Describing Open Distributed Systems: A Foundation

Andry Rakotonirainy†, Andrew Berry‡, Stephen Crawley§, Zoran Milosevic†
†CRC for Distributed Systems Technology
‡Department of Computer Science
The University of Queensland, Qld 4072, Australia
§Defence Science and Technology Organisation,
PO Box 1500, Salisbury, SA 5108, Australia
{andry,andyb,zoran}@dstc.edu.au, stephen.crawley@dsto.defence.gov.au

Abstract

In this paper we outline a semantic model for open distributed systems which provides a foundation for a corresponding architecture description language. This semantic model is based on architecture models reported in [2][5], with a number of refinements to support abstraction and composition. The model is specifically designed to describe open distributed systems independent of implementation details such as communication protocols and middleware systems. The modelling concepts in the semantic model are: object (a model of an entity), event (a unit of interaction between an object and its environment), event relationship (a specification of behaviour defining the relationships amongst a set of events), interface (an abstraction of an object's interaction with its environment) and binding (a context for interaction between objects). The binding concept is particularly important because it can describe any kind of interaction in an open distributed system, ranging from remote procedure calls and multicast to more complex, enterprise interactions.

Special attention is given to the problem of composition and abstraction of events and behaviour in the model. This is needed to reflect the reuse, evolution and interworking requirements of open distributed systems. Our approach allows for the effective modelling of asynchrony, concurrency and complex flows of information in open distributed systems.

1 Introduction

The evolution of middleware products over recent years and the new capabilities of emerging technologies (e.g. multimedia, WWW and Java) suggest several key issues that will contribute to the success of open distributed systems. First, an open distributed system architecture model should be powerful enough to model both current and future technologies, in particular the behaviour of objects, their complex interactions and different abstraction and composition mechanisms. Second, the precise semantics of the architectural concepts should be reflected in a corresponding architecture description language.

In this paper we describe a semantic model that provides a foundation for architecture description languages in open distributed systems. The semantic model includes powerful composition and abstraction mechanisms, and is intended to be independent of detail such as communications media, network protocols, middleware systems and implementation paradigms.

Our semantic model is aimed at formalising and refining the concepts previously defined in the DSTC Architecture Model [2]. The DSTC Architecture Model defines a set of high-level concepts that encapsulate the requirements of open distributed systems, while maintaining a correspondence with lower-level infrastructure. This architecture model was developed both to serve the needs of a diverse organisation engaged in distributed systems research, and to fuel participation in the ISO ODP[5] standardisation process. The semantic model described in this paper is the next step towards an architecture description language for open distributed systems, i.e. architecture model → semantic model → language.

Such a language will allow the designer to specify the architecture of a distributed system using design abstraction and refinement techniques, and lead to support for truly composable distributed systems. The ultimate goal is a practical language with a sound formal basis, and tools that support rapid prototyping and software engineering activities associated with open distributed systems.

1.1 Describing object interactions

The *event relationship* is the central concept in our model for the description of behaviour (the notion of behaviour in our model corresponds to a 'real world' concept—it should not be equated with its use in process alge-

bras). A complete specification of an event relationship consists of a set of events, an ordering of the events, the relationships of their parameters, and the events' timing constraints.

An interaction implies event ordering and we introduce a rich set of operators to construct different kinds of *event orderings* as they are required for open distributed systems.

The causal ordering of events is necessary but not sufficient to describe all properties of events in open distributed systems. Distributed systems essentially provide a conduit for the flow of information between physically distributed objects. This flow of information is supported in our semantic model by defining of arbitrary relationships between parameters of events. This gives the semantic model considerable expressiveness and enables the specification of many useful transformations. For example, it is possible to describe the correspondence between events occurring in different middleware platforms (e.g. CORBA and DCE), or the events associated with legacy and emerging platforms. This novel aspect of our work is of particular importance for open distributed systems.

We recognise that, in spite of the inherent difficulty of representing time in distributed systems, time constraints are important for many applications. We therefore provide facilities for the definition of *temporal relationships*.

Another novel aspect of our approach is enabling specification of the relationships between events in different *abstractions*. We recognise that it is useful to describe architectures in different abstractions, as opposed to the relevant formal description techniques [9][4][3], which focus on the specification of behaviour in a single abstraction and assume the atomicity of events. We allow an event in one abstraction to be a composition of events in another abstraction, and define an abstraction semantic with its associated consistency constraints. The support for abstraction means that, in general, events are non-atomic, and we introduce operators to deal with this added complexity.

Since the effective reuse of objects can involve various forms of *composition* mechanisms, we introduce a rich set of operators to support composition. The introduction of the operators for describing composition and abstraction further enhance the expressiveness and flexibility of the semantic model.

1.2 Interfaces and binding

In addition to the interactions, event relationships are used to describe the semantics of basic modelling concepts in our model, the interfaces and bindings. An *interface* is a particular abstraction of an object's behaviour, and is described by an event relationship for the events that may occur at the interface. A *binding* is introduced to specify interactions between events occurring at the interfaces of different objects. Interfaces and bindings provide structuring and encapsulation, and define an environment for execution of behaviour described by event relationships.

1.3 Asynchrony

The goal of our semantic model is to enable concise and flexible specification of distributed object interaction. Systems built through interconnection of distributed objects can only be asynchronous, since there is always a measurable time between the occurrence of an event at one location and the occurrence of a causally related event at a different location. To make our specifications concise, it is sensible for our semantic model to reflect this reality, thus our model is based on an assumption of asynchrony. We recognise the difficulties of analysing and reasoning about asynchronous systems, but feel that the advantages outweigh the disadvantages. We note that synchronous behaviour can be modelled in an asynchronous environment.

The remainder of this paper is organized as follows: The next section introduces the semantics of our core behavioural concepts and operators. Section 3 concentrates on the definition of interfaces, bindings and their composition, illustrated with an example of a multicast_RPC binding. Section 4 discusses our work and section 5 compares it with related work. Conclusions and future work directions are given in section 6.

2 Specification of Behaviour

We begin by introducing definitions needed in the description of behaviour. These are mostly based on those in RM-ODP[5]:

- Type: For any domain of instances, values or entities, we can define predicates that match some or all elements of the domain. These predicates are called types. Note that using this definition, all entities can have a type, including events, event relationships, event parameters (i.e. data), objects and interfaces.
- Location: This can be of three kinds. A physical location is an interval in physical space [5]. A logical location is an interval in some logical space or spaces that is relevant to the model. For example, a network address and port number, or (in UNIX) a process id and virtual memory address constitute logical locations. A temporal location is an interval in time[5]. An instant is a point in time.
- Object: An object represents some physical or abstract entity in the system being modelled. An object has an identity in the context of the model. An object is

assumed to have a boundary that separates the object from its environment; i.e. the rest of the system. Examples of boundaries include physical boundaries, address space boundaries, boundaries between software layers and the notional "abstraction boundary" for software objects. An object is assumed to be encapsulated; i.e. any state changes for an object occur only as a result of a purely internal occurrence or an occurrence at the object's boundary that is explicitly modelled.

2.1 Core behavioural concepts

We first define the concept of an event and then describe event relationships in terms of event ordering, event parameters relationships and temporal relationships.

2.1.1 Events. An *event* is a unit of interaction between an object and its environment. Events are not necessarily atomic and may be decomposed into a set of events in a lower-level abstraction. Events notionally occur at the boundary of an object and at a particular temporal location. Each event has a (possibly empty) set of parameters. Each parameter in an event has a value which must be transmissible and which is immutable in the context of the event. Events also have a direction relative to the a boundary of an object; i.e. either *in* or *out*. Parameters of an *in* event are bound to a value by the environment and parameters of an out *event* are bound by the object.

Formally, an event is described by the tuple <*name, dir, loc, P*> where

- *name* is the denotable name of the event;
- *d*ir ∈ **{***in, out***}** is the direction of the event;
- *loc* is the location of the event;
- *P* is a set of event parameters, each described by a tuple <*name, type, value*>.

In this paper, we denote events in the form *e!(a,b)* where *e* is the event name, *!* or *?* denote an outgoing or incoming event respectively, and *a,b* are the event parameter names. Location and the additional parameter details are typically omitted for brevity. Once an event has been introduced, it is denoted in subsequent use by the event name alone. We use the notation *e.n* to denote the parameter named *n* of event *e.*

2.1.2 Event ordering. As stated previously, an important part of an event relationship specification is an event ordering specification. This can be produced by applying event ordering operators to a set of events. An event ordering is a relation defining the causal dependence or independence of a pair of events.

Our model provides the following event ordering operators:

- Causality, denoted $a \rightarrow b$: an event *a* causally affects event *b*. This operator is introduced to capture "cause-effect" relations between events. This operator is anti-symmetric and transitive
- Independence, denoted $a \parallel b$: events *a* and *b* are causally independent. Formally:

  $(a \parallel b) \Leftrightarrow \neg (a \rightarrow b) \wedge \neg(b \rightarrow a)$

  This is introduced to explicitly specify that there is no causal relationship between two events (this operator is similar to ||| operator in LOTOS). This operator is symmetric and non-transitive.

The notion of causality and the resulting partial order is used in defining consistency constraints related to event parameters and timing. Note that the sequential "happened before" relation used in process algebras is insufficient because of the underlying interleaved concurrency model that implicitly relates events with no causal relationship.

Events that are not explicitly or transitively related by an ordering specification are causally independent. Although this means the || operator is redundant in a complete specification, the operator is required for detection of conflicts between composed specifications.

2.1.3 Event parameters. Event parameter relationships describe relationships between the parameters of causally related events. In general, an event parameter relationship can specify any relationship between event parameters. To make this semantic tractable, we use the binding context to constrain both the visibility of events and the range of relationships that can be specified. In a binding context, an information flow specification has the following semantics:

> An event parameter relationship describes the relationships between the parameters of causally related events visible in the current binding. Relationships between the types of event parameters must be well defined in the binding context.

Consider the following example:

$e!(x:DCEint) \rightarrow f?(x:CORBAint) \wedge e.x = f.x$

This specifies that the output event *e* causes an input event *f*, and that their single parameters are related by the = relationship. For this to be a valid specification, it must occur in a binding that defines an equality (=) relationship between the data types *DCEint* and *CORBAint.* Although this example uses an equality relationship, values can have arbitrary relationships restricted only by the relationship definitions accessible in the binding context.

In this definition, we free ourselves from the need to specify a data type system, since it is defined by the binding context. This semantic allows us to describe middleware that is largely independent of the type system used for transferring information. Such middleware could, for

example, support bindings that connect a CORBA client with a DCE server, provided appropriate equality relationships are defined between the CORBA and DCE type systems in the binding context.

Note also that there is no reason why a binding context cannot provide the means for defining new relationships, allowing application-specific relationships to be defined. This is of particular use in legacy systems, since the relationships between parameters output by one legacy application and input by another are specific to the applications.

There are many situations where the behaviour associated with a high-level binding description can be largely independent of the actual types of parameters to events. In particular, the positional or name equivalence of parameters to complementary events is very commonly used (e.g. both DCE and CORBA RPC). To simplify the specification of these relationships, we predefine name equivalence operator '*=' for complementary events as follows:

$$e! \rightarrow f? \Rightarrow (e \mathrel{*=} f) := \forall f.n : \exists e.n: f.n = e.n$$

This operator specifies that all parameters of the input event *f* have an equivalence relationship with the parameter of a causally preceding event *e* having the same name *n*. Positional equivalence is an application of this operator in systems where parameters are named by sequentially increasing numbers. Note that correct behaviour depends on the equality relationship = being defined for the types of all corresponding parameters, and the names of parameters to event *f* being a subset of those in event *e*.

Although parameter relationships imply causality, causal relationships can exist without parameter relationships. Parameter relationships cannot, however, exist without a causal relationship. As a result, we require the explicit specification of causal relationships, although a language implementation might infer causal relationships from parameter relationships.

2.1.4 Event timing. An event relationship may include conditional constraints on the start and end times of events using these functions and the relationships such as equal, less than, greater than, and so on as normally defined for numeric values.

Time specifications support the expression of temporal constraints that cannot be modelled with partial ordering. To support expression of time properties, we define two functions: *i) Start*(event), which returns the initiation time of an event, and *ii) End*(event), which returns time of event termination. Both of these times are non-negative numbers and the values returned are accurate within some bounds. The correctness of a binding with respect to its timing relationships can only be determined when the accuracy bounds are known. These bounds will typically be defined by the system in which a specification is executed.

2.2 Composition

The transitivity of the causality operator $\rightarrow$ provides basic composition of event relationships, but is limited by its asymmetry (i.e. the same event cannot occur more than once in partial ordering specification). Since the independence operator || is non-transitive, it serves no use in composing event relationships. In order to support more complex composition, additional operators are required. This subsection defines a set of operators for composition of event relationships. Note that at present, no iteration operator is defined. Iteration will be added when the language requirements become clearer.

2.2.1 AND. The AND of two event relationship specifications, denoted "$\wedge$", is the logical AND of their ordering, parameter and timing relationship specifications. Events, denoted by their event name, may appear on both sides of the operator, e.g. $a \rightarrow b \wedge a \rightarrow c$ specifies that *a* causally precedes both *b* and *c*. The operator is symmetric.

2.2.2 OR. The OR of two event relationship specifications, denoted "$\vee$", specifies that one or both of the behaviours may be chosen. As with logical AND, events may appear on both sides of the operator, e.g. $a \rightarrow b \vee a \rightarrow c$ specifies that *a* causally precedes *b* or causally precedes *c* or both. The operator is symmetric.

2.2.3 Exclusive OR. The exclusive OR of two event relationships specifications, denoted "|", specifies that one of the behaviours may be chosen, but not both. This is equivalent to the choice operator of process algebras. The operator is symmetric.

2.2.4 Subset choice. Subset choice, denoted [<c>]{<E_1, E_2,..., E_n} specifies that a (possibly empty) subset of the listed event relationships may be chosen, subject to the optional cardinality constraint <c>. A cardinality constraint defines a maximum and/or minimum number of behaviours that must be selected, e.g *[# >= 2]{b, c, d}* specifies that at least two of the events *b*, *c* and *d* must be chosen. The keyword *all* can be used in the cardinality constraint to denote all event relationships whose behaviour is possible in the current context (i.e. whose causally preceding events have occurred and guards satisfied).

Although subset choice can be described in terms of the other composition operators, it provides a convenient mechanism for describing group communication, which is increasingly common in distributed systems.

2.3 Other semantic constructs

2.3.1 Guards. We allow events and event relationships to be guarded by logical expressions. Guard expressions

define conditions that must be satisfied at a particular point in a binding, or that must be satisfied before an event can be correctly executed.

The logical expressions in a guard can reference any parameter of a causally preceding event, or information made available by the binding, and can use any relationships defined by the binding context. For the purpose of this paper, we denote guards by enclosing logical expressions in square brackets '[]'.

2.3.2 Logical NOT. The logical NOT operator, denoted "¬", is an explicit specification that a particular behaviour should not occur. Although not strictly a composition operator, it is useful in many situations.

2.3.3 Precedence and grouping. Although our primary goal is not to define a language syntax, it is useful to define precedence and grouping for the operator to support specification examples. The precedence of the operators is defined as follows:

(1) Logical NOT and guards

(2) Causality and independence

(3) Logical AND

(4) Logical OR, Exclusive OR and Subset Choice

Operators with the same precedence are applied left to right. Parenthesis "()" may be used for grouping to override precedence.

2.4 Consistency

There is potential for conflict between ordering, parameter and timing specifications in an event relationship specification. A consistent specification must be free of such conflicts, in particular:

- an event specification must not contain any cycles in the partial order defined by its ordering specification, and there must be no contradictory statements of causal dependence and causal independence;
- event parameter relationships must not exist where there is no causal relationship between events;
- timing relationships must not contradict the ordering relationships, e.g. a constraint on the starting time of an event should not specify or imply that it occurs before the starting time of a causally preceding event;

Where these consistency requirements are applied to a non-deterministic specification (i.e. one involving OR, exclusive OR or subset choice), then the requirements must be met for all possible behaviours.

2.5 Abstraction

Abstraction is the process of hiding irrelevant detail to establish a simplified model, or the result of that process[5]. In terms of our semantic model, abstractions are used to hide those event relationship details and the building blocks that are irrelevant for the specific aspects of an architecture being modelled.

Consider for example the specification of a workflow model which describes the flow of contractual activities between two organisations. Typically, this include contract negotiation, validation, monitoring and enforcing activities[11]. The complexity of some of these individual activities and the different viewpoints of those involved in the interaction suggests the need for abstraction.

The concept of refinement is the inverse of abstraction. It is a process of transforming a specification into a more detailed specification. This is, for example, useful in verifying conformance to standards, such as ODP standards[5] or software engineering methodologies based on refinement.

Both the abstraction and refinement concepts represent powerful mechanisms for the specification of open distributed systems, and also represent an important part of our semantic model. We outline these aspects of our model by focusing on abstraction.

We define the semantics of our abstraction operators as follows. An abstraction *S'* of event relationship *S* is defined by the statement:

$S' \equiv_R S$,

where S, S' and R are all event relationship expressions. The abstraction operator specifies the logical AND of these three specifications ($S \wedge S' \wedge R$), and states that S is synthesised from S' using the event relationship R. R specifies the causality, parameter and timing relationships between the events of S and S'. The notion of synthesis implies the following rules:

- parameters of output events in S' must be expressible in terms of parameters of causally preceding events in S or S';
- parameters of input events in S' may be independent of any event parameters in S. This allows, for example, an abstraction of a service that hides unnecessary parameters.

For example, a single output event *e* can be synthesised from an event relationship of three events:

$$e!(p,q) \equiv_R a!(x) \rightarrow b?(y) \rightarrow c!(z), \text{ where}$$

$$R := c \rightarrow e \wedge e.p = a.x \wedge e.q = c.z$$

This states that the event *e* is synthesised from the events on the right hand side, subject to the constraint that *e* occurs after all those events, and that the parameters

(p,q) of *e* are equal to the parameter *x* of *a* and the parameter *z* of *c* respectively.

There can be many distinct abstractions of an event relationship, with no constraints apart from their reliance on common component events (i.e. there is no requirement for purely hierarchical abstraction). The abstraction operator is non-transitive and anti-symmetric, however, the normal transitivity and symmetry properties apply to event relationships resulting from the abstraction.

The event relationship resulting from the logical AND of the three abstraction components must satisfy the standard consistency requirements specified in 2.3. This is a mandatory consistency requirement and no further consistency constraints are necessary.

It is sometimes necessary, however, to define abstraction relationships with the constraint that the less abstract event relationship is substitutable for the more abstract event relationship (i.e. S is substitutable for S'). This substitutability is not guaranteed by the above consistency constraint. We do not define the additional rules required to achieve substitutability as it is beyond the scope of this paper. We note, however, that applying the principles of Morgan's refinement calculus[10] or Liskov subtyping[6] would achieve substitutability.

2.6 Abstract causality operators

The basic event ordering operators ($\rightarrow$ and $||$), while enabling the specification of causal relationships, do not easily allow the specification of more complex relationships resulting from abstraction, e.g. the relationship resulting from the specification of a workflow activity as a set of interwoven activities in another abstraction.

To deal with these complex relationships, we extend the basic set of event ordering operators. The additional event ordering operators provide a powerful mechanism which allows us to hide unimportant orderings in an abstraction.

In order to define the extended causality operators, it is necessary to use a descendants function, denoted *Desc*, that identifies the set of events that contribute to an event in a lower-level abstraction. Informally, the *Desc* function applied to an event *e* returns the set of all events that contribute to the behaviour associated with *e* (i.e. events that compose *e* either directly or indirectly). Note that if an event *e* is atomic then Desc(*e*)=*e*. A more formal definition has been derived, but is excluded for brevity.

Using the above definition, the extended event ordering operators are described as follows.

- Universal causality, denoted ($a \ \chi \ b$). This operator states that there is a causal relationship between descendents of events *a* and *b*, where the precise ordering is irrelevant. Formally:

  $(a \ \chi \ b) := \exists a' \in \text{Desc}(a), \exists b' \in \text{Desc}(b) : (a' \rightarrow b') \vee (b' \rightarrow a')$

  The χ operator is symmetric and non-transitive.

- Universal ordered causality, denoted ($a \ \vec{\chi} \ b$). This operator allows the expression of a specific "cause-effect" ordering between descendant of events *a* and *b*. Formally:

  $(a \ \vec{\chi} \ b) := \exists a' \in \text{Desc}(a), \exists b' \in \text{Desc}(b) : (a' \rightarrow b') \wedge \neg\exists (a'' \in \text{Desc}(a), b'' \in \text{Desc}(b)) : b'' \rightarrow a''$

  The $\vec{\chi}$ operator is anti-symmetric and non-transitive.

- Universal cyclic ordered causality, denoted ($a \ \overleftrightarrow{\chi} \ b$) . This operator allows the expression of a cyclic causality, whereby a descendent of *a* from the causally affects a descendent of *b* and vice versa. Formally:

  $(a \ \overleftrightarrow{\chi} \ b) := \exists a' \in \text{Desc}(a) , \exists b' \in \text{Desc}(b) : (a' \rightarrow b') \wedge \exists a'' \in \text{Desc}(a), b'' \in \text{Desc}(b) : (b'' \rightarrow a'')$

  The $\overleftrightarrow{\chi}$ operator is symmetric and non-transitive. Given our consistency constraint specified in 2.3, this operator implies that one of *a* or *b* is non-atomic.

We illustrate the use of these extended causality operators using the example of the serial execution requirement for ACID transactions:

> A transaction is an abstraction of a set of operations with a sequential, causal order. Distinct transactions conflict if any of their component operations conflict. An execution of transaction T_1 followed by T_2 is correct if and only if all causal relationships between their respective operations are consistent with the order of T_1 and T_2, that is, all their operations are independent or $T_1(OP_i) \rightarrow T_2(OP_j)$. This is also called a serial execution. The negation of the above rule (and with the transitivity properties of $\rightarrow$) means that T_1 or T_2 can precede itself (there is a cycle in the causal ordering). By generalizing this example to n transactions executed in a history H it can be stated that an execution of a set of transactions is a serial execution if $\forall T_i \in$ **H:** $\neg(T_i \ \overleftrightarrow{\chi} \ T_i)$.

This example shows how our abstract ordering operators allow specifications of behaviour at a more abstract level (e.g. in terms of transactions rather than the individual operations).

We note that the abstract causality operators also allow the description of non-deterministic ordering in behaviour, allowing an execution environment to make scheduling choices where multiple orderings are possible.

3 Modelling concepts

The preceding section introduced the basic concepts we use to model behaviour. In this section, we define *inter-*

faces and *bindings* which allow us to structure and encapsulate behaviour. We also introduce the notion of *roles*, which are placeholders for interfaces in a binding. These concepts give additional abstraction capability over that defined in 2.4.

3.1 Interfaces

An *interface* is an abstraction of an object that identifies a subset of the interactions of that object with its environment. An interface is defined by the tuple <*name, loc, ER*> where:

- *name* is the denotable name of the interface
- *loc* is the location of the interface
- *ER* is an event relationship describing the behaviour occurring at that interface. All events in ER occur at the interface location *loc*, noting that an interface location may change over time.

In this paper, we denote the execution of an event *e* by an interface *Int* as *Int.e.*

3.2 Roles

A role is a placeholder for interfaces within a binding, describing the common behaviour and expectations of interfaces filling that role. A role is defined by the tuple <*name, card, interfaces, ER*> where:

- *name* is the denotable name of the role
- *card* is a cardinality constraint specifying the maximum and/or minimum number of interfaces that may fill the role
- *interfaces* is the set of interfaces filling the role
- *ER* is an event relationship describing the allowable and expected behaviour of interfaces filling the role

A role can be filled by an interface if their event relationship specifications are compatible. A strong compatibility requirement can be a subtype relation defined in [6][13]. In this case, we say that an interface and a role are matched if the interface is a subtype of the role. Weaker compatibility requirements are quite common in practice, for example, DCE requires only that version identifiers share the same major version number.

The execution of an event by an interface filling a role may be non-deterministic, since a role can be filled by many interfaces. Where more than one interface can fulfil a role, we define the execution of an event by a role to have subset choice semantics as defined in 2.2.4, that is, a subset of the interfaces filling the role each execute the event at their interface.

In this paper, we denote the execution of event *e* by role *R* as *R.e*, noting that this can be prefixed with a cardinality constraint if the role can be filled by more than one interface. In role specifications, we typically omit the specification of interfaces filling roles, allowing interfaces to be associated with roles by the enclosing context.

3.3 Bindings

The most useful interactions in open distributed systems are those which occur between events located at the interfaces of different objects. In our semantic model, these interactions are encapsulated in a binding, which provides an environment for the execution of interactions. A binding is defined by the tuple <*name, roles, ER*> where:

- *name* is the denotable name of the binding
- *roles* is a set of placeholders for interfaces, defined in
- *ER* is an event relationship describing a composition of the behaviour executed by interfaces filling the roles. This behaviour must be consistent with any configuration of interfaces correctly filling the roles of the binding. More formally, the AND of the binding event relationship *ER* with the event relationships of interfaces filling the roles must be consistent using the rules defined in 2.3.

Figure 1. illustrates a binding with 5 roles, three of which are filled by single interfaces. Although not illustrated, an interface can fulfil multiple roles, e.g. an interface which supports transaction semantics should fulfil roles of (1) executing normal operations such as read or write and (2) executing transaction verbs such as commit and prepare. It is important to note that a roles in a binding need not be complementary, i.e. there does not need to be a *receive* event for every *send* event.

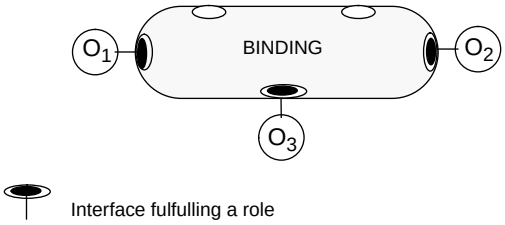

Figure 1. Binding

To illustrate the utility of our modelling concepts, the following example specifies a multicast binding. *The syntax used in this example is arbitrary and should not be taken as an indication of likely language syntax.* In this example, comments are preceded by "--" and role definitions are terminated by ";" and the role and behaviour specifications are introduced by keywords and enclosed in braces "{}". We also leave the association of interfaces and roles unspecified, allowing the binding to be used between any set of objects with compatible interfaces.

```
Binding MultiCast {
  Roles {
    S:[#=1],send!;     -- 1 sender
    R:[#>1],rec?;      -- >1 receivers
  }
  Behaviour {
    S.send! → [#>1] R.rec? ∧
    [#=all]R.rec? *= S.send!
  }
}
```

This specifies that in a binding with one sender *S* and more than one receivers *R*, a send event by the sender interface results in multiple receive events at distinct receiver interfaces, and the parameters to those receive events have name equivalence with those of the send event.

3.4 Composition of bindings

In order to support re-use and abstraction of bindings, it must be possible to use pre-specified bindings in a higher-level specification. To support this, we define that the use of a binding within a behaviour specification corresponds to the specification of that binding's event relationship *ER*. This allows the use of a binding as an argument to any operator defined for event relationships, including all of those defined in section 2.

When re-using a binding specification, the association of interfaces and roles in the binding can be either explicitly specified within the binding, or specified by the context in which it is used. Where the enclosing context is another binding, the interfaces filling roles of the enclosing binding can be associated with the roles of the enclosed binding.

This concept is best illustrated by an example. Our example uses the previously defined multicast binding and a new choose-reply binding to construct multicast RPC. We first define choose-reply and then define the composition that gives us multicast RPC. Choose-reply is defined as follows:

```
Binding ChooseReply {
  Roles {
    S:[#>0], send!;    -- >0 senders
    R:[#=1], rec?;     -- 1 receiver
  }
  Behaviour {
    [#>0]S.send! ∧
    [#=1]S.send! → R.rec? ∧
    R.rec? *= [#=1]S.send!
  }
}
```

This specifies that in a binding with one or more senders *S* and one receiver *R*, at least one sender sends, and exactly one of those send events results in a receive event at the receiver. The parameters to the receive event have name equivalence with those of the casually preceding send event. In other words, a message is non-deterministically chosen from a set of possible messages.

The multicast RPC binding is created by composing ChooseReply and Multicast bindings as follows:

```
Binding MulticastRPC {
  Roles {
    Client: [#=1] send! → rec?;
    Servers:[#>1] rec? → send!;
  }
  Behaviour {
    Multicast(S=Client,R=Servers) →
    ChooseReply(S=Servers,R=Client)
  }
}
```

This specifies that a multicast RPC binding has a single client, which executes a send followed by a receive, and multiple servers, each of which may execute a receive followed by a send. The behaviour specifies that a multicast with the client interface bound to the sending role and server interfaces bound to the receiving role is executed, followed by a choose-reply with the server interfaces bound to the sending role and the client interface bound to the receiving role.

The underlying behaviour specifications of the composed bindings results in the a multicast RPC being specified. Note that neither the client nor the servers need to be aware that the request is being multicast, or that multiple servers may reply. The clear distinction of interface and interaction allow this flexibility.

This example demonstrates the power of our semantic model in structuring, composing and abstracting specifications of a distributed systems architecture.

4 Discussion

The focus of this paper is on defining a semantic model for describing the architecture of open distributed systems. While the model presented is extremely powerful and introduces a number of novel features, there are a number of areas that require further research and refinement.

The eventual goal of our work is develop a architecture description language, and there are a number of useful language features that are not reflected in our model. In particular, we believe that the ability to dynamically define and modify behaviour through a well-defined meta-level is a necessary feature of a language. While our current semantic model does not preclude this feature, further work is required to introduce the necessary facilities.

Our model currently does not address the issues of binding configuration, instantiation and termination in an executable environment. On these issues, we expect to be guided by practical work on building Hector, an execution environment for the DSTC architecture model[1].

While we are easily able to describe complex behaviour in our current model, there are no iteration semantics for describing repetitive behaviour. This is at least partly because the requirements and hence appropriate semantics are not clear. This is an important issue that will be dealt with in the near future.

5 Related work

Our work embodies much of the existing theoretical and practical aspects of modelling open distributed system architectures.

The theoretical aspects of our model are similar to several event-based formal description techniques including as CCS[9], CSP [3] and LOTOS[4]. These provide similar notions of ordering and gates or ports (interfaces). Based on the assumption of synchrony and event atomicity, however, these techniques have limited ability to describe complex interaction or support abstraction. For example, it is quite difficult to describe multicast in any of these languages. They also have limited ability to describe non-trivial parameter relationships, and none for describing explicit temporal constraints. Their advantages lie in the ability to analyse behaviour using exist tools and in the simplicity of the synchronous approach.

The practical aspects of our work are similar to ongoing research on software architecture[7][8][12][16]. Moriconi et al[12] present a method for the stepwise refinement of an abstract architecture into a correct lower-level architecture that is intended to implement it. Shaw et al[16] use a fixed set of connection architectures, rather than providing a model for describing arbitrary architectures. The notion of binding in our model is similar to the concept of connector in these works. Our work extends these, particularly in the handling of event parameters, and the power and flexibility of our behavioural specifications and abstraction.

Darwin[8] is a declarative binding language used to define hierarchic compositions of interconnected components. The focus in Darwin is on constructing a correct configuration of components, rather than describing the behaviour of a configuration of objects, and as such it is not directly comparable to our work. The concepts of Darwin could be useful, however, when we address the issues related to binding instantiation. Darwin has a formal semantics given in on π-calculus based largely on CCS[9].

Rapide[7] is a concurrent event-based simulation language for defining and simulating the behaviour of architectures for distributed systems. It is probably closest to our work in terms of semantics, but with a focus on simulation and analysis rather than supporting the construction of distributed systems. It includes the concept of event pattern mapping to provide abstraction with similar results to our abstraction operator. We believe that our semantic model based on event relationship specifications allows more concise expression of rich interactions, and has different but equivalent facilities for abstraction.

Recent work on a Unified Modelling Language (UML)[16] from the object-oriented analysis and design community aims at providing a common, stable, and expressive object-oriented development method. While the initial version of UML did not deal with distribution and concurrency, recent developments do address these aspects. Although we have not yet analysed their work in detail, the notion of *pattern* in UML is quite similar to our binding concept.

6 Conclusion

In this paper we have presented a semantic model that provides a foundation for the description of interactions in open distributed systems. In deriving the model we were driven by the pragmatic requirements of open distributed systems: interworking between heterogeneous middleware platforms, reuse of objects, and evolvability of systems.

The specification of interaction behaviour is based on the notion of events and event relationships, and we use an asynchronous model to embody typical characteristics of interaction between objects in open distributed systems. Event relationships are expressed in terms of event ordering, parameter relationships, and temporal relationships. Each of these aspects reflects distinct needs of the builders and users of open distributed systems. The event parameter relationships in particular support the concise specification of interworking requirements in open distributed systems, for example, interworking between CORBA and DCE objects. The time relationships reflect the need to specify timing properties in many applications, for example, multimedia video transmission.

In our model, special attention is given to the problem of composition and abstraction to address the need for reuse and evolution in open distributed systems. The event composition and abstraction operators enhance the expressiveness and clarity of behavioural specifications, and provide a semantic foundation for the specification of interfaces, bindings and their composition.

Interfaces and bindings are the main building blocks for an architecture of open distributed systems. These concepts provide a powerful mechanism for structuring and reasoning about open distributed systems. The use of bindings in particular can facilitate the standardisation of common interactions such as multicast, and has significant potential for application to enterprise-level interactions.

We believe that this work, specifically oriented towards architectures for open distributed systems, is relevant to many existing and future software engineering practices.

In particular, it provides a basis for deriving architecture description languages for open distributed systems. The DSTC Open Environment[1] provides a complementary infrastructure for testing and refining the semantic model defined in this paper. Coupled with the capabilities of this infrastructure, an architecture definition language based on this work can provide both research and commercial platforms for implementing feasible and flexible open distributed systems.

Acknowledgments

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

References

[1] A. Bond, D. Arnold, M. Chilvers. *Hector: Designing and Building an ODP Environment.* Submitted to ICODP'97.

[2] A. Berry, K. Raymond., *The A1! Architecture Model*. In Proceedings of RM-ODP'95: Experiences with distributed environments. K. Raymond L. Armstrong Editors, Brisbane, February 1995.

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

[4] ISO/IS 8807. *LOTOS. A formal description technique based on the temporal ordering of observational behaviour.*

[5] ISO/IEC 10746-1 10746-2 10746-3. *Basic Reference for Open Distributed Processing.*

[6] B. Liskov, J.M. Wing. *A New Definition of the Subtype Relation*. In Proceedings of ECOOP'93, Kaiserslautern, Springer-Verlag, July 1993.

[7] D.C. Luckham, J. Vera. *An Event Based Architecture Definition Language*. IEEE Transactions on Software Engineering Vol. 21, No 9, September 1995.

[8] J. Magee, N. Dulay, S. Eisenbach, J. Kramer. *Specifying Distributed Software Architectures*. In Proceedings of 5th European Software Engineering Conference, Spain, 1994.

[9] R. Milner. C*ommunication and Concurrency.* Prentice-Hall 1989.

[10] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[11] Z. Milosevic, A. Berry, A. Bond, K.Raymond. *Supporting Business Contracts in Open Distributed Systems*. In Proceedings of SDNE'95, Whistler, Canada, IEEE Computer Society Press, June 1995.

[12] M. Moriconi, X. Qian, A.A. Riemenschneider. *Correct Architecture Refinement*. IEEE Transactions on Software Engineering, Vol. 21, No 4, April 1995.

[13] O. Nierstrasz. *Regular Types for Active Objects*. In Proceedings of OOPSALA '93, ACM SIGPLAN Notices, Vol. 28, No 10, October 1993.

[14] O. Nierstrasz, T. Dirk Meijler. *Requirements for Composition Language*. In Proceedings of ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution. Springer, 1995.

[15] M. Shaw, R. DeLine, V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, Vol. 21, No 4, April 95.

[16] http://www.rational.com/ot/uml.html