Using Coordinated Actors to Model Families of Distributed Systems*

Ramtin Khosravi[1,2] and Hamideh Sabouri[1]

[1] School of ECE, University of Tehran, Iran
[2] School of CS, Institute for Studies in Fundamental Sciences (IPM), Tehran, Iran

Abstract. Software product line engineering enables strategic reuse in development of families of related products. In a component-based approach to product line development, components capture functionalities appearing in one or more products in the family and different assemblies of components yield to various products or configurations. In this approach, an interaction model which effectively factors out the logic handling variability from the functionality of the system greatly enhances the reusability of components. We study the problem of variability modeling for a family of distributed systems expressed in actor model. We define a special type of actors called coordinators whose behavior is described as Reo circuits with the aim of encapsulating the variability logic. We have the benefits of Reo language for expressing coordination logic, while modeling the entire system as an actor-based distributed model. We have applied this model to a case study extracted from an industrial software family in the domain of interactive TV.

1 Introduction

Software Product Line Engineering (SPLE) focuses on proactive reuse to reduce the cost of developing families of related systems. The goal is to promote reuse from source code to other project artifacts as well, including models, documents, etc. [1]. A key factor to achieve this is the explicit modeling and management of commonalities and variabilities among the products in the family. Based on the domain and characteristics of the software family, suitable ways to manage variabilities in relevant models must be devised. In this paper, we deal with the class of distributed software systems that are modeled as actor systems, which is a well-known model for distributed and concurrent systems [2] (Sect. 2.1).

There are various approaches to variability modeling. Some use annotative techniques in which parts of the model are annotated with specific features and are present in the configurations that include those features. This technique is more effective for smaller variations. When two variants differ in more than a few lines of code, using annotations clutters the code and reduces maintainability. On the other hand, compositional variability management uses components to handle variability [3]. Each variant may be implemented by a separate component

* This research was in part supported by a grant from IPM. (No. CS1390-4-02).

M. Sirjani (Ed.): COORDINATION 2012, LNCS 7274, pp. 74–88, 2012.
© IFIP International Federation for Information Processing 2012

and alternatives for a variation point implement a common interface, through which other components have a uniform access to the variants. This method is more modular and is more flexible in the sense that the binding time for variation points can be deferred easily to runtime. This makes the compositional approach suitable for reconfigurable software, where decision about the variants is made at runtime and can be changed dynamically [4]. This is specifically useful for distributed systems in which failures or topology changes may require more flexibility in configuration.

As the actor model can be treated as a component-based approach to engineer a distributed system, compositional variability management seems a good match for engineering families of actor systems. In [5], handling variability in an actor-based modeling language Rebeca [6] is studied where both annotative and compositional methods are used. However, expressing variability logic in actors has the risk of making the core functionality of the actors messy.

Taking a compositional variability management approach for an actor system makes the variability handling logic more about wiring of actors and routing messages to appropriate actors. Having a way to model the variability logic in a modular way and separate from the functionality will make the actors more reusable and variability logic more manageable. This is almost the same as the objective of coordination language Reo [7] that aims to capture the glue code between components in a compositional and modular way (Sect. 2.2).

Hence, we define a special type of actors called *coordinators* and define their behavior using Reo circuits. Externally, coordinators can be treated as ordinary actors communicating with other actors/coordinators via asynchronous message passing. This is essential to keep message passing as the basic communication means in a distributed environment. Coordinators will be discussed in more detail in Sect. 3. To express the way we integrate Reo circuits in an actor system precisely, we present a formal operational semantics for the model in Sect. 4. Our approach to variability management is dynamic in the sense that the configuration parameters are modeled as inputs to the coordinators. This enables dynamic changes in the configuration, making our method closer to re-configurable architectures as opposed to statically generating coordinators from a given configuration (which has the benefit of less runtime overhead).

We use coordinated actors to model a part of a real-world distributed system in the domain of interactive TV systems (Sect. 5). We show how coordinators can capture variability logic among a set of components in a distributed environment, keeping the components free from the variability handling code.

To the best of our knowledge, little work has been done addressing product lines of distributed systems based on asynchronous message passing. In [8], a methodology is presented for design, implementation and verification of highly configurable systems, such as software product lines. This methodology is centered around the ABS language, which is a class-based language built on top of the active object concurrency model of Creol [9], using asynchronous method calls. The full ABS modeling framework extends ABS by delta modeling language (DML) that is based on delta-oriented programming [10] to describe code-level

variability. The implementation of a product family contains a core module and a set of delta modules specifying the changes that should be applied to the core module to obtain a new product. In [11] a method is presented for verification of families of services which is based on modal transition systems. Although the service-based computing is inherently distributed, the authors have used a synchronous model leaving support for asynchronous interactions to future work.

2 Preliminaries

2.1 Actor Model for Distributed Computing

Actor model is a well-known model for concurrent and distributed computing. The basic units of concurrency are called actors which communicate solely through asynchronous message passing. Each actor has a unique identifier and an unbounded message queue. An actor may know the identifier of a number of other actors to which it can send messages. An actor takes messages from its input queue one at a time. Processing each message will result in (a) a set of newly created actors, (b) a set of messages sent to other actors, and (c) the new behavior of the recipient actor. As soon as the new behavior is specified, it can take the next message from the input queue to process. The behavior replacement essentially makes the behavior of the actors history-sensitive. Hence the actor model is known as a concurrent object-based computation model.

In practice, it is possible to describe the behavior of the actors using functional or imperative paradigms. The way we treat actors in this paper is independent of the implementation paradigm.

2.2 Reo Coordination Language

Reo is a channel-based coordination language in which complex entities, called connectors, are constructed out of simpler ones. The behavior of each connector in Reo defines a specific coordination pattern, through which the system components perform I/O operations. The simplest connector types are called channels which are basic means of communication with exactly two ends. Channel ends are of two types: the source end through which data enters the channel, and the sink end through which data exits the channel. Reo does not impose any constraint on the channel behavior, so that each type can have its own policy for synchronization, buffering, sequencing, computation and data management. A number of channel types are commonly used in the literature. We use the following set of channels in this paper which are defined briefly in the following (Fig. 1).

Synchronous channel (Sync) Read and write operations are performed only if there is a data item on the source end and the sink end is ready to read. If one of the ends is not ready, the other end will be blocked.

Synchronous Drain channel (SyncDrain) The write operation is performed in this channel only if both ends are ready to write. If one of the ends is not ready, the other end will be blocked.

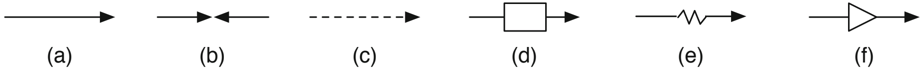

Fig. 1. Common channel notations: (a) Synchronous (b) Synchronous Drain (c) Lossy Synchronous (d) FIFO-1 (e) Filter (f) Transform

Lossy Synchronous channel (LossySync) A LossySynch channel is similar to a Sync channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink, the channel transfers the data item; otherwise the data item is lost.

One-place FIFO channel (FIFO1) In a one-place FIFO channel, the source end may put a data item on the channel if the buffer is empty. In this case, the sink end will be blocked. Furthermore, the sink end can perform the read operation if the buffer is full. In this case, the source end will be blocked.

Filter channel The read/write operation is synchronously performed (like Sync channel) with the condition that the data on the source end must satisfy a filter condition associated with the channel.

Transform channel The read/write operation is synchronously performed (like Sync channel) and a transformation function associated with the channel is applied to the input data to produce the output.

A connector is then constructed out of a number of channels organized in a graph of nodes and edges. A node consists of one or more channel ends. When all channel ends are of type source (resp. sink), the node is called a source node (resp. sink node), otherwise, it is called a mixed node. There are several ways to combine a set of channel ends of both types into a single node (Fig. 2).

The data items simply flow through a *flow through* node. A write on a *replication node* succeeds only if all outgoing channels are capable of consuming the written data. A *merge node* delivers a value out of one of the incoming channels non-deterministically. In our method, we usually face with the issue of prioritization over two or more inputs. In such cases, we use a special type of connector named *priority merger* which behaves like a merge node but one of its source ends has higher priority than the other (indicated by the exclamation mark). Whenever data is present on both its source ends, the data from the preferred input is passed to the sink end.

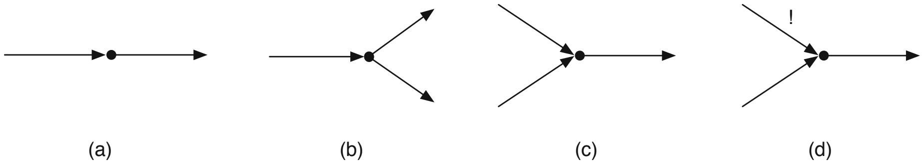

Fig. 2. Three types of mixed nodes: (a) flow through, (b) replication, (c) non-deterministic merge, and (d) a priority merger depicted as a node

3 Handling Variability through Coordination

We assume there are a number of special kind of actors named *coordinators* with their behavior specified as a Reo circuit (Fig. 3). The coordination logic, expressed as a Reo circuit, has one special input port for reading incoming messages from other actors, and a number of input ports specifying product configuration parameters. On the other side, there are a number of output message ports through which the coordinator sends messages to other actors, each port corresponding to a separate destination actor.

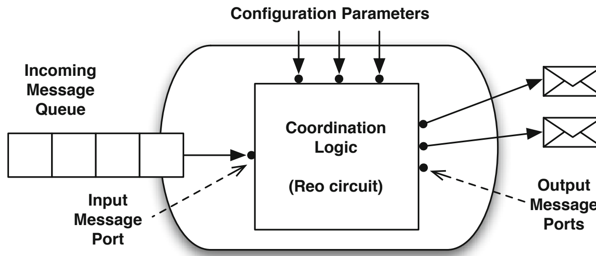

Fig. 3. A coordinator actor

Informally, the coordinator fetches one input message from the input queue and makes the message data ready on the input port of the Reo circuit. The Reo circuit then computes the output messages and provide them on the output ports. Whenever an output message appears on an output port, the coordinator takes the message and sends it to the corresponding actor. It is assumed that the coordinator is always ready to take the output messages, so the Reo circuit never blocks on the output ports.

An important assumption is made about the Reo circuit that is the circuit terminates after a finite number of steps when responding to a single input message. This assumption is made to make the behavior of the coordinators consistent with the semantics of the actor system given above. So, like other actors, a coordinator takes one message from its input queue, processes the message, generates a number of new messages, and possibly changes its own state. Note that the internal state of the Reo circuit (i.e., the contents of its buffer channels) may change when responding to a message, and to respond to the next input message, the Reo circuit continues from its last state.

3.1 Example: Handling Optional Features

As an illustrative example, consider a coffee machine that has an optional feature of adding milk to a coffee. To keep it simple, assume that if this feature is included

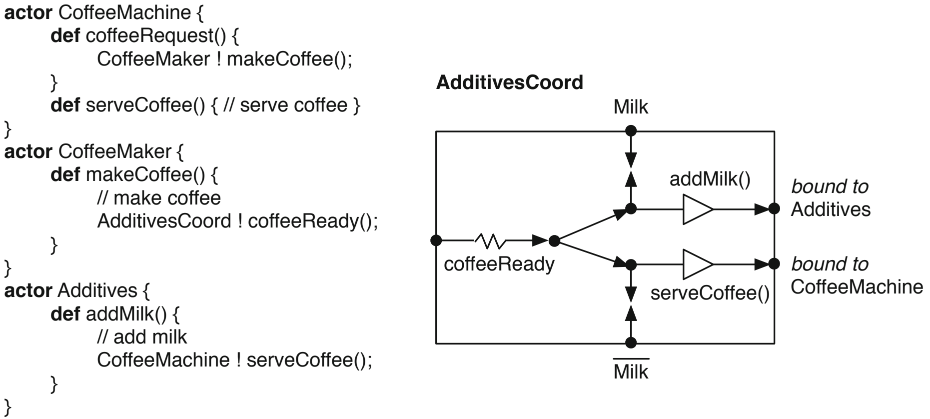

Fig. 4. Handling optional 'Milk' feature

in a product, then coffee is always served with milk. To handle this variability, we consider three functional actors and one coordinator actor as depicted in Fig. 4. The CoffeeMachine actor tells CoffeeMaker to make coffee. When it is done, CoffeeMaker informs a coordinator AdditvesCoord that the coffee is ready. The behavior of AdditivesCoord is described by a simple Reo circuit. To make it simple, we have provided both Milk and $\overline{\text{Milk}}$ configuration inputs to the circuit. The circuit first filters the incoming messages to process 'coffeeReady' only (to be complete, a mechanism must be added to discard irrelevant messages). Based on the presence of 'Milk' feature, the coordinator decides to tell Additives to add milk or just inform CoffeeMachine that the drink is ready.

Note that for this very simple example, the variability could be handled much more easily in the actors itself. As the interactions become more complex however, the benefits of using Reo as a compositional coordination model becomes more evident.

4 Formal Modeling of Coordinated Actor Systems

In this section, we formally demonstrate how the coordinators with their behavior described as Reo circuits are integrated into an actor system. The core material is presented in Sect. 4.3 and 4.4, but to make our description complete, we first give an operational semantics for actor systems in Sect. 4.1 and 4.2.

We use the following notations for working with sequences. Given a set A, the set A^* is the set of all finite sequences over elements of A. We write sequences as $[a_1, a_2, \ldots, a_n]$, where $a_i \in A$. The empty sequence is represented by $[\,]$, and $[h|T]$ denotes a sequence whose first elements is $h \in A$ and $T \in A^*$ is the sequence comprising the elements in the rest of the sequence. For two sequences σ_1 and σ_2 over A, $\sigma_1 \oplus \sigma_2$ is the sequence obtained by appending σ_2 to the end of σ_1.

We assume the following sets are given:

- *Id*: The set of all actor identifiers
- *Data*: The set of all data items that can be communicated in messages. We abstract from the details of this set, but note that it can contain actor identifiers, allowing a dynamic topology for the actor system.
- *State*: The set of all local states of the actors. We also abstract from the details of how actors keep track of their states. This can be a mapping from the actor's variables to values, or a function describing the actor's current behavior.

We also define $Msg = Id \times Data$ as the set of all messages. Each message is a pair (α, m) where α is the receiver and m is the message data. We usually write $\alpha.m$ as an alternative to (α, m).

4.1 Actor Systems

We define an actor as a tuple (α, q, s, b) where:

- $\alpha : Id$ is the unique identifier of the actor,
- $q : Msg^*$ is the unbounded FIFO message queue of the actor,
- $s : State$ is the local state of the actor,
- $b : Msg \times State \leftrightarrow 2^{Actor} \times Msg^* \times State$ is the behavior of the actor. Having $(A, \sigma, s') \in b(\alpha.m, s)$ means that the actor may respond to the incoming message $\alpha.m$ by creating a set of actors A, sending the sequence of messages σ, and changing its local state to s'.

Note that since the behavior of an actor may be modeled non-deterministically, in responding to an incoming message several outcomes may happen, hence b is defined as a relation, not a function. The uniqueness and freshness of identifiers of the actors in A are assumed (expressed later). For a message sequence $\sigma = [\alpha_1.m_1, \alpha_2.m_2, \ldots \alpha_k.m_k]$, we define $\text{recipients}(\sigma) = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$. If α is an actor identifier, then $\sigma_{[\beta]}$ denotes the sequence of messages in σ restricted to β as the recipient, i.e., it is obtained by removing the elements of σ whose recipients are not β.

If *Actor* denotes the set of all actors, then we define a configuration of actors as a set of actors. Since the actors in a configuration has unique identifiers, we define a configuration as a mapping from identifiers to actors: $C : Id \rightarrow Actor$. If no actor with identifier α exists in a configuration C, we write $C(\alpha) = \perp$. We use the notation $\text{ids}(C)$ to denote the domain of the mapping C, i.e., the set of all actor identifiers in C. We define *Conf* as the set of all configurations.

4.2 Transition System Semantics

For an actor system with initial configuration C_0, we define the transition system $TS = (Conf, Msg, \rightarrow, C_0)$, with *Conf* as the set of states, *Msg* as the set of actions (transition labels), C_0 as the initial state, and $\rightarrow \subseteq Conf \times Msg \times Conf$ as the transition relation defined as the smallest relation satisfying the following condition.

We have $(C, \alpha.m, C') \in \rightarrow$, (written alternatively as $C \xrightarrow{\alpha.m} C'$) if the following conditions hold:

1. $\alpha \in \text{ids}(C)$ and $C(\alpha) = (\alpha, [\alpha.m|q_\alpha], s_\alpha, b_\alpha)$
2. There exists $(A, \sigma, s'_\alpha) \in b_\alpha(\alpha.m, s_\alpha)$ such that
   (a) $\text{ids}(C') = \text{ids}(C) \cup \text{ids}(A)$
   (b) $C'(\alpha) = (\alpha, q_\alpha \oplus \sigma_{[\alpha]}, s'_\alpha, b_\alpha)$
   (c) $\forall \beta \in \text{ids}(C), \beta \neq \alpha \cdot C(\beta) = (q, s_\beta, b_\beta) \iff C'(\beta) = (q \oplus \sigma_{[\beta]}, s_\beta, b_\beta)$
   (d) $\forall (\gamma, q_\gamma, s_\gamma, b_\gamma) \in A \cdot C'(\gamma) = (\gamma, q_\gamma \oplus \sigma_{[\gamma]}, s_\gamma, b_\gamma)$
   (e) $\text{recipients}(\sigma) \subseteq \text{ids}(C')$

Condition 1 states that the recipient of the message $\alpha.m$ must exist in the configuration C and the $\alpha.m$ must be the first message in the actor's queue. The set of actors in the configuration must remain the same, except for the set of newly created actors (cond. 2a). After processing the message, the actor α changes its local state and possibly sends messages to itself (cond. 2b). All other actors possibly have new messages in their queues, but their state remains unchanged (cond. 2c). A number of new actors are created, and they may have messages in their queues initially (cond. 2d). Note that the recipients of the sent messages σ must exist at the time the messages are sent (cond. 2e).

To ensure uniqueness of identifiers in an actor system we make the following assumption on the behavior of every actor in the system. In every configuration C and an actor $(\alpha, [\alpha.m|q], s, b)$ in C, the behavior function b is defined such that if $(A, \sigma, s') \in b(\alpha.m, s)$, then we have $\text{ids}(A) \cap \text{ids}(C) = \emptyset$ and $|\text{ids}(A)| = |A|$.

4.3 Coordinator Actors

To describe the semantics of a coordinator, we assume the behavior of the Reo circuit is described using a Constraint Automaton [12], which is a well-known semantic model for Reo connectors. The notation we use in this paper is based on [13], as it includes finite runs of constraint automata too. An important point is that when handling variability using Reo circuits, we usually need priority merger which is an example of a context-dependent connector. The original constraint automata semantics for Reo does not handle context dependency (the situation where the behavior of a connector depends on other connectors). This issue is addressed in other automata-based semantics for Reo such as the one based on Reo automata [14]. Despite our use of context-dependent channels, we still express our semantics using constraint automata, as it is much simpler to explain. Furthermore, it is not hard to lift our semantic description to other types of automata-based semantics for Reo, supporting context dependency.

Since each coordinator is regarded as an actor, it will have a unique identifier. Let the set $CId \subset Id$ be the set of identifiers of coordinators. Suppose that the behavior of a coordinator with identifier ρ is defined by the constraint automaton $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \rightarrow, \mathcal{Q}_0)$. The set of states $\mathcal{Q}$, the set of initial states $\mathcal{Q}_0$, and the transition relation $\rightarrow$ are defined according to the standard semantics of Reo circuits in constraint automata. Assuming the usual partitioning of the node set

$\mathcal{N} = \mathcal{N}^{src} \uplus \mathcal{N}^{snk} \uplus \mathcal{N}^{mix}$ into source nodes, sink nodes, and mix nodes, we have $\mathcal{N}^{src} = \{in, cfg_{\rho_1}, cfg_{\rho_2}, \dots cfg_{\rho_k}\}$, where in is the input node, and cfg_{ρ_i} is the ith configuration port, $\mathcal{N}^{snk} = \{out_{\rho_1}, out_{\rho_2}, \dots, out_{\rho_n}\}$, the set of output ports, and $\mathcal{N}^{mix}$ is determined by the internal structure of the Reo circuit.

To specify the behavior of the coordinator when responding to the incoming message $\rho.m$, we must specify the messages it will create and the change in its local state. The local state of the coordinator is the current state of its behavior constraint automaton. So, suppose the coordinator is in state q and it receives the incoming message $\rho.m$. According to the constraint automata semantics for Reo circuits, and assuming that the behavior of the circuit terminates in finite steps, we will have maximal finite runs of the following form, where N_i is the set of nodes performing I/O operation and δ_i is a function giving data items on nodes.

$$q = q_0 \xrightarrow{N_1, \delta_1} \dots \xrightarrow{N_k, \delta_k} q_k \xrightarrow{\surd} q_k$$

Since the message m is put initially on the input node of the coordinator, we must have $in_\rho \in N_1$ and $\delta_1(in_\rho) = m$.

The set of output messages are obtained by taking the data on each output port and putting them as a sequence (to preserve the FIFO semantics of actors). More formally, let $\mu : \mathcal{N}^{snk} \to Id$ specify the binding of the output ports to the identifiers of the destination actors. For each I/O operation (N, δ), we define $\varsigma(N, \delta)$ as the sequence of messages sent during the I/O operation. The sequence is ordered (arbitrarily) based on the output port numbers of the Reo circuit:

$$\varsigma_i(N, \delta) = \begin{cases} [\,] & \text{if } out_{\rho_i} \notin N \\ [\mu(out_{\rho_i}).\delta(out_{\rho_i})] & \text{otherwise} \end{cases}$$

$$\varsigma(N, \delta) = \varsigma_1(N, \delta) \oplus \varsigma_2(N, \delta) \oplus \dots \oplus \varsigma_n(N, \delta)$$

Now, for the I/O-stream $IOS = (N_1, \delta_1) \dots (N_k, \delta_k)\surd$, we define the set of output messages as the sequence of $\varsigma(N_i, \delta_i)$, ordered by $i = 1, 2, \dots, k$, as follows:

$$outmsg_\rho(IOS) = \varsigma(N_1, \delta_1) \oplus \varsigma(N_2, \delta_2) \oplus \dots \oplus \varsigma(N_k, \delta_k)$$

It is important to note that initially (i.e., before processing the first message), the automaton is in one of the states in $\mathcal{Q}_0$. After processing of the first message, we assume the automaton saves its current state for processing the next message (hence, the coordination may be history-sensitive).

To integrate our coordination model into actor systems, we consider a coordinator as an actor in our semantic model, hence modeled as a tuple (ρ, q, s, b) as defined in Sect. 4.1. We assume the coordination logic of the coordinator with identifier ρ is defined by a Reo circuit specified by the constraint automaton $\mathcal{A}_\rho = (\mathcal{Q}_\rho, \mathcal{N}_\rho, \rightarrow_\rho, \mathcal{Q}_{0_\rho})$ with the set of port names $\mathcal{N}_\rho$ partitioned into $\mathcal{N}_\rho^{src}$, $\mathcal{N}_\rho^{snk}$, and $\mathcal{N}_\rho^{mix}$ as described previously. Also, we assume μ_ρ is the mapping of output ports into actor identifiers.

As said before, the state of a coordinator is the last state of the maximal finite run it has taken to respond to the last message, or one of the initial states of $\mathcal{A}_\rho$

if it has not responded to any message yet. So, we have $s \in \mathcal{Q}_\rho$ and to make it consistent with the definition of ordinary actors, we assume the following:

$$\biguplus_{\rho \in CId} \mathcal{Q}_\rho \subset State$$

The behavior of the coordinator ρ, expressed as the relation $b_\rho : Msg \times State \leftrightarrow 2^{Actor} \times Msg^* \times State$ is the smallest relation satisfying the condition $(A, \sigma, s') \in b_\rho(\rho.m, s)$ if

1. $A = \emptyset$
2. There exists a maximal finite run $s = q_0 \xrightarrow{N_1,\delta_1}_\rho \dots \xrightarrow{N_k,\delta_k}_\rho q_k \xrightarrow{\surd}_\rho q_k = s'$, such that
   (a) $in_\rho \in N_1$ and $\delta_1(in_\rho) = m$
   (b) $\sigma = outmsg_\rho((N_1, \delta_1) \dots (N_k, \delta_k)\surd)$

4.4 Initial Configurations

Initially, we have a set of (non-coordinator) actors of the form $(\alpha, q_{0_\alpha}, s_{0_\alpha}, b_\alpha)$. We assume the initial local states and message queues are unique. We also have a set of coordinators with the identifier set R. A coordinator with identifier $\rho \in R$, can start in one of the possible initial states $(\rho, [\ ], q_0, b_\rho)$ for some $q_0 \in \mathcal{Q}_{0_\rho}$.

5 Case Study

The case presented here is part of a larger project that is currently in progress in the domain of an interactive TV product line at Soroush company[1]. The project has a big feature model, as the product line ranges over various types of product in the subdomains of IPTV, Hospitality, Digital Signage, etc.

The external variabilities that correspond to different usages of the system lead to a relatively large number of internal variabilities too. For example, an IPTV with a large number of subscribers scattered through a vast geographical extent is different from a hotel offering hospitality services through its local TV network in terms of performance and availability requirements. In this section, we present a simplified view of the solution applied to handle an internal variability concerning load balancing of the offered services aiming to improve performance and availability.

5.1 The Video-On-Demand Use Case

In this example, an IPTV network is considered offering various services from which we focus on Video-On-Demand (VOD) service. The subscribers of this service can choose from a set of video contents, and receive the video via a streaming protocol.

[1] `http://www.soroush.net`

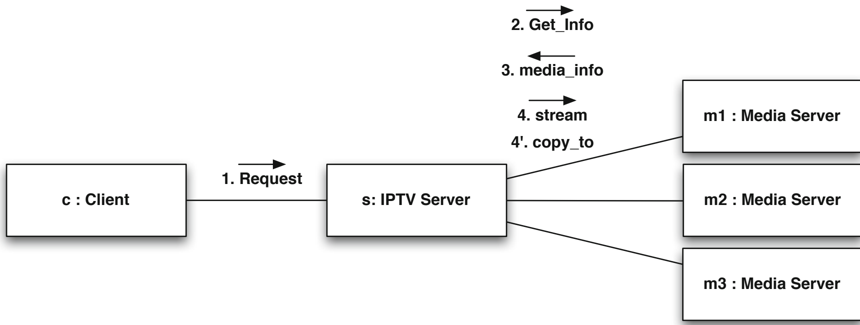

Fig. 5. Interaction diagram showing the basic scenario of VOD

To start the scenario, a client sends a request to the IPTV server. The server maps the requested video to a media resource description. The corresponding media may reside on a number of different media servers. Not every media server contains the specified media resource. If it does, it is possible that it is currently streaming media in its full capacity. So, the IPTV server first asks every media server about the requested media resource and if the server has free streaming capacity. If the IPTV server receives a reply from a media server which says it has the media resource and it has free capacity, then the media server is asked to stream the media to the client.

A variation point is defined for the case that no free server contains the requested media resource. In the simpler variant, a message is returned to the client to inform it that the requested media is not available at this time. In the other variant however, one of the servers that has the media is asked to copy the media into one of the servers that has free capacity. After the media is copied, the free server is asked to stream the media. This scenario is depicted in Fig. 5.

5.2 Modeling Video-On-Demand with Coordinated Actors

We now show how to model VOD interactions using coordinated actors. Due to space limitation, we focus only on the part that the IPTV server receives and processes the media information (messages 3, 4, and 4' in Fig. 5). The IPTV server is supposed to receive one media_info message from each media server. The message format is assumed to be of the form $\mathsf{media_info}(\mathsf{m}, \mathsf{s}, \mathsf{h}, \mathsf{f})$ where m denotes the requested media identifier, s is the identifier of the media server that is the sender of the message, h is a boolean value indicating the server s contains the media resource for m, and f is a boolean value showing that the server s has free capacity for streaming.

In the coordinated model, the media servers are supposed to send the media_info message to a coordinator actor, instead of the IPTV directly. The coordinator then decides how to continue based on the values received and the configuration parameter replicate_media which shows if the copying feature is enabled in the product configuration. (To keep the model simpler, we give the

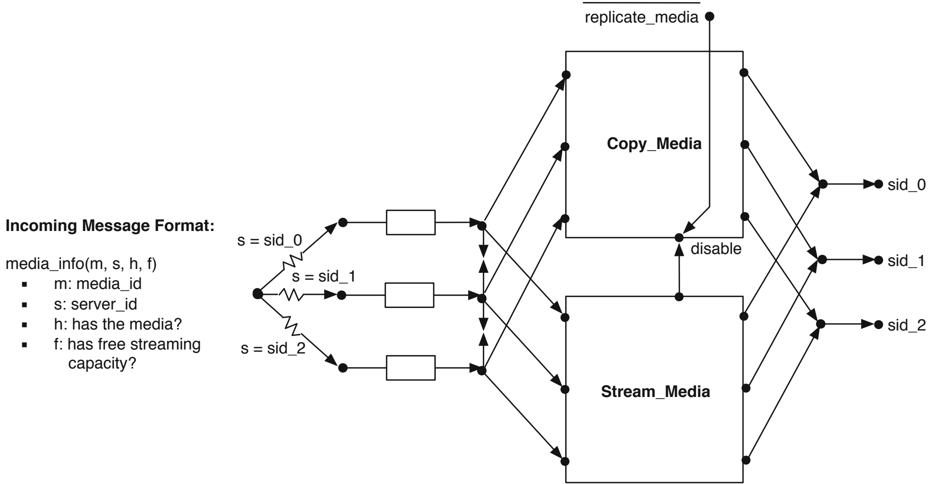

Fig. 6. The behavior of the coordinator for handling media_info messages

negation of this flag as the input to the coordinator.) The behavior of the coordinator for three media servers is depicted as a Reo circuit in Fig. 6.

Upon receiving a message media_info(m, s, h, f), the circuit first routes the message on one of the three channels corresponding to the server with identifier s. It is assumed that the identifiers of the media servers are sid_0, sid_1, and sid_2. The messages are buffered in FIFO1 channels, so that they are processed synchronously. When all buffers are full, the messages are replicated synchronously into two separate circuits Stream_Media and Copy_Media. The former decides if one of the media servers can be chosen to stream the media (i.e., both has the media resource and has free streaming capacity). In this case, a stream message is put on the corresponding output node. The circuit Copy_Media decides if one of the media servers must copy the media resource to another one, and puts the copy_to message on the corresponding output port. The output ports of the two circuits are then merged into single output ports of the coordinator.

The Copy_Media circuit has an input that disables its operation. We merge two values into this input. The first is $\overline{\text{replicate_media}}$ configuration parameter which is enabled if the media replication (copying media files) is not included in the product configuration. The second one is an output from Stream_Media which indicates a suitable server has been found and is sent stream message, hence no copying is necessary.

Figure 7 shows the internals of Stream_Media and Copy_Media circuits. The inputs to Stream_Media circuit are the media_info(m, sid_i, h, f) messages. The filter channels select those media servers that both have the media and have free streaming capacity. The merge node a is to activate at most one of these servers. Note that there are two outgoing channels from a. The SyncDrain is used to keep the passage of data synchronized with the input to the circuit (the choice of the first input is arbitrary, as they are all synchronized from outside of the circuit).

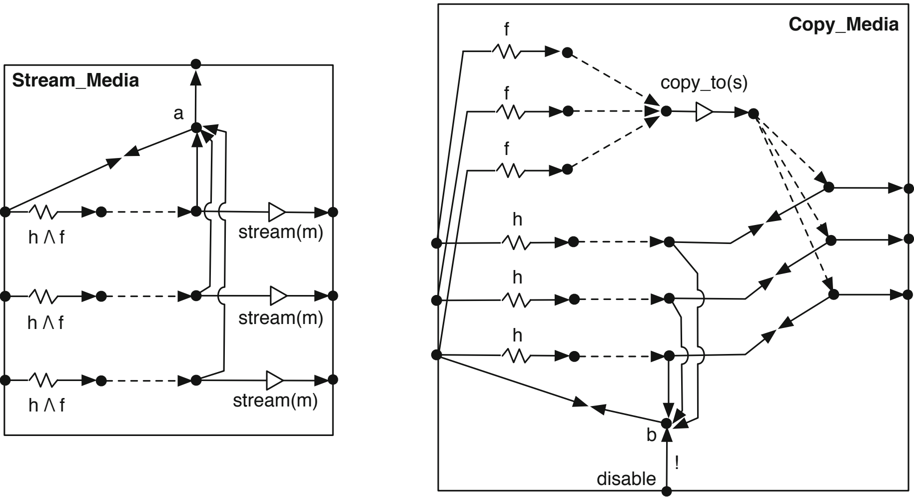

Fig. 7. The internal structure of the Stream_Media (left) and Copy_Media (right) circuits

The Sync channel ending at the circuit port is used to disable Copy_Media in case one of the servers is chosen to stream. The selected server is sent a stream message generated by the corresponding Transform channel.

The input to Copy_Media is the same. The upper set of filters pass the messages from those servers that has free capacity, from which one is chosen as the destination of the copy (hence the server identifier s is passed as the parameter of copy_to). The lower set of filters and subsequent channels choose a server that has the media resource to which the generated copy_to message is sent. The purpose of the merge node b is similar to the node a in Stream_Media, with the additional disable input that has higher priority over all other inputs. Hence, if the disable input is enabled, the data flow from all other inputs will be disabled and no output is sent by copy_to.

The circuits presented above only model the interactions needed to process media_info messages. There are other coordinators to handle other messages that are not discussed here.

We have implemented a simulator of VOD in Erlang language. The choice of the language was made due to its support of actor model, simplicity of the language, and support for distributed execution. A preliminary prototype of Reo simulator specifically designed to be used in our case study is implemented in Erlang. It still needs more work to fully cover Reo features in general case.

6 Discussion and Conclusion

In this paper, we defined a special type of actors named coordinators with their behavior defined by Reo circuits. We used coordinators to handle variabilities in a product line of actor systems. This keeps the variability handling logic

separate from the functionality of the components. An important point is that coordinators are still actors, so we can use them in a distributed setting.

Using Reo to describe the behavior of actors has a number of consequences. First, we benefit from the compositional characteristic of the model to reuse parts of the variability handling mechanism. Another advantage of Reo is its support for synchronized interactions. In actor modeling, we usually face with the problem of synchronizing a number of activities. Even thought there exist patterns for this, it still needs to be coded by the modeler and will clutter the main functionality of the system, especially in the absence of language-level abstractions for synchronization. However, as there are a several types of synchronized channels in Reo, various patterns for synchronization can be coded easily in the coordinators.

The BIP model [15] is another way to factor out interaction logic from a set of heterogenous components, by introducing connectors supporting rendezvous and broadcast synchronization patterns. BIP can be compared to our coordinated actors model as a whole. We think Reo is a better choice for our model, as it enables us to create and use various types of connectors in a rich, compositional method covering BIP interaction patterns too.

An issue that may limit the applicability of our model is the fact that the nature of Reo is not dynamic in the sense that channels can be added or removed dynamically. This will make a problem when the actor system changes its topology dynamically. The problem of reconfiguring a Reo circuit is studied in [16] and [17] based on graph transformations and coloring semantics. Still, we need proper language-level abstraction and parameterization mechanisms to easily describe reconfiguration in response to a message. Extending the model to support dynamic topology with an integrated precise formal model is a direction in which our work can be improved.

Another related work is the Dreams framework [18] which offers a distributed implementation of Reo using Scala actors. Although this work is not specifially designed to address software product lines, the fact that the coordination can be expressed in a single global Reo circuit, while executed in a distributed manner enables compositional description of the coordination logic. This is currently a weakness in our work, since having a number of different coordinators communicating through message passing we cannot benefit from the compositionality of Reo in reasoning about the coordination logic.

References

1. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc. (2005)
2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
3. Kästner, C., Apel, S.: Integrating compositional and annotative approaches for product line engineering. In: Proc. Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE). University of Passau (2008)

4. Lee, J., Kang, K.C.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proc. of the 10th International on Software Product Line Conference, pp. 131–140. IEEE Computer Society (2006)
5. Sabouri, H., Khosravi, R.: Reducing the model checking cost of product lines using static analysis techniques. In: FACS (to appear, 2012)
6. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae 63(4), 385–410 (2004)
7. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Comp. Sci. 14, 329–366 (2004)
8. Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability Modelling in the ABS Language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 204–224. Springer, Heidelberg (2011)
9. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365(1-2), 23–66 (2006)
10. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
11. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Model-Checking Tool for Families of Services. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 44–58. Springer, Heidelberg (2011)
12. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)
13. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Sci. Comput. Program. 74(9), 688–701 (2009)
14. Bonsangue, M., Clarke, D., Silva, A.: Automata for Context-Dependent Connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 184–203. Springer, Heidelberg (2009)
15. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Proc. of the Fourth IEEE Int. Conf. on Software Engineering and Formal Methods, pp. 3–12. IEEE Computer Society (2006)
16. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Sci. Comput. Program., 23–36 (January 2011)
17. Koehler, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of reo connectors triggered by dataflow. ECEASST 10 (2008)
18. Proenca, J.: Synchronous Coordination of Distributed Components. PhD thesis, Leiden University (2011)