Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 229 (2009) 95–113
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

The ARC Programming Model – Language Constructs for Coordination

Kevin Marth and Shangping Ren[1]

Department of Computer Science
Illinois Institute of Technology
10 West 31st Street, Chicago, IL 60616, USA

Abstract

The ARC (Actor, Role, Coordinator) model addresses the coordination requirements of open, distributed applications deployed in dynamic environments. This paper introduces the ARC programming model and the ARC-PL programming language, including the syntax and informal semantics of new language constructs designed to enable modular coordination in the ARC model. Several well-known problems are solved to illustrate the expressiveness and modularity of the ARC programming model.

Keywords: actor, coordination.

1 Introduction

As computing becomes pervasive, software applications are becoming more open and distributed. These open and distributed software applications must be concerned with the dynamic environments in which they are deployed. The environment of an open application is dynamic because computational entities are free to enter or leave the environment at any time. The number of computational entities present in the environment and interacting within an open application at any time may be quite large, so the application should be scalable and limited only by the resource constraints imposed by the environment. In addition, the functionality delivered by such applications is often subject to quality-of-service (QoS) requirements. To satisfy QoS requirements, strategies and policies that address concerns such as adaptation, evolution, security, reliability, fault-tolerance, and real-time constraints must be integrated into the applications. The integration of such concerns introduces the need for coordination among the autonomous and often asynchronous computational entities interacting within an open and distributed application.

[1] Email: `martkev@iit.edu`, `ren@iit.edu`

doi:10.1016/j.entcs.2009.06.031

Models such as CSP [15], the π-calculus [18], and the Actor model [1,2] are well-defined mathematical abstractions for concurrent and distributed computation. However, coordination is often realized in these computation models via message and behavior protocols tangled within the computational entities. Coordination may be modularized by composing a separate coordination model with the computation model. The Actor, Role, Coordinator (ARC) model [22] composes the Actor model of computation with an exogenous role-based coordination model that enables modular coordination of the asynchronous and autonomous actors in the basic Actor model, which we hereafter refer to as *basic actors*.

Coordination in the ARC model is distributed and is both local and non-local to basic actors. Local coordination enables a basic actor to use its internal state to determine when received messages may be dispatched to the actor for processing. Non-local coordination is based on the concept of a *role* and is partitioned into intra-role coordination and inter-role coordination, as illustrated in Fig. 1. A role is defined by an abstract behavioral interface. The dynamic nature of pervasive applications implies that an extensive number of ephemeral basic actors could be interacting within the application at any time. The stability and scalability of coordination policies are difficult to maintain if coordination is based solely on the identities of individual basic actors. Fortunately, the basic actors interacting within an open and distributed application typically play a limited number of well-known roles. Coordination based on roles is therefore relatively stable.

A basic actor must implement the abstract interface of a role in order for the basic actor to acquire membership in the role. A *role member actor* is a basic actor with membership in one or more roles. A *role actor* is a coordination actor that enacts intra-role coordination within its set of role member actors. Multiple role actors may exist for each role. A role actor may also serve as a parameter to a coordination actor called a *coordinator actor*. A coordinator actor enacts inter-role coordination upon role member actors via its role actor parameters and avoids persistent knowledge of individual role member actors.

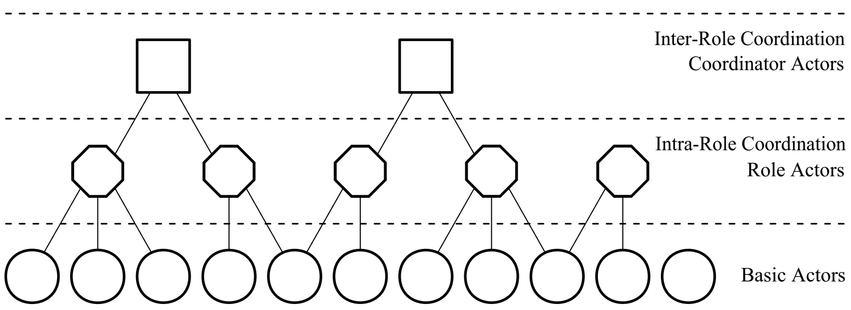

Fig. 1. The ARC Model

The coordination distributed to basic actors, role actors, and coordinator actors is examined in detail in this paper. We introduce the ARC programming model and the ARC Programming Language (ARC-PL), emphasizing the syntax and informal semantics of new language constructs designed to enable modular coordination in the ARC model. Section 2 discusses related work on coordination models. Sections

3, 4, and 5 discuss programming in the ARC model using several examples to introduce the syntax and semantics of language constructs designed to facilitate local coordination, intra-role coordination, and inter-role coordination, respectively. Section 6 concludes and discusses future work. The formal syntax of ARC-PL and its operational semantics are introduced in an appendix.

2 Related Work

Coordination has been an active area of research for over two decades. A broad survey [21] of coordination models and languages concluded that coordination models can be categorized as data-driven or control-driven. In data-driven models such as Linda [14] and its extensions, coordination tends to be endogenous and embedded within computational entities. In control-driven models, coordination tends to be exogenous and isolated from computational entities. Control-driven models such as ABT [4], ROAD [7], IWIM [3], and CoLaS [9] isolate coordination by considering functional entities as black boxes. Both IWIM and ABT address computation and coordination concerns in separate and independent levels. ABT treats both computation and coordination components as composable Abstract Behavior Types. Hybrid approaches such as tuple center [20] and ReSpecT [19] combine the data-driven and control-driven models.

Some control-driven models, such as ROAD, CoLaS, and Finesse [5], target the scalability issues of open systems through group-based coordination models. Most current role-based coordination models [6] are based on organizational concepts, where roles abstract coordination behaviors required of participants that play the roles. The ARC model uses roles to define behaviors required of participant actors but also introduces active, first-class entities (coordination actors) that enact coordination among participant actors in response to meta-level events.

Several coordination models address decentralization. TuCSoN [8] provides distributed tuple centers, each with its own local coordination rules. CoLaS partitions a distributed system into multiple coordination groups, and each coordination group enacts an independent set of coordination policies. ROAD provides a recursive structure that composes fine-grained coordination groups into coarse-grained groups. LGI [25] provides a controller for every object in the system and therefore implements completely decentralized coordination. The ARC model differs from these models by separating intra-role coordination and inter-role coordination and logically distributing the responsibility for coordination based on roles.

The ARC model is influenced by earlier coordination work based on the Actor model, including hierarchical coordination [23], multi-level meta architectures [24], and synchronizers [13]. However, the ARC model also differs significantly from earlier work. Synchronizers are closed in the sense that all participant actors must be individually specified when a synchronizer is instantiated, whereas role-based coordination is open, dynamic, and collectively based on actor behavior. Synchronizers coordinate existing messages sent by basic actors, whereas ARC coordination actors may also send messages required to enact coordination policies. The hierarchical coordination model intentionally avoids a meta-architecture and enacts coordination via hierarchical groupings of actors, but the groupings are not based on role behavior. The use of role-based coordination also distinguishes the ARC model from the multi-level meta architectures.

3 Local Coordination in the ARC Model

In the basic actor model, a message received in an actor's mailbox is dispatched to the actor for processing without regard for the current internal state of the actor. Consequently, a message that violates a local behavior invariant at the time of dispatch must be treated as an error or explicitly deferred. The logic required in such cases is often tedious, inelegant, and error-prone. Local coordination in the ARC model enables a basic actor to specify *when* a message is processed as well as *how* a message is processed while separating the two concerns. Local coordination is often a component of complex inter-actor coordination and is also sufficient by itself to solve interesting problems such as the bounded-buffer problem, which we consider below. The bounded-buffer problem requires mutual exclusion and conditional synchronization. The basic actor model guarantees mutual exclusion, since an actor processes its messages serially and to completion, while local coordination facilitates the required conditional synchronization.

The Behavior Construct

Local coordination in the ARC model is specified within the `behavior` constructs associated with basic actors, such as the `BoundedBuffer` behavior in Fig. 2. An ARC actor behavior is expressed as an Extended Finite State Machine (EFSM) [12] with one or more states. Each state has one or more inputs that correspond to messages received by the actor. The set of all state inputs for a behavior defines the public interface for the behavior. The public interface of the `BoundedBuffer` behavior is {`get`,`put`}. Execution begins in the `start` state of a behavior, where inputs are defined to process the `constructor` message dispatched to an actor upon instantiation, e.g. the message `constructor(10)` is dispatched to the actor instantiated by the expression `new BoundedBuffer[String](10)` to initialize a bounded-buffer with space for 10 strings. The value of the reserved `state` variable specifies the current behavior state. A state transition occurs immediately after the `state` variable is assigned. At transition to a state, optional entry logic specified preceding the state inputs is executed. The current state is unchanged after an input if the `state` variable is not assigned while processing the input.

Expressing actor behaviors as EFSMs enables seamless integration of local coordination via the message dispatch constraints explicit in the state inputs of a behavior. Specifically, a message received by an actor may be dispatched and processed by the actor only when the current state of the actor behavior defines an input that corresponds to the message. An actor executing the `BoundedBuffer` behavior may dispatch a `get` message only in the `fill` or `full` states and may dispatch a `put` message only in the `fill` or `empty` states. Thus, a declarative specification of dispatch constraints replaces the error handling or explicit buffering required in the original Actor model. Messages received by an ARC basic actor are implicitly buffered in the actor's mailbox in the order they are received until local dispatch constraints allow the messages to be dispatched.

The formal syntax of the ARC-PL `behavior` construct is specified in Appendix A. The syntax of ARC-PL is influenced by C++, while Pascal influenced the syntax where C++ has been criticized, e.g. variable declaration and assignment operators.

```
behavior BoundedBuffer[T]
{
  // The ARC model provides predefined collection types,
  // e.g. Queue, with synchronous call/return semantics.
  queue:Queue[T];
  state start {
    input constructor(space:integer) {
      queue := new Queue[T](space); state := empty;
    }
  }
  state empty {
    input put(item:T) {
      queue.push_rear(item); state := queue_full() ? full :: fill;
    }
  }
  state fill {
    input get(consumer:Consumer) {
      send_item(consumer); if (queue_empty()) state := empty;
    }
    input put(item:T) {
      queue.push_rear(item); if (queue_full()) state := full;
    }
  }
  state full {
    input get(consumer:Consumer) {
      send_item(consumer); state := queue_empty() ? empty :: fill;
    }
  }
  // local functions have synchronous call/return semantics
  boolean queue_empty() { return queue.size() = 0; }
  boolean queue_full()  { return queue.size() = queue.space(); }
  void send_item(consumer:Consumer) {
    item:T := queue.head(); queue.pop_head(); consumer.consume(item);
  }
}
```

Fig. 2. Bounded-Buffer Behavior in ARC-PL

4 Intra-Role Coordination in the ARC Model

In the ARC model, basic actors may be organized in role-based sets to enact non-local coordination. A basic actor may become a role member if the public interface of the actor behavior implements the public interface required by the role. Multiple role actors may exist for each role. A role actor is instantiated from the `role` construct that defines the role. A role actor coordinates membership in its open and dynamic set of role member actors and also enacts intra-role coordination upon its role member actors.

We illustrate intra-role coordination in the ARC model by implementing a GUI selection list of items. Each list item is either selected or unselected. Selecting an item toggles the selection state of the item and deselects any previous selection in the list. Thus, the list may have either no selection or a single selection. The solution illustrated in Fig. 3 defines the `Item` behavior for selection items and the `ListItem` role to coordinate a list of selection items.

```
behavior Item
{
  state start { input constructor() state := unselected; }
  state unselected { input toggle() state := selected;   }
  state selected   { input toggle() state := unselected; }
  state selected, unselected {
    input asListItem(LI:ListItem[Item]) LI.join(self, state = selected);
    input join(LI:ListItem[Item], selection:boolean)
      state := selection ? selected :: unselected;
  }
}

role ListItem[Item]
{
  join(sender:Item, selected:boolean); selection:Item := null;
  state start { input constructor(); }
  event [self <- join(item:Item, selected:boolean)] {
    selected := selected & (selection = null);
    dispatch { item.join(self, selected); }
    if (selected) selection := item;
  }
  event [item <- toggle()] {
    if (selection != null & selection != item) {
      dispatch { item.toggle(); selection.toggle(); } selection := item;
    }
    else if (selection = item) { dispatch { item.toggle(); } selection := null; }
    else if (selection = null) { dispatch { item.toggle(); } selection := item; }
  }
}
```

Fig. 3. GUI Selection List in ARC-PL

4.1 The Role Construct

We examine the syntax and semantics of the `role` construct in this subsection. The formal syntax of the `role` construct is specified in Appendix A.

4.1.1 Role Signature

The signature of a `role` construct, e.g. `ListItem[Item]`, specifies the name of the role and the abstract behavior required of role member actors. The abstract behavior `Item` specified by the `ListItem` role requires that role member actors implement the public interface of the `Item` behavior, i.e. {`toggle`}. Basic actors with the `Item` behavior trivially satisfy this requirement, but other basic actor behaviors whose public interfaces include the `toggle` message would also satisfy the abstract behavior requirement.

4.1.2 Role Membership

A basic actor explicitly acquires membership in the set of role member actors coordinated by a role actor by exchanging the `join` message with the role actor. The signature of all `join` messages requires the actor sending the message as the first parameter, and a role may specify additional `join` parameters required to administer role membership. The `join` message defined by the `ListItem` role requires an additional Boolean `selected` parameter as explained below. A basic actor explicitly releases a role membership by exchanging the `exit` message with the associated role actor. We omit the `exit` details in this example, as they can be inferred from the discussion of the `join` details.

4.1.3 Role State

The coordination policies enacted by role actors are state-based and may vary over time. The state of a role actor is composed of the local variables defined in the associated `role` construct and the dynamic set of role member actors coordinated by the role actor. A `ListItem` role actor uses its local variable `selection` to identify the role member `Item` actor that is the current selection within the selection list. The coordination enacted by a `ListItem` role actor is based upon the value of the current list selection.

4.1.4 Role Behavior

Role actors are hybrid actors with capabilities of both basic actors and meta actors. As a basic actor, a role actor may exchange `join` and `exit` messages with role member actors. As a meta actor, a role actor is able to react to meta-level messages, called *events*, that involve itself or the role member actors currently coordinated by the role actor. A role actor begins execution in the `start` state defined in the associated `role` construct. After processing a `constructor` message in the `start` state, the behavior of a role actor transitions to an implicit event-processing loop where events are processed serially and to completion. Events may be triggered for a role actor in three scenarios.

- The role actor receives a `join` or `exit` message in its mailbox.
- A role member actor receives a message in its mailbox.
- A role member actor sends a message.

4.1.5 Intra-Role Event Processing

The events of interest to a role actor are identified in the associated `role` construct. Each event has a signature that identifies essential information about the event.

- The event is either a receive event or a send event.
- The event has a focus message identified by a specific message signature.
- The event has a focus actor, either the role member actor that received the focus message or the role member actor that sent the focus message.

The parameters in the focus message signature are initialized from the values transmitted in the message and are available as local variables when reacting to the event. Events may be constrained by specifying an optional Boolean condition based on local variables. If the Boolean condition for an event is omitted, the default condition `if (true)` is assumed. Only messages that satisfy the Boolean condition will trigger events. A message that triggers a receive event is discarded from the mailbox of the focus actor after triggering the event and is therefore never dispatched and processed by the focus actor.

Two events are of interest to a `ListItem` role actor.

(i) The event `[self <- join(item:Item, selected:boolean)]` is triggered by the focus actor `self` (the `ListItem` role actor) receiving a `join` message from the `Item` actor specified by the `item` parameter. The `selected` parameter specifies whether the `item` is selected at the time role membership is requested.

(ii) The event `[item <- toggle()]` is triggered by a role member `Item` actor receiving a `toggle` message. The focus actor is specified by `item`.

The logic to process the second event has three mutually exclusive alternatives.

(i) The current list selection is non-null, and the `item` that received a `toggle` message is not the current selection.

(ii) The current list selection is the same `item` that received a `toggle` message.

(iii) The current list selection is `null`.

4.1.6 The Dispatch Primitive

The coordination logic enacted by role actors often includes the sending and atomic dispatch of actor messages. Modular coordination is facilitated if role actors have the capability to send actor messages, since role member actors would otherwise be required to send the messages relevant in coordination scenarios, which implies that coordination would be tangled within basic actor behaviors. The capability to atomically dispatch a set of messages to a set of actors is fundamental to many coordination scenarios. The ARC programming model provides a novel synthesis of message send and dispatch capabilities via the `dispatch` primitive. The body of a `dispatch` statement executed by a role actor must be composed of a sequence of message sends to a set of target role member actors. The messages are atomically sent and dispatched using an adapted three-phase commit protocol.

(i) In the first phase, all target actors are locked by the role actor, ensuring the target actors will dispatch only messages sent by the role actor until the target actors are unlocked. Target actors are locked by the ARC-PL implementation sequentially and in the order determined by their respective identifiers. A lock on a target actor completes only when the target actor is in a state that defines an input for its associated target message. Several deadlock and livelock scenarios are therefore avoided. A total order of actor identifiers may be achieved without global synchronization, e.g. by using the host machine address and a local sequence number to generate an identifier when an actor is instantiated.

(ii) In the second phase, the sequence of messages is sent by the role actor to the target actors, and the received messages are dispatched by the target actors. Checking at run-time ensures that only one message is sent to each target actor within a given `dispatch` statement. Since the actor model guarantees reliable message delivery, and the target actors are locked and sensitized to input messages from the role actor, the second phase must ultimately complete.

(iii) In the third phase, the target actors are unlocked in the order determined by their respective identifiers, execution of the `dispatch` statement is complete, and the role actor continues execution after the `dispatch` statement.

For example, the `dispatch { item.toggle(); selection.toggle(); }` statement used in the first alternative of the `ListItem` event `[item <- toggle()]` atomically toggles the two `Item` role member actors indicated by `item` and `selection`. The atomic action results in a new list selection, since the actor indicated by `selection` is deselected after dispatching the `toggle` message it is sent, and the actor indicated by `item` is selected after dispatching the `toggle` message it is sent. The `ListItem` role actor therefore updates its local `selection` variable upon completion of the `dispatch` statement.

The capability to send messages via the `dispatch` statement distinguishes ARC coordination actors from constructs (e.g. the `synchronizer`) that coordinate only existing messages sent by basic actors. The tension between the capability to send messages and the separation of computation and coordination is tempered by ensuring that coordination actors send messages only to role members in reaction to events. A coordination actor does not initiate arbitrary computation.

4.2 Role-Aware Actor Behaviors

Since role membership is acquired and released via explicit message exchange between a basic actor and a role actor, basic actor behaviors must be aware of roles, and ARC-PL must ensure that role membership is processed atomically. Role membership capabilities are integrated into the EFSM of a basic actor behavior via the designated `join` and `exit` messages discussed above. Like other messages, the `join` and `exit` messages are dispatched only in states that define inputs for the messages. A basic actor is implicitly locked and sensitized to a role actor immediately after the basic actor sends a `join` or `exit` message to the role actor, effectively suspending the basic actor (after any further processing of the current input) until the `join` or `exit` reply message is received from the role actor.

For example, the `Item` behavior defines the input `asListItem` in the `selected` and `unselected` states. The parameter `LI` in the `asListItem` message specifies the `ListItem` role actor of interest. In the `join` message sent to the role actor to acquire role membership, the `selected` parameter specifies whether the `Item` actor is currently selected. The `selected` parameter is used when the role actor processes the subsequent `join` receive event. If the `Item` actor is selected when requesting role membership and the selection list has no current selection, the `Item` actor may remain selected and become the current list selection. Otherwise, the `Item` actor must transition to the `unselected` state upon dispatching the `join` message sent by the role actor to confirm role membership. The `Item` actor is implicitly locked and sensitized to the role actor immediately after sending the `join` message to the role actor and is implicitly unlocked after dispatching the `join` message sent by the role actor to confirm role membership.

The treatment of roles within the ARC programming model enables basic actor behaviors to become role-aware in an incremental and modular fashion as an application evolves. Existing behavior often remains oblivious to role memberships, and a behavior need not be aware of all roles it will ultimately be asked to assume when the behavior is initially specified.

4.3 Multiple Role Memberships

A basic actor may be a member of multiple roles simultaneously. A behavior has conflicting role memberships if the behavior permits multiple role memberships and two or more of the roles define a receive event with the same focus message. An ARC-PL static semantic check disallows behaviors with conflicting role memberships. Role conflicts must be resolved by defining a composite role in which overlapping events defined by the conflicting roles are appropriately unified. The decision to disallow conflicting role memberships is motivated by the fact that the trigger condition for a receive event may depend on variables local to individual role actors. Without conflicting roles, a message will be processed by the basic actor that received the message if a receive event is not triggered at a role actor. If conflicting roles are allowed, an event may be triggered at one role actor but not at a second role actor, resulting in an undesirable scenario in which the first role actor assumes the received message is not processed by the basic actor, while the second role actor assumes the received message is processed by the basic actor.

5 Inter-Role Coordination in the ARC Model

Inter-role coordination is necessary when the coordinated interaction of multiple roles is required. We illustrate inter-role coordination in the ARC model by solving a version of the dining philosophers problem in which several philosophers seated around a table alternate between dining and thinking. A philosopher must atomically acquire assigned tableware before dining and release the tableware after dining. We extend the traditional problem and allow the tableware assigned to a philosopher to change dynamically to emphasize the strengths of role-based coordination. The ARC solution is structured using the roles of `Process` and `Resource`, as illustrated in Fig. 4 for philosophers one and two.

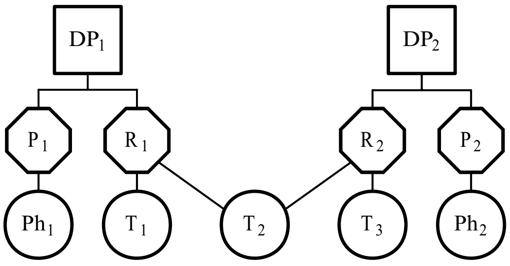

Fig. 4. The Dining Philosophers Problem in the ARC Model

- Each basic `Philosopher` actor ($\texttt{Ph}_i$) plays the role of a `Process` and is coordinated by a dedicated `Process` role actor ($\texttt{P}_i$).
- Each tableware utensil ($\texttt{T}_j$) plays the role of a shared `Resource`. The tableware (e.g. fork, spoon, chopstick) required by each philosopher is coordinated by a dedicated `Resource` role actor ($\texttt{R}_i$). Each tableware utensil (e.g. $\texttt{T}_2$) is coordinated by multiple `Resource` role actors, reflecting the inherent resource contention.

- The `Process` role actor and the `Resource` role actor for each philosopher are coordinated by a dedicated `Dining Philosopher` coordinator actor ($\mathtt{DP}_i$).

Before considering the required inter-role coordination, we summarize the behaviors of basic `Philosopher` and `Tableware` actors illustrated in Fig. 5. A `Tableware` actor alternates between its `acquired` and `released` state and may acquire `Resource` role membership only when in the `released` state. The behavior of a `Philosopher` actor is expressed as the following EFSM.

- The philosopher transitions from the `start` state to the `thinking` state after acquiring `Process` role membership.
- At transition to the `thinking` state, the philosopher sets a timer to expire after a random interval of thinking. When the timer expires, a message (T) is dispatched to the philosopher, and the philosopher transitions to the `hungry` state.
- At transition to the `hungry` state, the philosopher sends an `acquire` message to `self` with an empty set (`#{}`) of tableware as a parameter. The empty `acquire` message will trigger an event (detailed below) at its dedicated `Process` role actor that will ultimately result in the dispatch of an `acquire` message to the philosopher with the appropriate set of tableware. The philosopher transitions to the `dining` state after acquiring its tableware.
- At transition to the `dining` state, the philosopher sets a timer to expire after a random interval of dining. When the timer expires, a message (T) is dispatched to the philosopher, and the philosopher releases the set of acquired tableware and returns to the `thinking` state.

5.1 The Coordinator Construct

A coordinator actor is instantiated from the associated `coordinator` construct. We examine the syntax and semantics of the `coordinator` construct in this subsection. The formal syntax of the `coordinator` construct is specified in Appendix A.

5.1.1 Coordinator Behavior

A coordinator actor, like a role actor, is a hybrid actor that enacts state-based coordination policies while executing an implicit event-processing loop. As a meta actor, a coordinator actor is able to react to events triggered by the role actors specified as parameters in the `constructor` message received by the coordinator actor during instantiation. As illustrated in Fig. 5, each `DiningPhilosopher` coordinator actor is instantiated with a `Process` role actor and a `Resource` role actor. A coordinator actor begins execution in the `start` state defined in the associated `coordinator` construct. After processing a `constructor` message in the `start` state, the behavior of a coordinator actor transitions to an implicit event-processing loop where events are processed serially and to completion. Events may be triggered for a coordinator actor only when a coordinated role actor receives a message other than a `join` or `exit` message. This constraint implies that coordinator events are triggered only when a coordinated role actor receives a message sent to itself.

```
behavior Philosopher
{
  tableware:Set[Tableware];
  state start { input constructor(); }
  state thinking { T:Timer; T.expire(random()); input T() state := hungry; }
  state hungry { self.acquire(#{}); input acquire(tableware) state := dining; }
  state dining {
    T:Timer; T.expire(random()); input T() { @tableware.release(); state := thinking; }
    // The @ operator iterates over a set, applying a specified method to each element.
  }
  state start {
    input asProcess(process:Process[Philosopher]) process.join(self);
    input join(process:Process[Philosopher]) state := thinking;
  }
}

behavior Tableware
{
  state start { input constructor() state := released; }
  state released {  input acquire() state := acquired; }
  state acquired {  input release() state := released; }
  state released {
    input asResource(resource:Resource[Tableware]) resource.join(self);
    input join(resource:Resource[Tableware]);
  }
}

role Process[Philosopher]
{
  join(sender:Philosopher);
  state start { input constructor(); }
  event [self <- join(philosopher:Philosopher)] { dispatch { philosopher.join(self); } }
  event [philosopher <- acquire(tableware:Set[Tableware]) if (tableware = #{})] {
    coordinator; self.acquire(#{});
  }
}

role Resource[Tableware]
{
  join(sender:Tableware);
  state start { input constructor(); }
  event [self <- join(tableware:Tableware)] { dispatch { tableware.join(self); } }
}

coordinator DiningPhilosopher
{
  process:Process[Philosopher]; resource:Resource[Tableware];
  state start { input constructor(process, resource); }
  event [process <- acquire(tableware:Set[Tableware])] {
    philosopher:Philosopher  := process.each().only(); // access singleton set member
    tableware:Set[Tableware] := resource.each();
    dispatch { philosopher.acquire(tableware); @tableware.acquire(philosopher); }
  }
}
```

Fig. 5. Dining Philosophers in ARC-PL

5.1.2 Inter-Role Event Processing

The events of interest to a coordinator actor are identified in its `coordinator` construct. The single event of interest to a `DiningPhilosopher` coordinator actor is triggered when its `Process` role actor receives an `acquire` message with an empty set of tableware. As described above, a basic `Philosopher` actor sends an `acquire` message with an empty set of tableware to itself at transition to the `dining` state. This message is discarded upon triggering an event at the `Process` role actor coordinating the philosopher. The `Process` role actor reacts to the event by executing the `coordinator` statement and then sending an empty `acquire` message to itself, thereby propagating the event to its `DiningPhilosopher` coordinator actor. The

`Process` role actor is implicitly locked and sensitized to its `DiningPhilosopher` coordinator actor by executing the `coordinator` statement, effectively suspending the `Process` role actor and blocking the processing of any additional events until the `Process` role actor is unlocked by its `DiningPhilosopher` coordinator actor.

The coordination enacted by coordinator actors often includes the sending and atomic dispatch of actor messages. However, scalability concerns dictate that coordinator actors not have persistent knowledge of individual role member actors. Instead, two queries with synchronous call/return semantics are implicitly exported by a coordinated role actor to its coordinator actor.

- The `each` query returns the current set of role member actors.
- The `some` query returns a subset of the current set of role member actors.

A coordinated role actor may be queried by a coordinator actor only when the role actor is locked and sensitized to the coordinator actor. Each role actor has a reserved `each` variable and a reserved `some` variable. The value of the `each` variable is implicitly updated whenever the role actor processes a role membership event triggered by a `join` or `exit` message. The value of the `some` variable may be maintained by the role actor to select a subset of current role member actors based on a local policy. The `each` and `some` queries simply return the current values of the `each` and `some` variables, respectively. The use of queries enables a `DiningPhilosopher` coordinator actor to obtain a snapshot of the `Tableware` actors currently required by a `Philosopher` actor and then atomically dispatch `acquire` messages to the `Philosopher` actor and its required `Tableware` actors. A snapshot of role member actors should always remain local to a single event at a coordinator actor, since persistent knowledge of role member actors at a coordinator actor is inconsistent with open and dynamic role-based coordination.

The requirement that coordination be indivisible is fundamental. While an event is being processed, no intermediate states can be visible or accessible, and information required to process an event must remain valid throughout the event. Thus, all role actors specified as parameters to a coordinator actor are implicitly locked when the coordinator actor is processing an event, ensuring the role actors defer additional events, including role membership events, until the coordinator event is processed. An attempt by one actor to lock a second actor that is already locked by the first actor has no effect in the ARC programming model. As a result, deadlock does not occur when a role actor that has previously executed a `coordinator` statement to lock and sensitize itself to a coordinator actor is again locked by the coordinator actor prior to processing an event triggered by the role actor. Indivisibility is ensured in the ARC-PL solution to the dining philosophers problem through the locking of role actors and the combination of the `dispatch` primitive and local coordination. Specifically, the `dispatch` primitive ensures the atomic dispatch of the `acquire` messages sent by a `DiningPhilosopher` coordinator actor to a `Philosopher` actor and its required `Tableware` actors, and local coordination ensures that each `Tableware` actor dispatches its `acquire` message only in the `released` state.

5.2 Advantages of Role-Based Coordination

The advantages of role-based coordination are evident in the dynamic version of the dining philosophers problem illustrated above, where role-based coordination enables a `Philosopher` actor to acquire an open and dynamic set of `Tableware` actors during the course of execution. A `Philosopher` actor has explicit knowledge of acquired `Tableware` actors only in the `dining` state, and the set of `Tableware` actors acquired by a `Philosopher` actor may change at each transition to the `dining` state. For example, each `Philosopher` actor may initially require two chopsticks for dining. A subsequent requirement for each `Philosopher` actor to additionally acquire a globally shared spoon could be dynamically satisfied simply by directing the spoon to acquire membership with the `Resource` role actors dedicated to the `Philosopher` actors. The openness of role-based coordination also facilitates the satisfaction of QoS requirements in dynamic environments. For example, a more reliable version of the dining philosophers problem is possible with enhanced coordination that addresses the dynamic replacement of a broken chopstick. Further, the coordination is modular and not tangled in the `Philosopher` and `Tableware` behaviors, enabling each behavior to be expressed as a simple and concise EFSM that is oblivious to the myriad coordination details inherent in the atomic acquisition of a potentially dynamic set of shared resources. Implicit locking and the avoidance of some deadlock and livelock scenarios enhance the quality of open and distributed applications and the productivity of the software engineers developing such applications.

6 Conclusion and Future Work

In this paper, we have introduced the ARC programming model and used several examples to survey the syntax and informal semantics of the ARC-PL programming language, including new language constructs designed to facilitate local, intra-role, and inter-role coordination in the ARC programming model. The examples demonstrate the expressiveness, modularity, and simplicity of the language constructs and illustrate how role-based coordination satisfies the coordination requirements of an open and distributed application deployed in a dynamic environment.

We used the `Maude` system [17] to specify the formal executable semantics of ARC-PL. An introduction to the ARC-PL operational semantics is included in Appendix B. We are developing a compiler for ARC-PL that targets the executable Maude specification as well as the UML profile for SDL [16]. The Maude target will serve as the reference implementation of the ARC programming model and the definitive meaning of an ARC-PL program. An intuitive mapping to the UML SDL profile is attractive to software engineers developing applications that reflect the convergence of telecommunications and pervasive computing. In the future, we plan to continue research concerning the satisfaction of complex QoS requirements in the ARC programming model and explore a formal calculus for role composition based on directions suggested by the Traits model [11].

References

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

[2] Agha, G., I. A. Mason, S. F. Smith and C. L. Talcott, *A foundation for actor computation*, Journal of Functional Programming (1997), pp. 1–72.

[3] Arbab, F., *The IWIM model for coordination of concurrent activities*, in: *COORDINATION*, 1996, pp. 34–56.

[4] Arbab, F., *Abstract behavior types: a foundation model for components and their composition*, Science of Computer Programming (2005), pp. 3–52.

[5] Berry, A. and S. M. Kaplan, *Open, distributed coordination with finesse*, in: *SAC*, 1998, pp. 178–184.

[6] Cabri, G., L. Ferrari and F. Zambonelli, *Role-based approaches for engineering interactions in large-scale multi-agent systems*, in: *SELMAS*, 2003, pp. 243–263.

[7] Colman, A. and J. Han, *Coordination systems in role-based adaptive software*, in: *COORDINATION*, 2005, pp. 63–78.

[8] Cremonini, M. et al., *Coordination and access control in open distributed agent systems: The tucson approach*, in: *COORDINATION*, 2000, pp. 99–114.

[9] Cruz, J. C. and S. Ducasse, *A group based approach for coordinating active objects*, in: *COORDINATION*, 1999, pp. 355–370.

[10] d'Amorim, M. and G. Rosu, *An equational specification for the scheme language*, Journal of Universal Computer Science **11** (2005), pp. 1327–1348.
URL http://www.jucs.org/jucs_11_7/an_equational_specification_for

[11] Ducasse, S., O. Nierstrasz, N. Schärli, R. Wuyts and A. P. Black, *Traits: A mechanism for fine-grained reuse*, ACM TOPLAS **28** (2006), pp. 331–388.

[12] Ellsberger, J., D. Hogrefe and A. Sarma, "SDL - Formal Object-Oriented Language for Communicating Systems," Prentice Hall, 1997.

[13] Frølund, S., "Coordinating Distributed Objects: An Actor Based Approach to Synchronization," MIT Press, 1996.

[14] Gelernter, D., *Generative communication in linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[15] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

[16] International Telecommunication Union, *Recommendation Z.109: SDL Combined with UML* (2007).
URL http://www.itu.int/rec/T-REC-Z.109/en

[17] Maude System (2008).
URL http://maude.cs.uiuc.edu

[18] Milner, R., "Communicating and Mobile Systems: the Pi-Calculus," Cambridge University Press, 1999.

[19] Omicini, A. and E. Denti, *Formal ReSpecT*, in: *Declarative Programming – Selected Papers from AGP'00*, Electronic Notes in Theoretical Computer Science **48**, Elsevier Science, 2001 pp. 179–196.

[20] Omicini, A. and E. Denti, *From tuple spaces to tuple centres*, Science of Computer Programming (2001).
URL citeseer.ist.psu.edu/denti99from.html

[21] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, Advances in Computers (1998), pp. 330–401.

[22] Ren, S. et al., *Actors, roles and coordinators - a coordination model for open distributed and embedded systems*, in: *COORDINATION*, 2006, pp. 247–265.

[23] Varela, C. A. and G. Agha, *A hierarchical model for coordination of concurrent activities*, in: *COORDINATION*, 1999, pp. 166–182.

[24] Venkatasubramanian, N. and C. L. Talcott, *Reasoning about meta level activities in open distributed systems*, in: *PODC*, 1995, pp. 144–152.

[25] Zhang, W., C. Serban and N. H. Minsky, *Establishing global properties of multi-agent systems via local laws*, in: *E4MAS*, 2006, pp. 170–183.

Appendix

A Formal Syntax

We define the formal syntax of the `behavior`, `role`, and `coordinator` constructs using the following notation.

- Grammar terminals appear in bold font, e.g. `terminal`.
- Grammar nonterminals appear in italic font, e.g. *nonterminal.*
- The symbol | separates grammar alternatives; other symbols denote themselves.
- The notation $\mathit{nonterminal}^{?}$ denotes an optional instance of *nonterminal.*
- The notation $\mathit{nonterminal}^{*}_{,}$ denotes zero or more (comma-separated) instances of *nonterminal.*
- The notation $\mathit{nonterminal}^{+}_{,}$ denotes one or more (comma-separated) instances of *nonterminal.*

$$
\begin{array}{rcl}
\mathit{behavior} & ::= & \texttt{behavior}\ \mathit{type}\ \{\ \mathit{unit\text{-}b}^{*}\ \} \\
\mathit{role} & ::= & \texttt{role}\ \mathit{type}\ \{\ \mathit{unit\text{-}rc}^{*}\ \} \\
\mathit{coordinator} & ::= & \texttt{coordinator}\ \mathit{type}\ \{\ \mathit{unit\text{-}rc}^{*}\ \} \\
\mathit{unit\text{-}b} & ::= & \mathit{state} \mid \mathit{routine} \mid \mathit{variable\text{-}d} \\
\mathit{unit\text{-}rc} & ::= & \mathit{unit\text{-}b} \mid \mathit{event} \mid \mathit{signature\text{-}d} \\
\mathit{state} & ::= & \texttt{state}\ \mathit{name}^{+}_{,}\ \{\ \mathit{statement}^{*}\ \} \\
\mathit{routine} & ::= & \mathit{type}\ \mathit{signature}\ \{\ \mathit{statement}^{*}\ \} \\
\mathit{signature} & ::= & \mathit{name}\ (\ \mathit{variable}^{*}_{,}\ ) \\
\mathit{signature\text{-}d} & ::= & \mathit{signature}\ ; \\
\mathit{variable} & ::= & \mathit{name} \mid \mathit{name}\ :\ \mathit{type} \\
\mathit{variable\text{-}d} & ::= & \mathit{name}\ :\ \mathit{type}\ ; \\
 & \mid & \mathit{name}\ :\ \mathit{type}\ \texttt{:=}\ \mathit{expression}\ ; \\
\mathit{event} & ::= & \texttt{event}\ [\ \mathit{name}\ \texttt{<-}\ \mathit{signature}\ \mathit{constraint}^{?}\ ]\ \mathit{statement} \\
 & \mid & \texttt{event}\ [\ \mathit{name}\ \texttt{->}\ \mathit{signature}\ \mathit{constraint}^{?}\ ]\ \mathit{statement} \\
\mathit{constraint} & ::= & \texttt{if}\ (\ \mathit{expression}\ ) \\
\mathit{statement} & ::= & \mathit{expression}^{?}\ ; \\
 & \mid & \mathit{variable\text{-}d} \\
 & \mid & \{\ \mathit{statement}^{*}\ \} \\
 & \mid & \texttt{input}\ \mathit{signature}\ \mathit{statement} \\
 & \mid & \texttt{dispatch}\ \{\ \mathit{statement}^{*}\ \} \\
 & \mid & \texttt{if}\ (\ \mathit{expression}\ )\ \mathit{statement} \\
 & \mid & \texttt{if}\ (\ \mathit{expression}\ )\ \mathit{statement}\ \texttt{else}\ \mathit{statement} \\
 & \mid & \texttt{return}\ \mathit{expression}^{?}\ ; \\
 & \mid & \mathit{statement\text{-}case}
\end{array}
$$

$$\begin{array}{rl} & \mid \ \textit{statement-loop} \\ \textit{type} ::= & \textit{name} \\ & \mid \ \textit{name} \ [\ \textit{type}^{+}_{,}\ ] \\ & \mid \ \texttt{void} \\ & \mid \ \texttt{byte} \\ & \mid \ \texttt{boolean} \\ & \mid \ \texttt{unicode} \\ & \mid \ \texttt{integer} \\ & \mid \ \texttt{real} \end{array}$$

The ARC-PL *expression* syntax is similar to the expression syntax in the `C` programming language. The *statement-case* syntax and the *statement-loop* syntax are omitted, since these selection and iteration statements are not used in the examples in this paper.

B Operational Semantics

The `Maude` system [17] was used to specify the operational semantics of ARC-PL. The `Maude` specification of ARC-PL operational semantics includes both equational theories and rule-based rewrite theories that are applied to the term that represents an ARC-PL computation. The rewrite rules are applied modulo the equations by the `Maude` engine, ensuring that the term is simplified to canonical form via the equations before and after each application of a single rewrite rule to the term. This rewrite strategy enables the convenient expression of concurrency. Operations that must potentially be interleaved to correctly model concurrency are defined by rules, while operations that have no observable impact on concurrency may be defined by equations, since the intermediate states of the term during the successive application of multiple equations need not be visible. In the ARC model, as in the Actor model, concurrency exists among actors, but each individual actor executes within a dedicated sequential thread of control. The semantics of ARC-PL routines, statements, and expressions is therefore a sequential equational semantics based on continuation-passing style that is similar to the semantics defined in [10].

Concurrency is expressed using conditional rewrite rules with actors, messages, and threads. An actor is defined as `< A | B | E | L | R | I >`, where `A` is the unique identity of the actor, `B` is the behavior of the actor, `E` is the initial environment for the actor, `L` is the coordination actor that has locked the actor (the actor is unlocked when `L` is `'`), `R` is the role member set associated with the actor, and `I` is the input message set that will trigger coordination events. A message is defined as `< S -> A . N : VL >`, where `S` is the actor that sent the message, `A` is the actor that will receive the message, `N` is the name of the message, and `VL` is the list of message parameter values. A thread is defined as `thread[A | M]{C}`, where `A` is the actor associated with the thread, `M` is the thread-local memory for the thread, and `C` is the current continuation for the thread. The composition of an environment and memory implements an imperative store. An environment maps an identifier to a location in memory, and the memory maps a location to a value.

Conditional rules use the following syntax to specify that *term-1* is rewritten to *term-2* if the *condition* is true.

`crl [`*`label`*`] :  `*`term-1`*` => `*`term-2`*` if `*`condition`*` .`

We present several examples of conditional rewrite rules in the rest of this section, including message dispatch to a basic actor, locking a basic actor, and triggering an event at a role actor when a role member actor receives a message.

The conditional rule in Fig. B.1 is applied when an actor (`A`) with the `Item` behavior described in section 4 is waiting for input, receives a `toggle` message, and the *condition* succeeds because the following are true.

- The value of the `state` variable for actor `A` is `unselected`.
- The sender (`S`) of the `toggle` message is the same actor (`L`) that has locked actor `A` because an event is in progress, or the `toggle` message is not a member of the input message set (`I`) that triggers events.

When the rule is applied, the `toggle` message is dispatched to actor `A` and consumed by executing an anonymous routine using the current environment (`E'`) in the thread for actor `A`. The anonymous routine implements the logic specified for the `toggle` input in the `unselected` state of the EFSM for the `Item` behavior, and the `state` variable is assigned the value `selected`.

```
crl [Item:unselected:toggle] :
    < A | 'Item | E | L | R | I > thread[A | M]{input[E']} < S -> A . 'toggle : VL >
 => < A | 'Item | E | L | R | I >
    thread[A | M]{[routine(nil, {'state := state('selected);})(VL); @ E']}
 if ([['state @ E] @ M] == state('unselected)) and ((L == S) or not ('toggle in I)) .
```

Fig. B.1.

The conditional rule in Fig. B.2 is applied when an `Item` actor (`A`) is unlocked and waiting for input in state `unselected`, and a coordination actor (`L`) must lock actor `A` during the first phase of a `dispatch` statement that will ultimately dispatch a sequence of messages (`EL`) that includes a `toggle` message to actor `A`. When the rule is applied, actor `A` is locked by actor `L`, and the first phase of the `dispatch` statement continues.

```
crl [Item:unselected:toggle:dispatch.1] :
    dispatch.1[L : (actor(A).('toggle)),EL]
    < A | 'Item | E | ' | R | I > thread[A | M]{input[E']}
 => < A | 'Item | E | L | R | I > thread[A | M]{input[E']}
    dispatch.1[L : EL]
 if ([['state @ E] @ M] == state('unselected)) .
```

Fig. B.2.

The conditional rule in Fig. B.3 is applied in the following scenario.

- An `Item` actor (`A1`) is unlocked, waiting for input, and receives a `toggle` message.
- A `ListItem` role actor (`A2`) is unlocked and waiting for an event.

- The `Item` actor A1 is a role member actor coordinated by `ListItem` role actor A2. The role member set (`A2 ; R1`) for actor A1 includes actor `A2`, and the role member set (`A1 ; R2`) for actor A2 includes actor `A1`.

When the rule is applied, the `toggle` message is redirected from actor `A1` and instead triggers an event at actor `A2`. The focus actor of the event is actor `A1`. The event is processed by executing an anonymous routine using the current environment (`E2'`) in the thread for actor `A2`. The anonymous routine implements the logic specified for the event `[item <- toggle()]` for the `ListItem` role. The routine logic is elided below to simplify the figure.

```
crl [ListItem:Item:event:toggle] :
    < S -> A1 . 'toggle : VL >
    < A1 | 'Item          | E1 | ' | A2 ; R1 | I1 > thread[A1 | M1]{input[E1']}
    < A2 | 'ListItem:Item | E2 | ' | A1 ; R2 | I2 > thread[A2 | M2]{input[E2']}
 => < A1 | 'Item          | E1 | ' | A2 ; R1 | I1 > thread[A1 | M1]{input[E1']}
    < A2 | 'ListItem:Item | E2 | ' | A1 ; R2 | I2 >
    thread[A2 | M2]{[routine('item, {...})(actor(A1),VL); @ E2']}
 if ([['state @ E2] @ M2] == state('event)) .
```

Fig. B.3.