Comparing Three Coordination Models: Reo, ARC, and PBRD ⋆

Carolyn Talcott *

Computer Science Laboratory, SRI International Menlo Park, CA 94025, USA [1]

Marjan Sirjani

Electrical and Computer Engineering Department, University of Tehran; School of Computer Science, IPM, Tehran, Iran; and School of Computer Science, Reykjavik University, Iceland

Shangping Ren

Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA [2]

Abstract

Three models of coordination—Reo, Actors-Roles-Coordinators (ARC), and Policy-based Russian Dolls (PBRD)—are compared and contrasted according to a set of coordination features. Mappings between their semantic models are defined. Use of the models is illustrated by a small case study.

1 Introduction

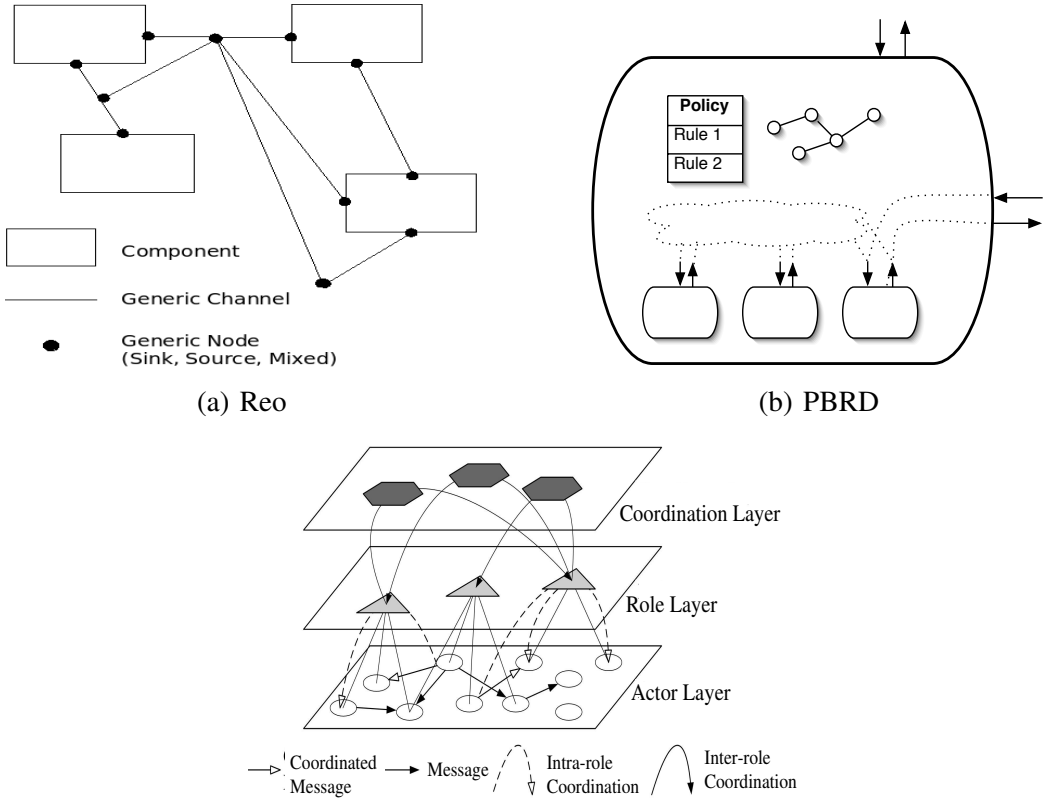
Coordination is becoming an increasingly important paradigm for systems design and implementation. With multiple languages and models for coordination emerging it is interesting to compare different models and understand their strengths and

⋆ Expanded version of a conference paper presented at the FOCLASA07.

* Corresponding author.
Email addresses: clt@cs.stanford.edu (Carolyn Talcott), msirjani@ut.ac.ir (Marjan Sirjani), ren@iit.edu (Shangping Ren).

[1] Supported in part by NSF grant CCR-0311348
[2] Supported in part by NSF grants CNS-0431832 and CNS-0746643

weaknesses, find common semantic models and develop mappings between formalisms. This will help us to gain a deeper insight into coordination concepts and applications, and also to establish a set of features/criteria for defining and comparing coordination models. There are two main classes of coordination models, exogenous and tuple-space based. In this paper, we focus on the exogenous case and compare and contrast three coordination models: Reo [1], Actors-Roles-Coordinators (ARC) [2], and Policy-based Reflective Russian Dolls (PBRD) [3]. These three models cover a wide spectrum of communication mechanisms including synchronous and asynchronous, message-based and channel based. They also represent a variety of organization principles such as roles, hierarchical reflection, nesting of circuits, and mechanisms for hiding. Thus we believe they serve as a good sample set for a first study. Future work includes considering tuple-based models such as Linda [4] and its mobile extension, Lime [5], and Klaim[6] and its stochastic extension [7].



(a) Reo

(b) PBRD

(c) ARC

Fig. 1. Three Different Coordination Models

Figure 1 gives a graphical impression of the Reo, ARC, and PBRD. Reo is a channel-based exogenous coordination model for component composition. In Reo, complex connectors are compositionally built out of simpler ones. The simplest connectors are channels with well-defined behaviors. These connectors are repre-

sented graphically as circuits. Similar to electronic circuits, connectors show how distributed coordinatees are connected [3]. The emphasis in Reo is on the connectors, and the coordination and communication patterns which they impose on the components, but not on the components which are the coordinatees. Compositional semantics of Reo circuits can be given by Timed Data Streams (TDS) [8] and by constraint automata [9,10]. Constraint automata can also be used for analyzing and model checking Reo systems.

ARC uses the separation of concern principle to partition coordination into two disjoint categories, i.e., intra-role and inter-role coordination, and uses roles and coordinators, respectively, to abstract these behaviors. The coordinatees in the ARC model are actors, entities that interact by asynchronous message exchange. Coordination is through message time-space manipulations which are transparent to the coordinatees. Reasoning in the ARC system is based on message dispatches in time (when) and space (to whom).

Reflective Russian Dolls (RRD) is a model of reflective distributed object computation [3]. It uses reflection and hierarchical structure to provide a general layered coordination model. Each layer (meta-object) controls the communication and the execution of objects in the layer below. Policy-based RRD (PBRD) is a restricted form of RRD in which communication control is specified by declarative policies. The objects being coordinated are actor-like objects. The semantics of PBRD coordinators and coordinatees is interaction semantics [11,12] which is compositional both horizontally (composing object or coordinated object configurations) and vertically (composing coordinators and coordinatees).

Although the underlying computation model of both ARC and PBRD is the actor model, they represent very different approaches to coordination. ARC splits coordination into two aspects, one that deals with dynamicity and manages a group of actors providing similar services to make the dynamicity transparent, while the second aspect deals with enforcing given constraints on communication patterns, between groups of actors rather than between single actors. PBRD is concerned with enforcing communication constraints between individual actors, using a declaratively specified policy. ARC is concerned with systems with large numbers of actors having a much smaller number of individual behaviors, while PBRD takes a global view of systems with diverse actors.

Related work. A broad survey [13] 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 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. Reo, ARC and PBRD are all control driven models. Control-driven models

[3] We use the term *coordinatees* to refer to the entities being coordinated.

such as ABT [14], ROAD [15], IWIM [16], and CoLaS [17] 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 [18] and ReSpecT [19,20] combine the data-driven and control-driven models.

Some control-driven models, such as ROAD, CoLaS, and Finesse [21], target the scalability issues of open distributed systems through group-based coordination models. Most current role-based coordination models are based on organizational concepts, where roles abstract coordination behaviors among participants that play the roles. Role-based coordination models are surveyed in [22].

Several coordination models take decentralization into account. TuCSoN [23] distributes communication abstractions (tuple centers) to multiple Internet nodes, and every tuple center produces and maintains 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 [24] provides a controller for every object in the system and therefore implements completely decentralized coordination.

Reo is built upon the IWIM model of coordination and the coordination language Manifold and allows sophisticated exogenous coordination of active entities in a system. It can also be considered as a concrete instance of the application of the ABT model and demonstrates the expressive power of ABT composition. Reo can be used as a glue language for compositional construction of connectors that orchestrate component instances in a component based system. Connectors and their composition are the main focus in Reo. The entities are connect to, communicate, and cooperate through these connectors. Each connector in Reo imposes a specific coordination pattern on the entities (e.g., component instances) that perform I/O operations through that connector, without the knowledge of those entities.

Earlier coordination work based on the Actor model includes hierarchical coordination [25], multi-level meta architectures [26], and synchronizers [27]. However, the ARC and PBRD 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 (events) required to enact coordination policies. The hierarchical coordination model intentionally does not include a meta-architecture and enacts coordination via hierarchical grouping of actors, but this grouping is not based on role behavior. The use of role-based coordination also distinguishes the ARC model from the multi-level meta architectures.

PBRD is an instance of Reflective Russian Dolls formal model of distributed object reflection based on rewriting logic [28] and several models of distributed actor reflection such as the onion skin model [29,30] and the two-level actor machine model [31,32] have been shown to be special cases of the RRD model. PBRD may take a global view of coordination, with all coordinatees nested in one coordinator objects. As shown in [33], such models can be systematically transformed into multi-coordinator systems that exhibit equivalent behavior, the extreme being on coordinator per object as in the LGI model. The PAGODA (Policy And GOal based Distributed Autonomy) architecture, for specifying and prototyping autonomous systems, uses PBRD coordination to combine local components (aspects) into a single agent behavior and a higher-level PBRD specification to coordinate behavior of multiple distributed agents [34].

Plan. The remainder of the paper is organized as follows. In Section 2, we spell out the features to be compared and contrasted. Section 3 describes the three models and compares and contrasts them according to the listed features. Section 4 describes representations in the three models of a simple coordination task. In Section 5 we make a step towards a common semantic foundation for the three models. Conclusions and future work are discussed in Section 6.

2 Coordination Features

Coordination languages and models are being developed to address the problem of managing the interactions among concurrent and distributed processes. The underlying principle is separation of computations by components and their interactions [35,36]. In our study of the three chosen models of coordination we considered a number of features (dimensions in the design space) including those summarized below.

Computation model. Is communication message-, event-, or channel-based? Is it synchronous or asynchronous? Is state localized or is there a shared global memory? Is the state space discrete, continuous, or hybrid?

Control. Is the coordinator in control or is it a passive information store (control oriented versus data oriented coordination)? Do the coordinated components have explicit actions for effecting the coordination?

Semantic model. How is the semantics of components and/or coordinators specified? An operational semantics could be given as a state transition system, such as automata or rewrite systems. Denotational semantics might be expressed in terms of observable events, traces/streams, or signals.

Modularity and Compositionality. An important issue is compositionality of

system descriptions and semantics at all levels, both vertically and horizontally. Does the model provide mechanisms for structuring or modularizing coordination activities?

Distribution. Coordination is inherently a system wide phenomena. However when the system itself is distributed there are issues of managing distributed state and actions. Does the coordination model support explicit expression of distribution aspects or is this addressed at the implementation level?

System Dynamics. To what extent can the system architecture change during execution. For example can new components or coordinators be created? Can the communication topology change?

Specification. Coordination models typically focus on how a coordinator achieves its goals. But how are the goals specified? How can you decide if a coordinator achieves its goals? Examples of different kinds of goals include: serializing requests to a component; ensuring a given group communication semantics; ensuring atomicity of a group of messages; providing fault tolerance; and balancing resource usage, quality and timeliness.

Analyzability. An important and often ignored aspect of specifications is analyzability. To what degree do different coordination models support analyzability, verification of certain properties? And how?

3 Three Models of Coordination

Each of the three models is described in some detail, followed by a feature-wise comparison. As a simple case study we consider the *alternating display* problem. The idea is that there are several sensors producing readings, and a display device that renders each reading it receives. The requirement is that displayed readings should alternate amongst the available sensors. A practical example is the billboard display that alternately displays time and temperature. A fancier system might repeatedly display date, time, temperature. The coordinator's task is to ensure that the display receives readings in the right order.

3.1 Reo

Reo is an exogenous coordination language based on a calculus of channel composition. A channel is an abstract communication medium with exactly two ends and a constraint that relates the flow of data at its ends. A channel represents a primitive interaction (protocol), explicitly represented as a binary constraint. There are two types of channel ends, source-end where data enters into the channel, and sink-end

where data leaves the channel. A channel can have two sources, two sinks, or a source and a sink. The channel relation can be defined by users which allows an open-ended set of different channel types, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc.

Channels are connected to make a circuit by *joining* channel ends together to form *nodes*. A node is a *source node* if all of its channel ends are source ends. It is a *sink node* if channel ends are sink ends. Otherwise it is a *mixed node*. A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items, by a take operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

We may put a Reo connector (circuit) in a box to make a component out of it. The inner nodes become hidden and the source or sink nodes which are the interfaces of a component and its environment are called (input or output) ports. Mixed nodes cannot be used as ports and are not available for other components to connect to, they shall all be included in inner/hidden nodes. Assuming a Reo connector as a component, it has well-defined behavior and interface (ports) and can be reused.

Constraint automata: Compositional Semantics of Reo Constraint automata are proposed in [9,10] as compositional semantics of Reo, based on timed data streams [8]. Each element of a timed data stream is a pair of time and a data item, where the time indicates when the data item is being input or output. A transition fires if it observes data item in a port of the component and according to the observed data, the automaton may change its state. Therefore, the automata-states stand for the possible configurations (e.g., the contents of the FIFO-channels of a Reo connector) while the automata-transitions represent the possible data flow and its effect on these configurations.

Definition [Constraint Automata]: *A constraint automaton (over the data domain Data) is a tuple* $A = (Q, Names, \longrightarrow, Q0)$ *where:*

Q *is a finite set of states,* $Names$ *is a finite set of names (e.g. I/O ports of a component),* $\longrightarrow$ *is a finite subset of* $Q \times 2^{Names} \times DC \times Q$*, called the transition relation of* A*, and* $Q0 \subseteq Q$ *is the set of initial states.* DC *is set of data constraints that play the role of guard for transitions. For example* $d\_A = d\_B$ *is a data constraint that requires that the observed data on ports* A *and* B *be equal.*

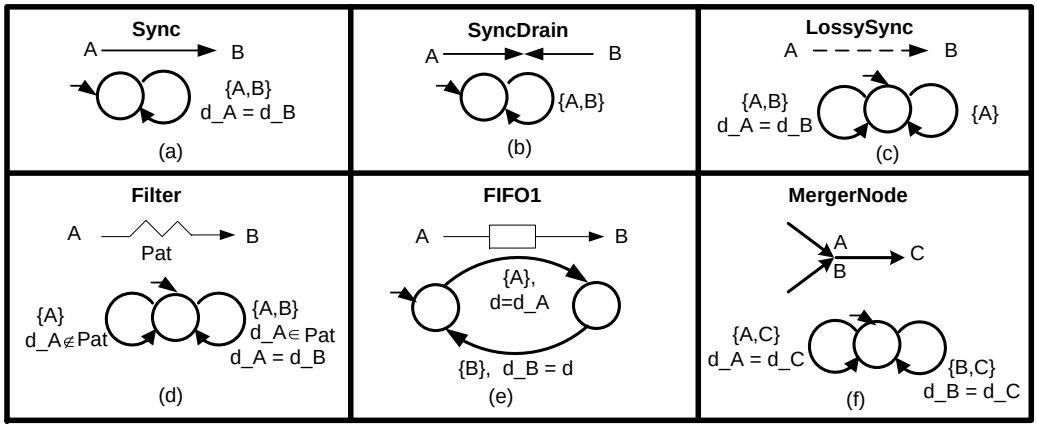
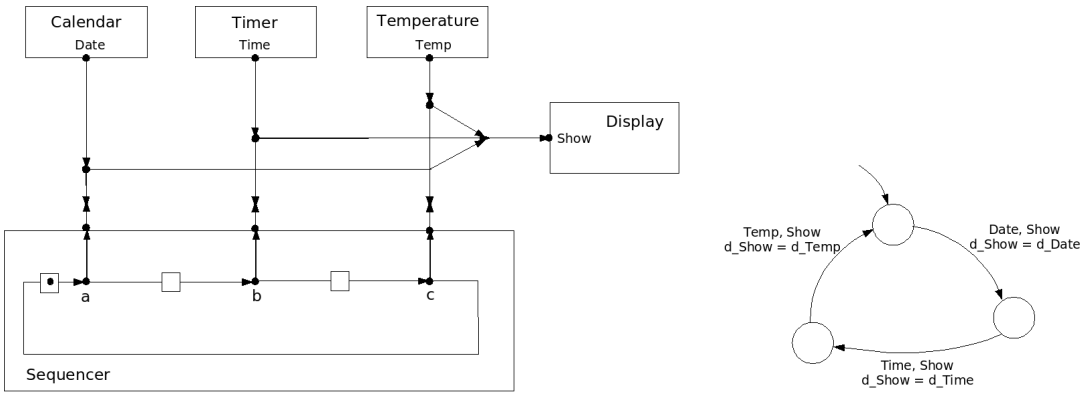


Fig. 2. Basic Reo channels and the merger node, together with their deterministic constraint automata

Figure 2 shows five primitive Reo channels and their corresponding constraint automata and also the constraint automaton of the merger node. A *Sync*, (Figure 2.a) channel has a source (A) and a sink (B) end. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. *SyncDrain* which is shown in Figure 2.b is a channel with two source ends (A and B). It accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost. The channel in Figure 2.c is a *LossySync* channel. This channel is similar to the Sync channel, except that it always accepts all data items through its source end (A). If it is possible for it to simultaneously dispense the data item through its sink (B) the channel transfers the data item; otherwise the data item is lost. A *Filter* channel which is shown in Figure 2.d behaves like the Sync except that it loses all data that do not match the specified pattern of the filter (*Pat* in the figure). The *FIFO*1 channel (Figure 2.e) has a source (A) and a sink end(B), and a bounded buffer with capacity of 1 data items (the box in the figure). The accepted data items are kept in the internal FIFO buffer of the channel. The appropriate I/O operations on the sink end of the channel obtain the content of the buffer in the FIFO order. The constraint automaton of the *MergerNode* in Figure 2.f shows that the output data on C is nondeterministically chosen from one of the inputs of A or B.

Example: An Alternating Display. Figure 3 shows the Reo circuit and constraint automata for a Display which shows date, time, and temperature in an alternating sequence. There are three components (Calendar, Timer, Temperature) which are responsible for generating date, time and temperature data. The fourth component (Display) displays the data which is given to it through its *show* port. These components are shown as black boxes and we know their behavior and interfaces (ports), but not their internal structure. We have a circuit, *Sequencer*, which orders the three inputs into Display component. By putting a box around Sequencer we can view it as a component. This component is further used in example of Section 4.



(a) ReoTimeTemp (b) TimeTempCA

Fig. 3. Date-Time-Temperature-Display

3.2 Actor-Role-Coordinator (ARC) Model

The Actor-Role-Coordinator (ARC) model [2] is also an exogenous coordination model. It is based on static behavior abstractions, but targeted on dynamic and large scale applications. In particular, actors [37,38] are used to model asynchronous and distributed computations. Roles, on the other hand, are static abstractions for behaviors shared by a set of underlying computational actors. The role member actor set may dynamically change, but all role player actors share the statically defined behaviors. Such abstraction decouples behaviors from their implementation and eliminates static binding between behavior coordination (which is the responsibility of coordinators) and computational actors. Coordinators in the ARC model coordinate different behaviors. Compared to the number of actors involved in an application, the number of behaviors, i.e. roles, is usually order(s) of magnitude smaller than the number of contributing actors. Therefore, the coordination model is not only stable, but also scalable.

To make this point more intuitive, consider a simplified space surveillance scenario in which infrared and radio wave sensors are deployed in an open space for detecting foreign objects. As shown in Figure 4, depending on where the foreign object occurs, different groups of sensors are active and generate data. In order for a control center to take an appropriate action, data from the two types of sensors must be semantically consistent (i.e., indicating the same type of object) and their arrivals at the center must be within a specified time range.

Clearly, it is a *must* that the infrared and radio wave sensors be *coordinated* in a *timed* fashion, but the nature of the problem prohibits us from statically pairing them up. Hence, two roles are introduced to abstract the dynamic groups of infrared and radio wave sensors, respectively. Further, the coordination of time synchronization among different type of data, from infrared sensors and radio wave sensors, can now be specified based on the two roles introduced, rather than the dynamic sensor sets.

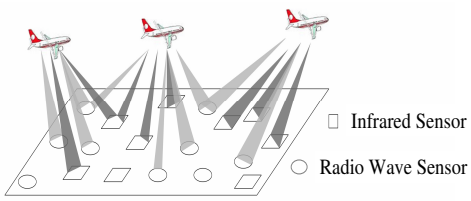


Fig. 4. The ARC Model

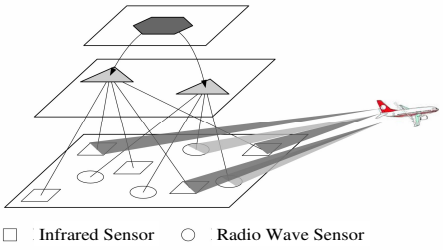


Fig. 5. The ARC View of an Open Space Surveillance System

In addition to serving as behavior abstractions, roles also share coordination responsibilities. They manage actor messages' space domain, i.e., deciding to which member actor a message should be delivered. However, *when* an actor message shall be dispatched is decided conjunctively by both roles and coordinators.

Under the ARC model, there are two types of *active* coordination objects in the model: *roles* and *coordinators*. The coordination within a role is called *intra-role* coordination which focuses on coordination among members that have the same behaviors, while coordination among the roles is called *inter-role* coordination. Both inter-role and intra-role coordination constraints are enforced transparently to the actors through actor message time-space manipulations.

Example: Coordination in an Alternating Display. Consider again the Date-Time-Temperature-Display example given in the previous subsection. In this example, the date, time and display are modeled by a single actor, respectively. However, in order to get more accurate temperature, multiple sensor actors are deployed in the region. The functionality of the date, time and temperature sensor actors are to send the display actor their corresponding value at their frequency. The display actor is to read values from the date, time, and sensor actors and display the values alternately.

Under this setting, we introduce four different roles, i.e., the *DateRole*, *TimeRole*, *TemperatureRole*, and *DisplayRole*. For the *DateRole*, *TimeRole*, and *DisplayRole*, there is only one actor as their member actor. Hence, these three roles are pass-through roles without any coordination functionality. The *TemperatureRole*, on the other hand, may have multiple sensor actors and its coordination functionality is to decide how to provide a value for the display actor. This value can be the first sensor value that reaches the role, or average sensing value depending on the intra-role coordination policy. The coordinator in this example is to coordinate the different value change frequencies. For instance, the display of the date changes every 86400 seconds, and time every second, while the temperature changes every hour only. In other words, the ratio among date, time and temperature changes should

be 86400:1:3600. Assuming we use the first available sensor value for display, the intra-role and inter role coordinations are given below.

Temperature role:

$$\gamma_{TP}(Z = 0 \wedge FirstValueArrived == false) :$$
$$P_1 : [\epsilon_{tp.send(dp,temp)}]$$
$$if(tp \in \gamma_{TP} \wedge tp \neq \alpha_{\perp_{TP}})\ \texttt{become}\ (\gamma_{TP}(z++));$$
$$FirstValueArrived = true;$$
$$\texttt{tell}\ (Z = z) \rightarrow tp.out\ (d, temp)\ \Box$$
$$ask\ (Z \neq z) \rightarrow \texttt{reroute}\ (temp, dp, \alpha_{\perp_{TP}});$$

Assume we take the first value from temperature sensor actors *tp*. The rule states that only the first message sent to the display actor *dp* will be sent out, the rest will be rerouted to the temperature role's sink actor $\alpha_{bot_{TP}}$, while the frequency control is done at the coordinator below. The role interacts with the coordinator through *ask* and *tell* operations on their observed events. These two operations are applied to a constraint store defined by concurrent constraint programming model.

Date role:

$$\gamma_{DT}(X = 0) :$$
$$P_1 : [\epsilon_{dt.send(dp,date)}]$$
$$if(dt \in \gamma_{DT} \wedge dt \neq \alpha_{\perp_{DT}})\ \texttt{become}\ (\gamma_{DT}(x++));$$
$$\texttt{tell}\ (X = x) \rightarrow dt.out\ (dp, date)\ \Box$$
$$ask\ (X \neq x) \rightarrow \texttt{reroute}\ (date, dp, \alpha_{\perp_{DT}});$$

As there is only one actor under the *Date* role, we only need to control the display frequency. The unused *date* messages sent to *display* are re-routed to its sink actor. The time role has a similar policy, but the contributing value is y.

Time role:

$$\gamma_{TP}(Y = 0 \wedge FirstValueArrived == false) :$$
$$P_1 : [\epsilon_{t.send(d,temp)}]$$
$$if(t \in \gamma_{TP} \wedge t \neq \alpha_{\perp_{TP}})\ \texttt{become}\ (\gamma_{TP}(y++));$$
$$FirstValueArrived = true;$$
$$\texttt{tell}\ (Y = y) \rightarrow t.out\ (d, temp)\ \Box$$
$$ask\ (Y \neq z) \rightarrow \texttt{reroute}\ (temp, d, \alpha_{\perp_{TP}});$$

Coordinator:

$$\theta(X:Y:Z=86400:1:3600):$$
$$P_1:[\epsilon_{\gamma_{DT}.become\ (\gamma_{DT}(x=0))} \cup \epsilon_{\gamma_{TM}.become\ (\gamma_{TM}(y=0))} \cup$$
$$\epsilon_{\gamma_{TP}.become\ (\gamma_{TP}(z=0))}]\ \texttt{become}\ (\theta(X:Y:Z=86400:1:3600))$$

The constraint is to reset the counters in different roles and ensure the display changes at the specified frequency

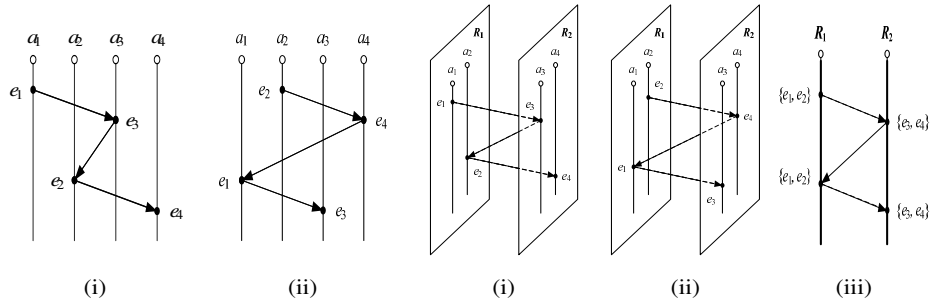
It is worthwhile to point out that the separation of *computation* and *intra-role* and *inter-role coordination* advocated by the ARC model is clean and orthogonal. Such separation mitigates the complexity of each individual type — coordinators only concern coordination of a *small scale* of roles while roles care *only* about actors of the *same behavior*, it provides the ground for independent modeling and compositional reasoning.

Formal Reasoning. The denotational semantics of the actor model is often described by externally observable event traces. Without external coordination constraints, valid traces of an actor computation are a set of all possible linear orderings of events. When coordination constraints such as timing constraints are imposed on the computation, valid traces must take into account the maximal allowed timespans between event pairs [39]. Hence, some otherwise valid event traces may be prohibited by the constraints.

The role introduced in the ARC model captures a group of actors sharing the same behaviors and makes these actors indistinguishable from a coordination perspective. It therefore extends the set of allowed observable traces and provides certain degree of relaxation to the constraints.

To be more specific, consider two actor computations represented by the event diagrams [40] shown in Figure 6(a) (i) and (ii), respectively, where actors a_1 and a_2 share the same behavior, and so do a_3 and a_4. Based on the traditional actor trace equivalence [38,41], these two computations are not trace equivalent because Figure 6(a)(i) has trace $\{(e_1, e_3, e_2, e_4)\}$, whereas Figure 6(a)(ii) has trace $\{(e_2, e_4, e_1, e_3)\}$. However, because a_1 and a_2 are indistinguishable from a coordination's perspective, so are a_3 and a_4, Figure 6(b) (i) and (ii) can hence both be reduced to Figure 6(b)(iii). In other words, the two computations are equivalent under role abstractions. Such a reduction avoids unnecessarily strong equivalence requirements and leads to an equivalence that is based on coordinated behaviors.

Satisfying Concurrent Constraints. Encapsulating coordination constraints within independent and distributed coordinators promotes the model's modularity, scalability and other features that a distributed system may provide. However, the benefits will only be fully realized if the following two issues can be addressed for the ARC model.



(a) Two actor computations (b) Two equivalent computations under role abstractions

Fig. 6. Event Diagrams for Actor Computation and Role Abstraction

The first issue is to satisfy concurrent constraints. We have currently adopted a Concurrent Constraint Programming (CCP) model [42,43] to communicate coordination between coordinators and roles [44]. operations on their observed events. By choosing a proper constraint system, such as one that resembles finite domain constraints [45–47] to define timing constraints, constraint consistency assurance can be reduced to the problem of checking the entailment of the constraint store after each *tell* event that carries a time-stamp of its occurrence.

The second issue is to resolve conflicting constraints encapsulated within different coordinators. The soft CCP model [48] provides a way to express preferences and priorities for different constraints. Such global prioritization is feasible under the CCP model because all the constraints are centralized in a constraint store, but is difficult to apply in the ARC model because of the model's distributed coordination nature. The approach we take is to avoid global prioritization, instead, associate each constraint with an award value if the constraint is satisfied under the specified statistic guarantee and resolve the conflicts in a way that maximizes the system award value.

3.3 Policy-based Russian Dolls (PBRD)

Reflective Russian Dolls (RRD) [3] is a 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. In this formalism, a coordinator is an object with a distinguished attribute that holds a nested configuration of objects and messages. The nested configuration itself could consist of base-level objects or coordinators each with their configuration of coordinated objects. The rewrite rules that specify the behavior of a coordinator object control delivery of messages in its

contained configuration as well as specifying how external messages are processed.

RRD provides a very general coordination mechanism. In [33] a special form of RRD called Policy-Based Russian Doll coordination (PBRD) was introduced. Here each coordinator has additional distinguished attributes: a policy attribute, a policy state attribute, that maintains processing state, and a queue of messages pending delivery. In PBRD rewrite rules interpret the policy attribute, in the context of the policy state, selecting messages from the pending queue to process and specifying what to do with them. Simple policies include ordering of message delivery, serializing requests, and recording a history of events. Policy languages can be simple tables, automata, or expressive functional languages.

Formalizing Policy Based Coordination. We explain the key features of PBRD coordination as formalized in Maude [49,50] a system based on rewriting logic used for developing, prototyping, and analyzing formal specifications. *Rewriting logic* [51] is a logical formalism designed for modeling and reasoning about concurrent and distributed systems. 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. Such a rule can be applied when a system term has a subcomponent matching t, such that c holds. That subcomponent can then be rewritten to t', possibly concurrently with changes described by rules matching other parts of the system state.

A PBRD coordinator is represented using Maude object syntax, by terms of the following form. [4]

```
[a : A | {_}, policy: P, policyState: pS, pending: pQ
       | inQ:(rcp,iMsgQ),outQ: iMsg,up:(ids,uMsgQ),dn: dMsgQ]
```

where `a` is the coordinators identifier and `A` is it class identifier, a subclass of `PBRDCoordinator`. Between the vertical bars are attributes holding the coordinators state. `{_}` is a place-holder for a set of objects being coordinated by `a` (these objects have a similar structure). The remaining attributes have a label-value format. `P` is the coordinators coordination policy (with label `policy:`); `pS` is its policy state; and `pQ` is a queue of messages pending delivery.

To the right of the second vertical bar are the coordinators *interfaces*—its points of interaction with its environment. Each interface has a label and a queue of messages. Incoming interfaces additionally have an associated set of object identifiers constraining the receivable messages. The interface `inQ: (rcp,iMsgQ)` is the

[4] Here we only discuss single level coordination. In general, there can be a hierarchy of coordinators, with lower-level coordinators being coordinated by higher level ones.

interface for messages coming in from the external world. This interface has label `inQ:` and message queue `iMsgQ`. `rcp` is a set of identifiers of *receptionist* objects, that are visible outside the coordinated component. Messages have the form `msg(id,mb)` where `id` is the identifier of the target object and `mb` is the message body. Only messages with target in `rcp` can be received at the `inQ:` interface. The interface with label `outQ:` and message queue `oMsgQ` is for outgoing messages. The interface `up: (ids, uMsgQ)` is for messages coming "up" from coordinatee objects with identifier in `ids`. The interface `dn: dMsgQ` is for messages going "down" to coordinatee objects. Messages sent by coordinatees have the form `msg(id,mb)@o` where `o` is the sender's identifier, exposed to the coordinator but not to the eventual receiver. Each coordinator interface can be thought of as a collection of ports, each port corresponding to an object that is the sender/receiver of the messages in the interface queue. In this interpretation, new ports open up when ever the coordinator learns of a new actor.

Communication rules. There are communication rules that move messages from a configuration of coordinators and messages into a coordinators input queue and from a coordinators output queue into the configuration. The rule, beginning `rl[in]` moves a message `msg(o,mb)` to the input queue of its target object, i.e of the object with `o` in the set of receptionists.

```
rl[in]:
[a : A |  atts | inQ: ((o rcp), iMsgQ), ips] msg(o,mb)
=>
[a : A |  atts | inQ: ((o rcp), (iMsgQ,  msg(o,mb))) , ips]
```

We use the Maude convention for writing object rewrite rules, making explicit only the attributes or interfaces read or written by the rule. The remaining attributes and interfaces are represented by variables that are bound to specific values when the rule is applied. In the above rule the variable `atts` stands for all of the coordinators attributes, while the variable `ips` stands for the remaining interfaces (`outQ:`, `up:`, and `dn:`). The term `(o rcp)` matches any identifier set containing the message target identifier, `o`.

The rule beginning `rl[up]` moves a message `msg(x,mb)@o` sent by coordinatee with identifier `o` into the up queue of its coordinator (the coordinator with `o` in the up queue identifier set).

```
rl[up]:
[a : A |  atts |  up: ((o ids), uMsgQ), ips] msg(x,mb)@o
=>
[a : A |  atts |   up: ((o ids), (uMsgQ, msg(x,mb)@o)), ips]
```

The rules for moving a message from an output or down queue are essentially the dual/reverse of the input/up rules and are omitted.

Finally, there are internal coordinator rules that move messages from up and input queues into the pending queue. The up case is given by the rule labelled `rl[up2pQ]`.

```
rl[up2pQ]:
[a : A | pending: pQ, atts
       | up: (ids,(msg(x,mb)@o,uMsgQ))],ips]
=>
[a : A | pending: (pQ,msg(x,mb)@o), atts
       | up: (ids,uMsgQ),ips]
```

Coordination rules. Coordination policies are specified by axioms for a function `next` that determines the next coordination actions—messages to deliver and state update. The axioms defining this function have the form

```
next(P,pS,pQ) = {dQ, oQ, pS1, pQ1}  if cond
```

where `P` is a policy, `pS` is a policy state, and `pQ` is a queue of messages pending processing. On the right, `dQ` and `oQ` are lists of messages. Those in `dQ` (resp. `oQ`) are for delivery to nested (resp. external) objects, by placing them in the coordinators down (resp. out) queues. The rule for policy interpretation (labeled `rl[next]`) uses the `next` function to determine the next action, if any.

```
rl[next]:
[a : A | {_}, policy: P, policyState: pS, pending: pQ, atts
       | dn: dMsgQ, outQ: oMsgQ, ips]
=>
[a : A | {_}, policy: P, policyState: pS1, pending: pQ1, atts
       | dn: (dMsgQ,dQ), outQ: (oMsgQ,oQ), ips]
if {dQ, oQ, pS1, pQ1} := next(P,pS,pQ)
```

where `pS1` and `pQ1` are the updated policy state and pending interaction queues, respectively.

Composition. There are two forms of composition in the PBRD coordination model. Horizontal composition is simply forming multisets of objects with distinct identifiers, using rules such as `rl[in]` for inter-object communication. More interesting is the composition of a coordinator with a set of objects to be coordinated, typically the `ids` part of the coordinators up queue is the set of identifiers of these objects. In this case, the configuration place holder `{_}` is filled by the object configuration `{C}` and the in/out rules for object communication are replaced by rules that move messages from an objects out queue to the coordinators up queue, and that move messages from the coordinators down queue into the target objects in queue. This vertical composition is a form of reflection and preserves message

ordering, while horizontal composition corresponds to standard actor system composition and asynchronous messaging.

Alternating display coordinator policy. The policy `altP` for the alternating display described at the beginning of this section has a policy state of the form `(od,oQ)` where `od` is the name of the display object and `oQ` is a queue of reader object ids (thus it will work for alternation of any number of sensor inputs). The requirement for alternation of messages to the display is expressed by

```
next(altP,(od,(o1 o2 ...)), (pQ0, msg(od,r)@o1, pQ1))
 =
{msg(od,r),nil,(od,(o2 ... o1)), (pQ0, pQ1)}
if not(containsSender(pQ0,o1))
```

where `containsSender(pQ,o)` is true just if `pQ` has the form `pQ0, msg(id,mb)@o, pQ1`. It should be clear that this policy ensures that the sensor ids of the sequence of messages received by the display alternates according to the list of object ids in the policy state.

3.4 Feature Analysis

The following table summarizes the features of the three models using the features listed in Section 2. Each feature is discussed in more detail below.

Comparing Features			
Feature	Reo	ARC	PBRD
Computation Model	channel based, synchronous and asynchronous	message based, asynchronous, reflective	
Control	message routing	global constraints, message ordering/routing	message ordering/routing
Semantic Model	CA, State transition, TDS	State transition	rewriting logic, interaction semantics
Modularity	circuit composition from basic channels; CA and TDS semantics are compositional	role based	component algebra, horizontal and vertical compositional semantics
Distribution	global logical view, distributed implementation	global coordination constraints, distributed roles, actors, and implementation	global logical coordination, transformation to distribute
Dynamics	components dynamically connect and disconnect	actors, roles dynamically created, changing communication topology	actors, coordinators dynamically created, changing communication topology
Specification	multiple temporal logics	rule based integration of first order logic and concurrent constraint programming	search patterns, LTL
Analyzability	model checking	constraint satisfactions analysis and constraint store consistency analysis	search, model checking

Computation model. Reo is a channel based language. Channels may be either synchronous or asynchronous. A channel is called synchronous if the pairs of operations on its two ends can only succeed atomically; otherwise it is called asynchronous. There is no shared global memory. Both ARC and PBRD are based on

the actor model of computation [37,41,38] with the coordinated objects being actors and the coordinators being meta-actors. Actors encapsulate their state and thread of control and communicate by asynchronous message passing. Meta-actors control the communication semantics of their base level actors.

Control. Coordination is imposed by a Reo circuit on connected components by determining when data can be accepted on input ports and when it can be taken from output ports, blocking components attempting write or read until the operation is available. The decision to connect to a port is made by the coordinatee, but once connected the coordinatee has no control over how the data is routed.

In the ARC model, role meta-actors intercept and control the delivery of base level messages. Formally, each base level action generates events that must be handled by the appropriate role before further base-level computation can take place. Role and coordinator meta-actors also communicate by events. The base-level actors have no active role in the coordination. However, roles are aware of the higher level coordinator and participate actively in their coordination. A novel aspect is that individual actors in a role are transparent to the coordinator layer. In the PBRD model, coordination is exogenous at all levels. At each level, lower-level objects execute as if there were no coordination layer, while the coordination layer controls delivery of message.

The actor model has a built in notion of communication / message delivery. This is modified by coordinators in ARC and PBRD using reflective mechanisms. In contrast, Reo components have individual behavior but there is no built in communication semantics for collections of components. This is provided by Reo connectors.

Semantics. Reo has several semantics including an operational semantics given by constraint automata (CA) [9,10] and a denotational semantics based on Timed Data Streams (TDS) [8,52]. In CA, states represent Reo configurations and transitions encode maximally-parallel stepwise evolution. Transition labels show maximal sets of active nodes and sets of data constraints. Timed data streams model the possible flows of data on connector ports, assigning a time to each interaction (input or output of a data element). A Structural Operational Semantics for Reo is given in [53] and a graph coloring semantics is given in [54]. The semantics of ARC coordinators, roles and actors is given by the composition of a state transition system that allows concurrent transitions and a concurrent constraint system that restricts the order and location of certain transitions. The operational semantics of PBRD coordinators and components is a rewriting logic system, a state transition system that allows concurrent transitions. The denotational semantics is a set of interaction paths—sequences of interactions, both peer-peer and object-metaobject. It is derived from the event partial order generated by executions of the rewriting semantics. The relationship between timed data stream and interaction path semantics is discussed in Section 5.

Modularity and compositionality. In Reo, more complicated connectors are made out of simpler ones. Nodes can be hidden by putting a box around a Reo connector, giving the connector a well-defined interface and making it a reusable entity. Both the CA and the TDS semantics are compositional—the behavior of a system can be constructed from the behavior of its constituents. The behavior of components as well as connectors can be given using CA or TDS, and so, we may have the behavior of the whole system as a CA or a TDS.

The key structuring mechanism of ARC is the notion of role, with overall coordination layered on top of the per role coordination. ARC semantics is compositional when certain restrictions are obeyed by the configuration of roles and coordinators, i.e., neither roles nor coordinators share coordinatees [2].

The essence of PBRD is the nested hierarchical structure of coordinators. This structure is preserved by basic composition operations. Event based semantics and interaction semantics are compositional both for pure actor systems and reflective systems—the semantics of a composition of objects and coordinators can be computed from the semantics of the parts (see [11,12]).

Distribution. In Reo, both components (coordinatees) and connectors (coordinators) may be distributed over a network. Physical locations are not captured by the semantics of Reo. In Reo implementations nodes may change physical location. Although this mobility of channel ends has significant consequences both for the application as well as implementation of channels (considering efficiency), it is transparent to Reo semantics and does not change the topology of channel connection. In ARC the overall inter-role constraints are conceptually stored in a centralized constraint store. The implementation of ARC distributes constraints based on criteria such as expected rate of change and locality of use [55]. RRD coordinators are specified as centralized controllers with global knowledge of the communication state of the controlled objects. As discussed in [33] RRD models can be systematically transformed into flat object system specifications that exhibit equivalent behavior, thus providing a principled path to distributed implementation. It is also shown how coordination policies can be modified, semi-systematically, so that a coordinator can be split into several distributed coordinators, each managing a subset of the original coordinated configuration.

Dynamic Behavior. In the Reo model channels can be created through active objects inside the components. Both components and connectors in Reo are mobile. Reo connectors are dynamically reconfigurable. We may have two kinds of reconfiguration: first, physical relocation of channel ends (by *move*) which is possible in Reo but entails no semantics consequences; and second, changing the placement of channel ends in nodes (by *join* and *split*) which changes the topology of the connector and its semantics. Replacements can also be done through active objects inside the components. Hence, dynamic behavior is under control of the environment/connecting components, and not part of the connector behavior.

Actor systems are inherently dynamic: new actors can be created, and actors names can be communicated in messages, thus changing the communication topology. Similarly for meta-actors. Beyond the topological dynamics inherent in actor systems, roles can re-route messages to different member actors and actors may change roles. RRD coordinators can be specified to deal with such dynamics in their contained configuration, and new coordinators can be created dynamically. Using the full reflective power of RRD, coordination of mobile objects can also be modeled.

Specification. A Reo circuit may be specified by a constraint automaton. Then this constraint automaton can be compared with the constraint automaton obtained as operational semantics of a Reo circuit to check (bi)simulation or language equivalence. Temporal logics for specifying properties of Reo circuits are presented in [52], [56,57], and [58], with main focus on real-time, reconfiguration, and model checking, respectively. Timed scheduled data stream logic (TSDSL) is introduced in [52] for reasoning about real-time constraints of Reo networks in the linear time setting. In [56,57], ReCTL* is introduced which combines the well-known CTL* logic [59] with SDSL for reasoning about Reo reconfiguration connectors. Branching time stream logic (BTSL) is introduced in [58], which deals with a branching time, time-abstract variant of TSDSL, and ignoring some minor differences, is contained in ReCTL*.

In ARC there are two types of coordination constraints, namely intra- and inter-role constraints. For intra-role constraints, we use guarded action to specify when a message should be re-routed to another destination within the group, or re-ordering within an actor in order to satisfy the coordination constraints. In contrast, inter-role constraints are a set of boolean properties that the roles being coordinated must satisfy. Requirements for PBRD coordinators have been specified by informal constraints on the resulting interactions of the coordinated actors (see [33,34]). Behaviors of specific ARC and PBRD coordinators can be specified in rewriting logic. No formal logic has been developed or adapted to date for either ARC or PBRD. Promising candidates include Event Logic [60] and temporal logic of rewriting [61,62].

Analyzability. Compositionality means that coordinators and coordinatees can be analyzed separately in any of the models. For Reo, regular model checking approaches can be adapted for constraint automata [58,56,57]. ARC's analyzability lies in the satisfiability and schedulability of composed inter-role and intra-role constraints. Although the satisfiability and schedulability in general are undecidable, certain techniques, such as graph theory, can be applied to identify infeasible situations. Furthermore, if the roles and coordinators are well partitioned, the complexity of constraint analysis can be reduced. The Maude rewriting logic language provides search and model-checking functions that can be used to analyze RRD systems. Use of policies expressed in restricted form can make coordinators easier to analyze.

4 Car Factory Case Study

In this section we look at how each of the three models addresses a particular coordination problem, namely coordinating different jobs in a factory. This example is taken from [2].

4.1 Specification

There are three factory jobs (called roles in [2]) to be coordinated: an assembler and some number of wheel and chassis producers. The requirements for job components (role players) are the following.

- An assembler receives car requests from a buyer, and parts (wheel or chassis) from producers. For each car request, it sends four part requests to wheel producers and one to a chassis producer. When four wheels and a chassis have been received it sends a car reply to a buyer.
- A wheel producer receives wheel requests and sends wheel replies.
- A chassis producer receives chassis requests and sends chassis replies.

The car factory system has one assembler, a, one chassis producer, c, and n wheel producers, $w_1, \ldots, w_n$. The assembler knows the chassis producer and one or more wheel producers, each producer (wheel or chassis) knows the assembler [5]. The assembler is the only receptionist (the only actor that can receive messages sent from outside the system). The requirements for the factory coordinator are

1. The ratio of chassis to wheel deliveries to the assembler is $1 : 4$.
2. The $1 + 4$ parts are delivered atomically.
3. Work is uniformly distributed amongst the wheel producers.

In the following subsections factory coordinators are described in Reo, ARC and PBRD.

4.2 Reo Factory

The actors—assembler, wheel producers, and chassis producers—are modeled as components. By putting an unbounded FIFO where an actor is connected to a Reo circuit, the inherent non-blocking and asynchronous behavior of actors is kept unchanged (i.e., Reo connectors cannot block actors sending messages). A Reo circuit

[5] In the actor setting one actor must 'know' another in order to send a message. In a channel based setting, 'knows' means sending on a suitable port.

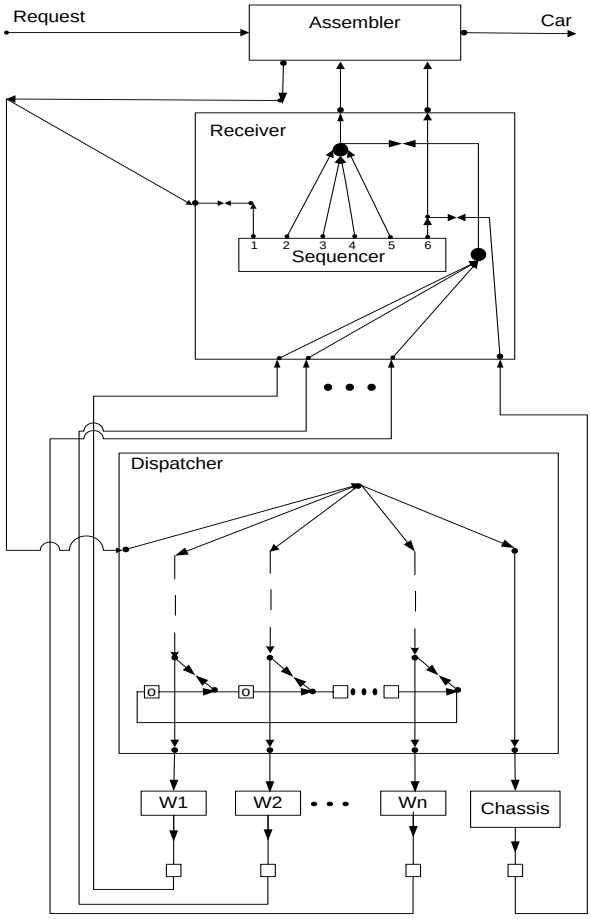


Fig. 7. Factory example using Reo

to coordinate these actors that satisfies the three requirements is shown in Figure 7. Using boxes, we may distinguish two modules: *request dispatcher* and *part receiver* in the Reo circuit, which we call as Dispatcher and Receiver, respectively. The Dispatcher sends chassis requests to the chassis producer and incorporates a round-robin policy in sending requests to "four out of n" wheel producers (to satisfy Requirement 3). The produced parts (messages) go from the producer actors to the Receiver. A Sequencer is used in the Receiver to send the parts atomically to the Assembler, satisfying Requirements 1 and 2.

The Reo circuit has been mapped to constraint automata compositionally. We first constructed the constraint automata of the Dispatcher and Receiver using the tool presented in [63], and then applied the hiding to avoid state explosion. Actors are modeled as CA and are composed with the CA of the rest of the circuit to

obtain the overall behavior. The resulting CA is used to show that the requirements are satisfied (the constraint automaton is not included in this paper for the lack of space).

The CA of the Receiver show that a request from the Assembler is received by the Receiver, then four wheels are received and sent to the Assembler, and finally a chassis is received and sent to the Assembler. The Sequencer guarantees the desired ratio and atomicity of the operations. The round-robin policy is shown by the CA of the Dispatcher.

Note that the round-robin dispatcher works properly because of the actor-nature of the wheel and chassis components. Without this assumption a request will get lost if none of the components are ready to receive it. In the general case, we use a four-way exclusive router [1] instead of LossySyncs. This can also be seen in the CA of the Dispatcher.

4.3 ARC Factory

$\gamma_A(busy = false)$:

$P_1 : [\epsilon_{a.receive(carReq)}]$

$if(busy == true)$ `reroute` $(carReq, a, \alpha_{\perp_A})$ $else$ $(busy = true)$;

$P_2 : [\epsilon_{a.send(buyer,car)}]$ $busy = false$;

$\gamma_W(x = 0)$:

$P_1 : [\epsilon_{w.send(a,wheel)}]$

$if(w \in \gamma_W \wedge w \neq \alpha_{\perp_W})$ `become` $(\gamma_W(x++))$;

`tell` $(X = x) \rightarrow w.out\ (a, wheel)$ $\square$

$ask\ (X \neq x) \rightarrow$ `reroute` $(wheel, a, \alpha_{\perp_W})$;

$P_2 : [\epsilon_{A.send(buyer,car)}]$ `become` $(\gamma_W(x = 0))$;

$P_3 : [\epsilon_{w_i.receive(wheelReq)}]$

$if\ (\exists j, 1 \leq j \leq n, s.t., |\mu_{w_j}| = \min_{1 \leq k \leq n} |\mu_{w_k}|)$ `reroute` $(wheelReq, w_i, w_j)$;

$\theta(X : Y = 4 : 1)$:

$P_1 : [\epsilon_{\gamma_W.become\ (\gamma_W(x=0))} \cup \epsilon_{\gamma_C.become\ (\gamma_C(y=0))}]$ `become` $(\theta(X : Y = 4 : 1))$

Fig. 8. Factory example using ARC

The ARC specification of the car factory coordination is shown in Figure 8. γ_A, γ_W, and θ denote structure of assembler role, wheel role, and the coordinator, re-

spectively, with initial states [6] . The P_i specify the coordination behavior associated with the coordinating actors. Expressions of the form $[\epsilon_{action}]$ denote events that trigger role and coordinator actions, $A \square B$ denotes that either A or B will take place; and $\mid \mu_\alpha \mid$ represents the size of actor α's mail box. The intra-role coordination for the assembler role is to ensure that if its member actor is busy (represented by the role's state variable *busy*), the role will buffer further incoming requests by rerouting them to its sink actor, $\alpha_{\perp_A}$. Upon observing the assembler actor finishing a car, the role resets its busy state to `false`. The wheel role has a state variable x, initially 0, that tracks the number of wheels produced since the last delivery to the assembler. The wheel role not only synchronizes with the chassis role through the coordinator by the primitive `tell` and `ask` operations to ensure a 4:1 ratio, but also reroutes wheel requests to ensure that they are evenly distributed. The coordinator specifies the inter-role coordination requirement. In this example, it ensures that wheel role and chassis role' productivity must be a 4:1 ratio.

4.4 PBRD Factory

A PBRD factory coordinator has the form

```
[FC : Factory | {_},
      policy: FP, policyState: wQ, pending: pQ,
     | in: ((FC a), iQ), out: oQ,
       up: ((a c w1 .. wk), uQ), dn: dQ]
```

The factory receptionist set includes the factory coordinator and the assembler, represented by `(FC a)` in the `in:` interface. The identifier set `(a c w1 .. wk)` in the `up:` interface gives the identifiers of the coordinated objects. The factor coordinator policy is represented by the constant `FP`. A PBRD policy state is a wheel actor queue, `wQ`, used to decide which wheel actor will receive the next request. There are five equational rules axiomatizing the `next` function for the policy, `FP`.

r1. if `pQ` has 4 wheel replies and at least 1 chassis reply addressed to the assembler a, remove them from `pQ` and deliver them to a (put them in `dn:`)

```
next(FP,wQ,pQ) = {mQ,nil,wQ,remove(pQ,mQ)}
    if mQ is a sublist of pQ containing
       4 wheel replies and at least 1 chassis
```

r2. if `pQ` has a car request, deliver it to a

```
next(FP,wQ,(pQ0, msg(a,carReq)@a, pQ1))
    = {msg(a,carReq),nil,wQ,(pQ0,pQ1)}
```

[6] As the chassis role has similar behavior to the wheel role, we omit its discussion.

r3. if pQ has a wheel request for some w, deliver it to the next wheel in wQ and rotate wQ

```
next(FP,(w wQ),(pQ0, msg(wh,wheelReq)@a, pQ1))
     = {msg(w,wheelReq),nil,(wQ w),(pQ0,pQ1)}
```

r4. if pQ has a chassis request, deliver it to c

```
next(FP,wQ,(pQ0, msg(c,chassisReq)@a,pQ1))
    = {msg(c,chassisReq),nil,wQ,(pQ0,pQ1)}
```

r5. if pending has a car reply put it in out:

```
next(FP,wQ,(pQ0, msg(c,carReply)@a, pQ1))
    = {nil,msg(c,carReply),wQ,(pQ0,pQ1)}
```

It is easy to see from the equations axiomatizing next(FP,wQ,pQ) that the PBRD Factory coordinator satisfies the three requirements. In particular the only parts messages delivered to the assembler are by rule r1, and each delivery consists of 4 wheels and a chassis, thus guaranteeing the 4:1 ration (requirement 1) and atomicity (requirement 2). The only requests delivered to wheel actors are by rule r3, which uses a round robin policy, thus guaranteeing uniform load distribution (requirement 3) in the sense of the number of requests to any two wheel actors differ by at most 1 at any time.

Discussion. Although the three models use different basic coordination primitives, there is a clear correspondence in the organization. Requirement 1-2 are addressed by the Reo Sequencer module, by the ARC coordinator rule plus the wheel rule P1, and by the PBRD rule r1. Requirement 3 is addressed by the Reo Dispatcher module, the ARC wheel role (P3) and the PBRD rule r3.

5 Semantic Foundations

In addition to comparing coordination models according to qualitative features, one can consider when coordinators represented in the different models are equivalent. For this purpose a common semantic foundation is needed. For the present, we focus on coordinating actor-like communication, that is asynchronous message passing. We assume an unbounded FIFO buffer at each connection point between a component and a Reo connector. We also assume Reo components send messages—pairs consisting of a target name and a data element [7]. Under these conditions we establish mappings between the TDS semantics of Reo components and

[7] Although communication of Reo components is "untargeted", nothing prevents a connector from using information in the data to redirect it (by using a filter channel). Dually, although actor messages are targeted, a coordinator in ARC or RRD may redirect it.

connectors [8,52] and the Interaction Semantics of actors and meta actors [11,12] for ARC and PBRD coordination.

5.1 Basic Definitions

We first define the two semantic domains and some auxiliary notation and give a small example.

Sequences. Following the Reo convention, we assume sequences are infinite and can thus be treated as functions from the natural numbers to the domain of sequence elements. We write $s(i)$ for the ith element of sequence s.

Timed Data Streams (TDS). A TDS over a set E is a pair (a, α) where a is a sequence with elements from E and α is a monotonically increasing sequence with elements from the non-negative reals. The semantics of a Reo connector with m ports is a set of m tuples of timed data streams, one for each port (i.e. an m-ary relation).

As an example, consider an alternating display (see section 3) with two sensors: a clock c producing time readings $hr : min$ of sort $Time$ (Time, 24 hour format); and a thermometer t producing temperature readings $n\ C$ or $n\ F$ (for Centigrade or Fahrenheit) of sort $Temp$. A Reo alternating display connector for this case would have three ports, two for input from the senders and one for output to the receiver. The TDS semantics for this connector is a set of triples with a TDS for each port:

$$((a_c, \alpha_c), (a_t, \alpha_t), (a_d, \alpha_d)).$$

A possible run is given by

- port c: $((9:00, 10:00), (0, 5))$
- port t: $((15:C, 17:C), (1, 3))$
- port d: $((9:00, 15\ C, 10:00, 17\ C), (2, 4, 6, 7))$

where for each port we have a pair of sequences of the same length. The first element of each pair is a data sequence, and the second element is a time sequence (time being represented by natural numbers). Thus $9:00$ is sent on port c at time 0 and received on port d at time 2.

Interaction Paths (IP). Given a set of object identities O and a data domain D, an interaction is a triple (ϕ, o, d) where ϕ is an interaction point of the form (x, dir) for $x \in O$ and $dir \in \{in, out, up, dn\}$; and (o, d) corresponds to a message, with target $o \in O$ and contents $d \in D$. An interaction path is a sequence of interactions. The semantics of a PBRD coordinator is a set of interaction paths corresponding to its possible sequences of interactions. Interactions correspond to firing of rules

that move messages between the coordinators interface queues and the external environment. For example firing `rl[up]:` with message `msg(o,d)@x` corresponds to an interaction $((x, up), o, d)$; and firing `rl[in]:` with message `msg(o,d)` corresponds to an interaction $((o, in), o, d)$.

Consider a PBRD alternating display coordinator for sensors c, t and display d. It doesn't accept messages from the environment, thus the interface for input from the environment has the form `in: (mt, nil)`. Its up interface for messages sent by coordinatees has the form `up: (c t d), uMsgQ`. The interaction path for this PBRD coordinator corresponding to the Reo run above is:

$$((c, up), d, 9:00), ((t, up), d, (15\,C)), ((d, dn), d, 9:00), ((t, up), d, (17\,C)),$$
$$((d, dn), d, (15\,C)), ((c, up), d, 10:00), ((d, dn), d, 10:00), ((d, dn), d, (17\,C))$$

The projection, $\pi(\theta, \phi)$, of an interaction path, θ, onto an interaction point ϕ is the subsequence of elements of θ of the form (ϕ, o, d) (preserving order). The function $ix(\theta, \phi)(j)$ returns the index of the jth element of $\pi(\theta, \phi)$ in θ. Thus if $ix(\theta, \phi)(j) = n$, then $\theta(n) = (\phi, o, d)$ for some (o, d), and there are j occurrences of interactions with interface ϕ in θ before n (since sequence indices start at 0). Given a correspondence of PBRD interaction point ϕ_i to Reo port i and letting $E = O \times D$, the projection $\pi(\theta, \phi_i)$ corresponds to the data stream on port i, and the function $ix(\theta, \phi_i)$ corresponds to the relative temporal ordering of events on port i.

Formalizing Requirements for the Alternating Display.

The alternating display semantics for the Reo connector is a relation Alt_{TDS} on TDS triples where

$$((a_c, \alpha_c), (a_t, \alpha_t), (a_d, \alpha_d)) \in Alt_{TDS}$$

just if for $i \in \mathbf{Nat}$

$$a_d(2i) = a_c(i), \quad a_d(2i+1) = a_t(i), \quad \alpha_c(i) < \alpha_d(2i), \text{ and } \alpha_t(i) < \alpha_d(2i+1).$$

The interaction semantics for a PBRD alternator is the set of interaction paths that satisfy Alt_{io} where

$$\begin{aligned}\theta \in Alt_{io} \;\Leftrightarrow\; & \pi(\theta, \phi_d)(2i) = \pi(\theta, \phi_c)(i) \;\wedge\; \pi(\theta, \phi_d)(2i+1) = \pi(\theta, \phi_t)(i) \;\wedge \\ & ix(\theta, \phi_c)(i) < ix(\theta, \phi_d)(2i) \;\wedge\; ix(\theta, \phi_t)(i) < ix(\theta, \phi_d)(2i+1)\end{aligned}$$

Thus, if we identify $\pi(\theta, \phi_i)$ with a_i and $ix(\theta, \phi_i)$ with α_i we see that the two relations correspond.

5.2 Factory Specification

Having introduced the semantic model, we can make the Factory Coordinator requirements more mathematically precise as constraints on the interaction paths θ of the coordinator semantics. We let $m, i, j, i', j', j_1, \ldots$ range over the natural numbers. The interfaces are (a, in), (x, out) (assembler communication with customer x), (a, up), (a, dn) (assembler output/input), (c, up), (c, dn) (chassis output/input), and (w_i, up), (w_i, dn) (ith wheel output/input), for $1 \leq i \leq n$.

Requirements 1, 2. Given that interaction paths are infinite, the notion of ratio of deliveries is not so simple to define. Thus requirements 1 and 2 are reformulated as: if any part is delivered to the assembler, the remaining parts of the $1 + 4$ set are delivered in a sequence that is not interleaved with any other deliveries. Namely, there is a function g mapping numbers to sequences of numbers such that if $\theta(m) = ((a, dn), p)$ where p is chassis or wheel (a part delivered to the assembler) then $m \in g(m) = [j_1, j_2, j_3, j_4, j_5]$ where $j_1 < j_2 < j_3 < j_4 < j_5$ and $\{d \mid (\exists 1 \leq k \leq 5)\theta(j_k) = ((a, dn), d)\}$ consists of one $chassis$ and four $wheel$s. If $\theta(m') = ((a, dn), p')$ then either $g(m) = g(m')$ or $g(m) \cap g(m') = \emptyset$. For other m, $g(m)$ is the empty sequence.

Requirement 3. Uniform distribution of requests to wheel producers can be interpreted in at least two ways, one is essentially round-robin scheduling, the other is balancing the pending requests for each producer. These differ if the wheel producers have different production rates. The following formalizes the round-robin interpretation. If $\theta(m) = ((w_i, dn), wheel)$ (a wheel request delivered to wheel producer w_i), and m is the index in θ of the jth wheel delivery, then $i = j \ mod \ n$.

5.3 Mappings between TDS and IP

We define functions $tds2ip$ mapping timed data streams to sets of interaction paths, and $ip2tds$ mapping interaction paths to sets of timed data streams. The mapping of data sequences is one-to-one. The fact that the images of these mappings are sets is due to the fact that for each stream or path there are a number of streams/paths that are equivalent in the sense that they represent different temporal views of the same underlying execution. We characterize the temporal views by ordering constraints and show that related streams satisfy the same ordering constraints.

Let D and O be a data domain and set of object identifiers, as above, and let IF be a set of m interaction points. Let $\tau = ((a_i, \alpha_i) \mid 1 \leq i \leq m)$ be a TDS tuple over $E = O \times D$ for a connector with m ports, and let θ be an interaction path over IF, O, D.

To define the mappings it is convenient to introduce the notion of stage in a TDS.

The nth stage of data transmission of τ, $S(\tau)(n)$, is defined using auxiliaries $J(\tau)(n,i)$—the index of the remaining tail of α_i after the nth global time point and $N(\tau)(n)$—the set of ports active at the nth global time point as follows.

$$
\begin{aligned}
&J(\tau)(0,i) = 0\\
&N(\tau)(n) = \{i \mid \alpha_i(J(\tau)(n,i) \leq \alpha_l(J(\tau)(n,l)) \quad \text{for} \quad 1 \leq l \leq m\}\\
&J(\tau)(n+1,i) = J(\tau)(n,i) + \text{ if } i \in N(\tau)(n) \text{ then } 1 \text{ else } 0\\
&S(\tau)(n) = \{(i, J(\tau)(n,i)) \mid i \in N(\tau)(n)\}
\end{aligned}
$$

Thus $(i,j) \in S(\tau)(n)$ if data flows on the ith port at the nth global time point, $\alpha_i(J(\tau)(n,i))$. Note that if $(i,j) \in S(\tau)(n)$, $(i',j') \in S(\tau)(n)$, $n < n'$, and $(i'',j'') \in S(\tau)(n')$ then $\alpha_i(j) = \alpha_{i'}(j') < \alpha_{i''}(j'')$. Furthermore for any $1 \leq i \leq m$ and any j, there is some n such that $(i,j) \in S(\tau)(n)$, and if $(i',j') \in S(\tau)(n')$ with $n < n'$ then $\alpha_i(j) < \alpha_{i'}(j')$.

We restrict attention to semantic relations defining coordinator behavior to those specified by a (possibly infinite) set of timing constraints of the form $t(i,j) < t(i',j')$ and a set of constraints on the data streams. τ satisfies $t(i,j) < t(i',j')$ (written $\tau \models t(i,j) < t(i',j')$) just if $\alpha_i(j) < \alpha_{i'}(j')$ and θ satisfies $t(i,j) < t(i',j')$ (written $\theta \models t(i,j) < t(i',j')$) just if $ix(\theta,\phi_i)(j) < ix(\theta,\phi_{i'})(j')$. A set of constraints is satisfied if each element is satisfied. Here we do not further restrict the form of data constraints. Each TDS tuple, τ, or IP, θ, defines a set of temporal constraints, $C(\tau)$ or $C(\theta)$, characterizing its temporal view such that $\tau \models C(\tau)$ and $\theta \models C(\theta)$.

$$
\begin{aligned}
&C(\tau) = \{t(i,j) < t(i',j') \mid (\exists n < n')((i,j) \in S(\tau)(n) \wedge (i',j') \in S(\tau)(n'))\}\\
&C(\theta) = \{t(i,j) < t(i',j') \mid ix(\theta,\phi_i)(j) < ix(\theta,\phi_{i'})(j')\}.
\end{aligned}
$$

In a TDS tuple it is possible that more than one port is active at a given time, i.e. $S(\tau')(n)$ has more than one element for some n. Following [64], we interpret this as meaning that the two communications could have occurred in either order rather than requiring strict synchrony. We write $\tau' \sim \tau$ if τ' has the same ports and underlying data streams as τ, $S(\tau')(n)$ is a singleton for each n, and $\tau' \models C(\tau)$. We call such a τ' an interleaving of τ. Note that $\tau' \sim \tau$ implies that τ' satisfies any of the considered temporal and data constraints that τ does. By the non-zeno assumption for TDS, there are many such τ', each obtained by adding/subtracting small amounts to times at appropriate points in τ guided by the sets $S(\tau)(n)$.

To simplify the treatment of multiple 'simultaneous' communications we generalize interaction paths to sequences of multisets of interactions. A generalized interaction path stands for a (possibly infinite) set of interaction paths, each obtained by choosing some order for the elements of each multiset.

To define *tds2ip* we first define *tds2ipg* from timed data streams to a general-

ized interaction paths, then $tds2ip(\tau)$ is the set of interaction paths represented by $tds2ipg(\tau)$. $tds2ipg(\tau)(n)$ is the set of interactions that occur at the nth time point from the set of time streams of τ.

$$tds2ipg(\tau)(n) = \{(\phi_i, a_i(j)) \mid (i,j) \in S(\tau)(n)\}$$

$ip2tds(\theta)$ is the set of tuples of TDS such that the data part of the jth tuple component is the projection of θ onto the jth interface, and the time part is a monotonically increasing time sequence such that the ordering between interactions of θ is preserved.

$$\begin{aligned} ip2tds(\theta) = \{&((\pi(\theta,\phi_i),\alpha_i) \mid 1 \le i \le m) \\ &\mid (\forall 1 \le i,i' \le m)(\forall j,j')(ix(\theta,\phi_i),j) < ix(\theta,\phi_{i'},j') \Rightarrow \alpha_i(j) < \alpha_{i'}(j'))\end{aligned}$$

Lemma. The mappings between TDS and IP satisfy the following.

$$(1)\ \theta \in tds2ip(\tau) \Rightarrow \theta \models C(\tau) \wedge \tau \in ip2tds(\theta) \Rightarrow \tau \models C(\theta)$$
$$(2)\ \theta \in tds2ip(\tau) \Rightarrow (\exists \tau' \in ip2tds(\theta))(\tau' \sim \tau)$$
$$(3)\ \tau \in ip2tds(\theta) \Rightarrow \{\theta\} = tds2ip(\tau)$$

(1) says that every IP in the image of a TDS satisfies the temporal constraints of that TDS, and every TDS in the image of an IP satisfies the temporal constraints of that IP. (2) and (3) say that modulo choice of interleaving the mappings between TDS and IP define an isomorphism. Thus we see that we can move between the two forms of semantics preserving essential information.

Proof Sketch. For (1), assume $\theta \in tds2ip(\tau)$ and $(t(i,j) < t(i',j') \in C(\tau))$ then by the definition of $C(\tau)$ let $n < n'$ such that $(i,j) \in S(\tau)(n) \wedge (i',j') \in S(\tau)(n')$. If $\theta^* = tds2ipg(\tau)$, then $ix(\theta^*,\phi_i)(j) = n$, $ix(\theta^*,\phi_{i'})(j') = n'$ and $\theta^* \models (t(i,j) < t(i',j')$ as does any flattening of θ^*. Now assume $\tau \in ip2tds(\theta)$ and $(t(i,j) < t(i',j') \in C(\theta))$. Then $ix(\theta,\phi_i)(j) < ix(\theta,\phi_{i'})(j')$ and by definition of $ip2tds$, $\alpha_i(j) < \alpha_{i'}(j')$. For (2), the linearizing map used to obtain θ from $tds2ipg(\tau)$ can be used to transform τ to a linear form $\tau' \sim \tau$ satisfying the mapping conditions. For (3), note that $S(\tau)(n)$ is a singleton for any n.

6 Conclusions and Future Work

Each of the models is clearly highly expressive. The Reo model is more mature, with several formal semantics and tools for analysis. Reo is closer to being a programming model, while PBRD focuses on more abstract specifications. The ARC model is aimed at coordination of resource usage and QoS goals while PBRD has focused on logical communication constraints, as has much of the Reo work.

All three models provide for user definable coordination behavior, but in different ways: channel behavior (Reo), coordinator events (ARC), coordination policy rules (PBRD).

Although channels and messages seem very different operationally, denotationally they have similar semantics. To simplify details we focused on coordination of actor-like components that communicate asynchronously. The denotational semantic model does not distinguish between synchronous and asynchronous events, the difference is in the semantic function mapping specifications to denotations. Operationally, synchronous communication introduces some complexity. The PBRD coordination rules could be modified to model synchronous coordination by taking messages from coordinatee output queues or putting messages into coordinatee input queues only when they can be processed. The execution rules of ARC roles and coordinators could similarly be adapted to handle synchronous communication.

There are a number of topics for future work. One topic is covering a broader range of coordination languages and models. Preliminary work indicates that Reo specifications as CA can be used as a policy language for PBRD and that ARC can be embedded fairly naturally into PBRD. These mappings need to be worked out in detail. The full generality of rewriting logic and PBRD make it difficult to give simple mappings from PBRD to Reo or ARC. Logics for specification and reasoning about coordination are of great interest. Do the logics developed for Reo work more generally? Are new logics needed to express end-to-end properties emerging from coordination? An important topic is developing methods to combine coordination rules for different concerns: communication constraints, timing, resource usage, etc., and to assure safe composition.

References

[1] F. Arbab, Reo: A channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (2004) 329–366.

[2] S. Ren, Y. Yu, N. Chen, K. Marth, P.-E. Poirot, L. Shen, Actors, roles and coordinators: a coordination model for open distributed and embedded systems, in: Coordination Models and Languages, Vol. 4038 of LNCS, Springer, 2006, pp. 247–265.

[3] J. Meseguer, C. L. Talcott, Semantic models for distributed object reflection, in: European Conference on Object-Oriented Programming, ECOOP'2002, Vol. 2374 of LNCS, Springer, 2002, pp. 1–36, invited paper.

[4] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and SySystems (TOPLAS) 7 (1) (1985) 80–112.

[5] G. Picco, A. Murphy, G.-C. Roman, LIME: Linda meets mobility, in: 21 Int. Conf. on Software Engineering, 1999, pp. 368–377.

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

[7] R. De Nicola, J. Katoen, D. Latella, M. Massink, Towards a logic for performance and mobility, in: 3rd Workshop on Quantitative Aspects of Programming Languages, QAPL05, Vol. 153 of ENTCS, Elsevier, 2006, pp. 161–175.

[8] F. Arbab, J. Rutten, A coinductive calculus of component connectors, in: Proceedings of WADT'02, Vol. 2755 of LNCS, Springer, 2002, pp. 34–55.

[9] F. Arbab, C. Baier, J. J. Rutten, M. Sirjani, Modeling component connectors in Reo by constraint automata (extended abstract), in: Proceedings of FOCLASA'03, Vol. 97 of ENTCS, Elsevier, 2003, pp. 25–46.

[10] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in reo by constraint automata, Science of Computer Programming 61 (2006) 75–113.

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

[12] G. Denker, J. Meseguer, 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), Vol. 36 of ENTCS, Elsevier, 2000, pp. 405–425.

[13] G. A. Papadopoulos, F. Arbab, Coordination models and languages, Advances in Computers 46 (1998) 330–401.

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

[15] A. W. Colman, J. Han, Coordination systems in role-based adaptive software, in: Proceedings of COORDINATION'05, Vol. 3454 of LNCS, Springer, 2005, pp. 63–78.

[16] F. Arbab, The IWIM model for coordination of concurrent activities, in: Proceedings of COORDINATION'96, Vol. 1061 of LNCS, Springer, 1996, pp. 34–56.

[17] J. C. Cruz, S. Ducasse, A group based approach for coordinating active objects, in: Proceedings of COORDINATION'99, Vol. 1594 of LNCS, Springer, 1999, pp. 355–370.

[18] A. Omicini, F. Zambonelli, Tuple centres for the coordination of internet agents, in: Proceedings of the ACM Symposium on Applied Computing, 1999, pp. 183–190.

[19] A. Omicini, E. Denti, Formal ReSpecT, in: Declarative Programming – Selected Papers from AGP'00, Vol. 48 of ENTCS, Elsevier, 2001, pp. 179–196.

[20] A. Omicini, Formal ReSpecT in the a&a perspective, in: Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'06), Vol. 175 of ENTCS, Elsevier, 2007, pp. 97–117.

[21] A. Berry, S. M. Kaplan, Open, distributed coordination with finesse, in: Proceedings of the 1998 ACM symposium on Applied Computing (SAC'98), ACM, 1998, pp. 178–184.

[22] G. Cabri, L. Ferrari, F. Zambonelli, Role-based approaches for engineering interactions in large-scale multi-agent systems, in: Proceedings of SELMAS'03, Vol. 2940 of LNCS, Springer, 2003, pp. 243–263.

[23] M. Cremonini, A. Omicini, F. Zambonelli, Coordination and access control in open distributed agent systems: The TuCSoN approach, in: Proceedings of COORDINATION'00, Vol. 1906 of LNCS, Springer, 2000, pp. 369–390.

[24] W. Zhang, C. Serban, N. H. Minsky, Establishing global properties of multi-agent systems via local laws, in: Proceedings of Environments for Multi-Agent Systems III (E4MAS), Vol. 4389 of LNCS, Springer, 2007, pp. 170–183.

[25] C. A. Varela, G. Agha, A hierarchical model for coordination of concurrent activities, in: Proceedings of COORDINATION'99, Vol. 1594 of LNCS, Springer-Verlag, 1999, pp. 166–182.

[26] N. Venkatasubramanian, C. L. Talcott, Reasoning about meta level activities in open distributed systems, in: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC), ACM, 1995, pp. 144–152.

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

[28] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.

[29] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, D. Sturman, Abstraction and modularity mechanisms for concurrent computing, Parallel and Distributed Technology: Systems and Applications 1 (2) (May 1993) 3–14.

[30] G. Agha, S. Frølund, R. Panwar, D. Sturman, A linguistic framework for dynamic composition of dependability protocols, in: Proceedings of the IFIP Conference on Dependable Computing for Critical Applications, 1992.

[31] N. Venkatasubramanian, C. L. Talcott, Reasoning about meta level activities in open distributed systems, in: Proceedings of the fourteenth annual ACM Symposium on Principles of Distributed Computing, ACM, 1995, pp. 144–152.

[32] N. Venkatasubramanian, C. Talcott, G. A. Agha, A formal model for reasoning about adaptive qos-enabled middleware, ACM Transactions on Software Engineering and Methodology 13 (1) (2004) 86–147.

[33] C. Talcott, Coordination models based on a formal model of distributed object reflection, in: Proceedings of the first International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005), Vol. 150, Elsevier, 2006, pp. 143–157.

[34] C. Talcott, Policy-based coordination in pagoda: A case study, in: Proceedings of Second International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006), Vol. 181, Elsevier, 2007, pp. 97–112.

[35] R. Gorrieri, C. Hankin, Theoretical aspects of coordination languages (foreword), in: Theoretical aspects of coordination languages, Vol. 192 of Theoretical Computer Science, 1998, pp. 163–165.

[36] F. Arbab, What do you mean, coordination?, in: Bulletin of the Dutch Association for Theoretical Computer Science, NVTI , 1998, pp. 11 – 22.

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

[38] G. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1997) 1–72.

[39] D. L. Dill, Timing assumptions and verification of finite-state concurrent systems, in: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Springer-Verlag, London, UK, 1990, pp. 197–212.

[40] W. Clinger, Foundations of Actor Semantics, AI-TR- 633, MIT Artificial Intelligence Laboratory (May 1981).

[41] I. A. Mason, C. L. Talcott, Actor languages their syntax, semantics, translation, and equivalence, Theoretical Computer Science 220 (1999) 409 – 467.

[42] V. A. Saraswat, Concurrent Constraint Programming, The MIT Press, 1993.

[43] V. Saraswat, R. Jagadeesan, V. Gupta, Foundations of timed concurrent constraint programming, in: Proceedings of the Symposium on Logic in Computer Science, LICS'94, 1994, pp. 71–80.

[44] S. Ren, Y. Yu, N. Chen, K. Marth, P.-E. Poirot, L. Shen, Actors, roles and coordinators - a coordination model for open distributed and embedded systems., in: Proceedings of COORDINATION'06, Vol. 4038 of LNCS, Springer, 2006, pp. 247–265.

[45] B. Carlson, Compiling and executing finite domain constraints, Ph.D. thesis, Uppsala University, Sweden (1995).

[46] B. Carlson, M. Carlsson, D. Diaz, Entailment of finite domain constraints, in: Proceedings of the Eleventh International Conference on Logic Programming, MIT Press, 1994, pp. 339 – 353.

[47] P. V. Hentenryck, V. A. Saraswat, Y. Deville, Design, implementation, and evaluation of the constraint language cc(fd), Journal of Logic Programming 37 (1-3) (1998) 293 – 316.

[48] S. Bistarelli, U. Montanari, F. Rossi, Soft concurrent constraint programming, ACM Trans. Comput. Logic 7 (3) (2006) 563–589.

[49] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Mart´-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, Vol. 4350 of LNCS, Springer, 2007.

[50] http://maude.cs.uiuc.edu, The Maude Homepage.

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

[52] F. Arbab, C. Baier, F. de Boer, J. Rutten, Models and temporal logics for timed component connectors, in: Proceedings of IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, 2004, pp. 198–207.

[53] M. R. Mousavi, M. Sirjani, F. Arbab, Formal semantics and analysis of component connectors in reo, in: FOCLASA'05, Vol. 154 of ENTCS, 2006, pp. 83–99.

[54] D. Clarke, D. Costa, F. Arbab, Connector colouring I: Synchronisation and context dependency, in: FOCLASA'05, Vol. 154 of ENTCS, 2006, pp. 101–119.

[55] N. Chen, S. Ren, Building a coordination framework to support behavior-based adaptive checkpointing for open distributed embedded systems, in: Proceedings of Hawaii International Conference on Systems Sciences, HICSS-40, IEEE Press, Hawaii, 2007.

[56] D. Clarke, Reasoning about connector reconfiguration II: Basic reconfiguration logic, in: Proceedings of First International Symposium on Fundamentals of Software Engineering, FSEN'05, Vol. 159 of ENTCS, Elsevier, 2006, pp. 61–77.

[57] D. Clarke, A basic logic for reasoning about connector reconfiguration, Fundamenta Informaticae 82 (2008) 361–390.

[58] S. Klüppelholz, C. Baier, Symbolic model checking for channel-based component connectors, in: Proceedings of FOCLASA'06, Vol. 175 of ENTCS, Elsevier, 2007, pp. 19–37.

[59] E. M. Clarke Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[60] M. Bickford, R. L. Constable, A causal logic of events in formalized computational type theory, Tech. Rep. Technical Report 2005-2010, Cornell University (2005).
URL \url{http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2005-2010}

[61] J. Meseguer, The temporal logic of rewriting: A gentle introduction, in: P. Degano, R. de Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday, Vol. 5065 of LNCS, 2008.

[62] K. Bae, J. Meseguer, A rewriting-based model checker for the linear temporal logic of rewriting, in: 9th International Workshop on Rule-based Programming, RULE 2008, 2008.

[63] B. Pourvatan, N. Rouhy, An alternative algorithm for constraint automata product, in: Second International Symposium on Fundamentals of Software Engineering, FSEN'2007, Vol. 4767 of LNCS, Springer, 2007, pp. 412–422.

[64] F. Arbab, A behavioral model for composition of software components, L'Objet 12 (2006) 33–76.