Specification of coordination behaviors in software architecture using the Reo coordination language

Yongzhi Li

REPORT SEN-E0509 SEPTEMBER 2005

ISSN 1386-369X

Specification of coordination behaviors in software architecture using the Reo coordination language

ABSTRACT
One of the key goals of a software architecture is to help application designers analyze a software system at a higher level of abstraction than implementation. Software architects often use architecture description languages (ADLs) and their supporting tools to specify software architectures. Existing ADLs often lack formal foundations for design, analysis and reconfiguration of software architectures. The Reo language has a strong formal basis and promotes loose coupling, distribution, mobility, exogenous coordination, and dynamic reconfigurability. This thesis focus on assessing the Reo coordination language as an ADL by doing the following work: a) specify a distributed meeting scheduling system using the Reo coordination language; b) assess the Reo coordination language as an ADL using an existing method.

1998 ACM Computing Classification System: D.2.11, J.1, D.1.7, D.2.1
Keywords and Phrases: Reo, software architecture, architecture description languages, distributed meeting scheduling
Note: The author performed this work at CWI as a Master's degree project in the ICT in Business programme of Leiden University.

SPECIFICATION OF COORDINATION BEHAVIORS IN SOFTWARE ARCHITECTURE USING THE REO COORDINATION LANGUAGE

by

Yongzhi Li

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in ICT in Business

Leiden University

2005

Approved by ____________________________________
Chairperson of Supervisory Committee

____

____

____

Program Authorized
to Offer Degree____________________________________

Date ____________________________________

Keywords: the Reo Coordination Language, Software Architecture, Architecture Description Language (ADL), Distributed Meeting Scheduling

Abstract

One of the key goals of a software architecture is to help application designers analyze a software system at a higher level of abstraction than implementation. Software architects often use architecture description languages (ADLs) and their supporting tools to specify software architectures. Existing ADLs often lack formal foundations for design, analysis and reconfiguration of software architectures. The Reo language has a strong formal basis and promotes loose coupling, distribution, mobility, exogenous coordination, and dynamic reconfigurability. This thesis focus on assessing the Reo coordination language as an ADL by doing the following work: a) specify a distributed meeting scheduling system using the Reo coordination language; b) assess the Reo coordination language as an ADL using an existing method.

Acknowledgements

First of all, I would like to thank Prof. Farhad Arbab for giving me the opportunity to work on this topic.

This thesis could not have been written without the support of my supervisor Dr. Nikolay Diakov. He has given me very helpful comments along the way. I was also impressed a lot by his scientific attitude, his willingness to accept only carefully and objectively verified facts.

Thanks to my classmate Min Xie, who helped me to get further insights in the area of Reo and to clarify my ideas.

Last but not least, I like to thank my parents. I would not have been able to do this work without their help, and their support is greatly appreciated.

Thesis Supervisors:

Prof. Dr. Farhad Arbab
Senior Researcher, National Research Institute for Mathematics and Computer Science (CWI)

Dr. Nikolay Diakov
Scientific Staff Member, National Research Institute for Mathematics and Computer Science (CWI)

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

A software architecture is commonly referred to as "the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution [1]". One of the key goals of a software architecture is to help application designers analyze a software system at a higher level of abstraction than implementation. Software architects often use architecture description languages (ADLs) and their supporting tools to specify software architectures. The basic elements of architecture description are [2]:

- **Component**: It is a unit of computation or a data store in architecture;
- **Connectors**: They are architectural building blocks used to model interactions among components and rules to govern those interactions;
- **Architectural configurations**: They are the connected graphs of components and connectors that describe architectural structure.

In a distributed environment, component interaction often results in the complex coordination of multiple concurrent activities. The coordination language community has focused on the coordination aspect in a software system. In the area of coordination languages, *coordination* is "the process of building programs by gluing together active pieces [6]". A *coordination language* allows two or more components to communicate with each other for the purpose of coordinating their behaviors to accomplish a common goal. The Reo coordination language [7] is one of such promising language that provide the following features [7]:

- Loose coupling among components;
- Support for distribution and mobility of heterogeneous components;

- Exogenous coordination;
- Dynamic reconfigurability;
- Formal semantics based on a conductive calculus of flow and (alternatively) on constraint automata;
- Specification and verification methods using programming logic.

1.2 Problem Statement

There are several drawbacks of ADLs in general, as identified in [2][3][4][5],

- ADLs, such as Rapide [8][9], MetaH [10][11], and Darwin [12][13], do not allow explicitly specifying connectors as first-class modeling entities;
- ADLs, such as MetaH and Unicon [14][15], limit their ability to let new user-defined types of components and connectors;
- Most ADLs lack support for refinement of software architectures across levels of detail. There is no guarantee that the specified high-level coarse system behavior will be correctly implemented in fine details. Furthermore, they also lack support for strict synthesis and composition of existing commercial-off-the-shelf components into a new system design;
- Most ADLs lack facilities, such as tools and formal modeling notations to support dynamic reconfiguration;
- ADLs vary in their ability to support both functional and non-functional analysis of a modeled system at architectural level, the reason for this problem can be traced to the lack of appropriate formal semantics of ADLs.

In this thesis we focus on the suitability of the Reo coordination language to address these problems, thus we pose the following research question:

“What are the weaknesses and the strengths of the Reo coordination language as an ADL?”

1.3 Approach

In this thesis, we use the following research steps:

Step 1: To understand the essential requirements of ADLs, We first study "A Classification and Comparison Framework for Software Architecture Description Languages [2]". Then we study the Reo coordination language.

Step 2: We perform a case study to gain hands-on experience in using the Reo coordination language to specify and implement a complex distributed software system.

Step 3: Finally, based on the resulting system specification from step 2, we apply the framework from step 1 to assess the Reo coordination language as an ADL.

1.4 Goal

.

- To assess Reo as an ADL using an existing method: "A Classification and Comparison Framework for Software Architecture Description Languages".

1.5 Structure

In chapter 2, we present an overview of the Reo coordination language.

In chapter 3, we present a case study in which we specify and implement a distributed meeting scheduling system (DMSS) using Reo.

In chapter 4, we first give a short introduction to the selected existing evaluation framework. Then we present our analysis of the Reo coordination language applied to the case study.

In chapter 5, we give conclusions, discuss open issues and outline possible future work.

Chapter 2
An Introduction to the Reo Coordination Language

2.1 Introduction

In this chapter, we summarize the basic terms and concepts of the Reo coordination language. For a detailed specification of Reo, see Arbab's articles [7][17].

The coordination models and languages [16] have been introduced to deal with the increasing complexity of modern software systems, especially the concurrency in massively parallel and distributed systems. The Reo coordination language was proposed for composition of software components based on the notion of channels. Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors are compositionally built out of simpler ones. The simplest connectors in Reo are a set of channels with well-defined behavior supplied by users [7].

The Reo coordination language separates the computation part and coordination part of a software system by adopting the concept of "exogenous coordination". The Reo coordination language can be viewed as a triple <**I, C, N** >, where the **I** stands for component instances, the **C** represents connectors, and **N** the nodes as the "glue points" among the elements of **I** and **C**. In addition, Reo also provides a set of operations for components to manipulate connector topology and input/output data.

In the following sections, we introduce component instances, Reo nodes, Reo connectors, and Reo operations, respectively. Then we describe how in Reo one can encapsulate components.

2.2 Component Instances

A component instance contains one ore more active entities (e.g. processes, agents, threads, actors, objects, etc.) which communicate with its outside exclusively through connectors. The internal constituents of a component instance may also be other component instances that are connected by Reo connectors.

2.3 Connector

A Reo connector is constructed out of one or more channels. A Reo channel is also referred to atomic or primitive Reo connectors. Each channel has exactly two directed ends, each of which is either source or sink. A source end accepts data into its channel. A sink end dispenses data out of its channel. A channel can be attached at most one component instance at any given time. The Reo coordination language supports a collection of predefined channel types, each with its well-defined behavior. Figure 2.1 lists some examples of channel types.

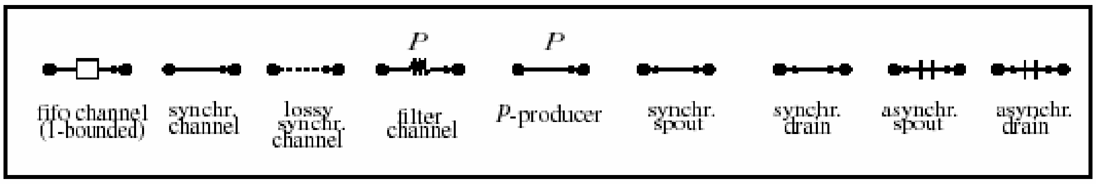
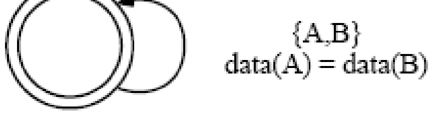


Figure 2.1.Some basic channel types in Reo

The formal semantics of Reo channel types can be defined using Abstract Behavior Types (ABT) [17], or Constraint Automata [19]. For example, a Sync channel type is defined in ABT as:

$$\langle \alpha, a \rangle \longmapsto \langle \beta, b \rangle \ \equiv \ \alpha = \beta \ \wedge \ a = b \ .$$

Or in Constraint Automata as:



.

A Reo connector with more than one channels are often referred to as a *composite Reo connector*. A composite Reo connector normally delivers more complicated behavior than an atomic one. For example, the sequencer connector (Figure 2.2) provides four nodes a, b, c, and d for other entities (component instance/ connectors) to connect with. The retrieving of the data item stored in the first FIFO1 can occur only in the strict left to right order, i.e. from node a to node d. One can find more examples of Reo composite connectors in [18].

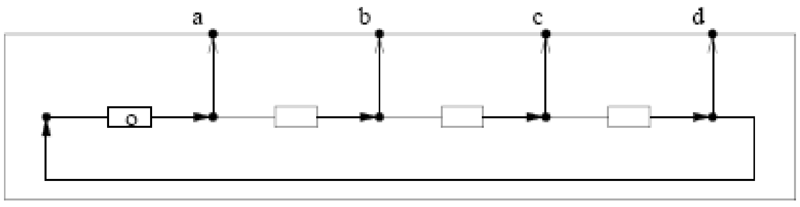


Figure 2.2.Sequencer Connector

2.4 Reo Nodes

A Reo Node is a location where more than one Reo channel end coincides. There are basically three types of nodes: *source, sink, and mixed.*

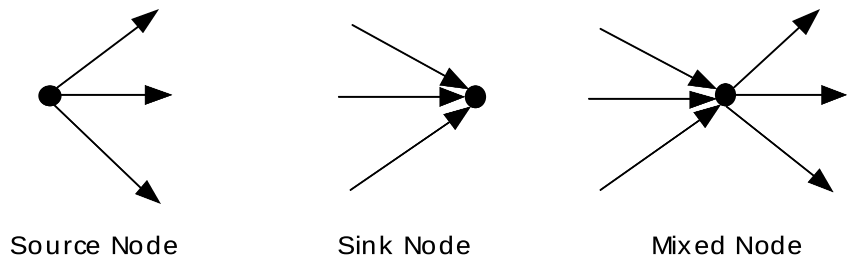


Figure 2.3 Nodes in the Reo Coordination Language (nodes are denoted as black bullets)

A source node consists of only source end of channels. It replicates a data item to its connected channels only when all the channels are ready to accept it. A sink node comprise only sink end of channels. It non-deterministically selects one of the data items from its connected channels when all the channels are trying to dispense the data items. The behavior of a mixed node is the combination of first two types of node, it selects a data item randomly from its input channels and replicates this data item to its output channels when they are ready to accept it.

2.5 Reo operations

Any active entity inside a component instance can perform Reo operations. Reo defines three types of operations: topological – ones that allow manipulation of connector topology, Input/Output – ones that allow input/output of data, and inquiry – ones that allow checking for conditions of interest.

2.5.1 Topological operations

Operation	**Informal Description**
create	This operation creates a channel with specific *type.*
connect	If node N is not a mixed node, N is connected to the component instance as a result of this operation.
disconnect	The component instance is disconnected from node N after performing this operation.
forget	The component performing this operation loses all its references to the node N.
join	This operation joins two distinct nodes, N1 and N2.
split	This operation splits node N by specifying the channel ends that the performer requires to coincide on the new node.
hide	This operation hides the node N such that it cannot be modified in any other operation.

2.5.2 Input/Output operations

Operation	**Informal Description**
read	If N is a sink node connected to the component instance performing this operation, this operation succeeds when a value compatible with pat is

	non-deterministically read from some channel end into the variable v.
take	Similar to read, but the value is also removed from the channel.
write	If N is a source node connected to the component instance performing this operation, this operation succeeds when a copy of the value in v is written to every channel end.

2.5.3 Inquiry operations

Operation	**Informal Description**
wait	Suspends the active entity that performs it (indefinitely or for the specified time-out, t) waiting for the specified conditions to become true.

2.6 Component Encapsulation

In analogy with electrical circuits, we call a design a *circuit* in Reo [29]. To facilitate component abstraction and modular design, in Reo one can define components using a box around a circuit and leaving some of the nodes as *ports* on the box. Note that a port is either input or output point where messages pass through a node [29].

One can instantiate a component in a circuit by drawing a box and the ports on it, without its internals. For example, a sequencer connector in Figure 2.2 can be instantiated as follows in Figure 2.4.

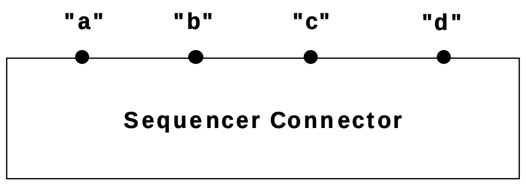


Figure 2.4.Black-Box Representation of Sequencer Connector

Chapter 3
Case Study: A Distributed Meeting Scheduling System

3.1 Introduction

In this chapter, we present our case study on using the Reo coordination language to specify a Distributed Meeting Scheduling System (DMSS). We focus on how the Reo coordination language can contribute to the software architecture description of the DMSS. The system we are modeling is based on an agent-based solution, called “RCal” [21], which is a particular implementation of Contract Net Protocol [20] for distributed meeting scheduling.

In this chapter, we first give an overview of the meeting scheduling problem and its current solutions. Then we present the use cases and the high level architecture of the DMSS. After that, we compose our specification of DMSS using the Reo coordination language.

3.2 Overview

The goal of meeting scheduling is to get a group of people to meet together [22]. Meeting scheduling involves three major concepts: ***participants*** *(who),* ***time*** *(when),* and ***location*** *(where).* Generally speaking, the more independent the participants are, the more difficult the meeting is to be scheduled. For example, a meeting with all participants at the same place is much easier to organize than one involving geographically distributed participants. A typical meeting scheduling process may involve changes of participants, time and location. For instance, participants may change their own decisions after a meeting being initially scheduled, and a meeting time or location may need to be rearranged after having been confirmed by all participants.

We look at two groups of solutions for automating meeting scheduling processes, a centralized and a distributed approach, depending on where the participants’ calendar information is located. Centralized solutions, such as done by MS outlook [23] and IBM Lotus Notes [24], provide basic facilities for calendar sharing based on maintaining calendar information on a central server. These solutions leave many manual tasks for the organizers, such as negotiating of a common meeting time and an appropriate location. Centralized solutions have additional drawbacks such as privacy exposure.

Recent academic researches [25][26][27][28] favor distributed solutions based on intelligent agent technology, where a group of agents performs negotiation on behalf of the meeting participants. The benefits of this approach are that meeting participants no longer need to share their private information with others, and they also do not need to take part in a potentially intensive scheduling process.

3.3 Functional Requirements

We capture the functional requirements of Distributed Meeting Scheduling System using use case diagrams. In figure 3.1, we describe the use cases in two groups, i.e. the human use cases and the agents use cases. The “include” associations indicate the ordering of use cases. In the remainder of this section, we first describe the actors involved in the use cases, followed by the description of the relevant use cases.

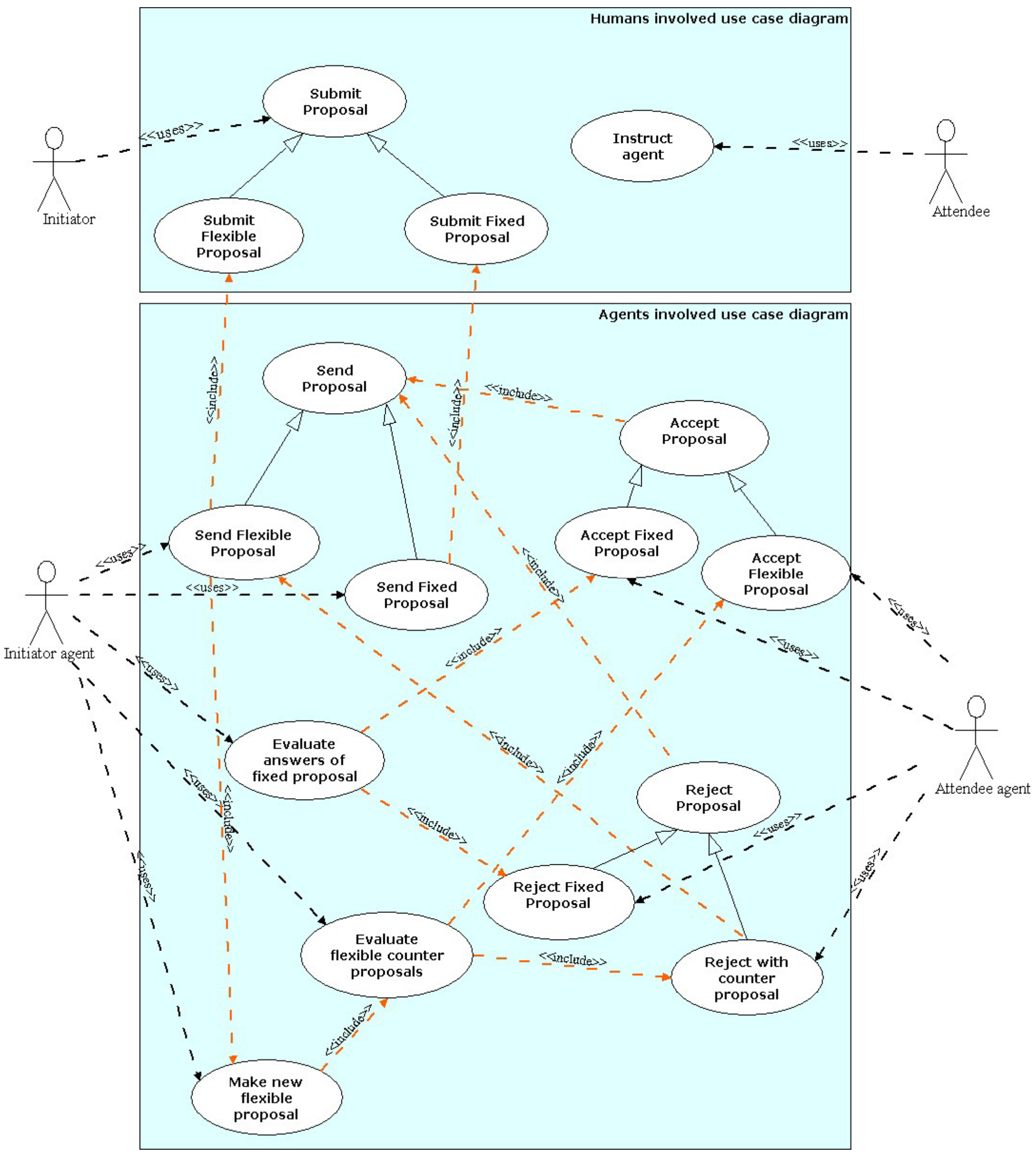


Figure 3.1.Use Case Diagrams of the DMSS

3.3.1 Actors:

The main actors involved in the meeting scheduling process are:

- **Initiator**

An initiator can initiate a meeting scheduling process by providing a meeting proposal to negotiate with all attendees, which may include start time, duration, end time, and attendee list. To simplify the matter, we consider location preferences of attendees non-negotiable.

- **Attendee**
  - **Required attendee**

    All required attendees must attend the meeting so that a meeting can be successfully scheduled.

  - **Optional attendee**

    An optional attendee is a person interested in attending the meeting, but the absence of him/her does not result in the a meeting scheduling process to fail. Note that we further simplify the case by not taking into account the optional attendees, since they do not affect the success of the scheduling process.

- **Meeting Agent**

  A meeting agent is an actor that acts on behalf of an initiator or an attendee during a meeting scheduling process. We distinguish two kinds of meeting agents:

  - **Initiator agent**

    An initiator agent works on behalf of an initiator in a meeting scheduling process.

  - **Attendee agent**

    An attendee's agent works on behalf of an attendee in a meeting scheduling process.

Note that in our case study, an initiator agent is only engaged in one meeting scheduling process at a time, while an attendee agent can be involved in many ongoing meeting scheduling processes at a time.

3.3.2 Use cases

Submit fixed proposal - An initiator submits a fixed proposal for scheduling a meeting. A fixed proposal does not allow any negotiation of meeting parameters - attendees can only accept or reject it.

Send fixed proposal – An initiator agent takes a submitted fixed proposal and sent it to its designated attendee agents for approval.

Instruct agent – An initiator instructs its agent to manage its calendar. This happens once, after which the attendee agent fully automates the scheduling activities on behalf of the attendee.

Accept fixed proposal - An attendee agent accepts a fixed proposal when the attendee is free during the time slot indicated in the fixed proposal.

Reject fixed proposal - An attendee agent rejects a fixed proposal when the attendee is busy during the time slot indicated in the fixed proposal.

Evaluate the answers of fixed proposal - If all necessary attendee agents accept a fixed proposal, the meeting negotiation succeeds, otherwise if any of them rejects the fixed proposal, the meeting negotiation fails.

Submit flexible proposal - An initiator can also submit a flexible proposal for scheduling a meeting. In contrast to a fixed proposal, a flexible proposal (a) may offer an interval within which the meeting can be scheduled and (b) allows attendees to respond the current proposal by negotiating counter proposals for alternative time slots of scheduling.

Send flexible proposal - An initiator agent takes a submitted flexible proposal and sent it to its designated attendee agents for approval.

Accept flexible proposal – An attendee agent accepts a flexible proposal if it finds out that the attendee is free during the interval specified in the flexible proposal.

Reject with counter proposal – An attendee agent rejects the flexible proposal if there is no time slot available for the attendee during the interval indicated in the flexible proposal. If there are some available time slots in the interval, an attendee agent responds with a counter proposal.

Evaluate flexible counter proposals – If any of the attendee agents rejects the flexible proposal, the meeting negotiation fails. Otherwise, the initiator agent evaluates all the alternative time slots and looks for the earliest common time slot of all the attendees. If it finds one, then the meeting negotiation succeeds. Otherwise it fails.

Make new flexible proposal – If the previous use case (evaluate flexible counter proposals) does not succeed, the initiator agent can start a new round of negotiation by making an updated flexible proposal.

Use case relations for meeting agents

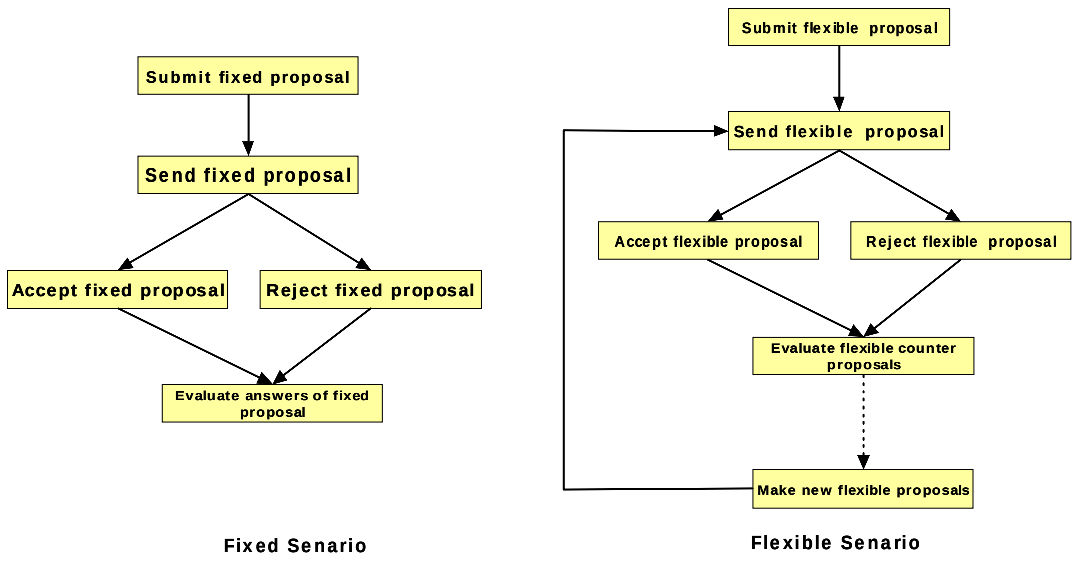


Figure 3.2. Relations of Use Cases

Figure 3.2 shows the relations between the use cases of the meeting agents. In the fixed scenario, an initiator first submits a flexible proposal, then its agent sends the fixed proposal to the attendee agents; upon receiving the fixed proposal, an attendee agent either accepts or rejects the fixed proposal; then the initiator agent evaluates answers of a fixed proposal. In the flexible scenario, an initiator submits a flexible proposal and its agent sends the flexible proposal to the attendee agents; upon receiving the flexible proposal, an attendee agent either accepts the flexible proposal or rejects it with a counter proposal; then the initiator agent evaluates the flexible counter proposals, if it fails to find an earliest common time slot, the initiator agent makes a new flexible proposal and start the negotiation all over.

3.4 High level architecture

At a high level of abstraction, the distributed meeting scheduling system (DMSS) consists of the following types of components (Figure 3.3):

1) Calendar Database

2) **Meeting Agent**

3) **Agent Coordination Hub**

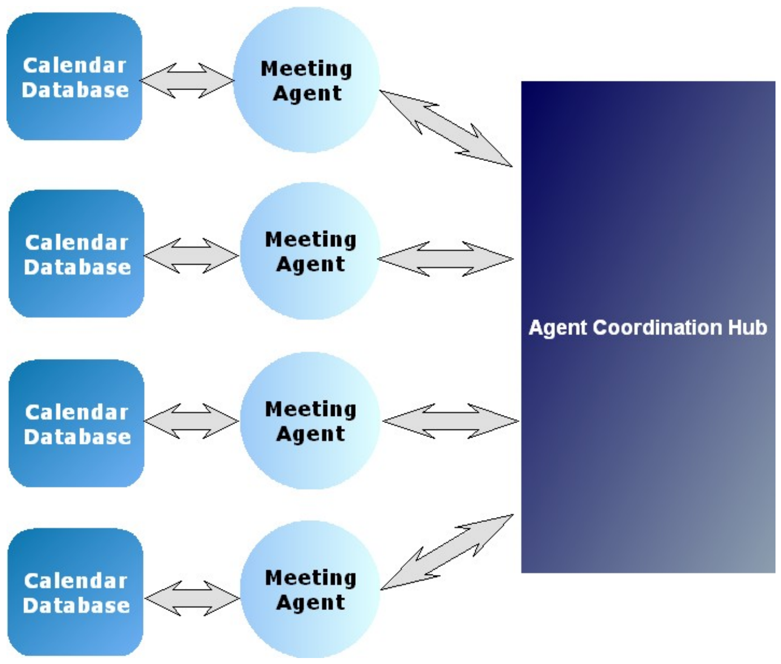


Figure 3.3.The High Level Architecture of DMSS

Each meeting agent interacts with one calendar database that provides calendar information. The Agent Coordination Hub coordinates collaborations among these meeting agents by ensuring the proper routing of all messages to and from meeting agents.

Depending on the role of the participant, a meeting agent can behave in three modes: “initiator mode”, “attendee mode”, or “dual mode”. In “initiator mode”, the meeting agent is responsible for initiating the meeting scheduling process, sending out proposals, evaluating responses, generating results, etc. While in “attendee mode”, the meeting agent responds to proposals for meeting in which its attendee may participate. Moreover, it’s also possible for a meeting agent to act in both “initiator mode” and “attendee mode” at the same time, which is called “dual mode”. In “dual mode” the agent represents an initiator who also participates in the meeting. We focus on initiator and attendee mode and leave the decision of attending and initiating the same meeting to users.

An attendee agent can respond to multiple meeting proposals from different initiators concurrently. To avoid the situation in which the same time slot is allocated in more than one proposal, we use the time slot(s) reserving mechanism [21] where the accepted time slot or the alternative time slots are marked as “reserved” in the calendar database and are either to be confirmed or aborted later in the meeting scheduling process.

In the following sections of this chapter, we present our DMSS software architecture specified using the Reo coordination language.

3.5 DMSS software architecture

In this section, we refine the high level architecture of DMSS into formal software architecture using the Reo coordination language. We specify the DMSS in two parts: fixed DMSS specification and flexible DMSS specification. The fixed DMSS specification handles fixed proposals, while the flexible one handles flexible proposals.

We model the software architecture into components, each of which offers one or more input or output Reo ports (nodes on the border of components). The components with solid gray color in both specifications are implemented externally and thus are not specified using Reo. The Reo channels are depicted as lines with arrowhead(s), the Reo nodes are depicted as black circles.

In the remainder of this section, we first describe the basic components and connectors used in both specifications, and then we present in detail these two specifications.

3.5.1 Basic components

The basic components below (Figure 3.4) provide data processing facilities.

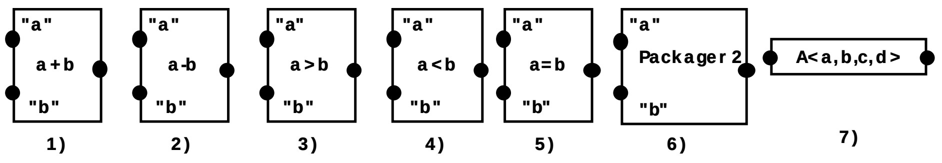


Figure 3.4.Basic Data Processing Components

The first and second components output the result of a + b and a – b. The third, fourth and fifth component output "true" only if a > b, a < b, or a =b, and "false" otherwise. The sixth component wraps two inputs, from node "a" and "b", into a pair <a, b>, it can be parameterized into packager N. The last component, "tuple 4", takes as input an tuple <a, b, c, d> and output the first element of the input tuple, B<a, b, c, d> outputs the second element of the tuple, and so on. The tuple component on the figure can be parameterized to tuple N.

We do not specify these basic components further. An algebraic specification can be done similar to the one provided for the sum (+) component in [17].In addition, we also introduce two basic data storing components: *Variable Component* and *Constant Writer Component.*

Variable Component

The Variable Component (Figure 3.5) serves as placeholder for data items similar to a variable in imperative programming language [29].From the "write" port, a user can set the value of the Variable Component, while the data item can be read from the "read" port.

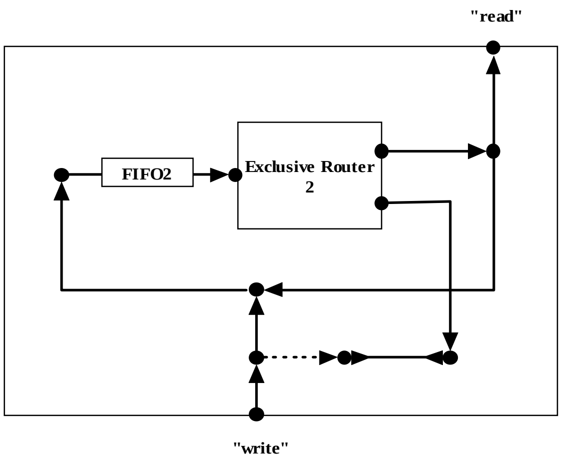


Figure 3.5.Varable Component

Constant Writer Component

A constant writer component serves as a constant value provider, and its value can only be set once during the component instantiation [29].

3.5.2 Basic connectors

Exclusive Router N

The Exclusive Router N (Figure 3.6) routes synchronously its input to precisely one of its outputs [29]. We depict an instance of the Exclusive Router 3 as a circle with three outgoing arrows.

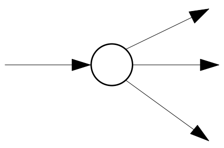


Figure 3.6.Exclusive Router 3

Inclusive Router N

The Inclusive Router N routes synchronously its input to K (K ≤N) of its outputs. We depict an instance of the Inclusive Router 3 (Figure 3.7) as a square with three outgoing arrows.

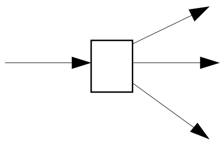


Figure 3.7.Inclusive Router 3

Initially Closed Valve (ICV)

An initially closed valve [18] (Figure 3.8) regulates the flow of data. The ICV initially does not allow flow of data. It has one node through which one can toggle its state from closed to opened and the other way around.

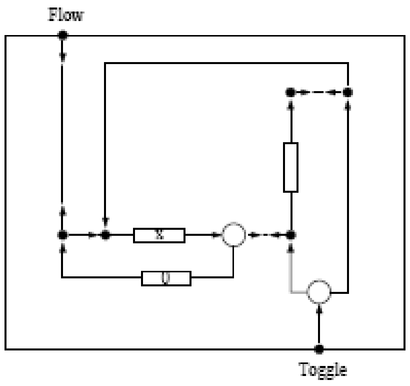


Figure 3.8.Initially Closed Valve(ICV)

3.5.3 Fixed DMSS specification

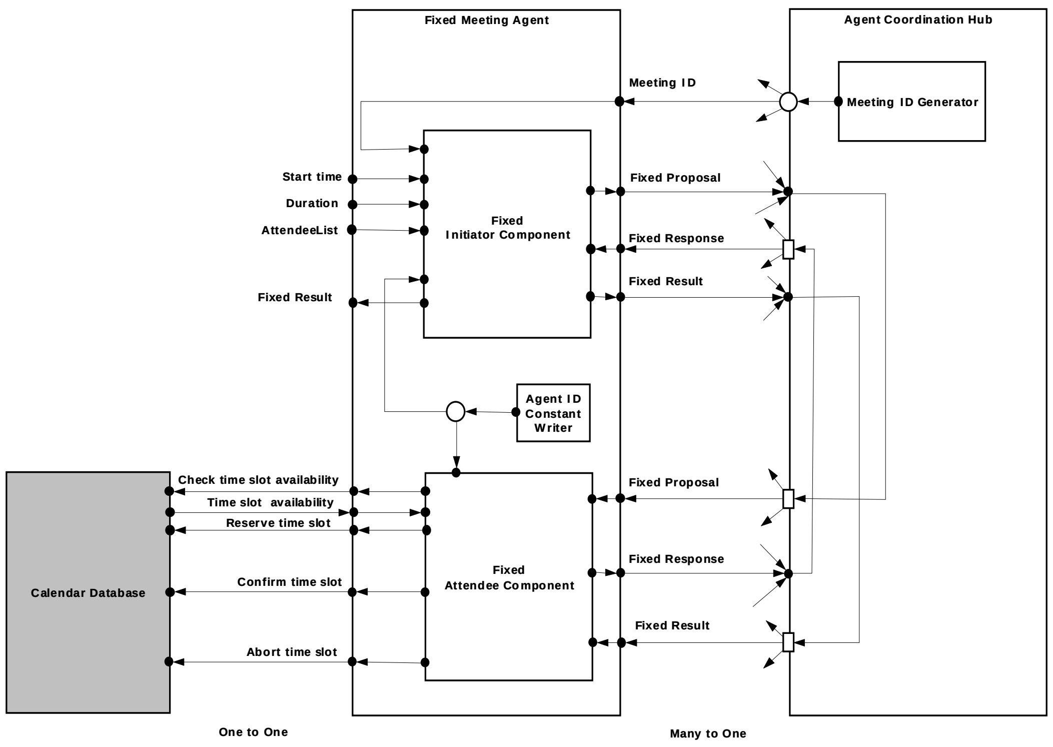


Figure 3.9.The High Level Model of fixed DMSS

The high-level model of the fixed DMSS (Figure 3.9) consists of three parts, i.e. Fixed Meeting Agent, Calendar Database, and Agent Coordination Hub. They work together to automate the meeting scheduling process for fixed proposals. The Fixed Meeting Agent, as depicted in the middle of the diagram, is the core part of the system. It further consists of the Fixed Initiator Component and the Fixed Attendee Component. The Fixed Initiator Component is activated when the user inputs the proposed meeting information, i.e. start time, duration, and attendee list. The Fixed initiator Component is responsible for sending out the fixed proposals, evaluating the fixed responses from the attendees' meeting agents, and generating fixed meeting scheduling results. The Fixed Attendee Component receives the fixed proposals from the initiator's meeting agents, generates fixed responses by querying its external calendar database, and receives the fixed meeting scheduling results. The responsibility of the Agent Coordination Hub is to properly route messages (fixed proposals, fixed responses, and fixed results) to the involved meeting agents. The Agent ID Constant Writer keeps each agent's unique ID (e-mail) so that all the meeting agents can be easily identified. The Calendar Database holds the individual calendar data of the meeting attendees and can be assessed through a set of dedicated interfaces.

In the Agent Coordination Hub, the “Meeting ID generator” (Figure 3.10) generates unique ID numbers for meeting scheduling processes. The Exclusive Router is to ensure that a Meeting ID can only be assigned to one particular meeting agent. Three Inclusive Routers are used to deliver messages (fixed proposals, fixed responses, and fixed results) to proper meeting agents.

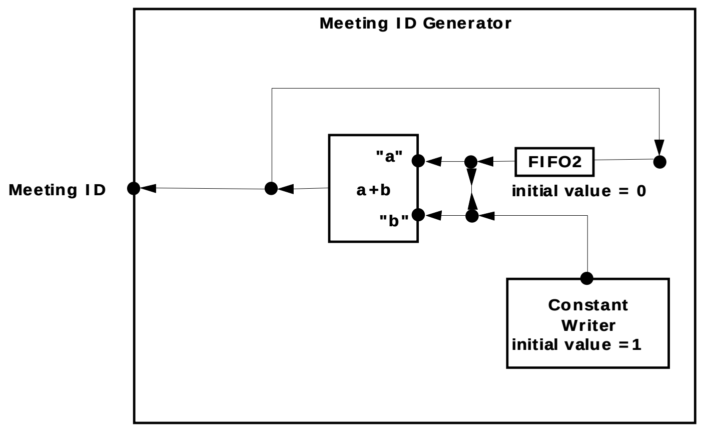


Figure 3.10.Meeting ID Generator

Fixed Initiator Component

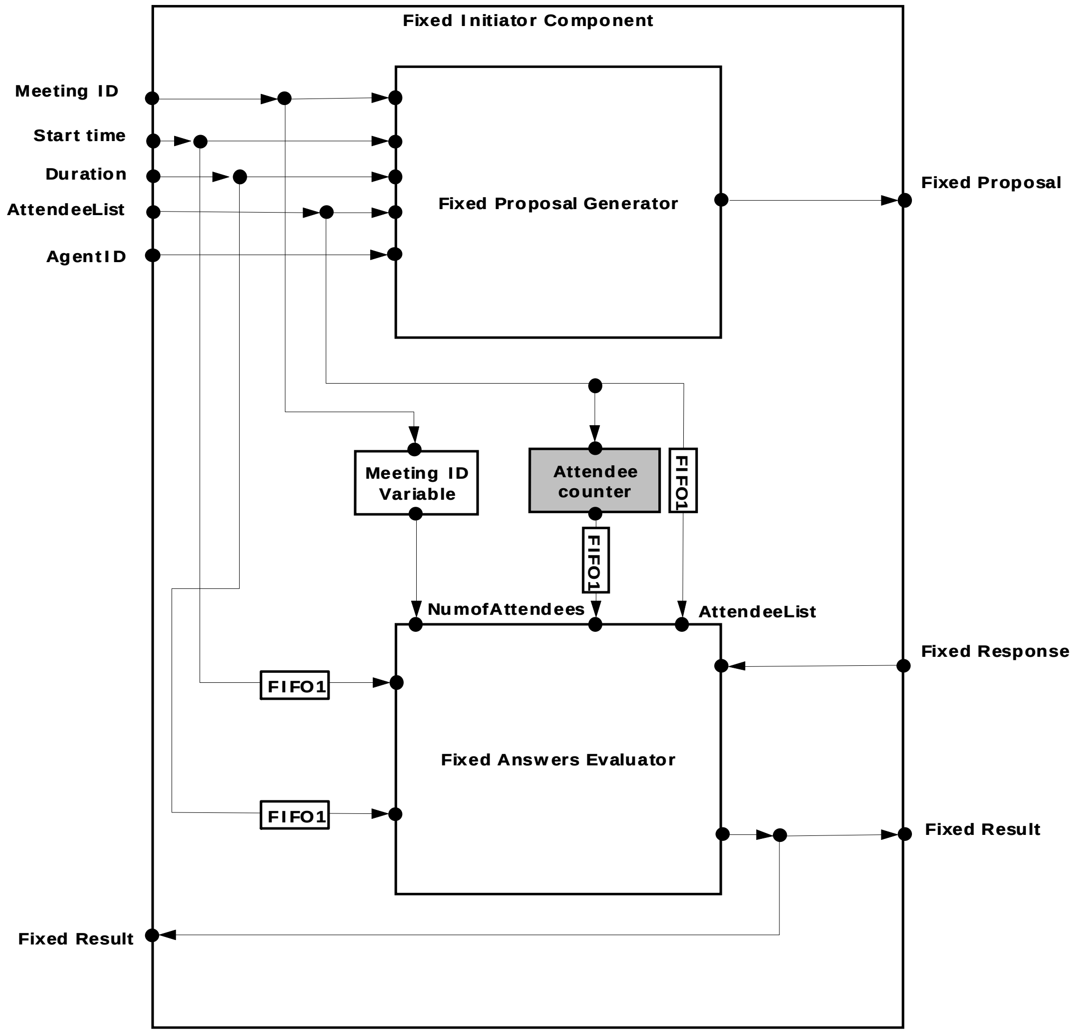


Figure 3.11.Fixed Initiator Component

The Fixed Initiator Component (Figure 3.11) consists of two main subcomponents. The Fixed Proposal Generator (Figure 3.12) gathers all the proposal information, including Meeting ID, start time, duration, and attendee list. An example of fixed proposals is listed as follows:

Fixed Proposal

Meeting ID: 00002324

Start Time: 26/05/2005 10:00:00

Duration: 45

Attendee list: user1@com.com, user2@com.com, user3@com.com.

The "Packager 4" makes a fixed proposal when and only when all the necessary information is available. The Attendee Counter", marked as solid gray in the Fixed Initiator Component, counts the number of attendees in the attendee list and stores it in a FIFO1 channel, which is used by the Fixed Answers Evaluator.

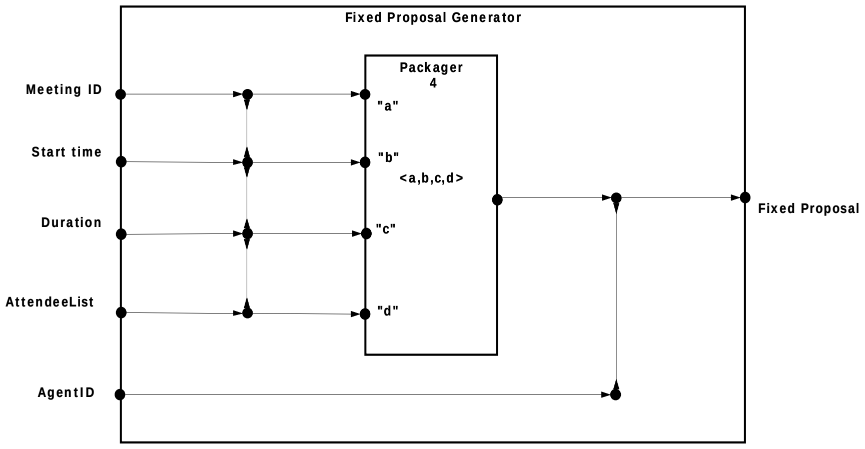


Figure 3.12.Fixed Proposal Generator

The Fixed Answers Evaluator (Figure 3.13) generates a fixed result after evaluating all the fixed responses from the attendees' meeting agents. The Response Switch (Figure 3.14), blocks the fixed responses from uninvolved attendee's agents by comparing the Meeting ID with the Meeting ID of the fixed responses. It also examines whether a fixed response has expired by comparing the expiry time of the response package with the current time. In case the fixed response has expired, the Response Switch will block the response and generates a "false" message, the Fixed Answers Evaluator puts a "fail" into the fixed result message.

Upon receiving a fixed response, the Fixed Answers Evaluator checks whether it contains a "reject" answer or not, if a "reject" answer is detected, the Fixed Answers Evaluator generates a "fail" message. Otherwise if all the fixed responses contain "accept" messages, the Fixed Answers Evaluator generates a “success” message. An example of a fixed result is shown as follows.

Fixed result

Meeting ID: 00002345

Status: success/fail

Attendee List: user1@com.com, user2@com.com, user3@com.com.

Start time: 26/05/2005 10:00:00

Duration: 45

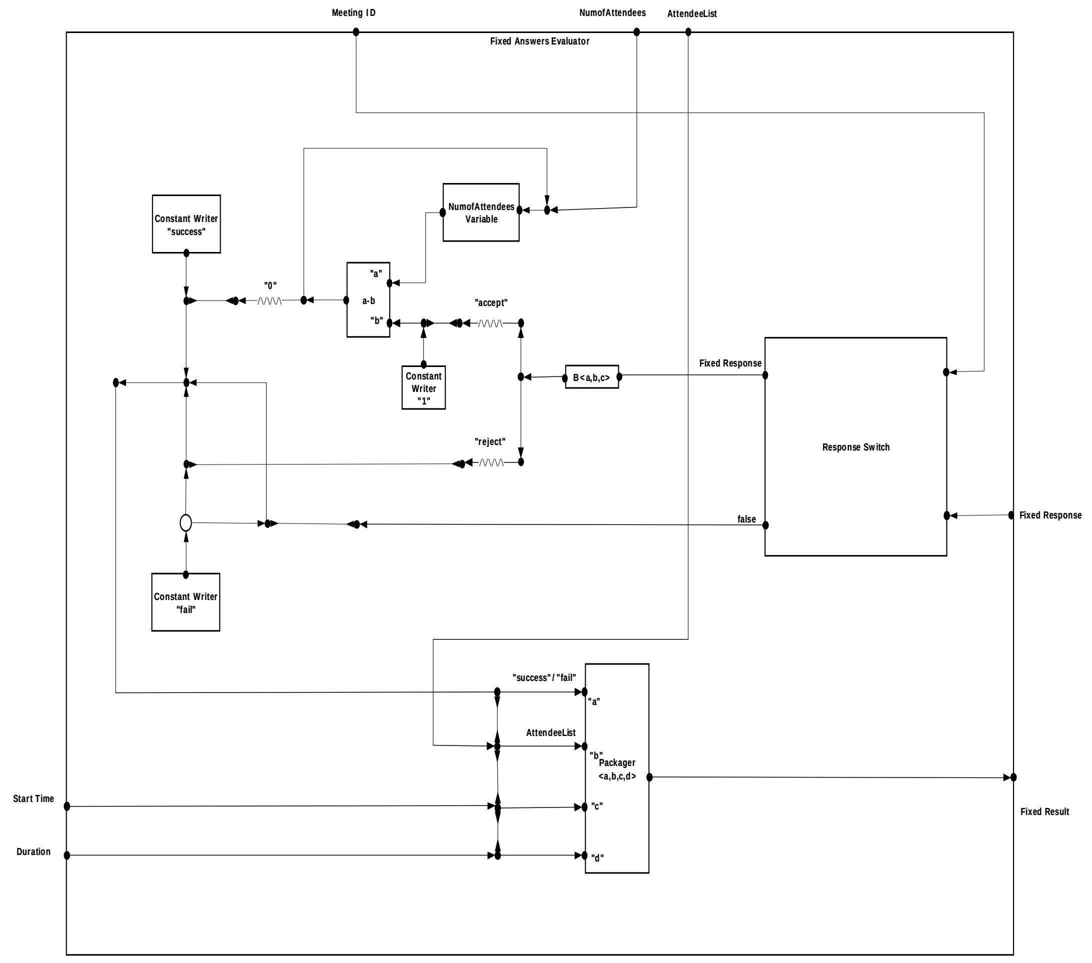


Figure 3.13.Fixed Answers Evaluator

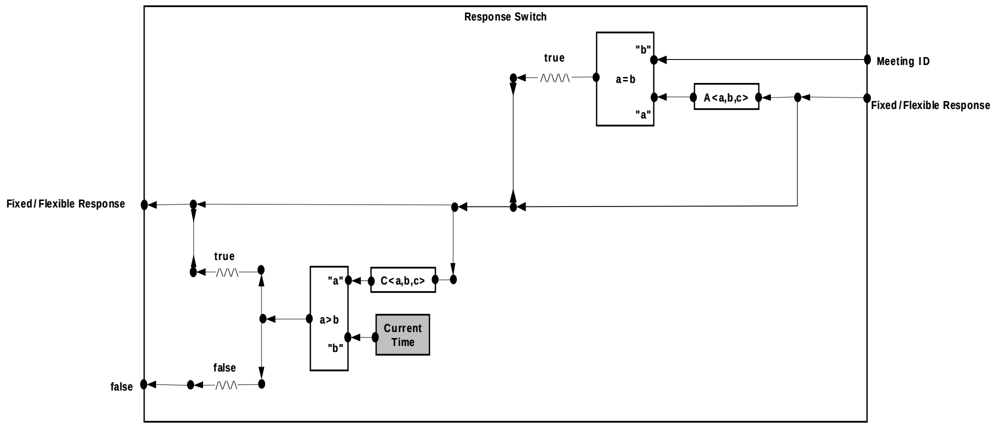


Figure 3.14.Response Switch

Fixed Attendee Component

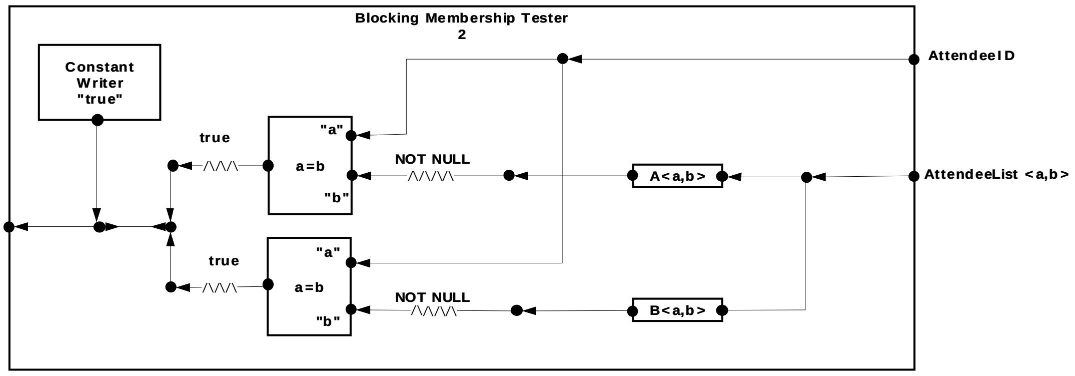


Figure 3.15.Blocking Membership Tester 2

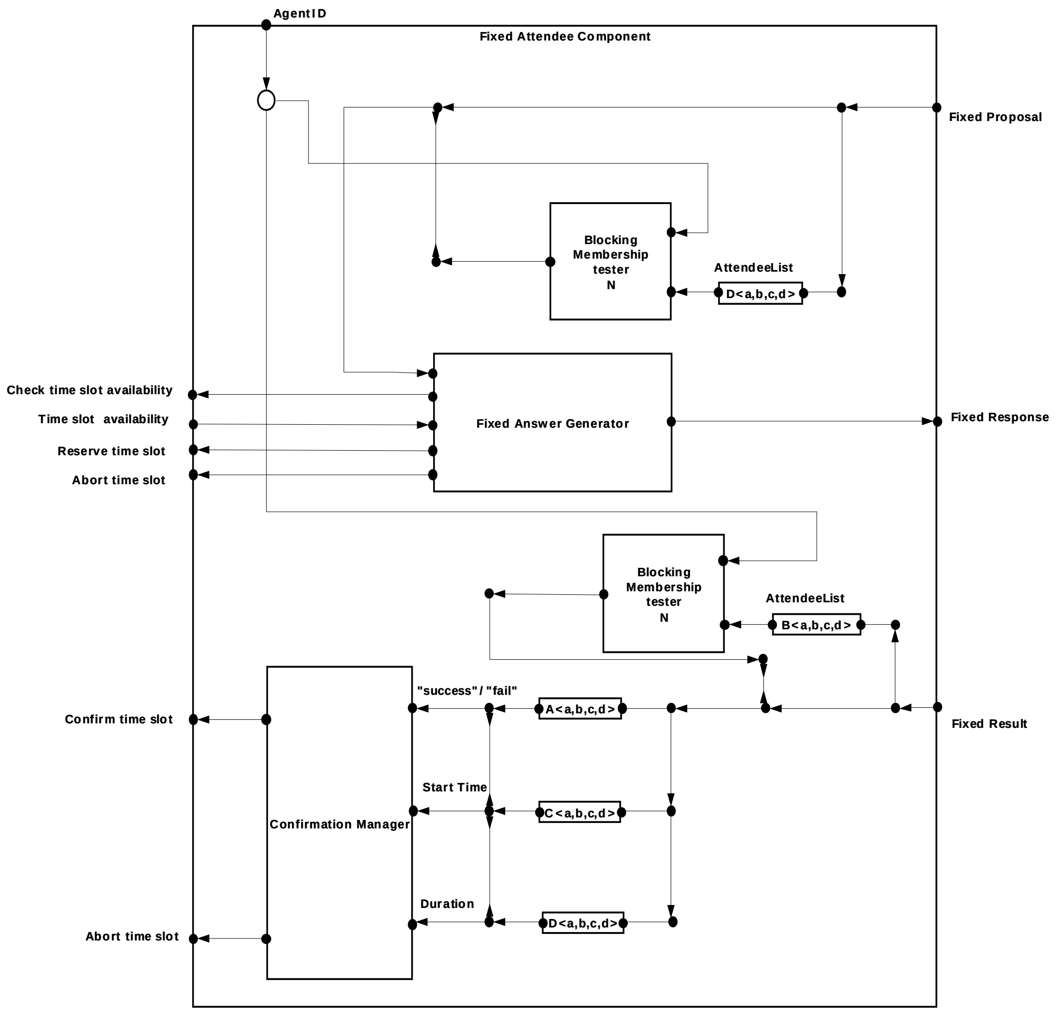


Figure 3.16.Fixed Attendee Component

By using Blocking Membership Testers (Figure 3.15), the Fixed Attendee Component (Figure 3.16) blocks any fixed proposals or results that are not destined for it. This is one way we implement "packet switching" as known from computer networks. The Blocking Membership Tester outputs a "true" message only when this meeting agent is in the attendee list of the fixed proposal or result, which enables the flow of a fixed response in the Fixed Attendee Component. Note that the specification of the Blocking Membership Tester shown in Figure 3.15 only deals with an attendee agent list of two, but it can be further parameterized to Blocking Membership Tester N before the DMSS instantiation. A Blocking Membership Tester N works with size K≤N of attendee list.

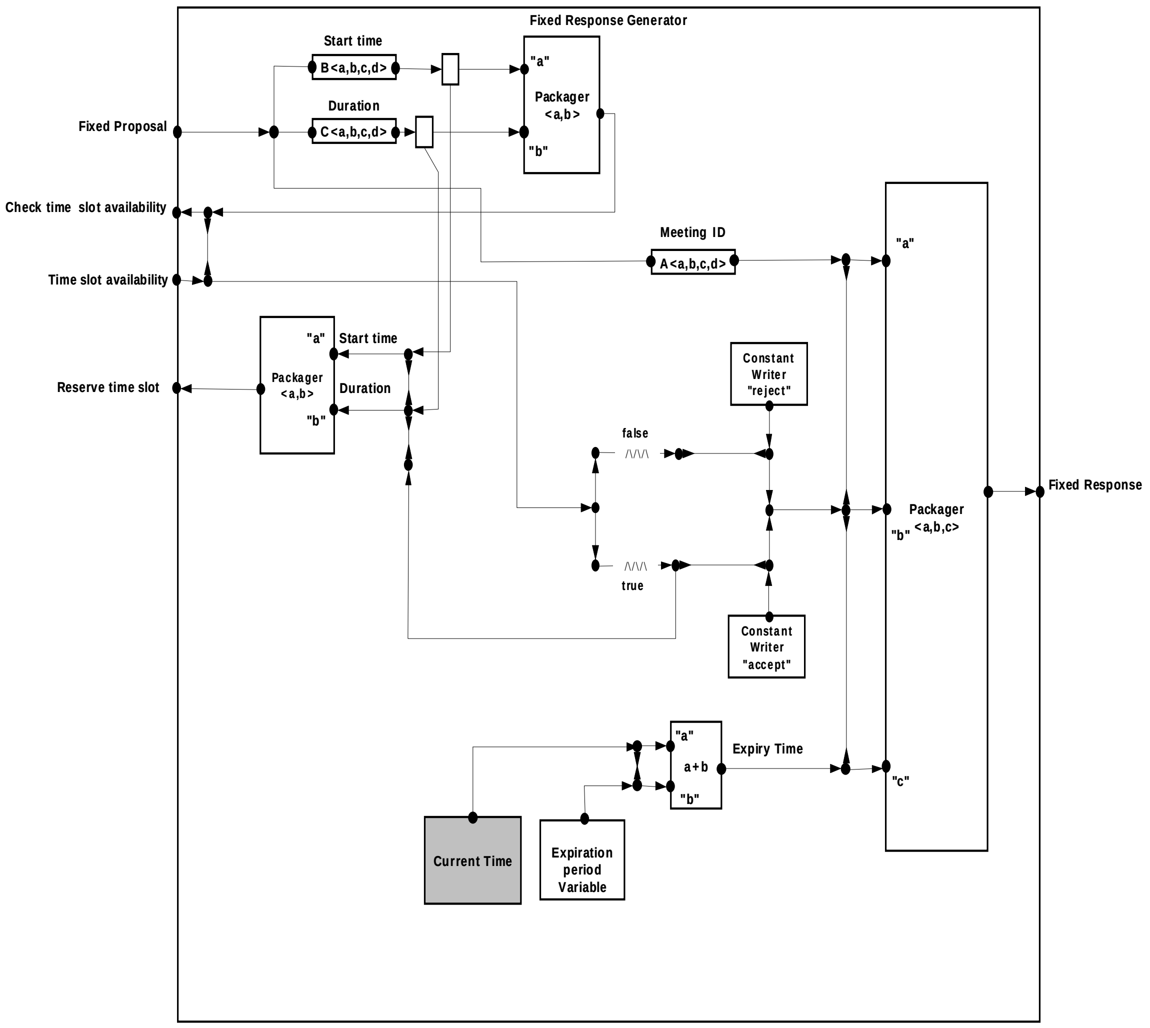


Figure 3.17.Fixed Response Generator

The Fixed Response Generator" (Figure 3.17) checks the availability of the time slot specified in the fixed proposal by querying its external calendar database. If it returns "true", which means that the time slot is free, it reserves the time slot in the calendar database and passes an "accept" for the response message. Otherwise, it only generates a "reject" message. In the response message, the fixed response generator issues an expiry timestamp by using the Current Time and the Expiration Period Variable. An example of fixed response is listed below.

Fixed response

Meeting ID: 00002324

Answer: accept/reject

Expiry time: 23/05/2005 10:00:00

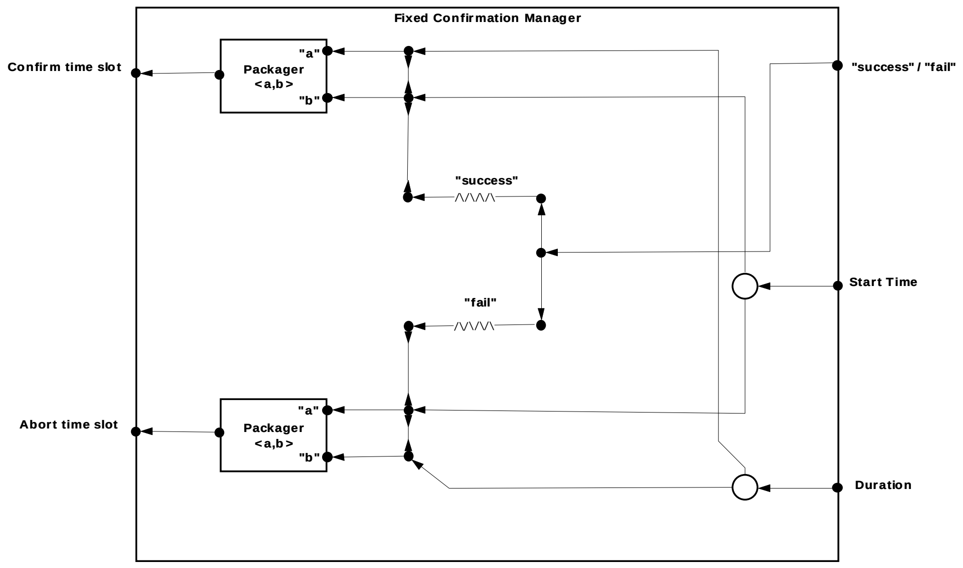


Figure 3.18.Confirmation Manager

The Confirmation Manager (Figure 3.18) confirms the time slot when it receives a “success” message; otherwise, it aborts the reserved time slot.

3.5.4 Flexible DMSS specification

Since the flexible DMSS shares a lot of functions with the fixed one, we extend the fixed DMSS by modifying some of its circuitry. In the high level structure of flexible DMSS (Figure 3.19), we add two ports to the Flexible Initiator Component: start and end time. "Start" input enables a new meeting scheduling process. The "end time" input indicates that the meeting can be scheduled at any time slot between the start time and the end time.

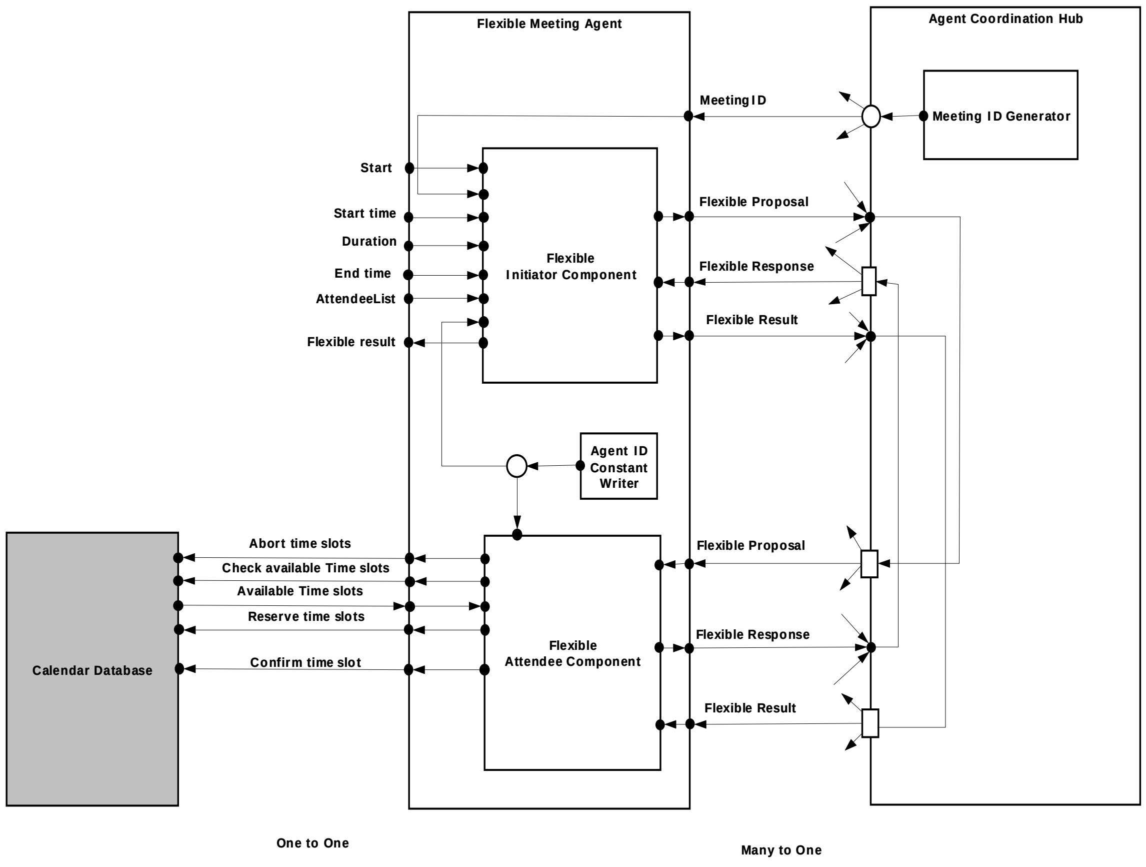


Figure 3.19.The High Level Structure of Flexible DMSS

Flexible Initiator Component

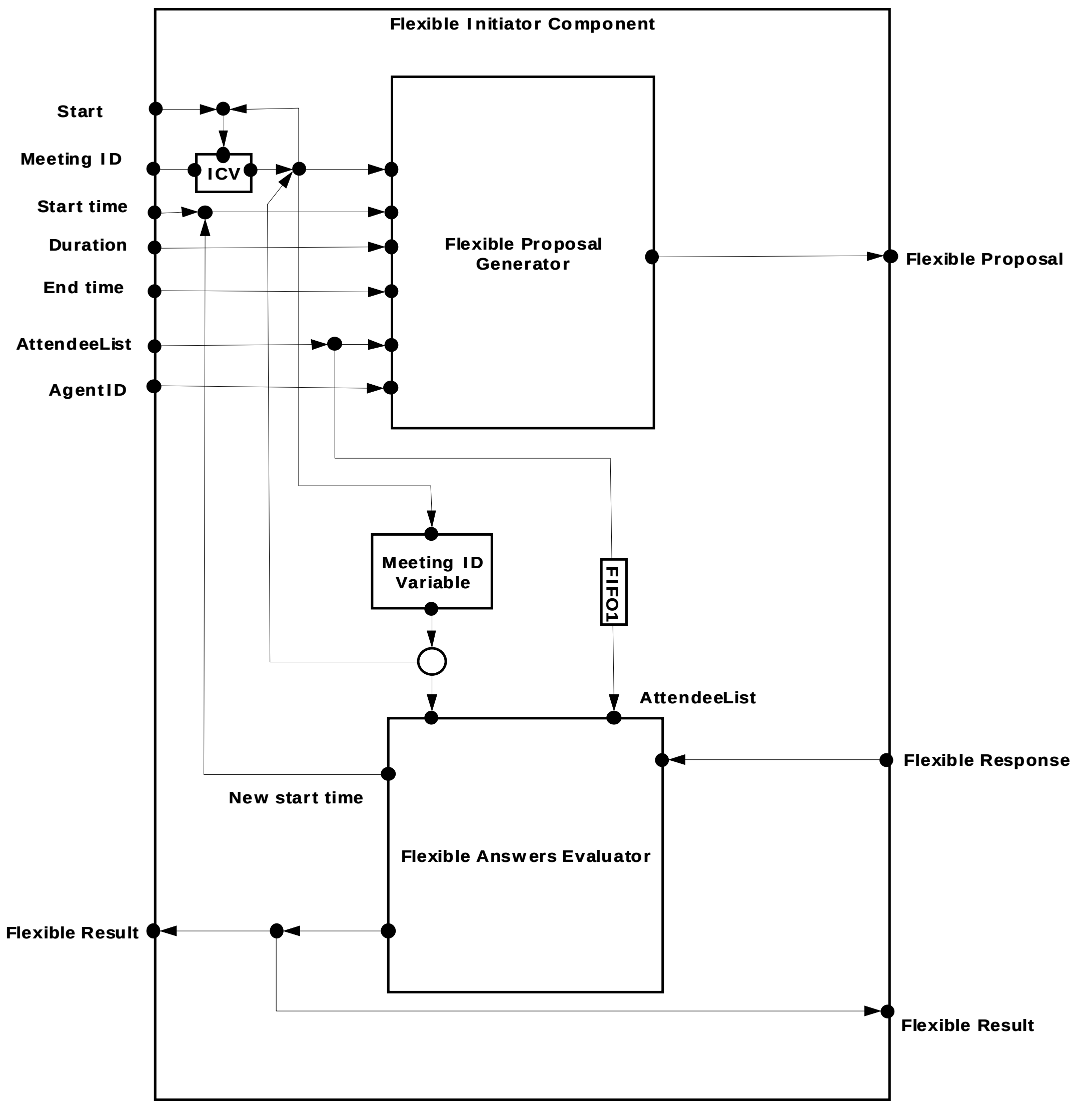


Figure 3.20.Flexible Initiator Component

To start a meeting scheduling process, the initiator first inputs a “start” signal to the Flexible Initiator Component (Figure 3.20), which toggles the state of Initially Closed Valve (ICV) from closed to opened and hence enables the flow of data. Then the initiator inputs meeting information, such as start time, duration, and end time, to the Flexible Proposal Generator. After this information is inputted, the state of ICV is changed to “closed” again, and the Flexible Proposal Generator (Figure 3.21) makes a meeting proposal. An example of a flexible proposal is listed as follows:

Flexible Proposal

Meeting ID: 00002345

Start Time: 28/05/2005 14:00:00

Duration: 30

End Time: 10/06/2005 18:00:00

Attendee list: user1@com.com, user2@com.com, user3@com.com

In addition, the Flexible Proposal Generator also keeps copies of duration, end time, and attendee list in FIFO channels, and when the Flexible Responses Evaluator generates a new meeting start time, the Flexible Proposal Generator makes a new flexible proposal automatically.

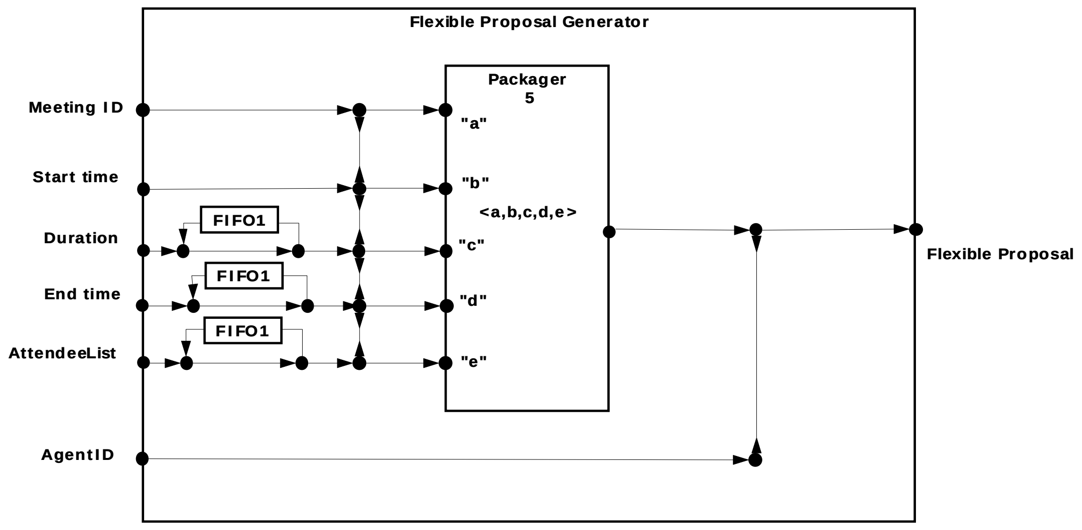


Figure 3.21.Flexible Proposal Generator

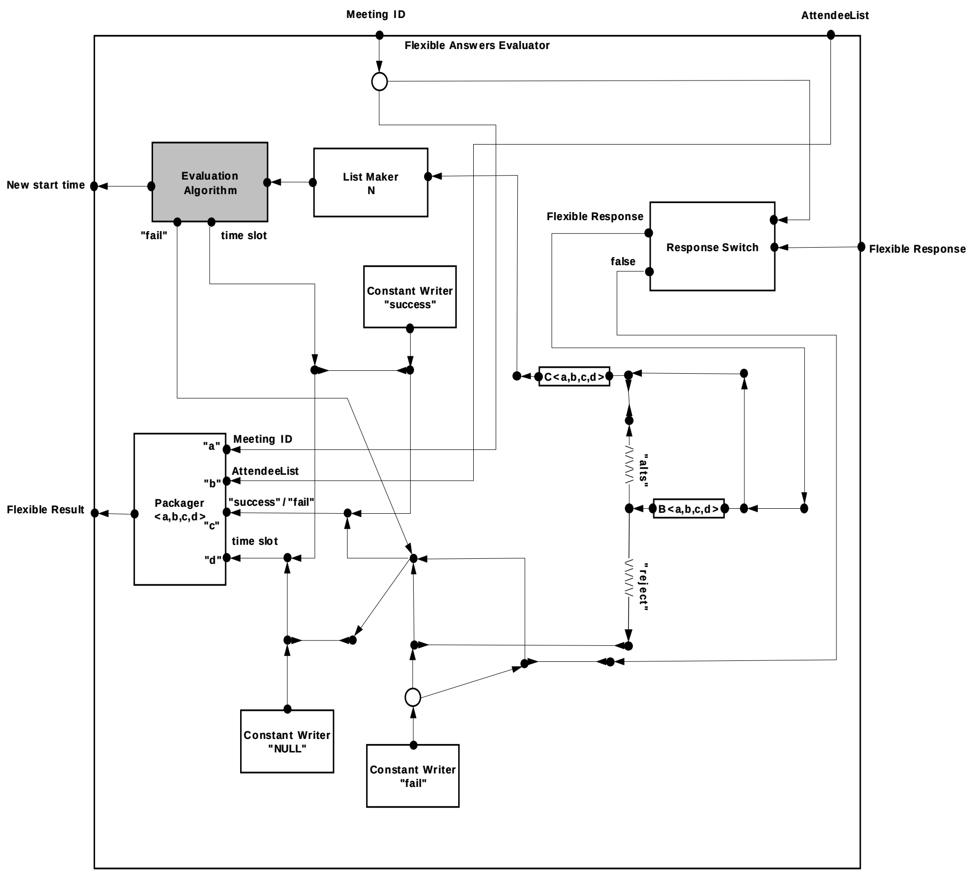


Figure 3.22.Flexible Responses Evaluator

The Flexible Responses Evaluator (Figure 3.22) blocks all the expired and illegal responses using a response switch. Illegal flexible responses are those responses not designated for this meeting agent. If any flexible response expires, it returns a “fail” message and a “NULL” time slot. Otherwise, the flexible responses evaluator uses the evaluation algorithm to search all the alternative time slots of the attendees for an earliest common time slot, if there is such a time slot, it puts the time slot into the result message with a “success” message. If no such time slot exists, it generates a new start time earlier than the end time; otherwise the meeting scheduling process fails. An example of a flexible result is shown below.

Flexible result

Meeting ID: 00002345

Attendee List: user1@com.com, user2@com.com, user3@com.com.

Status: success/fail

Time slot: {26/05/2005 10:00:00, 45}/NULL

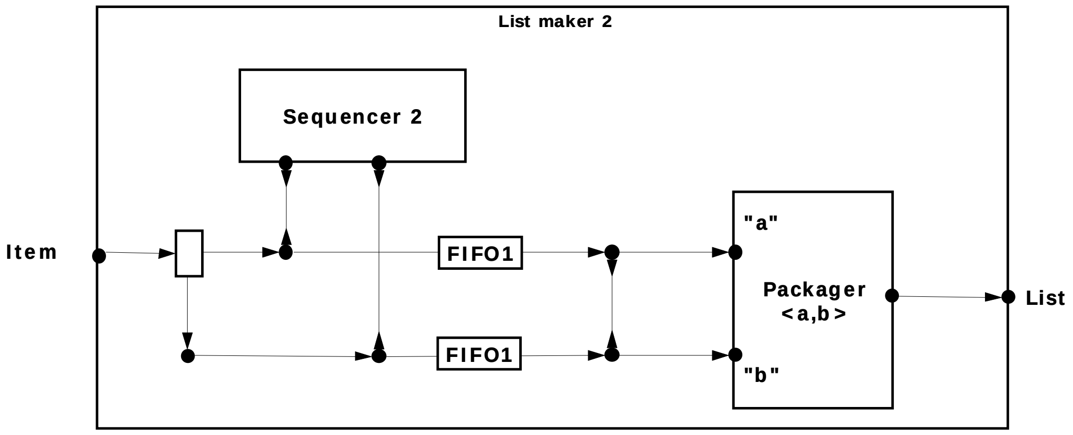


Figure 3.23.List Maker 2

The “List Maker N” creates a list by consuming N items from it inputs port one after another. When the first item comes in, the sequencer N allows it to be stored in the first FIFO1 channel, and the second one can be accepted by the second FIFO1 channel, and so on. Once all the FIFO1 channels are full, a list of these items is made. In Figure 3.23 we show the list maker for two attendees only.

Flexible Attendee Component

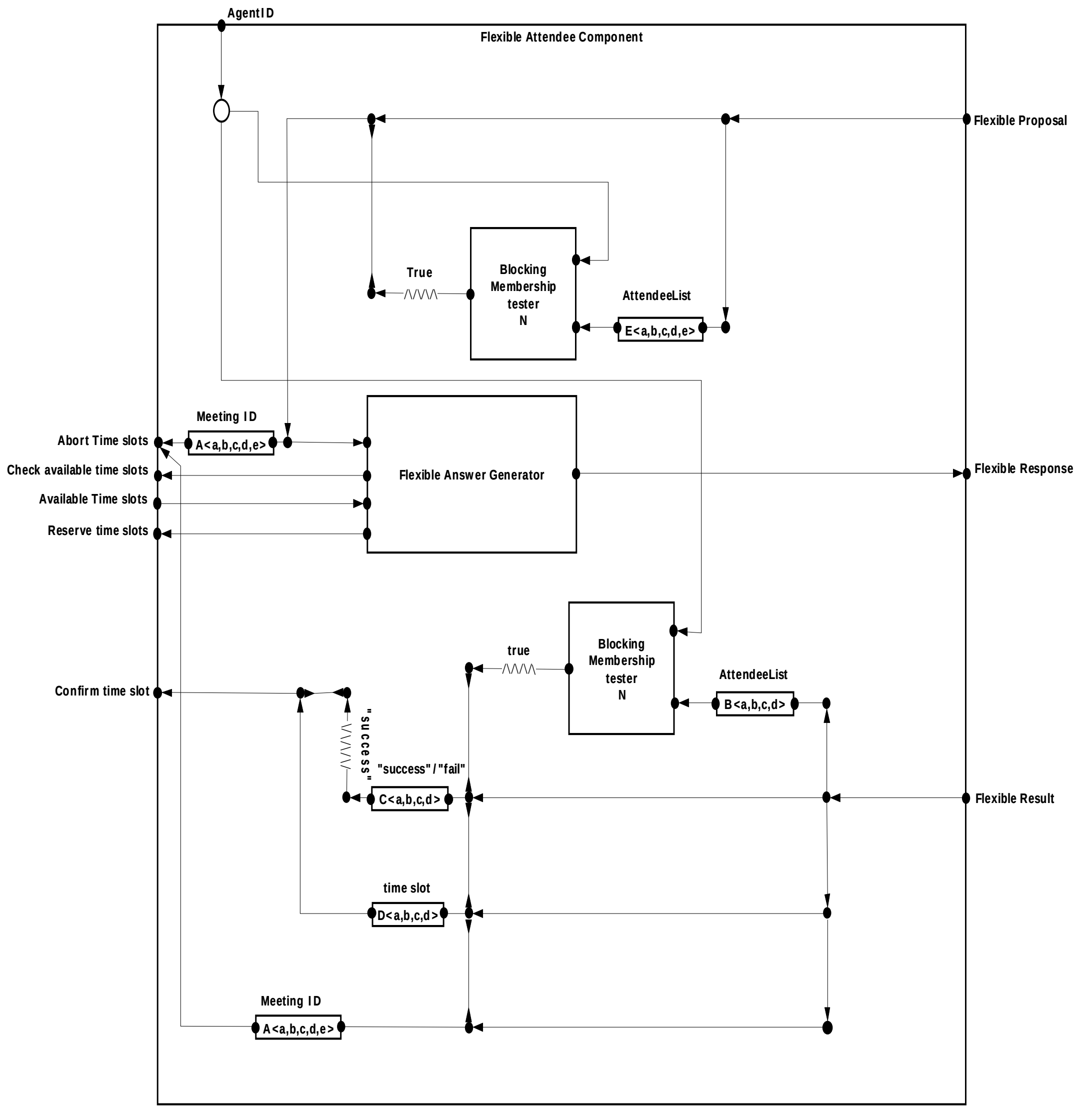


Figure 3.24.Flexible Attendee Component

The Flexible Attendee Component (Figure 3.24) blocks all illegal flexible proposals and results using Blocking Membership Testers. Upon receiving new proposals or any result form the initiator's agent, the "abort time slots" signals the calendar database to abort the currently reserved time slots identified by the Meeting ID contained in the new proposal or result. When a flexible result comes, the Flexible Attendee Component confirms the time slot if the result message is "success".

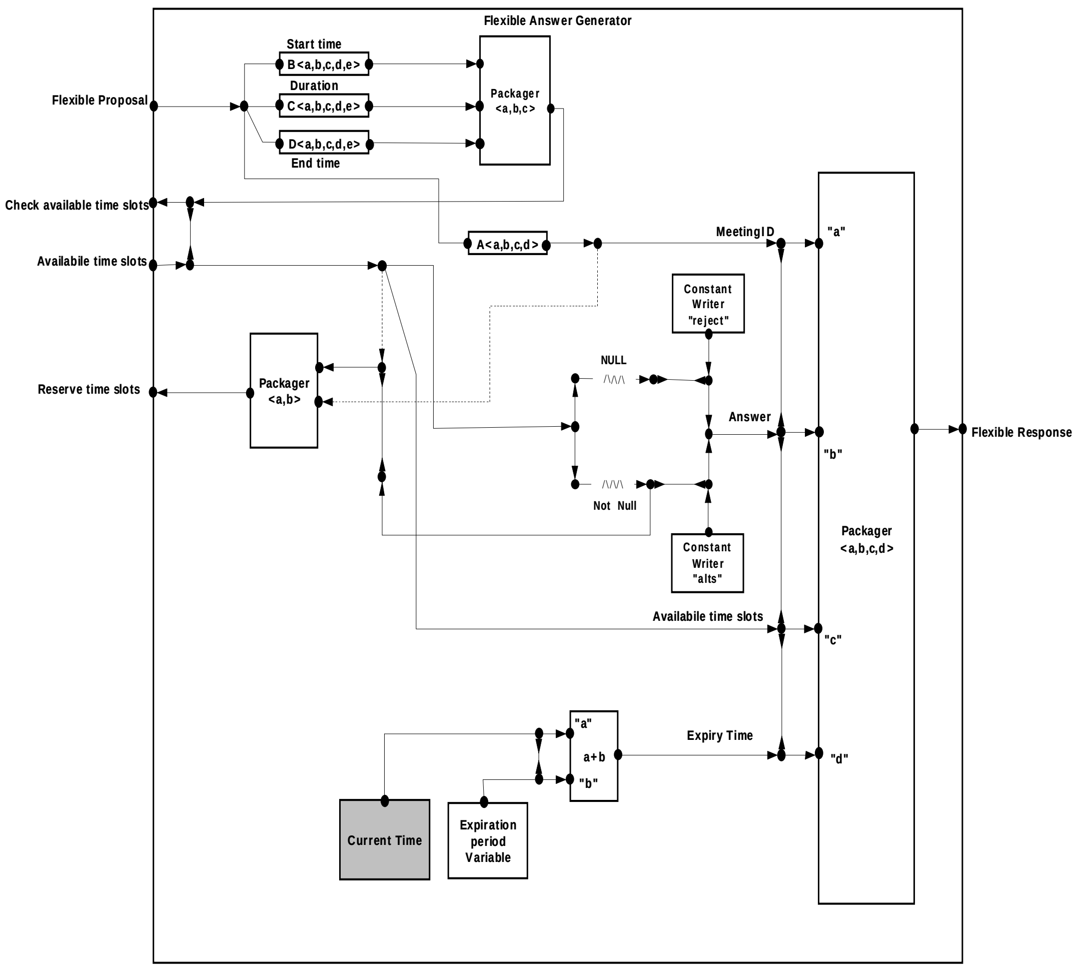


Figure 3.25.Flexible Response Generator

The Flexible Response Generator (Figure 3.25) checks the available time slots upon receiving a flexible proposal, and if no available time slot exists, it returns a “reject” response; otherwise it returns an “alts” answer and reserve these time slots identified by the Meeting ID. An example of a flexible response is shown as follows.

Flexible response

Meeting ID: 00002345

Answer: reject

Alternatives: NULL

Expiry time: 23/05/2005 10:00:00

Or

Meeting ID: 00002345

Answer: alts

Alternatives:

29/05/2005 14:00:00

30/05/2005 10:00:00

30/05/2005 14:00:00

31/05/2005 09:30:00

01/06/2005 10:30:00

Expiry time: 23/05/2005 10:00:00

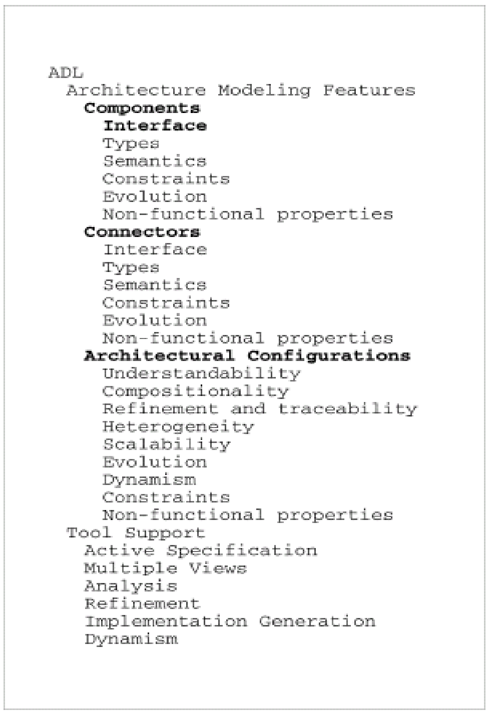
Chapter 4
An Analysis of Reo as an ADL

4.1 Introduction

In this chapter, we use an evaluation criteria to assess the Reo coordination language as an ADL. The analysis is based on observations during the case study, as well as on reports of previous work [29][18] [31] .

4.2 A Overview of the Evaluation Criteria

We base our criteria to a large extent on the *"Classification and Comparison Framework for Software Architecture Description Languages"* [2], as shown in Figure 4.1.



- ADL
  - Architecture Modeling Features
    - **Components**
      - **Interface**
      - Types
      - Semantics
      - Constraints
      - Evolution
      - Non-functional properties
    - **Connectors**
      - Interface
      - Types
      - Semantics
      - Constraints
      - Evolution
      - Non-functional properties
    - **Architectural Configurations**
      - Understandability
      - Compositionality
      - Refinement and traceability
      - Heterogeneity
      - Scalability
      - Evolution
      - Dynamism
      - Constraints
      - Non-functional properties
  - Tool Support
    - Active Specification
    - Multiple Views
    - Analysis
    - Refinement
    - Implementation Generation
    - Dynamism

Figure 4.1. ADL classification and comparison framework. Essential modeling features are in bold font.

This framework identifies the common features and requirements on what an ADL *should have* and *should be able to do.* As stated in the framework, an ADL must explicitly model components, connectors, and architectural configurations [2].

- **Component**: It is a unit of computation or a data store in architecture;
- **Connectors**: They are architectural building blocks used to model interactions among components and rules to govern those interactions;
- **Architectural configurations**: They are the connected graphs of components and connectors that describe architectural structure;

In addition to this, an ADL should provide an accompanying **tool support**, which renders an ADL more usable and reusable. In the remainder of this section, we select and describe the most important features that an ADL must support.

4.2.1 Component

Components are modeled using the feature *interface*, which is required by the ADL. Additional features are those for modeling component *type, semantics, and evolution.*

Interface – “A component's interface is a set of interaction points between it and the external world. The interface specifies the services (messages, operations, and variables) a component provides and requires. [2]”

Type – Components behave in identifiable, distinct ways, and they also interact with other components in similarly distinct and identifiable ways. These distinctions separate components into categories, or types. A component type captures the semantics of a component's behavior, the kind of functionality it implements, its performance characteristics, and its expectations of the style of interaction with other components [32]. The explicit identification of component types not only enhances the understandability and analyzability of software architecture, but also facilitates the reuse of software components by instantiating a component type multiple times [2].

Semantics – “Component semantics is defined as a high-level model of a component's behavior. Such a model is needed to perform analysis, enforce architectural constraints, and ensure consistent mappings of architectures from one level of abstraction to another [2].”

Evolution – “As architectural building blocks, components will continuously evolve. Component evolution can be informally defined as the modification of (a subset of) a component's properties, e.g., interface, behavior, or implementation [2].”

4.2.2 Connectors

The features characterizing connectors are their *interfaces, types, semantics, evolution.*

Interface – “A connector's interface is a set of interaction points between the connector and the components and other connectors attached to it. Connector interfaces enable proper connectivity of components and their interaction in an architecture and, thereby, reasoning about architectural configurations. [2]”

Type – “Connector types are abstractions that encapsulate component communication, coordination, and mediation decisions. A connector type captures the semantics of a class of interactions, assertions about that class, and the responsibilities and requirements that components must satisfy in an interaction from the class [32]. An ADL typically has either an *extensible connector type system*, defined in terms of interaction protocols, or a *built-in, enumerated connector type system*, based on particular implementation mechanisms [2].”

Semantics – “Similarly to components, connector semantics is defined as a high-level model of a connector's behavior. Unlike components, whose semantics express application-level functionality, connector semantics entail specifications of (computation-independent) interaction protocols.[2]"

Evolution – "Analogously to component evolution, the evolution of a connector is defined as the modification of (a subset of) its properties, e.g., interface, semantics, or constraints on the two. ADLs can accommodate this evolution by modifying or refining existing connectors with techniques such as incremental information filtering, subtyping, and refinement [2] ."

4.2.3 Architectural Configurations

We define requirements of architectural configurations as follows [2]:

Understandability – "One role of software architecture is to serve as an early communication conduit for different stakeholders in a project and facilitate understanding of (families of) systems at a high level of abstraction. ADLs must thus model structural (topological) information with simple and understandable syntax [2]."

Compositionality – "Compositionality, or hierarchical composition, is a mechanism that allows architectures to describe software systems at different levels of detail: Complex structure and behavior may be explicitly represented or they may be abstracted away into a single component or connector. Such abstraction mechanisms should be provided as part of an ADLs modeling capabilities. [2]"

Refinement and traceability – "ADLs must also enable correct and consistent refinement of architectures into executable systems and traceability of changes across levels of architectural refinement. This view is supported by the prevailing argument for developing and using ADLs: They are necessary to bridge the gap between informal, "boxes and lines" diagrams and programming languages which are deemed too low-level for application design activities. [2]"

Heterogeneity – It is important that ADLs be *open*, i.e., that they provide facilities for architectural specification and development with heterogeneous components and connectors [2]. An ADL should separate lower-level concerns, such as programming languages, middlewares, operating systems and computer networks, from high-level design/concepts.

Scalability – “Architectures are intended to provide developers with abstractions needed to cope with the issues of software complexity and size. ADLs must therefore directly support specification and development of large scale systems that are likely to grow further. [2]”

Evolvability – “Evolution, as we define it, refers to “offline” changes to an architecture (and the resulting system).An ADL should support the evolution of architectural configurations at the level of components and connectors with features for their incremental addition, removal, replacement, and reconnection in a configuration [2].”

Dynamism – “Known also dynamic reconfiguration, on the other hand, refers to modifying the architecture and enacting those modifications in the system while the system is executing. Support for dynamic reconfiguration is important in the case of certain safety- and mission-critical systems, such as air traffic control, telephone switching, and high availability public information systems. Shutting down and restarting such systems for upgrades may incur unacceptable delays, increased cost, and risk [2].”

4.2.4 Tool Support

Tool support includes programs as well as theories for working with ADL specifications. The kinds of tool support should be provided by an ADLs are: *active specification, multiple views, analysis, refinement, implementation generation and dynamism* [2].

Active specification – “ADL tools provide active specification support by reducing the space of possible design options based on the current state of the architecture. They can be either proactive, by suggesting courses of action or disallowing design options that may result in undesirable design states, or reactive, by informing the architect of such states once they are reached during design [2]. ”

Multiple views – Software architecture must be understandable to all involved stakeholders, including the customers who make decisions, and the developers who build the system. This is done by incorporating multi-views and provide the most appropriate view to a given stakeholder, and meanwhile, ensuring inter-view consistency.

Analysis – A comprehensive analysis of the software architecture before system implementation substantially reduces the errors. Furthermore, an ADL should allow *simulation* for testing of the software architecture.

Refinement & Implementation generation – “Refining architectural descriptions is a complex task whose correctness and consistency cannot always be guaranteed by formal proof, but adequate tool support can give architects increased confidence in this respect. It is therefore desirable, if not imperative, for an ADL toolkit to provide tools to assist in, e.g., producing source code. [2]”

Dynamism – “An ADL's ability to model dynamic changes is insufficient to guarantee that they will be applied to the executing system in a property-preserving manner. Software tools are needed to analyze the modified architecture to ensure its desirable properties, correctly map the changes expressed in terms of architectural constructs to the implementation modules, ensure continuous execution of the application's vital subsystems and preservation of valid state before and after the modification, and analyze and test the modified application while it is executing. [2]”

4.3 Analysis of Reo as an ADL

In this section, we present an analysis of Reo as an ADL based on the criteria we introduced in previous section.

4.3.1 Components

In Reo, a component is a software implementation whose instances can be executed on physical or logical devices [7]. As a connector-based language, Reo distinguish two kinds of components: external components (black-boxes) and Reo-specific components (encapsulated connectors). A component (external, black box) can be used (composed) by means of its interface only. In this section, we always refer components as “external components”, and we discuss Reo-specific components as connectors in next section.

Interface

Reo support specification of component interfaces. Such an interface describes the input/output ports, and the observable behavior of the component on these ports [30], this means that an interface defines what a component needs, but also what a component offers. Reo constrains the usage of a component by specifying its interface as the only legal means of interaction from within and without the component. An interface can have multiple ports, each of which is involved in the exchange of *untargeted*, *passive* messages [17]. An *untargeted* and *passive* message can be simply interpreted as nothing but a data item sent or received by a component. In Reo, a component input port attaches to a sink node of a connector, while an output port attaches to a source node. A component can either send a message (data item) by a “write” operation on its output ports, or receives it by a “take” operation on its input ports. The information flows through a port in one direction only (*unidirection*): either from the environment into its component instance (through take) or from its component instance to the environment (through write) [17].

In the case study we identified two actors: initiator and attendee –these we represent with two user components. We defined their interfaces and composed a larger circuit called the DMSS system to connect those component interfaces together. Furthermore, the DMSS also contains some additional external components, such as attendee calendar databases.

Types

Reo does not distinguish component types specifically, such as database, file, process, algorithm, etc. We use a naming convention to distinguish between those.

Semantics

Reo offers both synchronous and asynchronous channels. Composing of synchronous primitives together allows modeling of atomic behavior. Composing together third-party components using synchronous circuits allows enforcing of complex transactions [31]. In our case study, the interactions between an attendee agent and an attendee calendar database are transactional, as enforced by synchronous connectors between them (Figure 4.2). Semantic of an external component can be specified algebraically in terms of relations among its ports.

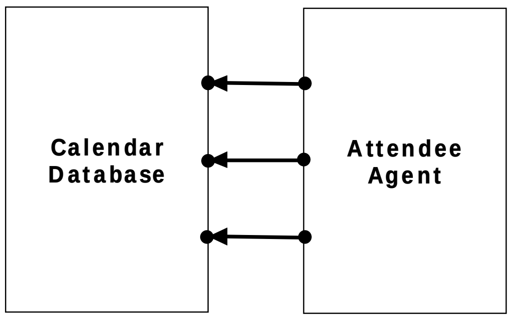


Figure 4.2

Evolution

External components in Reo serve as black boxes –hence internal evolution is possible as long as their interfaces remain intact.

4.3.2 Connectors

Reo treats connectors as first-class entities that exogenously coordinate inter-component activities in a component-based system.

Interface

When connecting a group of Reo channels and encapsulating them into a composite connector, its interface is modeled in the same way as a component: a collection of ports. Other connectors or (external) components with matching interface ports can connect to a connector interface. For example, the interface of a meeting agent matches the one of the agent coordination hub, hence they can be attached to each other point-to-point using several auxiliary synchronous channels.

Types

Reo identifies two general types of connectors: *synchronous* and *asynchronous*. A channel is called synchronous if it delays the success of operations among its ports such that they can only succeed simultaneously, as in as single transaction [31]; otherwise, it is called asynchronous. Channels in Reo are user-defined. In Chapter 2 we have introduced one useful set of channels that we further used in Chapter 3. We use naming conventions and connector encapsulation to specify different types of connectors.

Semantics

The semantics of a Reo connector is formally specified using either Abstract Behavior Type in terms of a (maximal) relation among a set of timed-data-streams [17], or Constraint Automata [19]. These formal semantics allow for the translation of a virtual model to a formal specification in order to perform verification and model checking. In our case study, we have provided a virtual model of a DMSS.

Evolution

Reo's topological operations allow for incremental composition of additional behavior in a circuit. We develop our case study model incrementally, by gradually extending the prototype to its full functionality. Reo also supports connector evolution via parameterization, where, e.g., the connector can be upgraded with more capacity. For example, an "Exclusive Router 3" can be parameterized to support exclusively routing of arbitrary number of messages.

4.3.3 Architectural Configurations

A configuration of components and connectors in Reo is often referred to as a *Reo circuit*. The overall DMSS model represents an assembly of (external) components and Reo connectors to implement a meeting scheduling system.

Understandability

Reo circuit can be explained and understood intuitively because of their strong correspondence to a metaphor of physical flow of data through channels [7] . In Reo, it is clear to see the flow of data items from one component to another through Reo channels. Reo combines flow of data with synchronization conditions on the entities that produce or consume data flows. In our case study, the uniqueness of meeting ID for each meeting is enforced by connecting meeting (initiator) agents to an exclusive router of the agent coordination hub, which enforces a new meeting ID only flow to one of them at any moment (Figure 4.3).

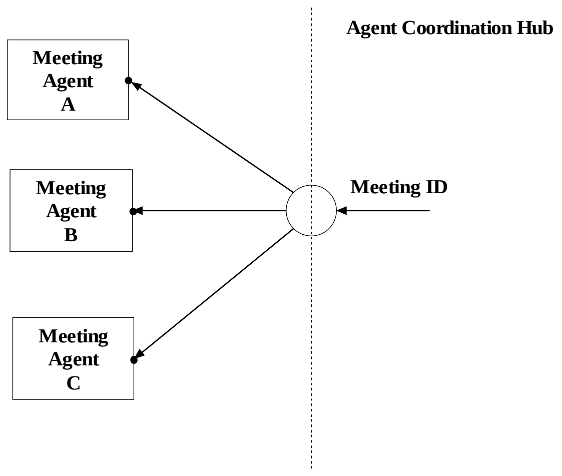


Figure 4.3.Explicit specification of a connector in Reo

Compositionality

As we observed in our case study, the Reo coordination language by design supports compositionality of configurations, allowing hierarchical breakdown of a software system into a group of components connected by connectors.

Refinement and traceability

Reo supports refinement through component/connector encapsulation; Reo does not provide sophisticated facilities for traceability.

Heterogeneity

Reo does not specify particular technology for implementation. It only offers blocking “write” and “take” operations to (external) components. Other non-Reo coordination mechanisms and interaction patterns, such as RPC, shared spaces, can be easily expressed by composing together Reo channels [7].

Scalability

Since a Reo node doesn’t constrain on the number of channels that can be attached to it, new components and connectors can be added without requiring modifications of existing component instances and connectors (Figure 4. 4). In our case study, we can have a flexible number of meeting agent connected to the *agent coordination hub*. Reo's composability allows for specifying large scalable systems.

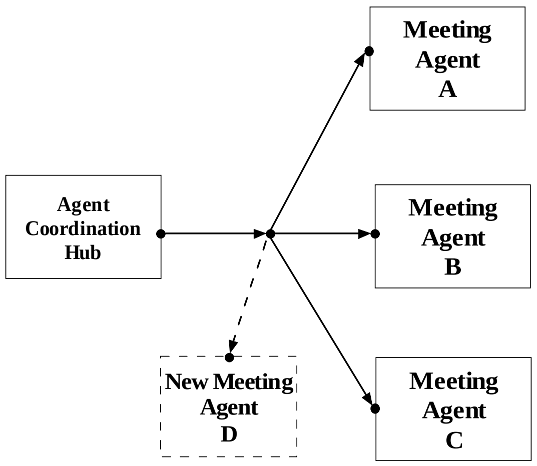


Figure 4.4.Scalability of Reo circuit

Evolvability

In Reo, to improve the functionality of a software configuration, one can add, remove, replace components and connectors, reconfigure the topology of an architectural configuration.

Dynamism

Reo allows dynamic reconfiguration of Reo connectors and nodes providing a set of topological operations that can be used during run-time. These operations are*: create channel, (dis) connect channel end, forget channel end, (dis) connect node, forget node, join nodes*, and *split node* [7].

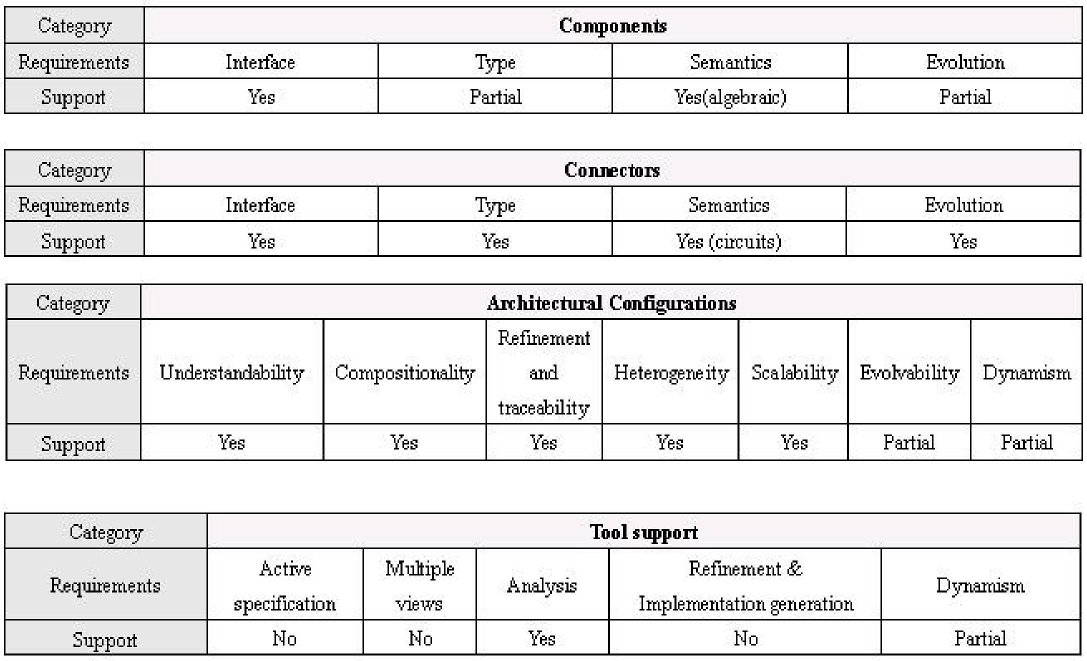
Reo only provides primitive topological operations. To insert or remove a whole component or connector during runtime, designers need to perform many primitive operations in sequence. The designers need to take care that safety and consistency of the system stay intact during and after the performing of the necessary for reconfiguration consisting of a set of topological operations.

4.3.4 Tool support

The Reo visual programming environment (under development) contains a simulator for Reo circuits. The Reo simulator tool is a non-distributed version of the Reo operational semantics, allowing running and testing Reo circuits. It only allows one to perform the basic operations on Reo circuits at this stage. It's not only able to verify the syntactic correctness of system models. Since the development of Reo toolset is still under way, we do not further assess the Reo tool support and leave this for future work.

4.4 Summary

Table 4.1 shows the summary of our evaluation of Reo as an ADL.



Category	**Components**			
Requirements	Interface	Type	Semantics	Evolution
Support	Yes	Partial	Yes(algebraic)	Partial

Category	**Connectors**			
Requirements	Interface	Type	Semantics	Evolution
Support	Yes	Yes	Yes (circuits)	Yes

Category	**Architectural Configurations**						
Requirements	Understandability	Compositionality	Refinement and traceability	Heterogeneity	Scalability	Evolvability	Dynamism
Support	Yes	Yes	Yes	Yes	Yes	Partial	Partial

Category	**Tool support**				
Requirements	Active specification	Multiple views	Analysis	Refinement & Implementation generation	Dynamism
Support	No	No	Yes	No	Partial

Table 4.1. Summary of the Evaluation of Reo as an ADL

Chapter 5
Conclusions

5.1 Summary

In this thesis we assess Reo as an Architecture Description Language. We have presented a case study on specifying a distributed meeting scheduling system (DMSS) using Reo. We provide an integrated model of the system by using some outstanding features of Reo, e.g., compositionality, refinability, dynamic reconfiguration. The implementation of the DMSS can be directly derived from the specification given a Reo coordination middleware (under development).

We used an existing evaluation framework to assess the capability of Reo as an ADL. Below we conclude some pros and cons of Reo as an ADL:

- Both Reo components and connectors have well-defined interfaces, allowing them to be accessed independently of their implementation details;
- Reo channels and nodes have formal semantics that ensures the precise specifications of inter-component coordination;
- Reo allows hierarchical breakdown of a software system incrementally into a group of components connected by connectors;
- A Reo circuit allows the (dynamic) (re)configuration of heterogeneous components on different platforms;
- Based on exogenous coordination, Reo directly supports specification and development of large-scale systems that are likely to grow further, in terms of both complexity and size;
- At current stage, Reo lacks support for component types (i.e. a set of generic Reo wrappers for external components, e.g., database, e-mail system, storage, etc.), so that the common behavior of a group of similar components can be captured and reused;
- Reo lacks facilities to guarantee safety and consistency of system's state for reconfiguration programs consisting of many topological operations;

- Reo still lacks tool support in some aspects: a) multiple views to satisfy the needs of different stakeholders (e.g. customers, users, designers); b) model refinement and implementation generation, which allows a software architecture to be converted into a running application (under development).

5.2 Discussion

During the requirement analysis of the thesis project, I described use case diagrams technically, which led to a lot of confusion and misunderstanding. A software designer should not start the design of a software system without completely understanding and fully documenting the system requirements from a user's perspective. A designer uses use case diagrams to capture "what" a system that supports some business process should do, as opposed to "how" it does it.

Another difficulty I experienced is the selection of appropriate high-level structure of the system, i.e. how to define and configure coarse-grained components and connectors. Initially I provided a "client-server" solution: an initiator agent act as a server interacting with attendee agents, which resulted in an inflexible design of the initiator agent component. After considering the original design, we adopted a peer architecture for the system, which renders a much more understandable and scalable design.

5.3 Future work

As future work, we expect some work to be done in following areas: **a)** Mechanisms for manipulating of coarse connectors (insertion, replacement, and removal) at runtime need to be added at language level. **b)** The Reo virtual programming environment should support generic types (e.g. database, e-mail system, storage), so that designers can easily and even automatically wrap external components. **c)** Implementation generation of Reo circuits to make software development using Reo more efficient (e.g. code generation) and effective (e.g. reducing human errors). **d)** Based on my personal experiences I feel that a knowledge base (e.g. developing processes, best practices, and patterns) based on previous work [18] , should be captured and documented better to help designers operate with the same vocabulary.

Reference:

[1]. ANSI/IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*.p.3. 2000.

[2]. N Medvidovic, RN Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, Volume 26, Issue 1, 70 – 93.2002.

[3]. Paul C. Clements. *A Survey of Architecture Description Languages*. Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD ’96). 1996.

[4]. Ariel D. Fuxman. *A Survey of Architecture Description languages*.2000.

[5]. N. Medvidovic. *ADLs and dynamic architecture changes*. In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints ’96) on SIGSOFT ’96 workshops, pages 24–27, San Francisco, CA, 1996.

[6]. N. Carriero and D. Gelernter, *Coordination Languages and their Significance*, Communications of the ACM 35 (2), pp. 97-107. 1992.

[7]. Arbab.F. Reo: *A Channel-based Coordination Model for Component Composition*.Mathematical Structures in Computer Science, Cambridge University Press, Vol. 14, No. 3, pp. 329-366, June 2004.

[8]. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. “*Specification and Analysis of System Architecture Using Rapide*.” IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 336-355, April 1995.

[9]. D. C. Luckham and J. Vera. “*An Event-Based Architecture Definition Language*.” IEEE Transactions on Software Engineering, vol. 21, no. 9, pages 717-734, September 1995.

[10]. P. Binns, M. Engelhart, M. Jackson, and S. Vestal. “*Domain-Specific Software Architectures for Guidance, Navigation, and Control*.” International Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 2, 1996.

[11]. S. Vestal. “*MetaH Programmer’s Manual, Version 1.09*.” Technical Report, Honeywell Technology Center, April 1996.

[12]. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. “*Specifying Distributed Software Architectures*.” In Proceedings of the Fifth European Software Engineering Conference (ESEC’95), Barcelona, September 1995.

[13]. J. Magee and J. Kramer. “*Dynamic Structure in Software Architectures.*” In Proceedings of ACM SIGSOFT’96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pages 3-14, San Francisco, CA, October 1996.

[14]. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and Zelesnik. “*Abstractions for Software Architecture and Tools Support Them*.” IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 314-335, April 1995.

[15]. M. Shaw, R. DeLine, and G. Zelesnik. “Abstractions and Implementations for Architectural Connections.” In Proceedings the Third International Conference on Configurable Distributed Systems, May 1996.

[16]. G. A. Papadopoulos, F. Arbab, Coordination models and languages, in: M. Zelkowitz (Ed.), The engineering of Large Systems, Vol. 46 of Advances in Computers, Academic Press, 1998, pp. 329{400.

[17]. F. Arbab, *Abstract behavior types: a foundation model for components and their composition*, Science of Computer Programming 55 (2005) 3-52.

[18]. Dave Clarke and David Costa, *A Compendium of Reo Circuits*, working in process, CWI , 2005

[19]. F. Arbab, C. Baier, J. Rutten, M. Sirjani, *Modeling component connectors in Reo by constraint automata*, in: Proceedings of FOCLASA 2003, a satellite event of CONCUR 2003, Vol. 97 of ENTCS, Elsevier Science, 2004

[20]. R. G. Smith. "The Contract Net Protocol: High-Level Communications and Control in a Distributed Problem Solver", *IEEE Transactions on Computers*, C29(12), 1980.

[21]. Rahul Singh. RCal: An Autonomous Agent for Intelligent Distributed Meeting Scheduling. Master Thesis, the Robotics Institute, Carnegie Mellon University.2003.

[22]. M. Shaw, D. Garlan, R. Allen, D. Klein, J. Ockerbloom, C.Scott, M. Schumacher. Candidate Model Problems in Software Architecture. Unpublished manuscript, November 1995. Available from: http://www.cs.cmu.edu/afs/cs/project/compose/www/html/ModProb/.

[23]. Microsoft Outlook, Web address: http://www.microsoft.com/office/outlook/

[24]. IBM Lotus Note, Web address:http://www.lotus.com/products/product4.nsf/wdocs/noteshomepage

[25]. Sen S & Durfee E H. A Formal Study of Distributed Meeting Scheduling: Preliminary Results. *ACM Conference on Organizational Computing Systems*.1991.

[26]. Sen S and Durfee E H.On the Design of an Adaptive Meeting Scheduler. *Proceedings of the Tenth IEEE Conference on Artificial Intelligence Applications*, pp 40—46.1994.

[27]. Sen S. An automated distributed meeting scheduler. *IEEE Expert*, 12, No 4, pp 41-45.1997.

[28]. Leonado Garrido, Katia Sycar.Multi-agent meeting scheduling: Preliminary experimental results. *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*.1996.

[29]. Zlatev, Z., Diakov, N., Pokraev, S.: Construction of negotiation protocols for ecommerce applications. ACM SIGecom Exchanges (2004) 11-22

[30]. F. Arbab, F.S. de Boer, M.M. Bonsangue, and J.V. Guillen Scholten. A channel-based coordination model for components. Technical Report SEN-R0127, CWI, Amsterdam, 2001.

[31]. Diakov, N.K., Arbab, F. *"Adaptation of Software Entities for Synchronous Exogenous Coordination: An Initial Approach"*, Proceedings of The Second International Workshop on Coordination and Adaptation of Software Entities, W-CAT'2005, Glasgow, July 25, 2005, United Kingdom

[32]. Gregory, Z, http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/