable RPC

The following binding describes a parameterisable RPC interaction with two roles, client and server. The *Roles* section defines the behaviour of the participants. The *Behaviour* 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 {
  -- simple, parameterisable 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 executes a receive followed by a send
       receive?(IN) -> send!(OUT);
    }
  }

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

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

The following syntactic elements are used:

- $Client(IN, OUT)$ introduces the client role, parameterised by a set of sent values sent (IN) and a set of received values (OUT).
- $send!(IN)$ indicates an event where the client outputs the IN values
- $receive?(OUT)$ indicates an event where the client accepts the OUT values
- $->$ indicates a causal relationships between events, that is $A -> B$ specifies that A affects B hence must occur before B.
- $Client.send$ refers to the execution of the client send event.

- $* = prev$ indicates that the parameters of the current event should be set equal to parameters having the same name in the previous event (i.e. name equivalence).

A.2 Example: File Access using RPC

Use of this parameterisable RPC binding is demonstrated in the following binding definition for file I/O:

```
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); }

       open -> read *+ -> close;
    }
    Server {
       open  { RPC.Server ((string name), (handle fh)); }
       read  { RPC.Server ((handle fh, int bytes), (buffer buf, int bytes)); }
       close { receive?(handle fh); }

       open -> read *+ -> close;
    }
  }

  Behaviour {
    -- Client operations result in corresponding server operations.
    -- Operations are performed sequentially.
    RPC(Client.open, Server.open) ->
    RPC(Client.read, Server.read) *+ ->
    Client.close -> Server.close {*= prev};
  }
}
```

The $*+$ operator indicates sequential, causally dependent repetition of the preceding event. A similar operator, $*-$, is used for parallel, independent repetition.

A.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; }
  }

  Behaviour {
     -- 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.

A.4 Replicated File Access

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 implement open/read/close operations, as
    -- before.  Only Server cardinality has changed.
    Client { FileIO.Client }
    [#>=1] Server { FileIO.Server }
  }

  Behaviour {
    -- 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.