Coordination Models Based on a Formal Model of Distributed Object Reflection

Carolyn L. Talcott [1,2]

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

We propose a family of models of coordination of distributed object systems representing different views, with refinement relations between the different views. We start with distributed objects interacting via asynchronous message passing. The semantics of such a system is a set of event partial orders (event diagrams) giving the interactions during possible system executions. A global coordination requirement is a constraint on the allowed event diagrams. A system coordination specification consists of a meta-level coordinator that controls message delivery in the system according to a given global policy. The system-wide coordination can be refined/distributed using coordinators for disjoint subsystems that communicate with their peers to enforce the global policy. By a further transformation the meta-level can be replaced by systematically transformed base-level objects communicating via a controller object. The coordination models are formalized in rewriting logic using the Reflective Russian Dolls model of distributed object reflection. The general ideas are illustrated with several examples.

Keywords: coordination, distributed object reflection, policy, event diagram

1 Introduction

We present ongoing work to develop semantic models of coordination of distributed object systems and formal executable specifications from multiple points of view. The view points range from a global view of the possible interactions among a system of objects to a local view of how controller objects achieve coordination. The former can be thought of as an end-to-end requirements level. The latter is closer to system design or implementation. The eventual goal is a theory of refinement and composition for coordination of distributed object systems.

[1] The work was partially supported by NSF grant CCR-023446.

[2] Email: clt@cs.stanford.edu

The underlying computation model is based on the actor model of distributed objects interacting via asynchronous message passing [15,5,1]. The semantics of such a system is a set of event partial orders, called event diagrams [14,9,26]. Executable specifications are formalized in rewriting logic [18,20] using the Reflective Russian Dolls (RRD) model of distributed object reflection [19].

In this setting, a coordination requirement is a constraint on the allowed event diagrams. A system-wide coordinator is specified by specifying a coordination policy enforced by a meta-level coordinator object that controls message delivery in the system according to that policy. The system-wide coordination can be distributed using coordinators for disjoint subsystems that communicate with their peers to enforce the system-wide policy. The meta-level can be replaced by systematically transformed base-level objects communicating via a controller object.

Plan. Section 2 gives a brief introduction to rewriting logic and the RRD model. Section 3 defines the event model and the notions of coordinator and coordination requirement. Section 4 defines the policy based executable specification of a coordinator and the notion of a coordination policy satisfying a coordination requirement. Section 5 discusses distribution of system-wide coordinator to multiple local coordinators, and a transformation to remove the meta-level in favor of an object level controller. Section 6 discusses some related work and section 7 concludes with a discussion of future directions.

2 Background

To provide context we give a brief introduction to rewriting logic and the Reflective Russian Dolls (RRD) model of distributed object reflection.

Rewriting logic [18] is a logical formalism designed for modeling and reasoning about concurrent and distributed systems [17]. It is based on two simple ideas: states of a system are represented as elements of an algebraic data type; and the behavior of a system is given by local transitions between states described by *rewrite rules*. A rewrite rule has the form $t \Rightarrow t'$ if c where t and t' are terms representing a local part of the system state, and c is a condition on the variables of t. This rule says that when the system has a subcomponent matching t, such that c holds, that subcomponent can evolve to t', possibly concurrently with changes described by rules matching other parts of the system state. The process of application of rewrite rules generates computations (also thought of as deductions). Maude [7,8] is a formal language and tool set based on rewriting logic used for developing, prototyping, and analyzing formal specifications.

Reflective Russian Dolls (RRD) [19] is a generic formal model of distributed object reflection based on rewriting logic. The model combines logical reflection with a structuring of distributed objects as nested configurations of meta-objects (a la Russian Dolls) that can reason about and control their sub-objects. This model can be used to develop formal specifications of interaction as well as architectural, and behavioral aspects of distributed object-based systems. For example, the Inter-

net is not really a flat network, but a network of networks, having different network domains, that may not be directly accessible except through specific gateways, firewalls, and so on. As another example, a multimedia server is a nested collection of resource manager objects (load balancing, admission control, object placement, and so on) and an execution environment object that coordinates execution of contained objects generating media streams.

For the purpose of specifying and modeling coordination we use two broad classes of meta-object—*coordinators* and *customizers*. A coordinator has a distinguished attribute that holds a nested configuration of objects and messages and controls delivery of messages in its contained configuration. A customizer contains a single object and is used to locally manage object meta-data and adapt the objects communication.

3 Event Diagrams and Coordination Requirements

We use Maude-like syntax in describing the formal model. Objects are formalized as terms of the form

```
[a :  A | atts | inQ, outQ]
```

where `a` is an object identifier, `A` is a class identifier, and `atts` is an attribute set, giving the objects internal state. `inQ` and `outQ` are the objects input and output message queues. For simplicity we assume messages have the form `a<-mb` where `a` is the message target (addressee) and `mb` is the message body. The behavior of an individual object is given by message delivery rewrite rules of the form

```
rl[dlv]:  [a : A | atts | (msg inQ), outQ]
           =>
          [a : A | atts' | inQ, (outQ outQ')]  if cond
```

The first message of the input queue (`msg`) is delivered. As a result the object's attributes may be modified, becoming `atts'`, and messages `outQ'`, possibly none, are added to the output queue. The term `cond` is a boolean term constraining conditions under which the rule applies. [3]

An object system is a multiset of objects and messages with default communication infrastructure rules `[obj.out]` and `[obj.in]` that simply move messages from output queues to the system soup and move messages from the system soup into the input queue of the target actor. [4]

```
rl[obj.out]:
  [a : A | atts' | inQ, (msg outQ)] =>
         [a : A | atts' | inQ, outQ] msg
rl[obj.in]:
  [a : A | atts' | inQ, outQ] msg =>
         [a : A | atts' | (inQ msg), outQ]  if target(msg) == a
```

[3] In general, new objects may also be created. To simplify discussion, we omit that aspect.

[4] Here we consider closed systems. It is straightforward to extend the ideas to open systems where messages may arrive from external objects and may be sent to external objects by adding external interaction rules.

Rules such as the message delivery and communication rules apply to a multiset of objects and messages when the rule left-hand side matches a sub-multiset and the rule condition, if any, evaluates to `true`. In which case, the matched sub-multiset is replaced by the rule right-hand side.

A computation is a possibly infinite sequence of rewrites:

$$S_0 \xrightarrow{l_1} \ldots \xrightarrow{l_k} S_k \ldots$$

where l_i is a label determined by the rewrite rule applied, a delivery rule, or one of the infrastructure communication rules. For a rewrite using a delivery rule the label has the form `dlv(a<-mb,i,b,j)` where `a<-mb` is the message delivered, `i` is the objects local time (modeled as the number of messages sent or received) and `b,j` is the message identifier represented using the sender identifier and local time.

Event diagram semantics. The event diagram associated with a computation is the set of events `(mb,a,i,b,j)` such that `dlv(a<-mb,i,b,j)` is the label of a rewrite in the computation. The partial ordering on events is the transitive closure of the arrival and activations orders, where the *arrival order* is given by

```
(mb,a,i,b,j) < (mb',a,i',b',j')  iff i < i'
```

and the *activation order* is given by

```
(mb',b,j, a',j') < (mb,a,i,b,j'')
if a<-mb was sent in the rewrite in which b<-mb' was deliviered.
```

The event diagram semantics of a system S, $ED(S)$, is the set of event diagrams associated to the possible computations of S. [5]

Coordinators and Coordination Requirements. A *coordinator*, C, constrains the interactions of a system S. We write $C\{S\}$ for the application of a coordinator C to a system S and require

$$ED(C\{S\}) \subseteq ED(S)$$

A *coordination requirement* is a predicate Φ on event diagrams. The coordinator C_Φ associated with a requirement Φ is defined by

$$ED(C_\Phi\{S\}) = \{ed \in ED(S) \mid \Phi(ed)\}$$

Note that not all requirements are realizable. In the next section we will define a class of realizable requirements given by executable specifications of policy-based coordinators.

Requirements Example 1. Consider a system with a ticker object with identifier `t` used by other objects as a clock. A ticker has a local counter. Initially there is a mes-

[5] To formalize conditions such as the above, we instrument each object with a counter representing the objects local time and augment the communication rules by annotating messages in the soup with the sending object identifier and local time. Because we allow multiple messages to be sent at once, the sending time of the ith message generated by a message delivered at time `j` is `j + i`.

sage `t<-tick`, and objects may request the time using messages `(t<-time@a)`. When `t<-tick` is delivered, a ticker increments its counter and sends `t<-tick`. When `t<-time@a` is delivered, a ticker sends `a<-timeReply(n)` where `n` is the current value of its counter.

The requirement $\Phi_+(t)$ requires that each at least one tick is delivered between any two time requests. That is, $\Phi_+(t)(ed)$ holds if for any `(time@b',t,j,b,i)`, `(time@a',t,j',a,i')` in ed if `j < j'` then `(tick,t,k,t,k')` is in ed for some `k,k'` such that `j < k < j'`.

Requirements Example 2. In this example we specify coordination requirements corresponding to ordering guarantees that might be provided by a group communication service [6]. We consider two common guarantees: fifo and causal ordering. There are several other standard ordering guarantees that can be treated similarly. To simplify the discussion we assume a predicate that identifies group communications—messages sent to all objects in the system. Fifo delivery semantics requires that messages from the same sender are delivered in the order sent (possibly interleaved with messages from other senders). Causal delivery semantics requires fifo delivery and in addition, all group messages delivered to the sender prior to sending must be delivered to a receiver first.

The fifo ordering requirement, $\Phi_f(ed)$, holds if for all events `(mb,a,j,b,i)`, `(mb',a,j',b,i')` in ed such that `mb,mb'` are group message bodies, `i' < i` implies `j' < j`.

To define the causal order on group communication events we first define the preorder on message send identifiers, `(a,j)` $\prec_{ed}$ `(b,i)`, as the transitive closure of the following clauses: `(a,i)` $\prec_{ed}$ `(a,i')` if `i < i'`, and `(b,i)` $\prec_{ed}$ `(a,j)` if `(mb,a,j,b,i)` is in ed and `mb` is a group message body.

The causal ordering requirement, $\Phi_c(ed)$, holds if $\Phi_f(ed)$ holds and for all `(mb,a,j,b,i)`, `(mb',a,j',b',i')` in ed such that `mb,mb'` are group message bodies, if `(b',i')` $\prec_{ed}$ `(b,i)` then `j' < j`.

4 Coordination Policies and Executable Coordinators

We specify coordinators as RRD meta-objects with a policy attribute that determines when messages in the contained configuration can be delivered. To this end, we extend the specification of basic object and message data types with annotations and specification of events, finite sets of events, pending events, and policies. An *annotated object* has the form

```
[ a :  A | atts | inQ, outQ, i, status ]
```

where `i` is the objects local time, incremented each time the object is rewritten using a message delivery rule, or a message is removed from the output queue. The flag `status`, one of (`ready`, `busy`), is used to maintain the causal relation between delivery of received messages and the resulting messages sent. An *event* is a tuple `(mb,a,i,b,j)` as above, and a (finite) event diagram is a finite set

of events. A pending event is a message annotated with the sender and sending time, `(a<-mb : b,i)`, providing a unique message identifier. Pending events are messages that have been sent but not yet delivered. We declare a sort `Policy` and a satisfaction relation, `ed,M,m |= P`, where `ed` is an event diagram, `M` is a set of pending events and `m` is a pending event. If `(ed,M,m |= P)` rewrites to `true` then `m` can be delivered in a situation where the events of `ed` have happened and the remaining pending events are those in `M`.

A *coordinator* is a meta-object of class `Coord`. It is an instance of a general class of meta-objects called *containers*. A coordinator has an attribute `{_}` whose value (filling the blank) is a configuration consisting of a multiset of annotated objects and pending events. In addition to the configuration attribute, a coordinator has an attribute named `events` whose value is the event diagram of the computation of the contained configuration from the initial state to present, and an attribute named `policy` whose value is the coordination policy being enforced.

A coordinator is initially determined by its identifier `c` and policy `P`, and is applied to a system `S` as follows

```
C[c,P] = [c : Coord | {_}, events: none,policy: P | nil,nil]
C[c,P]{S} = [c : Coord | {S*}, events: none,policy: P | nil,nil]
```

where `S*` is obtained from `S` by annotating the objects with local time `0` and status `ready` and converting each message `(a<-mb)` in the configuration into an initial pending event `(a<-mb : *,*)` with unspecified sender.

The default communication infrastructure rules for the contained object system are replaced by the rules of class `Coord` for object level communication. The rule `[obj.out]` is replaced by `[coord.out]` which converts sent messages to pending events and updates the object status.

```
rl[coord.out]:
  [ c : Coord | {S [a : A | atts | nil, mQ, j, busy ]}
                policy : P, events : ed | inQ, outQ]
  =>
  [ c : Coord | {S [a : A | atts | nil, nil, j + k, ready ]  M},
                policy : P, events : ed | inQ, outQ]
    if k := length(mQ)  /\ M := mkPend(mQ,a,j)
```

where `mkPend(mQ,a,j)` is the set of pending events `(b<-mb : a,j+i)` such that `b<-mb` is the ith element of `mQ`, counting from 0.

The rule `[obj.in]` is replaced by `[coord.in]` which only applies when the policy is satisfied.

```
rl[coord.in]
 [c : Coord | {S (a<-mb : b,i)[a : A | atts | nil, nil, j, ready]}
              policy : P, events : ed | inQ, outQ]
 =>
 [c : Coord | {S [a : A | atts |  (a<-mb), nil, j + 1, busy]
              policy : P, events : (ed  e) | inQ, outQ]
   if  ed,pend(S), (a<-mb : b,i) |= P
   /\ e := (mb,a,j,b,i)
```

where `pend(S)` is the set of pending events in `S`.

We can see from the rules that a coordinator maintains the invariant that there is at most one message in an objects input queue so that the message delivery causing messages in the output queue can be determined. When the input queue of an object is `nil` and its status is `busy` this indicates that a message has been delivered, thus messages in the output queue can be turned into pending events and the local time can be incremented. If instead the status is `ready`, this indicates that the object is waiting for the next message to deliver. In a computation of a coordinated system, if `(a<-m : b,j)` is among the pending events, then the activation predecessor event (having the form `(mb',b,i,b',i')` with `i` less than `j`) is in events attribute of the coordinator, as well as all arrival predecessors. Furthermore if `(mb,a,j,b,i)` is in the events attribute, then the local time of object `a` is greater than `j`.

Defn: Coordinated system event diagrams. The event diagram semantics for a coordinated system `C[c,P]{S}` (with objects in `S` having empty input and output queues) is defined as follows. Let π be a computation for `C[c,P]{S}` and let ed_i be the value of the `events` attribute of the coordinator in the ith state. Then the event diagram associated to π, $ED(\pi)$ is the union of the finite event sets

$$ED(\pi) = \bigcup_{i \in \mathbf{Nat}} ed_i.$$

Defn: Policy satisfies Requirement. We say that a policy `P` satisfies a coordination requirement Φ (written `P` $\models \Phi$) if for each computation π of `C[c,P]{S}` we have

$$\Phi(ED(\pi))$$

Examples of Coordination Policies

Now we give examples of policies satisfying the example requirements given at the end of Section 3.

Policy Example 1. The policy `P+(t)` for coordination of communication with a ticker is specified by the equation

```
( ed, M, (b<-mb, a,j) |= P+(t) ) =
   if b == t and  mb == time@a
   then msgBody(last(ed,t)) == tick  else true fi
```

where `last(ed,t)` is the last event delivered to `t` in `ed` and `msgBody` selects the message body component of a pending event. (If there are no delivered events the equality will fail due to the initial semantics of Maude modules.)

Proposition 1. The ticker policy `P+(t)` satisfies $\Phi_+(t))$.

To give an idea of how to reason about policies, we sketch a proof of *Proposition 1.* Let π be a computation of a coordinated ticker system, and let $ed = \bigcup_{i \in \mathbf{Nat}} ed_i$ be the associated event diagram. It is sufficient to show $\Phi_+(t)(ed_i)$ for each i, which we do by induction. The base case is trivial. Assume $\Phi_+(t)(ed_i)$ and con-

sider ed_{i+1} which is reached by applying one of the rewrite rules. The only rule that changes the `events` attribute is `coord.in`, and the only event addition that could violate $\Phi_+(t)$ is one of the form `(time@b,t,j,b',i)`. The policy requires that last event in ed_i for `t` has the form `(tick,t,j',t,i')` and by the coordinator invariants `j'` is less than `j`. Thus we have the required intermediate `tick` event.

Policy Example 2f. The fifo order coordination policy `P-f` is axiomatized by the equation

```
(ed, M, (a<-mb, b,j) |= P-f ) =
    group(mb) and before-f(M,a,b,j) == none
```

where `before-f(M,a,b,j)` is the set of pending events in `M` of the form `(a<-mb',b,j')` such that `j' < j`.

Proposition 2-f. The fifo order policy `P-f` satisfies Φ_f.

Policy Example 2c. The causal order coordination policy `P-c` is axiomatized by the equation

```
(ed, M, (a<-mb, b,j) |= P-f ) =
    group(mb) and before-c(M,a,b,j) == none
```

where `before-c(M,a,b,j)` is the set of pending events in `M` of the form `(a<-mb',b',j')` such that `(b',j')` $\prec_{ed}$ `(b,j)`.

Proposition 2-c. The causal order policy `P-c` satisfies Φ_c.

We omit proofs of propositions 2-f and 2-c, noting that the 'before' sets being empty ensure that all required predecessor messages have been delivered.

5 Refining Coordinator Specifications

Now we briefly consider two refinements of the policy-based coordinator: distribution of coordinators, and flattening (elimination of meta-level nesting).

5.1 Distributing Coordination

In practice, a system being coordinated may be distributed and thus can not be placed under the control of a single coordinator meta-object. Here we show how a system can be partitioned among several distributed coordinators that communicate with one another to enforce a system-wide coordination policy.

A distributed coordinator has two additional attributes: `sent` and `fwd`. The value of `sent` is the set of pending events that have been sent to a peer coordinator for delivery, and the value of `fwd` is a table giving for each remote object, the identifier of the peer coordinator containing that object. The coordinator communication rules are extended with rules for sending messages to and receiving messages from remote locations. A pending event with remote target is sent to the containing coordinator in a `dlv` message. When a `dlv` message arrives the contained pending event is added to the configuration.

```
rl[coord.xsend]:
  [c : Coord | {S (x<-mb : b,i)}, policy : P, events : ed,
               sent: M, fwd: locs | inQ, outQ]
  =>
  [c : Coord | {S},  policy : P, events : ed,
               sent: M  (x<-mb : b,i), fwd: locs
              | inQ, outQ c'<-dlv((x<-mb : b,i), c)]
  if  c' := lookup(locs,x) /\ c' =/= c

rl[coord.xrcv]:
  [c : Coord | {S}, policy : P, events : ed, sent: M, fwd: locs
             | (c<-dlv((a <-mb : x,i), c')) inQ, outQ]
  =>
  [c : Coord | { S (a<-mb : x,i) },  policy : P, events : ed,
                sent: M, fwd: locs | inQ, outQ]
```

In the case of ticker system coordination, this is sufficient. The coordinator containing the ticker only needs local events to check the policy and messages to other objects are not constrained.

For the group communication example, further adaptation is needed to ensure that each coordinator has sufficient information to make correct ordering policy decisions. One way of accomplishing this is to extend the annotations of pending events with a set of message identifiers (sender,send time) corresponding to messages that must be delivered before the pending message. For example, in the fifo case the set of message identifiers of all messages previously sent by the pending event sender is sufficient. The fifo policy can then be adapted to use only the extended annotations and the local event diagram to determine satisfaction.

In generally, annotation and policy adaptation can be done as a transformation of the global coordinator, and then the global coordinator can be distributed as for the ticker system, once decisions are localized. Localizing first means that the work of verifying that the localization is correct can be carried out at one level of abstraction rather than dealing with level crossing at the same time.

5.2 Coordination via object level controllers

To eliminate meta-level nesting, a coordinator can be replaced by a controller object where the base objects are adapted to communicate via the controller and to keep track of their local time and status. This transformation is independent of policy or object system. Each coordinator rule is split into rules for the controller and the adapted object.

A controller (class `Ctl`) has `policy` and `events` attributes as for a coordinator. In addition it has a `pend` attribute that represents the pending events of the coordinators configuration. At any given time, some pending events may still be in transit, i.e. in the configuration containing the control object. In addition there is a `wait4` attribute used to wait for the acknowledgment of a transmitted pending event, before proceeding. The controller only sends a pending event when the message is deliverable.

```
rl[ctl.snd]:
[c : Ctl | pend: M m, events: ed, policy: P, wait4: none
         | inQ, outQ]
=>
[c : Ctl | pend: M, events: ed, policy: P, wait4: m
         | inQ, outQ m]  if ed,M,m |= P
```

When an acknowledgment of a pending event is received, it contains the local time of the object when the message was delivered and the controller adds the corresponding event to its event diagram attribute.

```
rl[ctl.ack]:
[c : Ctl | pend: M, events: ed, policy: P, wait4: m
         | (c<-ack(m,j) inQ, outQ ]
=>
[c : Ctl | pend: M, events: ed e, policy: P, wait4: none
         | inQ, outQ]
if (a<-mb,b,i) := m  /\ e := (mb,a,j,b,i)
```

Pending events received by a controller are added to its pending events set.

```
rl[ctl.rcv]:
[c : Ctl | pend: M, events: ed, policy: P, wait4: w
         | (c<-snd(m)) inQ, outQ]
=>
[c : Ctl | pend: M m, events: ed, policy: P, wait4: w
         | inQ, outQ]
```

The base objects are adapted by wrapping them in a *customizer* object with the same identifier, that performs the additional bookkeeping and reroutes messages through the controller. Specifically, a coordinated object customizer has a configuration attribute containing a single annotated object, and an attribute `ctl` whose value is the controller identifier.

The customizer rule `[cust.in]` together with the controller rules `[ctl.snd]` and `[ctl.ack]` implement the coordinator rule `coord.in`. Since the controller only sends a pending event when the message is deliverable, the customizer puts the message in the object input queue and acknowledges receipt adding the local delivery time.

```
rl[cust.in]:
[a : CA | {[a : A | atts | nil,nil,j,ready]}, ctl: c
        | (a<-mb : b,i) inQ, outQ]
=>
[a : CA | {[a : A | atts | a<-mb,nil,s j,busy]}, ctl: c
        | inQ, outQ c<-ack((a<-mb : b,i),j) ]
```

The customizer rule `[cust.out]` together with the controller rule `[ctl.rcv]` implement the coordinator rule `[coord.out]`.

```
rl[cust.out]:
[a : CA | {[a : A | atts | nil,mQ,j,busy]}, ctl: c | inQ, outQ]
=>
[a : CA | {[a : A | atts | nil,nil,j+k,ready]}, ctl: c
        | inQ, outQ outQ']
    if k := length(mQ)  /\ outQ' := mkSnd(mQ,j,c)
```

where `mkSnd(mQ,j,c)` is the set of messages `c<-snd(b<-mb : a,j+i)` such that `b<-mb` is the ith element of `mQ`, counting from 0. Thus the pending events generated by the coordinator rule `[coord.out]` are packaged and sent to the controller.

In fact, we have not completely eliminated nesting, however customized objects can be flattened by straightforward module transformations [10]. The transformation to customized objects plus controller is also a way to distribute the object system, however the single controller may not be the most suitable solution.

6 Related Work

There are numerous languages for specifying or programming coordination whose semantics has been studied by a variety of techniques. The approach of the work presented here is to start with a semantic model of distributed object coordination, focusing on interactions rather than system state, and study language independent coordination mechanisms specified in a general formal logic. Object behavior and coordination mechanisms are specified separately and composition operations can be studied in the same framework.

Tuple space languages include Linda [13] and its mobile extension, Lime [23]. Actor languages with coordination abstractions include Synchronizers [12,11] and Real-Time Synchronizers [24]. These languages provide linguistic constructs for specifying coordination and come with compilation transformations for implementation in terms of standard actors. Reo [4,3] is a channel based coordination model where complex coordinators called connectors are constructed by composing smaller one. A semantic model based on timed data streams and co-inductive reasoning principles is given in [2]. Klaim[21] is a process-algebra based language.

In [22] a methodology is proposed for designing coordination between agents among software agents. The methodology starts with requirements and refines through several stages. Here coordination is viewed as a dependency between agents rather that ordering of events. The methodology has associated graphical notation, but lacks formal semantics.

The Mobile Unity language provides coordination primitives as well as a logic for reasoning about Mobile Unity specifications. Refinement from a high-level logical specification to mobile unity code is illustrated in [25]. Coordination properties are based on system state rather than interaction events.

The WS-Coordination specification [16] describes an extensible framework for providing protocols that coordinate the actions of distributed applications. It focuses on issues such as initialization, registration and security.

7 Future Directions

We have defined a simple notion of coordination requirement based on event diagram semantics of an object system and executable specifications of policy-based

coordinators. These ideas were illustrated with simple examples and ideas for transformation to distributed coordinators and base-level controllers were sketched.

An obvious direction of future work is to develop refinement rules that guarantee preservation of requirements satisfaction for different classes of policy. Are there general principles for deriving a distributed coordination protocol? Can the localization transformation sketched for the case of group communication be generalized?

Another interesting direction is to consider notions of composition of coordination requirements or of coordinators. Several notions of composition for container meta-objects exist that can be explored. In addition, composition based on policy composition is another possibility. Here we can consider composition with policies other than coordination, such as security.

Finally, coordination that involves explicit time or use of resources is of interest. For this, the semantic model will need to be extended appropriately.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[2] F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. Technical Report SEN=R0216, Centrum voor Wiskunde en Informatica (CWI), 2002.

[3] Farhad Arbab. A chanel-based coordination model for component composition. Technical Report SEN=R0203, Centrum voor Wiskunde en Informatica (CWI), 2002.

[4] F. Arbib. Coordination of mobile components, 2001.

[5] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[6] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. http://maude.cs.uiuc.edu.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.

[9] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[10] G. Denker, J. Meseguer, and C. L. Talcott. Rewriting semantics of distributed meta objects and composable communication services. In *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[11] S. Frølund. *Coordinated Distributed Objects: An Actor Based Approach to Synchronization*. MIT Press, 1996.

[12] S. Frølund and G. Agha. A language framework for multi-object coordination. In *ECOOP 1993*, volume 707 of *Lecture Notes in Computer Science*. Springer, 1993.

[13] David Gelernter. Generative communication in linda. *TOPLAS*, 7(1):80–112, 1985.

[14] I. Greif. Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC, 1975.

[15] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*, pages 235–245, August 1973.

[16] IBM, Microsoft, and BEA Systems. Web services coordination, 2004.

[17] Narciso Marti-Oliet, Jose Meseguer, and Miguel Palomino. Rewriting logic and applications bibliography. *Theoretical Computer Science*, 285(2), 2002.

[18] J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[19] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36, 2002. invited paper.

[20] José Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pages 89–117. Kluwer, 2000.

[21] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[22] Anna Perini, Angelo Susi, and Fausto Giunchiglia. Coordination specification in multi-agent systems: from requirements to architecture with the Tropos methodology. In *14th international conference on Software engineering and knowledge engineering*, pages 51–54. ACM Press, 2002.

[23] G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *21 Int. Conf. on Software Engineering*, pages 368–377, 1999.

[24] Shangping. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[25] Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang. Formal specification and design of mobile systems. In *Formal Methods for Parallel Programming: Theory and Applications*, 2002.

[26] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.