INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Responsive Units : Behavioral Elements for Coordination Oriented Development of Object Oriented Systems

by

Teodor Dumitru

a thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Applied Sciences

Ottawa - Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

Abstract

Theoretically, the thesis sustains that in object oriented systems -mainly event driven- (1) the dialog between objects is essential, (2) the dialog implies coordination, (3) a dialog coordination object -that separates coordination from activities - is useful for response design. *Practically,* it is proposed a design modality based on Responsive Units. It is a model which directly captures the logic of interactions between objects, without much overhead. The approach is an extension (adaptation) of 'Coordination Oriented Design' (COD) towards object orientation. It uses the basic principle of COD of separating coordination from execution.

We address the problem that in OO systems, even mono-threaded, the *event responses can be only partially defined.* This mainly happens because interaction diagrams can capture only a few interaction sequences and because object implementation hardly relates to their behavioral design. An improvement in the definition of event responses would be beneficial, mainly for systems in which the sequence of interactions is complicate to design, but critical.

The suggested approach sets the *focus on the goal-oriented dialog* between objects, saying that:
1) *The dialog is essential.* The interaction logic between objects and with the users is fundamental.
2) *Dialog implies coordination.* When a dialog runs for a goal, a coordination implicitly exists.
Two corollaries lead to pragmatics:
1) *Objects and systems are built through dialog.* They are shaped by the new goal-oriented dialog.
2) *Dialog Coordination can become an object.* We choose to implement it because a group of objects working for a goal has *goal values* and *goal interaction logic* that better fit in a *distinct object.* The goal concept includes error management.
We differentiate three types of Dialog Coordination:
(a) Master-Slave Coordination, for objects related by *strong* interactions in a *'unit'*.
(b) Team Coordination, for units related by *light* interactions in a *'team'*.
(c) Agreement Coordination, that refers to *set-up agreements* between objects.
The *Responsive Unit is introduced for good encapsulation of interaction logic.* It is a *container-object* composed of *strongly related objects* and their *Master-Slave Coordinator.* A Unit contains also *a Port object and a Data object.* To reflect the behavioral model, inside the coordinator we implement a state machine.
In a Team of Units one Unit may be Team Coordinator. The *state machine of the Team Coordinator calls for activities* in the other Units. Supplementary, at system level another Unit may be used as Agreement Coordinator, dealing with teams set-up and major error management.

We advocate that some of advantages of the approach are as follows:

- A good event response definition, which can be understood by both user and developer and based on which an early requirements verification can be done.
- Fast development and testing, since the dialog coordinators clearly define the interactions and the Responsive Units predefine groups of objects where Unit responses are captured.
- It is a 'design for change' approach, as the modifications are concentrated in dialog coordinators and localized in Responsive Units.

The method , illustrated in the case studies, is believed to have a good impact for the general object oriented development of event driven systems.

Acknowledgments

Beyond all, I wish to give thanks to Him who is 'scientiarum Dominus' and our graceful provider. The thought that all things are made through His word is a constant source of observations for me.

I address my sympathy and respect to my research advisor Dr. Moshe Krieger for his swift and essential distinctions, for vision and graceful leading.

Tom Lismer, Gabby Coifan, Voia Radonjic and Mark Robb, as friends and colleagues, offered me advice and meaningful feedback for which I am so grateful.

My wife Miki showed a high equilibrium and much understanding (I wish to note that her university grade was A+). She also directly helped me with drawings and readings.

My children Dariu, Dorothy, Johnatan and Ruth helped me to think about goal-oriented dialog and coordination while playing with me and about Responsive Units by showing me animal toys.

The main desire of my parents, Ana-Emilia and George, was to provide proper education to their three sons. I am thankful to them for their good-will, presence of spirit and hard work.

University of Ottawa and Nortel -my company- are great research & development environments, for which they deserve recognition.

Responsive Units : Behavioral Elements for Coordination Oriented Development of Object Oriented Systems

Part 2 Suggested Approach for Behavior Modeling

Chapter 1

Introduction

The theoretical purpose of the thesis is to underline the importance of interactions between objects and the need for their coordination, especially in behavioral intensive event-driven systems. As a practical goal, the thesis looks for a modality of development in which the coordinated interactions can be clearly captured in code and in documentation, with a minimum overhead beyond Object Oriented Development. The suggested development modality is based on 'Responsive Units' and their specific interactions. A responsive unit is a complex object that promotes a clear definition of the logic of interactions, both internal (between the objects in its composition) and external (with other similar units).

The approach represents an extension (adaptation) of 'Coordination Oriented Design' (COD)-a response oriented method [Krieger5]-towards object orientation. It uses the basic principle of COD of separating coordination from execution; and it uses the basic principle of Object Oriented Design (OOD) in that all system components are responsive units, i.e., macro-objects.

A question that may be raised is why such a method may be needed. First, in OOD -originally developed mostly for complex data systems- the interactions and system behavior can be *only partially modeled,* which is a disadvantage for many event-driven systems. On the other hand COD -being behavior driven- is not easily acceptable by engineers used to OOD, where the focus is set on objects. It is hoped that the suggested method can include the advantages of both. The author believes that this method provides a model that enhances the visibility of system dynamics by explicitly capturing the logic of interactions and the behavior of objects. Also, it is a 'design for change' method as the event responses can be modified with more accuracy if initially they are well captured.

1.1 On Event-Driven Systems

A large percentage of software development was and is done for data / information processing. As a consequence, the usual view of software is a transformational view, in which the output data is obtained from the input data via a set of transformations. For each new set of data a new output is obtained. These systems are often flow oriented, as there is a continuous flow of input data processed by modules that do the necessary work in a specific order, until completion.

Another increasingly important domain of software development is represented by event-driven systems. In such applications the system has to respond in a number of ways to external events or requests. In this category can be the real-time, behavior intensive systems but also the interactive systems that receive multiple inputs from users.

A response in an event-driven system depends on the respective event, on the sequence of preceding events and on the outcome of previous responses. The response may also be influenced by environment changes. Some of these ideas are presented in figure 1.1.

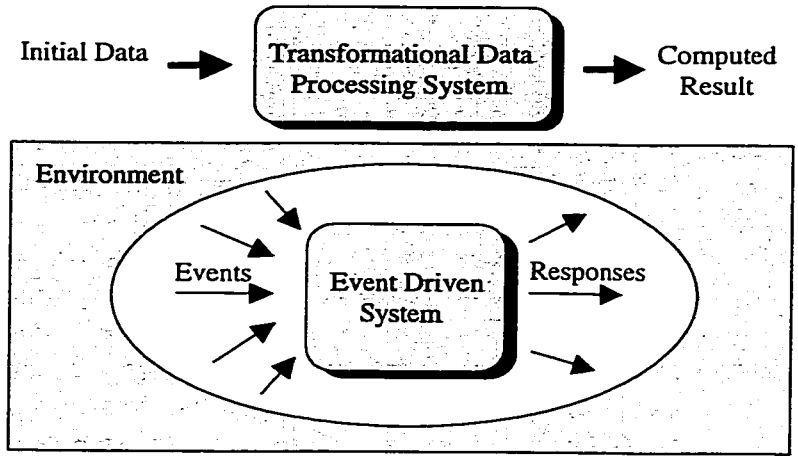


Figure 1.1 Transformational versus Event Driven Systems

In order to have a good response design in OOD cases, one needs a good definition of interactions between objects, besides the behavioral design of each object. Some issues on this topic are introduced in the next subchapter, 1.2.

More specifically this thesis addresses real-time embedded systems mainly, which include some of the following features:

- In the system there are some tens of objects or more, with multiple interactions. The granularity of objects is quite small and the interactions cover a major portion of project.
- The multiplicity of event sequences imply interaction diagrams with several possible paths and several ways in which the diagrams are related. This leads to partial design and later insufficient understanding.
- There are certain physical restrictions as in embedded systems or other tight applications (see Java applets in Case Study) ; the overhead added by using CAD tools for the behavioral definition is not acceptable.
- Specialized hardware components are used besides the processor and their status / behavior can modify the overall responses; these modifications are typically difficult to capture in the design of individual objects.
- The requirements (part of which are event responses and their sequences) are done by application engineers. Software engineers need to show that the interactions & objects implement the application responses.

Examples may be taken from the telecommunication industry (not a small domain) : network systems, wireless basestations; all these are using embedded processors with specific hardware. Having experience with such systems, the author was interested to add some visibility / clarity in the design of event responses and interactions. This appears to be more necessary when the design changes often and more developers are involved, which happens in most projects.

A design solution where objects communicate freely in a decentralized manner will need multiple interaction diagrams. For a bigger group of objects, these may not be able to capture each and every response in each and every case, as it is often necessary in our type of systems.

Our method is to group few objects (5-10) and add a COD type of coordinator for them, as it will be presented. The coordinator *captures all event responses of the group and is still easy to understand.*

Adding a coordinator looks as a design overhead. We consider this overhead acceptable as we get a major gain by defining completely the group responses. Actually an organized approach finally leads to a simpler design and without the coordinator there is an overall development overhead resulting from the partial design of object interactions. Between the groups of coordinated objects, the quantity of interactions may become manageable enough. However, we consider the option to add one more level of coordination. We propose this for an even better coverage of each response at higher level.

Thus, in this research we consider the described type of systems, for which it is believed that our coordination approach adds value to the response design, both in documentation and in implementation.

1.2 About Object Orientation and System Behavior

The Object Oriented Development is an approach that promotes the encapsulation of a group of related methods and data in an entity, which is the object. As such, in OOD a *system is composed of a set of objects that interact.* For system modeling, the design methodologies use mainly a structural view (for static characteristics) and a behavioral view (for dynamic characteristics). It is typically suggested that the components should match some real life objects. Because of their low level of abstraction, the design with objects is well suited for mid size projects, but the implementation of a well-engineered large system requires much expertise. A major problem is the complexity of interactions, which grows fast with the number of objects.

The OOD *methodologies* have much in common concerning *behavior modeling*. We will discuss later the differences, but most typical is to define *system behavior* as follows:
(1) The behavior of *each object with a state machine*
(2) The *interactions between objects with interaction diagrams.*
In these diagrams the objects are represented with parallel lines, as being concurrent, since they are seen as independent machines even if the implementation is done in a single process. One diagram can show only *one sequence of interactions, with no decision points*; in some cases, a plain English text can be added to explain *multiple choices* in the sequence. The methodologies justify this simplified model saying that the interactions are too complex to be better captured. For a *complete* definition of the system behavior, it would be necessary to define *each sequence of interactions that must be implemented.* However, this is an unrealistic expectation, since too many diagrams would result for an average sized system. Therefore the design document usually *contains only interaction diagrams for the most representative sequences*, letting the

implementor to decide about the rest of them. In other words, in an OOD behavioral model the *event responses* remain only *partially defined, a major impediment for many event-driven system that have quite complicate interaction sequences.*

In this thesis we examine several object oriented methodologies with respect to behavior modeling, driven by our interest for behavior intensive systems. We look for generic and specific concepts that attempt to describe the behavior beyond objects. The most recent with a major impact is 'A Use Case Driven Approach' of Jacobson, included in the Unified Modeling Language [UML] (the approach proposed in common by Booch [Booch], Rumbaugh [Rumbaugh] and Jacobson [Jacobson]). He suggests the development of systems from a user's perspective, as any system is built to be interactive. Such a view is a good support for ours, where we consider that the *logic of interactions between objects is essential.*

Concerning the behavior beyond objects, the domain of Behavioral Patterns [Patterns], offers certain valuable solutions. However these patterns "shift your focus away from flow of control to let you concentrate just on the way the objects are interconnected.". They *group the* objects for a specific design problem; thus they are more like a structural approach for solving specific behavioral issues. This thesis put the *focus on the flow of messages and on the need for their explicit coordination* and thus it takes a different direction than the patterns.

1.3 Brief Outline of Coordination Oriented Design

Coordination Oriented Design (COD) is an engineering approach for the development of event-driven computer based (software) systems, developed by Krieger at Ottawa University [Krieger1]. In COD the development process revolves around event-responses. The specifications, design, implementation and modifications are driven by the way in which the system must respond to events. Events can be requests, externally generated, for certain responses or can be changes in environment that require the system reaction.

In this methodology, the system behavior is analyzed in terms of the work it has to do as response to events or changes in circumstances. The work that needs to be done for a request is specified as the precedence in which the 'units of work' must be executed to fulfill the response. Also, it must be specified how the responses interact with each other and how the responses will be modified by changes in conditions.The relatively independent units of work are named 'activities'. An activity is considered a sequence of operations that once started executes to the end without needing to wait for other input or resource. Differently said, a response to an event is a set of activities that have to be executed in a given precedence relation. Thus, *the execution (what has to be done - the activities) is separated from coordination (when to do it).* As a result, the methodology views the system that will be implemented as a set of assignable resources (software modules, hardware components, interfaces, etc.) assigned to execute in a given order the activities (units of work) which form the responses.

The corresponding *architecture* describes the system as one or more *"coordinators"*, that specify the sequences of activities representing the responses to each event, and several *"execution*

units" that execute the activities for different responses. The lower level coordinators receive only the environment conditions or state changes that can influence the lower level execution.

The Coordination Oriented Design is the main stream of our research team where have been analyzed different directions, such as:

- The Multiactivity Paradigm, an approach for the design of embedded systems by application specialists [Krieger2].
- Restricted Object Based Design (ROBD). An architecture where the *"execution units"* are wrapped in restricted objects. These respond to calls from a manager. [Lemire]
- ROBD for network management.(in work)
- COD for Components Off The Shelf (COTS). It refers to complex applications where the parts are software-off-the-shelf, related by a distinct coordinator. [Cox]
- Coordination Oriented Architecture for Large Software Systems. Different software architectures are analyzed, presenting the need and the meaning of a 'coordination architecture'. [Coifan]
- An adaptation of COD for the behavior modeling in OOD, which is presented here.

Coordination is a general concept that is applicable for handling complexity and also for 'disciplined' integration. The various studies denote that the methodology has industrial potential for the development of different software products.

As noted, this research underlines the importance of the logic of interactions between objects and the need for their coordination, mainly for event-driven system. Since in OOD the interactions are only partially defined, this research suggests that a certain adaptation of COD may enhance the behavior modeling in OOD.

1.4 Responsive Units: an Overview

Originally this thesis was conceived as an investigation of the detailed relationship between *behavior modeling in the Object Oriented Design and the Coordination Methodology* [Krieger1]. Reviewing the different ways of defining the behavior in OOD methodologies, it became apparent that the *logic of interactions* between various objects is a concept with a *distinct value, deserving distinct attention.* This *logic of interactions is seen here as a dialog between parts,* which is designed for the execution of the event responses. As such, the thesis *focuses on the 'dialog'* between the objects in a system.

The problem we address is that usually in OOD behavior modeling the *event responses can be only partially defined.* This situation happens because, as said, it is practically unfeasible to describe each sequence of interaction with an interaction diagram. For behavior intensive systems it is a major impediment with implications in all the system life-cycle. The issue appears in OO systems, even mono-threaded, that have a certain number of objects exhibiting relatively complex interactions.

Our intent is to eventually find a better way of defining the interaction logic between objects, i.e., their dialog. A complete response definition may be too high of an expectation, but an improvement is certainly desirable.

We start with the observation that the **dialog is essential** at the design time and at Run-Time. The objects are considered primordial, but at least in event-driven systems the dialog is shaping *at design time* the objects, to implement the event responses as desired. *At Run-Time* the dialog activates the objects and makes them working as a system, responding to requests. According to its sequence, each message changes the state of the receiving object, as designed.
The next step is the observation that *whenever there is a dialog* between a number of entities, **implicitly there is some type of coordination** between them. The coordination is *goal oriented* if the entities of the group work together for a group level response. Often such coordination is done by defining a collaboration media and letting the *entities to decide how and when* to collaborate.
Distinctively, in *this thesis we want to take full advantage of the OOD behavior modeling and add the approach of COD, where the distinction between coordination and execution is driving the design.* We suggest to *capture the interaction logic between objects (of a group) in a separate object; this receives events and calls for 'actions' i.e., execution methods in the other objects. The separate coordinator object will be referred in this work as Dialog Coordinator.*

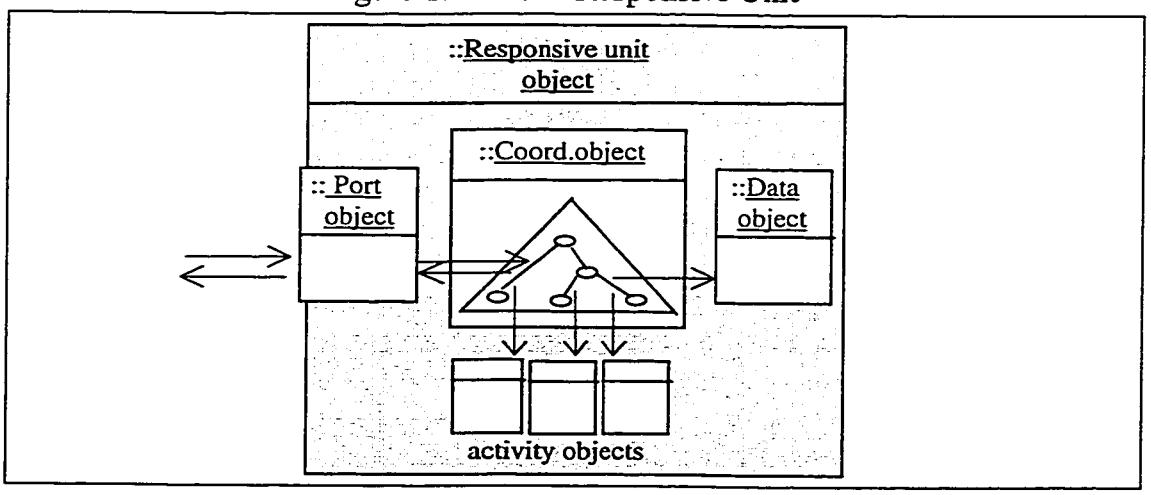
A group of objects has a set of event responses, the group goal, for which the group is responsible. One can distinguish goal oriented logic & data that belong to the group rather than to involved objects. These better fit in the Dialog Coordinator, instead of being dispersed. We believe that this way the *response definition for a group of objects* can be enhanced.

The thesis distinguishes *three types* of Dialog Coordination, based on the style of interactions:

1. *The Master-Slave Coordinator* deals with objects that *must* execute a request
2. *The Team Coordinator* deals with bigger components that may or may not execute a request. Such components have more responsibility and are related in a *team.*
3. *The Agreement Coordinator* deals with team building and modification, with system start-up and shut-down and with major error management.

The Responsive Unit is defined around a group of *tightly related objects with their Master-Slave Coordinator object.* A Responsive Unit, fig 1.2, is itself *an object that has in its composition several other objects. In addition to the Coordinator and Activity objects, a Responsive Unit object may have also a Port object and a Data object.* The Port is meant to be a connector with the exterior ; the Data object keeps the information that is common for the Activity objects.

Figure 1.2 The Responsive Unit



In *Coordinator is implemented a state machine of StateChart* type that receives requests coming to the Unit and organizes the responses, taking into account the history of the dialog.
The actions of this state machine are calls to the public methods of the Activity objects.

A number of Responsive Units can be grouped as a 'team' - with one of them being Team Coordinator. This one defines the intended dialog between the Responsive Units from its team and with the domain outside the team. It receives team level events and it calls for *actions, which are actually events* sent to the managed Units. Thus, from the behavioral point of view, the system may be designed using *hierarchically organized StateCharts*. One StateChart will describe the interactions between the others and with the external world.
A system can include a number of teams ; the concept is scaleable.

In addition, *an Agreement Coordinator, itself a Responsive Unit, may be used to provide mainly team building or modification.* The Responsive Units and the Teams may need to have some initial interactions, an 'agreement dialog', to be grouped for specific tasks. This 'agreement dialog' may be explicitly defined in the Agreement Coordinator. Also this one may provide controllable set-up or shut-down of the system and major error management. As we see, it does not deal with the user initiated requests-responses activity, but it has a supervising mandate.
The Team and Agreement Coordinators are presented in figure 1.3.

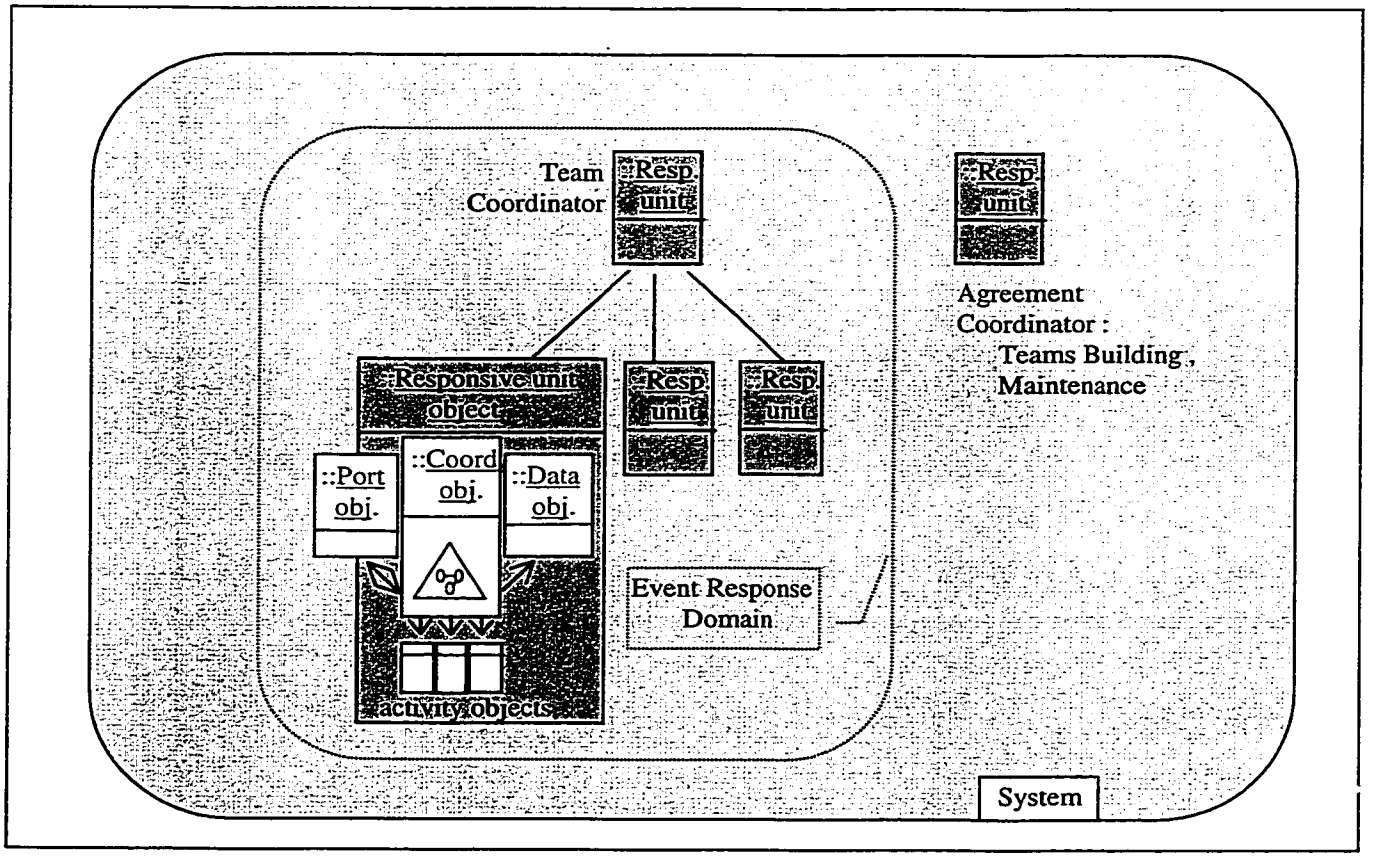


Figure 1.3 Team and Agreement Coordinators

Since each Responsive Unit has the *same format regardless of the design problem (an object in composition with Coordinator, Activity, Port, and Data objects),* we suggest the use of some ready-written code with this format. The programmer can cut-and-paste it as often as needed ; than he completes each Unit as required for the application under development. The ready-made code can also include a *skeletal state machine* in the coordinator, that will be completed according to the behavioral design of that application.

This Responsive Unit should not be seen as a design pattern since Design Patterns, as introduced by Gamma & all [Patterns], "are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". *Different* patterns reflect *different* design problems but the Responsive Unit *does not* represent a *specific* design problem. We consider the Responsive Unit a kind of *mega-object*, a generic element as the objects are. Actually the COD Coordination concept itself offers another direction than Patterns for managing complexity, direction that is *independent of any specific design problem.*

The author considers that the advantages of using Responsive Units, Team Coordinators and Agreement Coordinators (as outlined in 'Contributions', subchapter 1.5) by far outweigh what may be perceived as drawbacks. The development, maintenance and modification processes appear to be enhanced because the responses to events are better defined when the interaction logic is clearly localized.

1.5 Thesis Outline and Contributions

This thesis has two main areas:

- *The first part*, chapters 2 and 3, is a *critical analysis of the background information* related to behavior modeling ; we refer to OOD and CODE.
- *The second part,* chapters 4, 5, 6, 7, presents the *thesis contributions* for enhancing the behavior modeling. The results are studied in design examples, with practical suggestions.

The first part

Chapter 2 On Behavior Modeling in Object Oriented Methodologies

This chapter observes that one of the system views is the *behavioral view.* The chapter presents commonalties between methodologies concerning behavior modeling and underlines differences and particular enhancements. We group separately methodologies that are driven by system structure from methodologies with more interest in system behavior. From each one we select elements that help to justify our suggested approach and we discuss the issues of actual behavior modeling. In appropriate cases we discuss a major concept of the thesis: the advantage of defining the interaction logic between objects in a distinct component.

Chapter 3 Coordination Oriented Development (CODE): a Behavior Driven Approach

The Coordination Oriented Development *environment* is presented with the phases of the development process and the corresponding system architecture. The Coordination Based Design *methodology* is discussed in detail as a *behavior driven* approach for system design. Also we explain its origins, qualities and application domain. The relationship between CODE and OOD is discussed and also there are comments on some ideas from non Object Oriented methodologies. We consider also other Coordination related studies and their rapport with CODE. Finally we address some existing CODE application with a few comments.

The second part

Chapter 4 Introducing the Dialog Coordination with its Subtypes: Master-Slave, Team and Agreement Coordination

In the quest for an enhanced system behavior modeling we consider that it is important to firmly address the inter-objects dialog and its goal-oriented coordination. From the beginning we focus on the dialog between objects, when usually the focus is on objects. We start with two concepts:

- The dialog is essential
- Dialog implies coordination.

The concepts are followed by two corollaries, which add a pragmatical direction:

- Objects and systems are built through dialog
- Dialog coordination may be implemented as an object.

Noting that objects in a group work together for a goal, we consider that the goal-oriented interaction logic and the goal-oriented group data are better implemented in a Dialog Coordinator.
Three types of Dialog Coordination are further defined. We consider that between objects there are either strong or light interactions. A *Master-Slave* type is a *coordination* for strongly interacting objects related in groups, while *Team coordination* deals with light interacting groups. The *Agreement Coordination* cares about the set-up and reorganization of objects.

Chapter 5 Responsive Units: Behavioral Elements Based on Dialog Coordination

We build a box-object having in its composition a group of strongly interacting objects and their coordinator. This object with objects is named Responsive Unit. Beside acting objects and the coordinator, a Unit may also have in composition a Data object and a Port object. As this typical format can be repeated in any design, some ready-made code may be used to build a Responsive Unit. This prefabricated code can have a skeletal state machine inside the coordinator.
The response-oriented dialog between Units can be defined in a Team Coordination Unit. Its state machine calls for actions that are events for the state machines of the Units.
The Agreement Coordinator Unit defines the logic for the Teams building or reorganization.

Chapter 6 Case Study for the New Concepts

Three microprocessor simulators (CISC, RISC and STACK) have been designed in Java . A Team Coordinator relates the user interface with three Responsive Units, that are built for the Memory, for the Registers Bank and for the Arithmetic Logic Circuit. As *'designing for change'* is an important issue, we studied the advantages of the method for building the second and third simulator. The main changes were in the Coordinators inside Responsive Units and because we had such objects the transition was smooth.

Chapter 7 Conclusions and Future Research

The concepts of 'focus on dialog', Dialog Coordination and Responsive Units appear to be meaningful. Development and testing are well organized and the change is precise. We believe that further research may be done to use the concepts in multi-tasking and interacting systems.

Thesis contributions

In the following we concentrate the *distinctive ideas* in a short form.

As a *theoretical* part, the thesis sustains that (1) the dialog between objects is essential, (2) the dialog implies coordination, (3) the dialog coordination may be implemented as a special object in COD sense and this is an advantage for the design of event-driven systems.

The *practical* proposal is a design modality that adds value without much overhead. The Responsive Units with their specific interactions enable developers to understand and explicitly represent the logic of interactions between the objects in a system.

The contributions may be underlined as follows:

This work promotes several ideas with *the aim of enhancing the behavior modeling* in OOD. Since the interactions between object state machines represent an issue even in single-process systems, we mainly consider such cases, adding few suggestions for multi-process systems.

We start with a *change* of focus from objects to the dialog between them ; the dialog coordination concept is the base on which the work later evolves.

- The author proposes a change of *focus to the logic of interactions* between objects, *named here 'dialog,'* and advocates that :
    - *The dialog is essential*, both at design and at run time,
    - *Dialog implies coordination*, which is usually dispersed in the interacting parts.
    - *The dialog coordination should be captured in a distinct object* because:

- The *dynamic* logic of interactions can be better defined in a *dynamic* module rather than in a *static* document.
- The objects of a group have a common goal; there are some data and interaction logic that belong to the group, not to any object. These should be captured in a separate object, public to the group.

Thus, the thesis applies the COD Coordinator concept to the goal-oriented interaction logic between objects, the dialog, naming it Dialog Coordinator.

- Going in more detail, we consider three types of Dialog Coordination :
  1. *The Master-Slave Coordination* deals with objects that *must* execute a request
  2. *The Team Coordination* relates components that *may* respond positively.
  3. *The Agreement Coordination* defines the working agreements between entities.

- The thesis advocates the *direct implementation of a state machine* in each Dialog Coordinator, to define its behavior.

Concerning ideas that can be concretely implemented, the thesis introduces the following:

- *The Responsive Unit*, a mega-object that has in its composition : *a Coordinator object, a number of Activity objects, a Port object, and a Data object.* It may be built using some ready-made code, that includes a skeletal *state machine inside the coordinator.*
- *The Team Coordinator,* a Responsive Unit that manages a *team* of other Responsive Units.
- *An Agreement Coordinator*, also a Responsive Unit, may be used to provide *team building or modification.*

The *behavioral view* becomes a *'structured FSM design'* : the FSM of the Team Coordinator deals with the interactions between the FSM's of Responsive Units.

This model brings for users, developers and testers a number of advantages -which will be detailed and explained in second part of thesis- as:

- For users, the model allows better visibility of the system, which is valuable for requirements checking. The user may verify and give a feedback for the 'primary events' and can follow the behavior specification for event responses.

- For developers, it brings an *increased clarity in the definition of interactions*, with direct implications in the definition of event responses.
  The speed of design may be higher, as there is a ready made location to place the logic of interactions: the dialog coordinator.
  A better separation between designers and implementors may be made, as the sequence of events and responses in Responsive Units can be presented as a coordinator state machine. Also new designers can learn faster a project.

The method is a 'design for change' approach, which is highly desirable for the actual pace of development. The changes for the logic of interactions are concentrated in coordinators; new objects can be easier 'plugged-in' when one has a complete view of interactions in a group.

- The testing will benefit from the fact that the state machine of the coordinator can *be exercised before completing* the other objects. This may be seen as also a *rapid prototyping method* and intermediate testing can provide earlier feedback to designers.
  Also, messages can be logged for each event and state, showing clearly the executed path.

Chapter 2

On Behavior Modeling

in Object Oriented Methodologies

2.1 About System Views
2.2 Methodologies with Focus on Structure

- **J.Rumbaugh, Object Oriented Modeling and Design**
- **G.Booch, Object Oriented Analysis and Design**
- **Coad-Yourdon, Object Oriented Analysis and Design**

2.3 Methodologies with More Concern for Behavior

- **I.Jacobson, OOSE, A Use Case Driven Approach**
- **Shlaer-Mellor, Object Lifecycles, Modeling the World in States**
- **R.Wirfs-Brook, Designing OO Software, A Responsibility Driven Approach**

2.4 Unified Modeling Language (UML)
23 Other Methods for the Behavior Modeling of Object Oriented Systems

Introduction

The Object Oriented Development has a relatively long history, with the period of crystallization mainly in 80's. The early success of this approach came with a strong emphasis on the structural modeling. However, most object oriented methodologies agree to view the behavior modeling as quite important, underlining that the system dynamics must be part of the design. Systems, components and objects are interactive and the logic of interactions needs to be part of the project. In 'A Use Case Driven Approach' [Jacobson], one of latest methodologies, interactions even become the core of the development.
Thus, in the well known and utilized methodologies we find that the behavioral view is introduced as a natural thing to have. Normally, there are differences in the way of presenting it and concerning its importance. In order to have a better organization for this study, we consider the methodologies in *two groups*: one *structure oriented* and one with *more concern for behavior*. UML [UML], the common work of Rumbaugh [Rumbaugh], *Booch* [Booch], and Jacobson [Jacobson], promotes a 'Use Case' design approach underlining the importance of interactions.

The intention in this chapter is to see *essential commonalties and differences* in behavior modeling, to *observe and comment the limits and underline the advantages*. Specifically, we look for ideas *beyond the usual format of describing objects as state machines and their messages with interaction diagrams*. We discuss these specific ideas as they motivate and help to justify the behavior elements introduced in the thesis.

2.1 About System Views

Whenever one considers a system of any complexity, a single type of view is seldom sufficient to describe it. Development methodologies approach the systems under design from two or three perspectives. The idea of having more views can be traced back to antiquity[1]: "A passive view was proposed by Democritus, who asserted that the world was composed of matter called atoms. Democritus' view places *things at the center of focus*. On the other hand, the classical spokesman for the active view is Heraclitus, who emphasized the notion of *process*."
The *object view* puts emphasis on what is touchable, visible. *Objects* are perceived as *substance* (the question is: down to what level?), *interactions* are seen as an *abstract* layer.
However, one should note that objects of higher-level are built with objects *and interactions*. We could say that *interactions are substance* for the higher-level object.
On the other side, the *process view* underlines the flow of interactions while the meaning of individual objects with their boundaries is minimized. An observation (in the opposite direction) is that processes may even *generate a boundary for a new object.* Several objects with tight interactions are perceived as a system, or a new object.

In software we may discern an element that is common to both object and process views. One can observe that:

- The objects are built through a language with rules and words, related for a reason. *Correlated words* define objects.
- The logic of interactions between objects is done by related messages, a kind of words. *Correlated words* form the logic of interactions.

In software these *correlated words* are *essential both for objects & interactions.*

This observation relates to an other ancient thought, "In Essence is the Word". It says that the logical links (logos) are primordial and through them things are built at any level. In software this thought makes sense as systems are built through *correlated words.* Even the system's hardware is often built with some language -as HDL- on top of silicon.
Thus, one can complement the Booch's observation on Democritus and Heraclitus with an other view: Solomon notes "in the beginning is the intelligence, before any of the dust of the world." According to this view, objects & systems (even dust-silicon itself) are made from logical links.

Object oriented methodologies initially insisted on objects, as a manner of encapsulation, but lately the interactions became an equally major topic. In actual modeling, there is a balance between what objects are - *the structural view* and how they are used - the *behavioral view.*

The term 'behavior' is defined in Webster [www.dictionary.com] as *'The mode in which someone bears himself in the presence of others or towards them.'*. It is interesting to underline that it means *both inter-actions and internal actions*. In computing systems the term refers to the sequences of messages between objects & with users and to the activities executed as a result of

[1]Booch, in his OOA, cites C.H. Waddington with this text.

these exchanges. In the following paragraphs both aspects are studied, which are the internal behavior of objects and their interactions.

2.2 Methodologies with Focus on Structure

From the studied methodologies, we selected for this group those of *Rumbaugh*, [Rumbaugh], *Booch* [Booch] and *Coad-Yourdon* [CY]. They are well committed to the 'problem domain' objects and the design is driven by them.

2.2.1 The importance given to behavior in these methodologies

A sensitive issue for this study is the importance given to behavior in these methodologies. Approaching *James Rumbaugh* we see why the behavior model comes for him *as a second step*. He says that 'It is necessary to describe WHAT is changing or transforming *before* describing WHEN or HOW it changes.' In his opinion, Object Oriented technology insists on what an object IS, rather than how it is USED. From this position he compares Object Oriented Development with database development. This raises our comment that this is *not the case for behavior intensive system*, where complex sequences of interaction must be part of the system development.
Booch has the *same position*, even if he is putting more attention than Rumbaugh on the dynamic model.
For *Coad & Yourdon* the concept of 'Service', which essentially may be considered the behavior aspect, is built *after forming the objects based on 'problem domain'* criteria.

2.2.2 Key elements, commonalties and differences in behavior modeling

The methodologies selected in this 'structural' group have been developed on the *rising edge* of object orientation theory and therefore they *had* to introduce and defend the 'Object' as the main concept. (After this rising edge passed, the time came to observe that the logic of interactions between objects snould have a similar level of importance as the objects.)
The main view is similar with Rumbaugh's : 'The fundamental construct is the object, which combines *both data structure and behavior in a single entity*'. The term 'behavior' here refers to the methods -public and private- encapsulated together. When it comes to explain where these objects come from, he says, as most others, that '**objects are found in the application domain**.' Rumbaugh and Booch devoted a good part of their work to system dynamics even if they are driven by the structure of the system. Coad-Yourdon address the issue of behavior, but with less emphasis.

In the following paragraphs we will address separately:
a) The behavior inside objects
b) The system behavior, or what is seen as interactions or object cooperation.

a) The Behavior Inside Objects

Typically, the behavior of an object is seen as the ensemble of the execution paths of all methods. The concept of *state of the object* is based on the values of some or all data members. When (some of) those values change, the object changes its state ; as a result, the methods change their execution path. Rumbaugh notes that *'the attribute values and links held by an object are called its STATE'*. Booch and Coad&Yurdon express with other words the same type of thinking.

In this manner, the state machine of the object is not explicitly implemented, even if inside documentation it may be defined. However, according to the main methodologies we can define *explicitly some sort of state machines in each object worthy to be considered dynamic.*

It is interesting to observe the *balance between the concepts of Events and States*, as we will insist in focusing on events rather on states. In state machines (and implicitly in these methodologies) the focus is on State, obviously.
Rumbaugh says that **events mark changes** and the state defines the context for events.
We would raise much more the importance of Events as they generate the dynamics of the system, which responds and acts as it was designed for the sequence of events. When the response finishes, the system remains in a state. So, we see the state as the end of the event-response 'vibration' of the system.
Why should we put emphasis on the time-outs and not on the game?
Such change of attention generates in this work a few theoretical and practical consequences.

Rumbaugh considers an event an 'individual stimulus' from one object to another, but adds an interesting suggestion: to group events into EVENT CLASSES. There may be classes of events that may either be simply signals (something has occurred) or transport some data values. The significance of events is increased with these event classes : the essential part of interactions, the event, is considered here as any normal object, of the same kind as the 'problem domain' objects. This way the basic distinction between components and interaction is blurred.

From a practical point of view though, this procedure adds some difficulty. As in a system there are many events, it is not handy to build an object for each of them.
Our view will be to define an object that will capture the logic of interactions (the dialog) between a number of objects. Thus, the full dialog becomes an object instead of making an object for each event.
Rumbaugh considers in this sense some *control* objects, managing a group of objects, implemented as state machines, but these are 'not application objects'. They are 'part of the language substrate' as a support, not as an expression of design, in the same way classes are part of language. *We consider that such an object can be an essential application object, helping to define the dialog between the underlying objects.*

The position of *Booch*, concerning the importance of Events versus States, is that events help to
- define the boundaries of the system's behavior,
- assign responsibilities to classes that support the system's behavior.

In a way this may be seen as an *event driven design.* At the time of building the system, the events are establishing the boundaries and are the source for defining responsibilities.
This view is more in the spirit of our focus on dialog (the events-responses sequence). We believe that defining first the dialog, we can carve the needed subsystems or objects and we can build methods based on this dialog.

However Booch does not extend too much this approach. *He is mainly developing his behavior modeling from roles and responsibilities of objects*, granting the necessary credit to R.Wirfs-Brook (details on this last methodology in the next sub-chapter).
In this type of view an object is a *server offering services* to its clients, according to its responsibilities as assigned at design time. *A 'role' would be a group of responsibilities and the methods of an object represent its protocol.* This *protocol defines the 'envelope of an object's allowable behavior'*. Once this concept of 'Protocol' is introduced, we would expect to see a way of describing the relationship in time between methods. We *do not get* the suggestion of building some software as a protocol machine or a state machine to describe the given protocol.
According to our experience, the 'totality of attributes' defining the state is way too big to be kept under control by designers / implementors. With this definition of 'state' it is ambiguous and difficult to *implement* a state machine as it was designed.
If we really want to capture that 'protocol', an extra step appears necessary. One needs to implement a *distinct software machine that explicitly* executes the intended path, with a distinct State Attribute. As said, the methodology does not suggest such implementation.

Coad & Yurdon put less emphasis on dynamics ; they consider that 'Every change in Attribute value(s) reflects a change in state' and is supposed that the system's responsibilities include different behavior for different states.
If the object is 'responsible' with a method towards exterior, the behavior of the method may change based on an internal value, which is in the state's space.

As we can see, there is much emphasis on using state machines for object behavior design. Rumbaugh and Booch use a StateCharts type of state machine [Harel3] while Coad & Yurdon apply a simple model. Though, there are not many suggestions about how to implement the state machine design and the programmer may interpret the design in different ways.

b) System behavior

What happens between objects? What happens between the system and the users?
The ether is filled with events, or messages, or request-responses from any object to possibly any object. Because the object of the attention is the Object, the definition of interactions is secondary. How one builds these interactions? The engineer has to use his ingenuity somewhere and this is an interesting field to apply it.

Rumbaugh notes that '*A dynamic model describes a set of concurrent objects, each with its own state and state diagram*'. Therefore the *system state is the product of the states of the objects.*
For system level design Rumbaugh and Booch use nested and parallel state machines, as in Harel's StateCharts, with some modifications. Coad & Yourdon employ simple state machines.

These machines should define some supplementary organization beyond objects. It remains a problem where and how these system-level machines are reflected in objects ; actually in Unified Modeling Language [UML] the idea of defining the system behavior with state machines is avoided.

Another problem appears when one tries to relate states-drawing with programming. Nesting and parallelism of state machines have not an obvious representation in software. The solutions are let at the choice of the programmer, as for the case of simple state machines. There are some tools doing some type of implementation, but they come with much overhead which is avoided in many systems. Also *class inheritance* and *nesting of state machines beyond objects* are two issues that do not easily fit together. For these kind of reasons, UML later reduces the use of StateCharts at the level of objects.
These problems rolled out a lot of research [Colleman] [Harel3]. The usual proposals are new layers of complexity on top of the software, with their own language and image constructs trying to interleave with Object Oriented concepts. By contrast, our behavior elements -the Dialog Coordinators and the Responsive Units- are aimed for simplicity of design.

The **Dynamic Model of Rumbaugh** specifies and implements the *control aspects* of a system and the interactions between objects in the system. Some objects are 'more' dynamic than others, as in data-warehouses where a lot of objects are information bases, needing mainly set-get actions and a few are user interfaces.
The *Control part describes the sequences of operations that occur, regardless what the operations do internally.* This way of thinking is well in the Coordination Oriented Development style, that is promoting the separation of coordination from activities.

Relative to this control aspect, we find at **Booch** a supplementary design element :
the Mechanism, which is a 'decision about how collections of objects cooperate'.
It represents a 'pattern of behavior' appearing conceptually as a higher level object with

- internal behavior: the logic between internal objects,
- group responsibility, the dialog with other mechanisms.

An interesting step may have happened in Booch methodology with the idea of *'Mechanisms'*. These, being *related objects* fulfilling a higher level of responsibility (seen as 'patterns of behavior'), would have a *set of internal messages* and a *public interface,* much as an object has.
This concept is however not too much expanded with practical solutions.
We retain from 'Mechanisms' the ideas of

- Restricted dialog,
- Framing objects in a higher level entity,
- Keeping a portal to outside world.

These help us to motivate and to introduce the Responsive Unit, which is a part of our suggested model.

The objects interactions are represented with **Scenarios, at Rumbaugh, or Interactions Diagrams, at Booch.** Each object is represented as a vertical line, in the direction of time-flow. The messages are functions belonging to the called object. One diagram represents a unique sequence, as in figure 2.1, and to cover all possible cases one would need many diagrams.

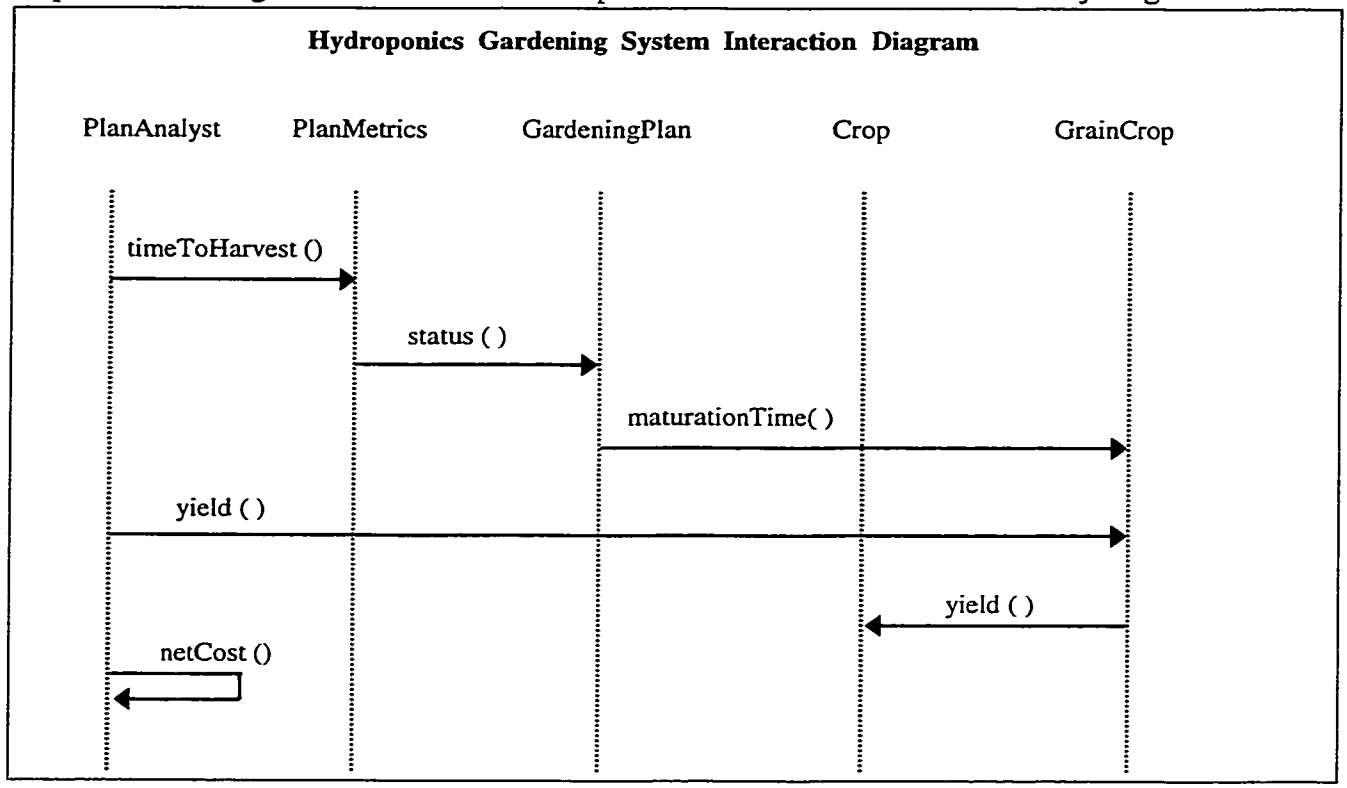


Figure 2.1 Example of Interaction Diagram

One comment on this model is that there is no restriction or rule from the methodology on who may communicate with whom. Thus any calls are basically allowed, as before Structured Programming a *GOTO* statement could lead anywhere.
An other comment on Interactions Diagrams and Scenarios, is that each object has a single path : *no internal decision* points, *no differentiation of paths for different events.* Booch suggests some supplementary 'scripts' beside the diagram, to define major decision points.
In this way the description is done for *one single case of execution* and as we cannot afford to draw a page for each possible case, we have to pick the most generic ones. But which are the most generic and how can we consider the design sufficient without the others?

Object Diagrams are another manner of representing the same things as Interactions Diagrams. The objects are 'clouds' and the order of execution is given by a sequence number. Again, it represents a unique path, eventually the most important.

As we see, the interaction logic, the Dialog, can not be completely designed with the actual procedures. Programmers can not pick the design document and proceed to their work, the implementation. Any meaningful enhancement would help and we believe that the use of the dialog coordination concept and of Responsive Units bring a certain advantage.

2.3 Methodologies with More Concern for Behavior

For the scope of this study, we grouped together some methodologies that address more directly the problem of *interactions between objects and with the user*. These are the methodologies of

- **Jacobson,** 'Object Oriented Software Engineering, a Use Case Driven Approach'[Jacobson],
- **Shlaer & Mellor**, 'Object Lifecycles, Modeling the World in States' [ShMellor],
- **Wirfs-Brook**, Designing Object Oriented Software, A Responsibility Driven Approach [Brook].

We will look in this part only at the manner of dealing with the *interactions* of objects ; the *internal* behavior of objects is defined as in the previously studied methodologies.

The logic of interactions, that is the dialog between objects, is not quite visible nor as tangible as the 'problem domain' objects. These objects are the first design elements in the previous methodologies. However, if the dialog is not visible, it surely is think-able. We can easily understand that a volleyball game means more that a few players, a net and a ball; the dynamic interactions of the game have an essential role in defining the 'system'. In the same way, a Parliament is not only a building with people ; La Parole is essential, being the dialog that links the people according to group goals and rules.

Ivar Jacobson, while dealing with the telecommunications company Erricson, had to face many problems concerning the interactions inside systems and at interfaces. He defined his methodology well after the approach of object oriented development was accepted in the software world. Therefore his energy could be focused at what happens between objects, rather than trying to convince software developers about the value of objects. This being his domain and his time, he could bring a *new layer* of modeling, on top of the usual object oriented paradigm. The approach is considered a *'use oriented' development*, since the design is driven by the way a system *is used* instead of being driven by 'problem domain' objects.
The success of this methodology may be surprising in a world of objects. One cannot forget the strong commitment of the object oriented camp to define first *what is* an object ; how *it is used* was seen as a secondary issue. However, with the basic reflection that objects have to be used and to interact, there should be no surprise that Jacobson succeeded. *He brought back a due balance by* saying that *what is* a component is based on how it *interacts*. In our words, Jacobson's view would mean that *components are defined through interactions*.

A definition for the central concept of 'use case' would be as quoted: *a user performs 'a behaviorally related sequence of transactions in a dialogue with the system. We **call** such a special sequence a **use case**'.*
A system may have several use cases and it may interact with several 'actors'.
In the requirement model, each use case is described in plain English as a sequence of interactions. In the analysis model the problem domain objects are generically clarified and for each use case there is drawn a scheme with the implied objects. These may be *entity objects, interface objects or control objects.*

The *entity* objects model information in the system that is held for a longer time.
The *interface* objects model part of the system interface or relate to it.
The *control* objects model 'functionality that is not naturally tied to any other object'.

To compensate this last 'definition', Jacobson gives an example with some 'behavior that consist of operating on several different entity objects, doing some computations and then returning the result to an interface object'.

The 'construction' model includes the design and the implementation. The generic objects from the previous model are considered here 'blocks'; these may be implemented with one or more objects.
In the design model *each use case* is represented as an *interaction diagram*, where the vertical lines represent blocks which are related for that case. By considering *all use cases* in which a block is involved we get a complete picture of its interface. A state machine captures every 'stimulus' (message) from any case and defines thus the 'protocol' of the block. For the implementation of blocks with several objects it is suggested the use of one or more interface objects.
We notice that the blocks and objects (the 'substance' of a system) are build based on *sequences of messages* described in use cases, as in figure 2.2.This perspective helps us to promote the observation that the dialog is essential, observation with a major role in this thesis.

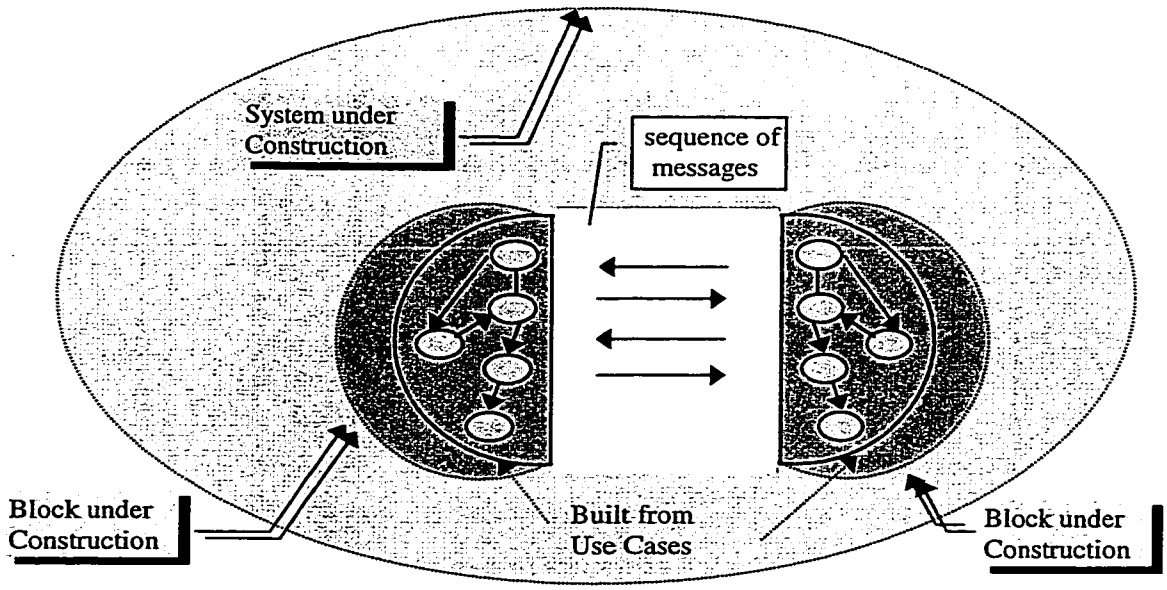


Figure 2.2 Blocks Built Based on *Sequences of Messages*

The methodology clearly introduces a new direction for development but the behavioral representation of the system is still typical : the behavior of objects is defined with state machines and the relationship of objects is defined with interaction diagrams. The issue that interaction diagrams can only partially represent responses is not alleviated.

Considering **'Modeling the World in States' of Shlaer & Mellor** we find that not only objects need to be modeled with state machines but *also the Relationships.*
The Relationships between problem domain objects are observed as having their own lifecycle:

- They evolve in time,
- A Relationship has an Instance, which is created or destroyed,
- They are active, selecting objects around for interaction.

Relationships are represented by state machines, using a specialized class named 'Finite State Model Class'. They mainly act as an 'Assigner', connecting objects that need one an other (for example, to a bank client will be assigned a free clerk).
Here we see a case where the *Relationship becomes a Tangible Object.*
However, not the dialog itself is objectualized, as we intend to suggest in this thesis, but the *instantiation of the Relationship is an Assigner; it connects dynamically Objects* that need to talk as suggested in figure 2.3.

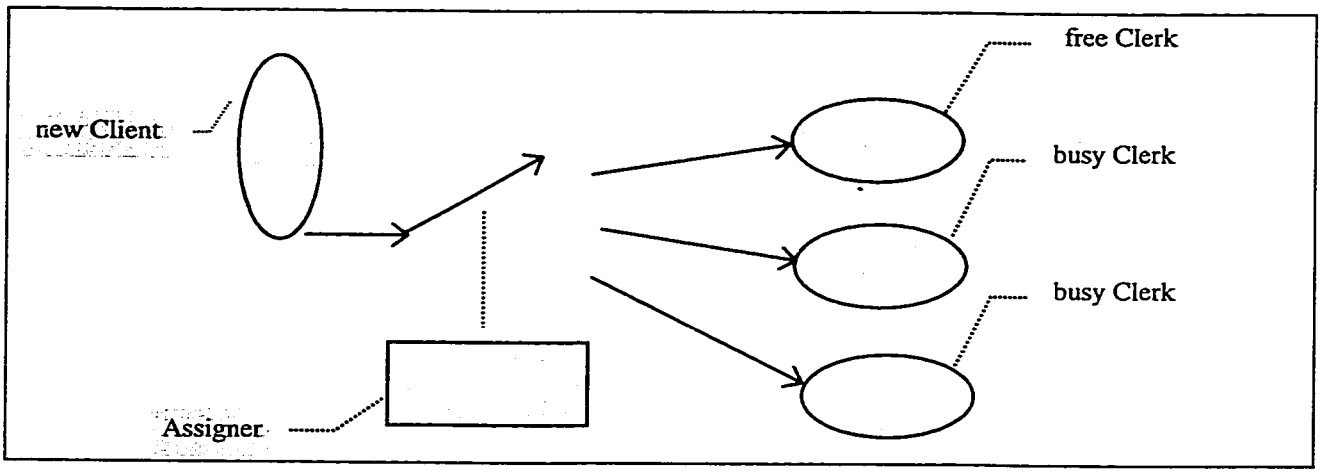


Figure 2.3 Relationship as an Entity

Again we compare with our vision, where the Dialog is seen as an Entity itself (figure 2.4) because in software we can build a 'world of words'.

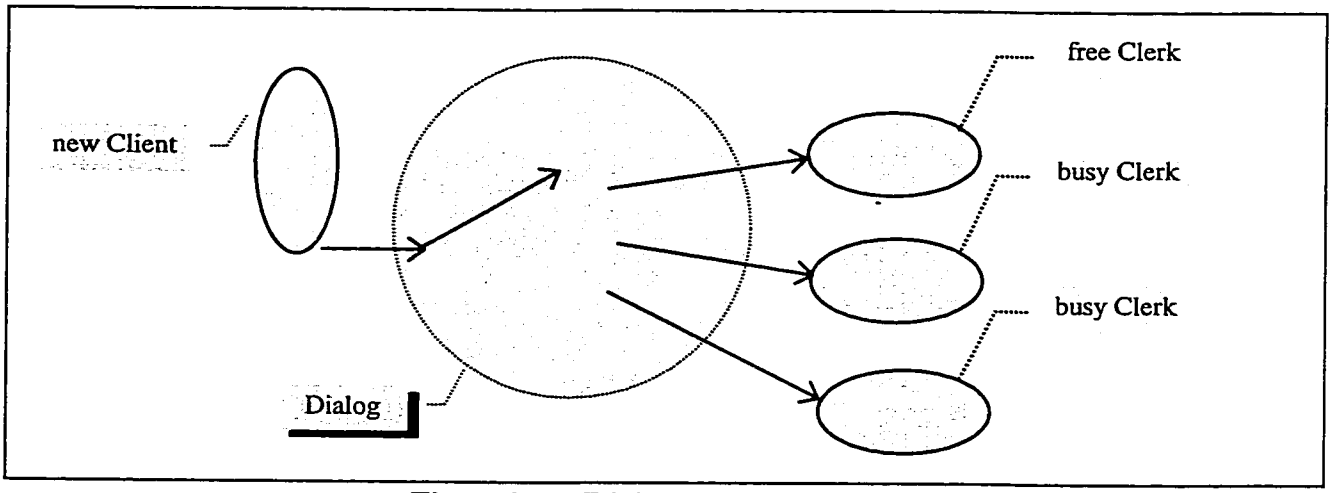


Figure 2.4 Dialog as an Entity

In software we have the chance to observe the duality Dialog-Object in a new light: we may have objects defining the desired interaction logic (the dialog) between the problem domain objects. These 'dialog' objects will coordinate the interactions. Moreover, we will suggest that also the other objects are carved by the intended logic of interactions, the dialog.

In **R. Wirfs-Brook methodology** -a **'Responsibility'** based design- the objects are *not anymore simple mirrors of some reality*. They are designed based on the 'Responsibilities' which must be shared between the parts of the system.
Observing the key word 'responsibility', we note that it corresponds to a part of the pair request-response. More such related pairs define the Dialog, on which we focus in this work.
The Responsibilities, that are parts of the dialog, shape the objects in this methodology.

An important issue about Responsibilities is how they are going to be allocated.
There are two extreme cases: *an even distributed intelligence* OR *having one object with full Responsibility*. Both extremes are hard to apply and therefore R. Wirfs-Brook develops the concept of *layering* Responsibilities 'by properly encapsulating' subsystems.
In each of these subsystems, there is designated an object able to *respond* at major *requests*. The communication with a (sub) system will be mainly through this object, which has a higher Responsibility (figure 2.5)

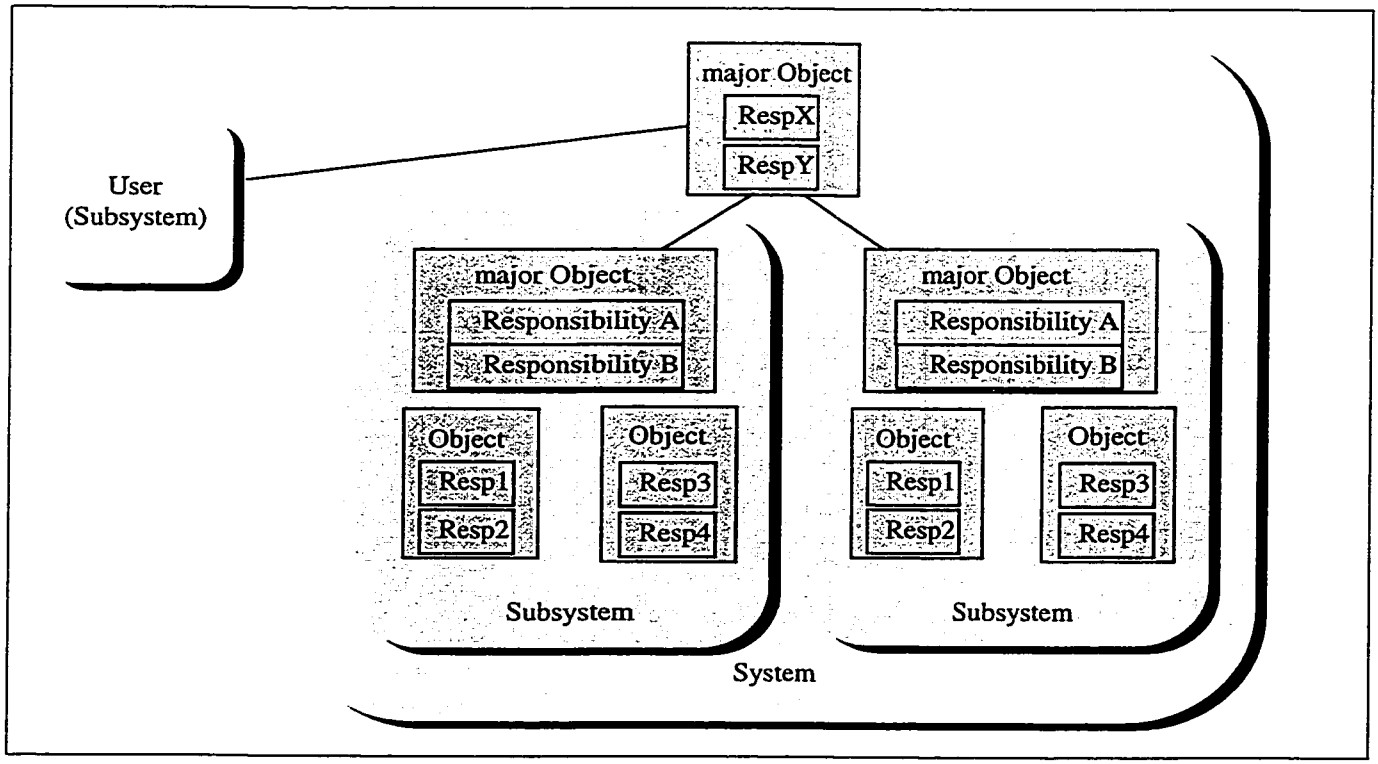


Figure 2.5 Layering Responsibilities

This model is structurally similar to the Coordination model ; but from the behavioral view here is no interest to explicitly implement the precedence relation in which the low level responsibilities have to be executed. The dynamic logic of interactions remains unaddressed.

2.3.1 About the utilization of State Machines and Interactions Diagrams

At **R.Wirfs-Brook** the concept of 'Responsibility'(and encapsulation based on it) may be partially seen as a Behavioral approach, but the real *flow in time of events-actions is not represented* in this methodology. *There is the suggestion of a 'walk-trough'* the design, in order to see how the classes will *'work together to provide the behavior'*.

Shaeler & Mellor and Jacobson *employ state machines* as the modeling facility for the dynamics of system and objects. This mainly is done as in the other methodologies; now we only extract new ideas *or* enhancements.

Shaeler & Mellor use a basic state machine model. Their novelty is not with the model, but with the mode of using it. As presented, the Relationship is seen as a candidate for instantiation. Interesting is the manner in which it is implemented. *The Lifecycle of a Relationship is described in terms of a state machine*, which is meant to capture the discrete evolution in time of this Relationship, based on the requirements.
We note that this model is not reflecting the Dialog itself, but the different states -as different ways- in which Relationships may exist.
As a matter of attitude -showing how important it is for the authors this view of the 'world in states'- they offer, for implementation reasons, a *ready-made state machine class* as a fundamental architectural element.

Jacobson chooses the type of state machine used in SDL. He comes from an environment where SDL is highly used as a CCITT standard in telecommunication projects.
Looking first to the state machine model itself, we appreciate the enhanced *importance given to events* and responses. Both are drawn as distinct areas in the space of the state machine, as an incoming, respectively outgoing arrow with defining text inside. Actions are shown also as distinct entities, as opposed to classic state machine where a response-message and an action are identically represented.

With such focus on the event-response pair, these machines could be better named Dialog Machines. We say so considering that in a system the important thing is not how it stays in a State but how it moves according to the intended Dialog.

Also it is meaningful to pay attention to the *Decision Diamond*, an element that is not found in the classic state machines. It is *not a State*, as the system does not stop its execution expecting an event, but however we get more branches from such a point.
Jacobson finds an advantage in using it because it makes a distinction not found in usual state machine models. A decision may be taken *outside* an object, transmitted by different events from an other object, or from *inside*, based on data values. Use Case uses this distinction, as the place of a *decision must be clearly designed.* A decision can be (1) internal, (2) made by an interaction partner or (3) it may be a group level decision. For the public/group decisions, we suggest that a supplementary object may be built in a group to handle them.

In figure 2.6 one can see the distinction between an *internal* and an *external* decision; also we note that the signs for actions and messages are different. These help to clarify the relationship with the user and thus, to design better the Use Cases.

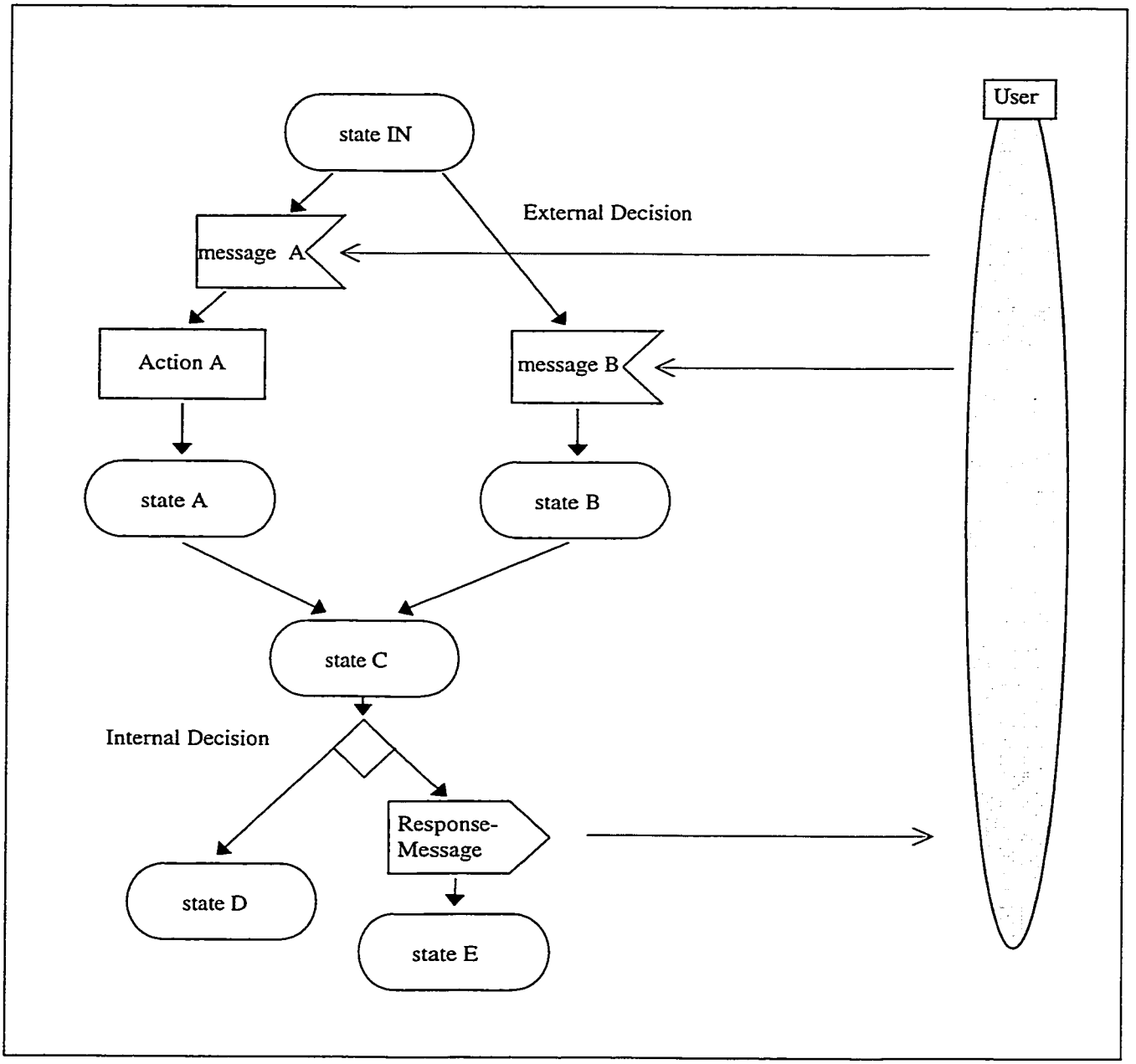


Figure 2.6 Internal and External Decisions

When the events are actually functions with data-parameters, the designer has to establish if decisions belong to the caller or to the receiver. For modeling it is important to define what an event is in this case. Each function may correspond to one single event or a function with two different values of a parameter may correspond to two different events. A uniform design may help for a good relationship between design and code.

The objects that are considered active, which are modeled with state machines, are of type 'interface' and 'control'. The third type, the 'entity', is mainly a repository object where data is 'set' or 'get'.
The 'interface' objects are considered more important than the 'control' objects. These last are actually elements that do *not* fall in the other two categories (interface or entity) and they do *not get their own* definition.

The **Interaction Diagrams** in these methodologies are quite similar to those observed in the previous chapter. However, we see at Jacobson a distinction, valuable to our research, when he speaks of two extremes of design with Interaction Diagrams :

- One is named Fork Diagram; it is a centralized manner of design where an object drives the others.
- In the other extreme, called Stair Diagram, each object has pointers to other objects that can offer a service and there is no central entity.

We believe that seeing these two ways of design as two extremes may be reconsidered. The interaction logic (the dialog) may be defined in an object that fundamentally is not a problem domain object and thus it should not be aligned with the others.

Even if our attention is set on what happens beyond and between objects, it is worth mentioning the position of Jacobson concerning multi-processes systems. Actually his ideas may be applied also to the other authors and we consider them good clarifications/delimiters for this research.
For multi-process systems OOSE (Use Cases) 'does not have *a priori* strong semantics' since '*various implementation environments provide various semantics*'. Since *synchronization and communication* are the main issues, he suggests that the semantics of the actual environment should be used, eventually to form '*specializations of OOSE*'. Also he notes that the idea of *object orientation is in good correlation with concurrent processes* and often objects fit directly in processes.
For the scope of this thesis we only say that starting a separate coordination process may be a useful portal for synchronization-communication calls between processes. We consider this idea for future research, in the 'Conclusions' chapter.

2.4 Unified Modeling Language (UML)

UML [UML] is the result of the common work of Rumbaugh, Booch and Jacobson who unified their views for the benefit of the development community. UML is a modeling *language*; the authors have defined in Rational Unified Process a mode of applying UML.

It is suggested that the development should start with *'use cases', which is a behavior driven approach for system design*. This decision of the authors represents a major novelty for the object oriented world. The original approach in OOD was to first define *what a component IS* and after that to define *how it is USED*.

For behavior modeling, UML uses a variant of *StateCharts [Harel3] named Object State Machine*, with differences resulting from the way in which the model is applied. Essentially, *in UML the Object State Machines represent the behavior of an object* rather than the behavior of a sub-system. Therefore:

- An event can carry parameters (as the methods) instead of being simple signals.
- Events are directed to an object, not broadcasted.
- The synchronous communication is supported, by direct method call.
- The transition time is based on the reality of software execution.

There are also several details from StateCharts that are minimized.

By choosing to define with state machines only the behavior of objects, UML avoids the major problems encountered when state machines are used beyond classes. In this approach, the interactions between objects have to complete the behavioral design up to the system level. The messages between objects are defined in *interaction diagrams* and this is (as usual) *only a partial definition*.
Besides *Object State Machines*, UML offers also an *'Activity Diagram'* to model computational processes. The states are *action* states ; when an *action finishes, the flow can continue* without an event. It is suggested to apply them at system level, for a kind of generic functional description [Fowler]. They are used beyond objects and it is a difficult task to translate them into the design of objects. Therefore the main behavior model is still the *Object State Machine* .

The UML team works from a company named Rational Rose. Lately they bought ObjectTime, which is a CAD design tool based on the Real Time Object Oriented Modeling (ROOM) method [Room]. Consequently, some concepts from ROOM are suggested on top of UML, as being useful for real-time design [UML2].
1) For *structure modeling*, three principal constructs are considered:

- A 'capsule', which is a complex object that interacts with the surroundings through one or more port objects.
- A 'port', which is an object part of a capsule that plays a particular *role in the collaboration* with other capsules. A *protocol* is associated with a port, defining the acceptable flow of signals.
- 'Connectors' are abstract views of communication channels.

A capsule can have an *internal network of collaborating* capsules.

2) The behavior modeling refers to:

- Capsules, which may have a state machine implemented to define the behavior.
- Ports, which need to define:
    - *Protocol roles,* i.e., groups of incoming and outgoing signals.
    - Optionally a state machine can implement the order of allowed execution.
    - A protocol may have a set of 'prototypical' interaction sequences. Interaction diagrams may be used to present them.

Some observations on this real-time extension of UML are as follows:
(a) The definition of capsules is in line with the usual view of building block or modules for structuring the system. Capsules can have sub-capsules at any depth.
(b) Their communication is free as in normal object design ; however the ports add more precision to the allowable sequences.
(c) The state machine inside a capsule (as in usual objects) offers only a partial definition of behavior, since more sub-capsules can form an internal network. Each has its own behavior and the interaction diagrams between them define only partially their interactions, as usual.

As major observations on UML, we underline that :
(a) *Object State Machines define precisely the behavior at object level and not beyond.* Other methodologies allow the use of state machines beyond objects, with related difficulties. UML takes note of these issues and clarifies the situation. Therefore in UML the *system behavior* is defined through the *interactions of Object State Machines.*
(b) By applying *'Use cases' , UML addresses the major interactions in the early design phase at a time when the objects are only identified. The design of objects is continued based on intended interactions.* In this thesis we will expand this direction, suggesting that the *'dialog'* between components -the sum of all their interactions- is essential and for small groups of objects we intend to *completely* design their interactions. To fulfill this intention, we will introduce the Dialog Coordinator and the Responsive Unit concepts.

2.5 Other Methods for the Behavior Modeling of Object Oriented Systems

Besides usual object oriented methodologies there a number of other approaches for capturing system dynamics. In the following study we look for essential ideas that are somehow distinct from the ideas already presented.
As major methods that are better known or used, we consider the following:

- *Object Modeling with StateCharts* [Harel3]
  StateCharts is an extended state machine initially created for general system design [Harel1]; lately its use was revised for OOD applications.
- *Specification and Description Language for OOD, SDL-2000* [Bjor]
  SDL is a model based on communicating state machines [SDL1] used in multi-process systems (not necessarily object oriented). SDL-2000 is an object oriented adaptation.

Other research offering some interesting aspects for event response modeling would be:

- Application Development through Object Composition by Means of Event and Object Environment [Rizman]
- Introducing ObjectCharts or How to Use Statecharts in Object Oriented Design [Colleman]
- Research on Message Sequence Charts and Message Flow Graphs [Leue]

2.5.1 Object Modeling with StateCharts

StateCharts is an extended state machine model introduced by Harel [Harel1] with the aim of representing the behavior of more complex systems. The model has two main characteristics : (1) it *hides* different layers of behavior in sub-machines and (2) it employs a certain *concurrency* between sub-machines. StateCharts was chosen, with small modifications, by Booch, Rumbaugh and by the Unified Modeling Language (UML) team to be used for behavior modeling.

In '*Object Modeling with StateCharts*' the use of the extended state machine is *reduced to the definition of object behavior*, as in UML; for system behavior Interaction Diagrams are added.

For this study our attention is directed towards two aspects:

1) The way the *interactions between objects are addressed.*
In this respect, it is proposed that the events may be asynchronous events queued by the object, or may be direct method calls. The Interaction Diagrams are applied as usual.

2) The way the required *event responses influence* the design of a StateChart *inside an object.*
For this reason, we will discuss the proposed modes of state decomposition.
Harel considers the State Machine model relevant for behavior modeling because 'the system changes *its state when an event* is received'. But state machines 'suffer from being flat and

unstructured...and give rise to an exponential blow-up in the number of states'. Therefore a certain state *decomposition* is introduced.
We consider that although such reason for decomposition is fair, there may be also an other motivation: the need of layering according to the abstraction level of events. This decomposition would be driven by the logic of interactions between objects.

The decomposition is done in two ways:
- The 'OR' decomposition from the need for *depth*,
- The 'AND' decomposition from the need for *orthogonality*.

a) The OR decomposition

The *depth of state machines* is built either as *clustering more states to one* (top-down manner) or a *refinement of a state in more states* (a bottom-up approach)

In Statemate the *Clustering* process is based on observing that the same event b changes the system from state A or C in B, as in figure 2.7.

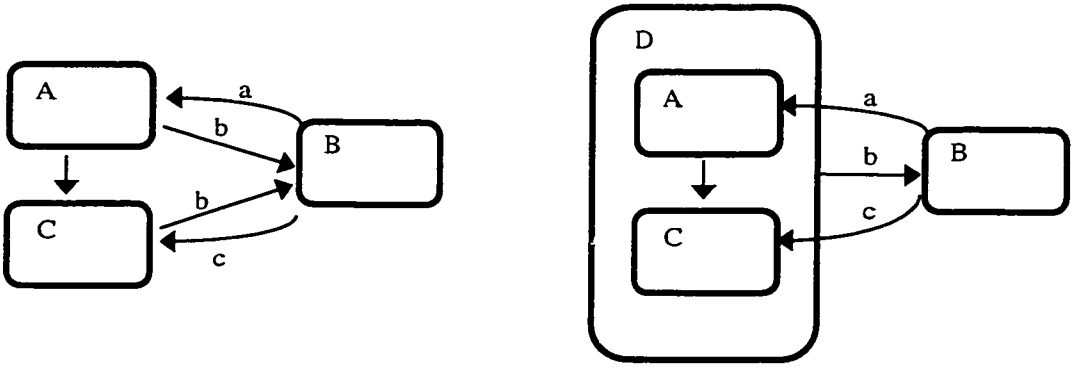


Figure 2.7 Clustering

One builds a *cluster state* D for states A, C as a result of the unifying process of the transition on event b. When the system is in state D, it is either in state A OR in state C, NOT in both. More precisely, the conceptual OR is an *exclusive* OR, XOR.

The *Refinement* is a process of building an OR decomposition starting top-down. First is defined the high level view and because events a and c are different, substates A, C are built.

Referring to way the event responses influence the design, we note that:

- *The response, as essential counterpart of the event, does not have any role in decomposition.* A response may be a message to an other object or an action, which are critical parts of requirements.
- *Also we do not see how the importance of events can influence a decomposition*

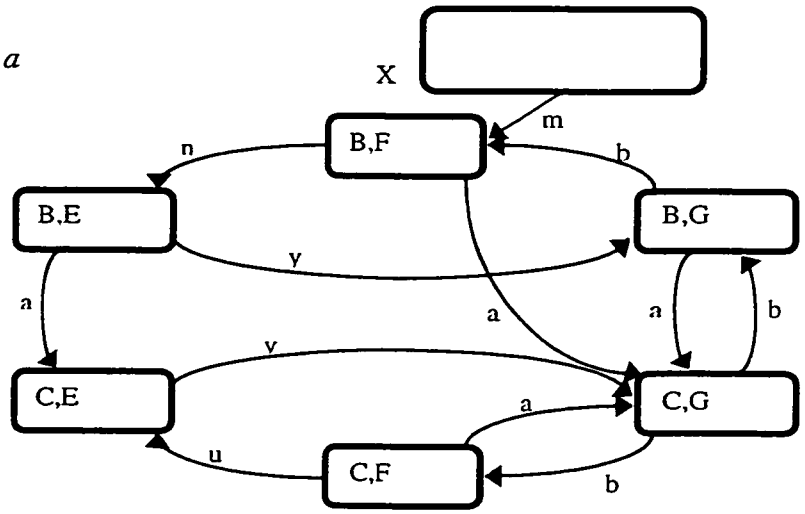
b) The AND decomposition

This kind of approach is driven by the need for orthogonality, which is offering independence of behavior and a *certain concurrency*. However, in the last proposal a StateChart is used only for one object and concurrency *inside* an object is usually not suggested.

According to AND decomposition a state Y may be composed of two orthogonal state machines (see figure 2.8 b). When the system is in state Y, it must be in one substate from the left side AND in one substate from the right side.

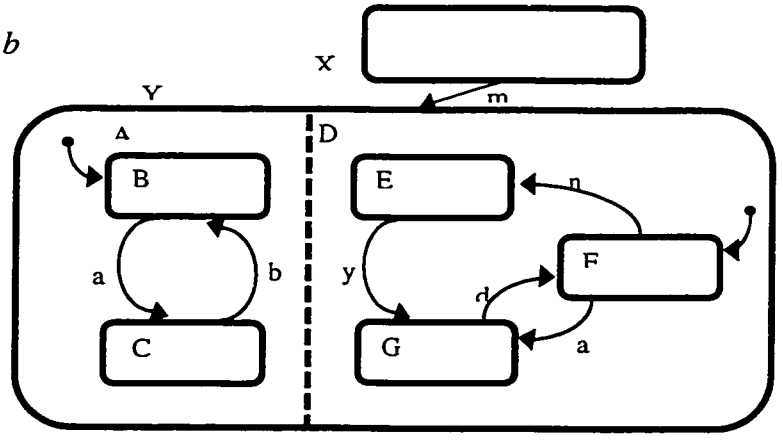
Figure 2.8 AND Decomposition

Figure a



AND decomposition:

Figure b



We do not see, as for the OR case, the role of the response in the decomposition.. Also the source of events, their sequence, their layers of importance have no visible role. These systems are seen more 'mechanisms' than 'dialog oriented' systems.

The logic sequence of events and responses is important also for system comprehension. The system model should clearly present:

- *Where events come from and where response-messages go,*
- *How they are related in a timely logic (the dialog history),*
- *How a group of responses can form a higher level response, etc.*

Here is a major *focus on the concept of 'State'* and we suggest that more *focus on Events/Responses and their dynamic logic* would only help. We underline that the inter-object *dialog is essential* and it should be clearly addressed in design.

2.5.2 Specification and Description Language for OOD, SDL-2000

SDL was defined for a formal specification and description of telecommunication systems. It was developed by the telecommunication companies of Sweden and Denmark [SDL1] and standardized by CCITT. The theory of Communicating Sequential Processes of Hoare [Hoare] had an important influence for the developers of SDL. SDL-2000 is an object oriented adaptation (mostly based on the original SDL) intended to fit UML. Thus a new convergence is under way.

The OO-SDL uses 'active objects' and 'passive data objects'. An active object (or 'agent') has its *own thread of control that can be triggered by a timer expiration.* As such, SDL-2000 addresses only multi-threaded systems. We are searching for an improvement in the behavior modeling of the general OOD, that has to apply also when multiple objects are in one thread or in mono-treaded systems. We will filter the SDL concepts through this perspective.

More active objects communicate by asynchronous message-passing. Thus, they correspond to the processes from the original SDL. The *system behavior* is the *combined* behavior of the active objects and their interactions.
Part of the development consists in defining *boundaries of behavior that delimit active objects* and establishing their *relationship. The development is behavior driven.*

The behavior of an active object is described as an *extended state machine*, independent, concurrent with the other active objects from the system. Why this choice? Because an active object 'does nothing (or runs around doing the same thing) until some request comes'. This is the usual wisdom about any apparatus: the system would be a slave that is *waiting* for a command and responds to it after it comes.

At this point we underline again the importance of the event, as it changes the system state as intended by designer. Moreover, the timely logic of signals (the dialog between active objects) is essential and it shapes at design time two or more state machines.

The active objects can communicate in an *asynchronous* mode using 'signals' (figure 2.9). Function calls are not applicable, which for the general design mode would be an impediment. Active objects may have more associated *Interfaces* (specific objects) linking them to the external world.

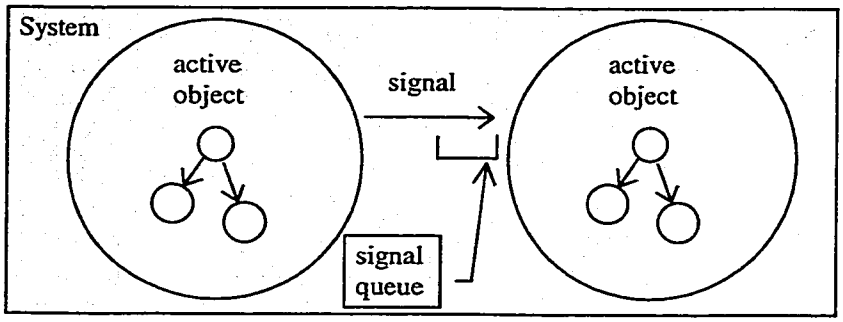


Figure 2.9 Asynchronous Communication

In order to organize the system, more active objects are contained in a Block-object. Here we observe an interesting idea: the *Structure is built based on behavior partitioning.* Inside a block, the connections between processes are done using Signal Routes. *Between blocks* the communication is accomplished through *Channels.* (Figure 2.10)

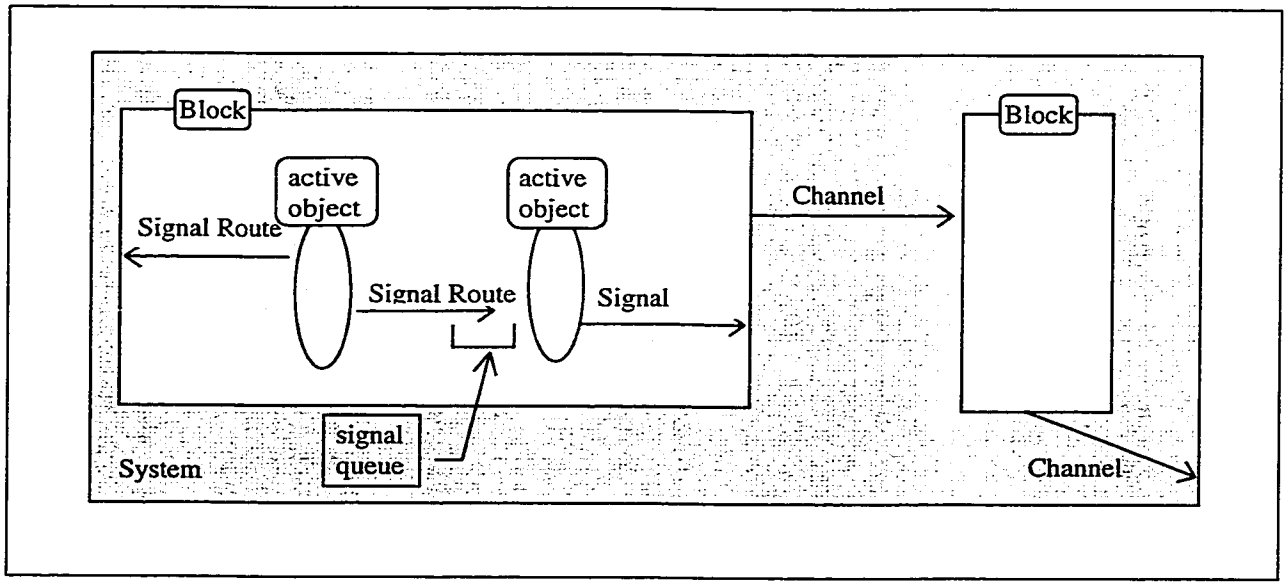


Figure 2.10 Communications

A channel also can make the link between a block and the environment. The concepts of Blocks and Channels scale up as needed. Inside a block there may be as many active objects as desired, allowing the use of one single block or too few for all active objects in the system. The signals park in a FIFO queue and are *consumed when used, even if they are not expected* in a certain state. There is no *mention* of priority, round robin input, preemption, sets of events, event reuse, or other techniques that are necessary in specific cases.

A signal can *carry some parameters* that are copied in the private memory. An *address* is defined for each signal: the *identifier of the receiving* active object and eventually the *signal route or the channel* may be named. *Activities* are executed *only on transitions.*

It is suggested that the expertise of SDL with interacting state machines can be applied for the usual OOD and can complement the behavior design in UML. However, for the general OOD with UML, the requirements of SDL to have active objects (with their own thread of control) and asynchronous message-passing appear as a too tight expectation.

We reiterate one of the main comments in this thesis : we see the events -and the dialog they form- as a major value of a system. The system is built through dialog and at run-time the system runs through this dialog. The value of a system is given by its logic of interactions, by its event-response sequence model. We look for a better way of capturing the dialog and this will be not only on paper, but dynamically in a separate software module ; this one will coordinate -in a way specific to COD- the interactions between the other modules.

2.4.3 Other research offering some interesting aspects for event response modeling

In this part we present some research papers that are concerned with the *express* definition of logic of messages.

1.Application Development through Object Composition by Means of Event and Object Environment [Rizman]

For an enhanced reusability of objects, the author suggests that a better definition of events is needed. The approach is based on the concept of 'object environment' and on events' exact description.

All interactions between objects have to be defined in the 'object environment' document in terms of event-actions rules. These interobject rules describe the required behavior of more objects and they form a virtual (in a document) manager. For the definition of each class, the respective event-action rules should be attached.

We consider that the idea of defining event-actions rules is valuable, but we raise a point: this way one defines *dynamic* rules only in a *static* language/document.

Our observation is that in software we are able to define the *dynamic* rules in a *dynamic* language, in a special object. This is not possible in construction engineering: the plans for the interactions between bricks cannot be captured in a brick. Biological cells however can do it ; teams of humans may have a manager; in hardware often is found a manager IC. In software we can for sure apply the idea and not only as an ad-hoc approach, but as a well-defined methodology.

2.Introducing ObjectCharts or How to Use Statecharts in Object Oriented Design [Colleman]

An extension of Statecharts is presented to allow the specification of transitions in state machines. In this approach any Statechart defines one object, as in the latest proposal of OO Statecharts.

At the original model two notations are added on each transition:

(a) The observer, which is the name of the method(service) that is allowed to provoke a transition. It is a better clarification of the event.
(b) Attributes, which are arguments of the methods/observers that influence the transition.

With this extra information, one should be able to define in some *lists* (in a document) all acceptable *traces* of interactions between object state machines.

We consider the approach a good solution but though partial: too many lists may be needed for a complete definition, as in the case of Interactions Diagrams. Also, the previous observation applies here too: a *dynamic* language would better define dynamic traces than only a static document.

3.Research on Message Sequence Charts and Message Flow Graphs [Leue]

In this paper it is underlined that the Message Sequence Charts (similar to Interaction Diagrams) need some enhancement as they only present one execution path in a chart. After some research it is concluded that a state machine in a separate plane than the plane of the chart will add value. This state machine will receive and redirect the messages, as appropriate in a given state of the system.

We see the advantage of taking in consideration the system state and events, and modeling them as a state machine in a document. However, the objects from the chart have also states on which the progress of a chart may depend ; these remain unspecified. It is interesting to see though that Message Sequence Charts themselves generated an entire research; it is hoped that definition of the interaction logic may be enhanced.

Chapter 3

Coordination Oriented Development Environment(CODE), a Behavior Driven Approach

Introduction

Coordination Oriented Development Environment (CODE) is a systematic engineering approach to the development of event-driven computer based (software) systems. The method was developed by Krieger at Ottawa University, originally for real-time event-driven systems (EDS) and applied for flexible manufacturing systems [Krieger1] and for embedded systems [Krieger2]. The method was seen initially as a way of managing multiple activities, as in multi-processor systems or in manufacturing systems ; the original name was Multiactivity Paradigm. Since the activities must work in a specified precedence relation to produce a requested outcome, *the specification of activities was seen distinctly from the specification of the precedence relation, i.e., the coordination of activities.*
After several applications have been developed using the paradigm, this one appeared to be enough independent of application and was considered as an *interesting way of addressing complexity*. In the general quest for enhanced software design methods, the Multiactivity Paradigm began to be seen as a noticeable alternative. Later it was better organized and the name was modified to *'Coordination Based Design'* [Krieger5];thus, in the name of the methodology, the focus changed from activities to their coordination. A development environment was defined around the method in *'Coordination Oriented Development Environment'* [Krieger3].

The purpose of this chapter is to present the CODE concepts, to make references to other research on Coordination and to present some existing applications. We do this since the next part of the thesis -containing specific views and novelties- is tightly bound to the coordination concept; a good explanation of CODE is therefore needed.

3.1 About the Paradigm and its implications

CODE method expressly addresses the *quality of the system development*, that has direct implications in testing, maintenance and for further modifications. Considering only the modifications issue, one can observe that there is enormous spending of time for any little change, since the comprehension of software and its hardware dependencies is difficult. In complex EDS

the problem is more obvious, as they are intensively interactive and it is hard to define the dynamic logic of interactions. One of the main issues is the recognized gap between design and implementation, that lets the code to be the ultimate document. CODE tries to fill this gap not only enhancing the documentation, but through a specific approach in implementation.

CODE is intended to be an *integrated development environment* [Krieger3] with:

- *A Methodology, as a set of unifying principles needed to define a system,*
- *A System Architecture corresponding to the behavioral view of the system.*
- *A matching Development Process*

The *methodology* of CODE describes a system from a behavioral point of view. [Krieger2] Originally CODE referred to usual event driven systems, but most interactive system can be seen as driven by inputs, and considered as event driven. The requirement specifications of EDS are *inherently behavioral*; they are focused on the interactions between system and the environment. Therefore, the EDS can be specified at different levels of abstraction: at highest level there are specifications of the responses to external events. These can be repeatedly partitioned, until the smallest components of the response are reached, the *activities*. An *activity* is a sequence of operations that can be executed completely without needing to wait for additional data or resources. In other words, a response is a set of activities that are executed in a given precedence relation to provide the required outcome to an external event.

The Event Driven Systems interact with the environment receiving requests or events and perform related actions to provide for each event the expected response.(see Figure 3.1)

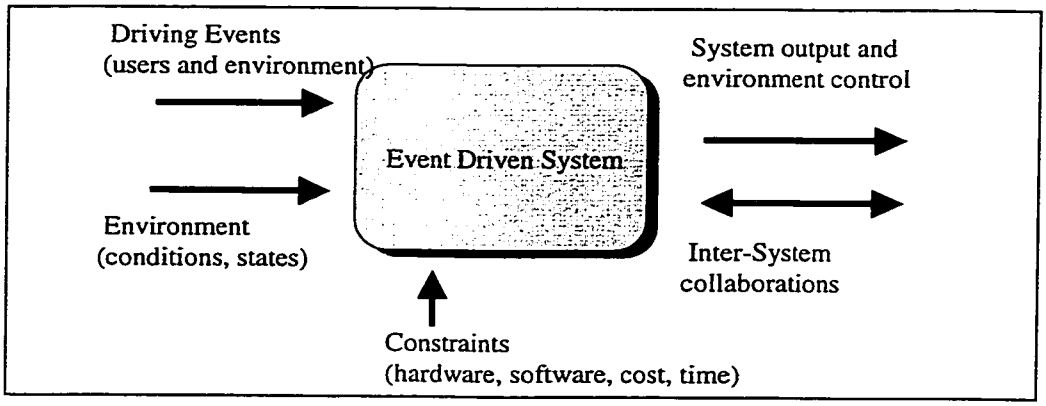


Figure 3.1 Event Driven Systems

In *transformational* data processing systems the environment is quite static. In these systems the data is the variable element.(Figure 3.2)

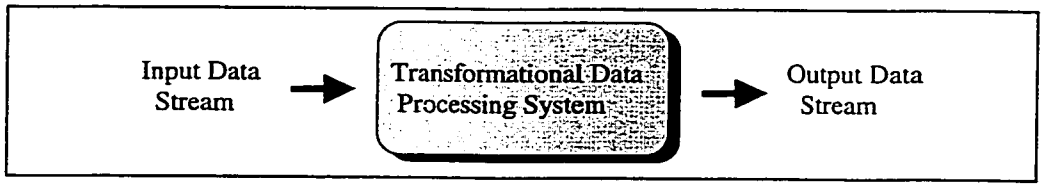


Figure 3.2 Transformational data processing systems

A major problem at design time is usually the *partition* of work to be done by the system, defining the appropriate units of work and their precedence relation. Such partition may be based on different criteria. The type of the system is important as it gives different focus for this partition. In EDS a proper partition is especially important because there may be a number of responses that need to interact. The critical resources have to be identified from early analysis ; than the partitioning of the work has to be done based on the way of accessing, managing, these resources.Considering the work as having two components, the activities that have to be executed and their coordination, one separates 'what to do' of 'when to do it'.

In Coordination based design a system is partitioned (Figure 3.3) in two major domains:

- **The coordination level,**
  which defines the sequence of activities that have to be executed as a response to an event,
- **The execution level,**
  which specifies the actions that have to be carried out in an activity.

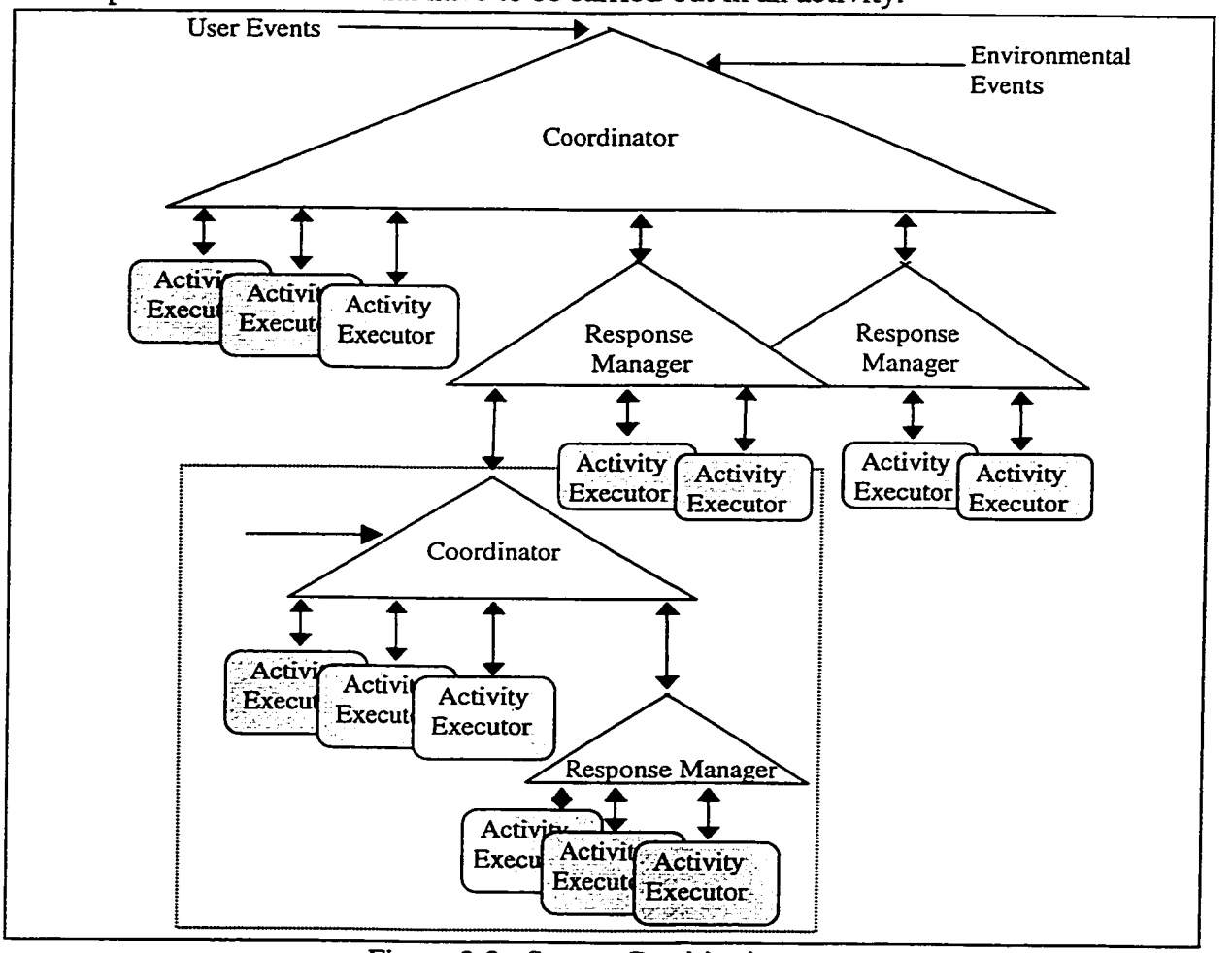


Figure 3.3 System Partitioning

An event driven system may be realized with one or more coordinators and their respective activity executors. The concept may be used with several layers. In this style, a computer system is seen as a group of hardware & software resources that have to be used in a certain precedence relation, to form the designed Response specific to each Event.

The architecture that corresponds to this methodology defines the system as one or more coordinators and a number of execution units. The coordinators define the sequence in which the execution units have to be called as a response to an external event.
The lower level coordinator does not receive driving events, but only environmental conditions.
The Coordinator is more like a manager, who controls when and what activities must be done, their order, priority, changing it eventually, but who does not do himself the jobs.
Response Managers deal with one specific event. In different states this event may have different responses. Each response is defined as a different sequence of activities.
The boundary of a Coordination Based System is the upper layer where it *collaborates* with an other entity. This is the limit where a new Coordinator cannot be added because of some physical limitation.

For the definition of coordinators, a design tool was developed. It is named Process Activity Language (PAL) and is a precedence graph representing the sequence of activities related to an event [Krieger2]. For the application of this thesis, behavior modeling in Object Oriented Design, our tool of choice is the finite state machine, as being typical in OOD.

3.1.1 Qualities of Coordination Based Systems

The Systems built this way are highly modular -an important expectation of software engineering- with good decoupling, encapsulation, separation of concerns, abstraction.

- The distinction of coordinators from executors clarifies and reduces the *development* effort:
  - *requirements* are defined independently for each one,
  - the *design of executors* is protected from the randomness of events, while in the designs without coordination the logic of requests must be spread and solved in executors too.
  - the *coordinator captures* all *re-publicae*, public things as the internal and external relationships, constraints logic, etc.
  - *implementation* is straight forward for executors, with less interface and logic while coordinators reflect the main requirements.
- *Testing* of coordinators relates directly to the design and back to requirements since the full logic of event-response activity is in a place.
  In the other systems the dispersion of this logic in each active entity makes testing harder.
  Testing Activity Executors should be a minimal task, as the input and logic are minimal.

- The CODE defines more *traceable* systems, as they are built with a Behavior Specification and most of the behavior is concentrated in Coordinators.
  Modifications, version control are better done for traceable designs.

- The use of '*component software*' and 'component off the shelf ', COTS, is easier considering the idea of a Coordinator that relates them.

- A *fault-tolerant* system will be also simpler to build, if one adds at the interface between coordinator and executors:

1. A test for acceptable values
2. Time-out routines
3. Maybe use the concept of 'imprecise computation' , to give a shorter response,

- The *instrumentation* for critical check points is cleaner in a Coordination System, again because the logic is concentrated. History may be logged, for trend analysis or testing.

The Development Process of CODE defines the evolution from a problem oriented model (the expectations/requirements) through solution oriented models to implementation. It has the following phases:

1.Architectural phase, capturing the problem to solve, with following steps:

- requirement acquisition

  In this phase the developer finds out from users
  - 'primary' events
  - a rough behavioral description of responses to each event
  - relationships, dependency between them
  - constrains of quality, performance, interface.

  With this data, the developer issues a requirement specification document based on event responses, as a functional diagram with coordinators and executors.
- architectural design

  in which the modules are finalized as it appears at requirements level, based on different coordinators and executors.

2.Design phase, capturing internal details of the system, not visible to the user, but necessary for implementation.

Logical modularity, executors with links only towards the coordinator that defines relationships, help for a good reflection of the requirements. Traceability is very high as the executors do not interact in any direction, all to all.

3.Implementation phase, where the design is cast in the code. This phase takes also advantage of the condensed logic in coordinators. The programmer can directly apply the directed graphs, PAL, or the state machines from the design phase and turn them into code.

4.Deployment phase deals with integration and testing. The interfaces being well defined in an event-driven manner, both external and internal, with a localized logic are a great advantage. The documentation can be built directly from the design.

5.Maintenance and upgrade phase. System instrumentation can be accurately built-in as the coordinators contain the main logic. For the same reason, faulty modules can be easily replaced or upgraded. The extension of functionality is also easier: one has to adapt the coordinator and to add the new executor, than test in limited area.

3.1.2 Ideas meaningful for CODE from some non Object Oriented Methodologies

We looked at few examples to underline some ideas, important to CODE.

Jackson System Development, JSD [Jackson], based on Jackson Structured Programming, JSP,[Budgen] emphases the hierarchical operational design, as a follow-up of the Dijkstra and Hoare solid intervention with Structured Programming [Dijkstra]. The key word '*structured*' is a major support for CODE, for its hierarchical approach at system level. In JSD there is no trend to define any kind of coordination and the 'data flow' drives the design.

We consider in a similar way the Structured System Analysis, SSA [Budgen], developed by L.Constantine and Ed Yourdon. We want to emphasize this because in Object Oriented style the hierarchical structure is required only at the static level, through class inheritance. On the operational side, the objects are allowed to communicate all-to-all, in a 'GOTO' manner as in un-structured programming.

CODE advocates a hierarchically structured behavior, where the activities are executed in a given precedence relation, defined in a coordinator.

In the 'functional' System Development MASCOT [Budgen] the Activities Processes are the main elements and Intercommunication Data Areas, IDA, are the fields through which Activities cooperate. The IDAs are made by ports, buffers and alike. We retain that IDAs and Activities are considered separately, but the *logic of interaction* is however dispersed in Activities. In CODE the coordination is implemented distinctly from activities.

3.1.3 CODE Relative to Object Oriented Design

CODE was built and can be applied in a *non Object* Oriented manner, being a generic engineering approach for system design. Also CODE may be applied *with Objects*, which can be used to encapsulate coordinators and executors.

In this sense there is an extension of CODE, named *Restricted Object Based Design, ROBD* [Krieger4] which suggests that Object Oriented designers can take advantage of CODE to *organize* objects. An Operational Structure, other than the class structure, would result by adding some Coordinator Objects to the usual Problem Domain Objects. Both CODE and ROBD are preambles for the ideas introduced further in this thesis. Thus ROBD is a structural approach while the approach in the thesis will be a behavioral one.

3.2 Related Coordination Studies

In most studies 'Coordination' is considered more as a passive or semi-passive media for the integration of *separate units* that facilitates cooperation and synchronization. Distinctively, in CODE the term Coordinator refers to an active decision entity enabled to 'command' executors.

A generic research paper, not specific for software, is "The Interdisciplinary Study of Coordination" [Malone], work that is referred in many other studies. We address here the driving thought of the paper, the definition of coordination : "Coordination is managing dependencies between activities". We consider that in a definition of coordination the *common goals* -for which activities are co-ordinated- should be more important. The *dependencies* should be the secondary part, since dependencies are set up *for a reason*. In the Appendix of the paper we find an older definition of Malone where he only addresses the goals. In this definition, coordination is "The additional information-processing performed when multiple, connected actors pursue goals that a single actor pursuing the same goal would not perform." His study is not focused on this definition.

CODE is a *goal oriented coordination*, since a coordinator defines explicitly the goals of the system (the expected responses for each request).

The coordination research for *the area between systems*, which is not the domain of CODE, is reflected in a vast amount of studies, conferences, coordination languages and building concepts. The 'coordination' is seen, as already mentioned, more as a passive or semi-passive media for cooperation, synchronization.

Our only intention here is to raise awareness of a the intense debate on Coordination between systems and to suggest that Coordination between objects should not be underevaluated.

A major conference was "Third Int. Conference on Coordination Models and Languages" named "COORDINATION '99" with web address www.cs.unibo.it/~coord99/.

In this conference, Coordination Models and Languages [Papadopoulos] is a critical survey with a presentation of many models and languages. All are referring to distributed systems, multi-platform, multi-language types of applications. As expected for such systems, Coordination does *not* become in any case a *self-standing* entity.This paper makes an important distinction between Data Driven Coordination and Control Driven Coordination; CODE would fit in the last one.

A 'classic' paper is "Coordination Languages and their Significance" by D.Gelernter, N. Carriero [Carriero1] , where "Linda" coordination language was introduced. We note their main claim, which fits to CODE view: "It is possible and desirable to treat *coordination as orthogonal to computation* for purposes of building programs." Their solution for orthogonality is a Data Driven, quasi *passive* blackboard for data exchange, supported with a 'coordination language'. CODE brings an *active* Control Driven coordination.

A Ph.D. student of Malone, C. N. Dellarocas [Del], in his thesis "A Coordination Perspective on Software Architecture" details the analysis based the same thought : "Coordination is managing dependencies". The concept of goal remains secondary.He does a valuable exhaustive study of generic dependencies (goal independent). His important decision, matching CODE style, is that a *run-time coordination process* should *implement the dependencies management.*

Our comment is that *each application may need specific Coordinators. They should be goal oriented*, based on event-responses, in the way intended by the system inventor for *that* case.

We consider that as the *interacting systems* may need a sort of coordination, also the definition of *interacting objects* may be enhanced by addressing their coordination.This is more necessary when objects and interactions are sufficiently complicated to make the design task nontrivial.

As mentioned in previous chapter, in 'A Use Case Approach' [Jacobson] is made an important differentiation between two types of interaction diagrams : (1) a centralized type, which is named 'fork diagram' and a decentralized type, which is named 'stair diagram'. In the centralized one, a distinct object controls the others while between these there is no other interaction. Because this concept is opening the field for our proposal, it is expanded in paragraph 4.2.2 of 'Dialog Coordination' chapter.

An other coordination approach is observed in the 'Mediator' pattern [Patterns] where a mediation object controls or 'coordinates the interactions' of a few objects. The behavior is represented through an interaction diagram similar with the 'fork diagram' of Jacobson. It is done an analysis of cases where the 'mediator' pattern would be preferred. Again, we discus more about this approach when we introduce the dialog coordination in next chapter.

A sample where the concept is addressed for OOD is the part "Coordinated Behavior" from "Obstacles in Object Oriented Software Development" [Bergmans]. The author criticizes the Object interactions as being mainly based on *message send* semantics, from an object to an other one (Figure 3.4).

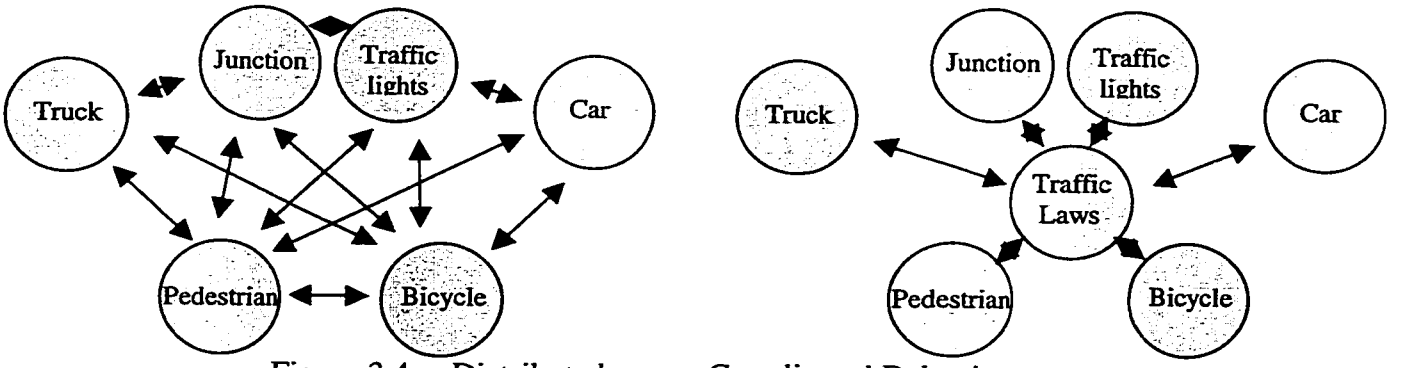


Figure 3.4 Distributed versus Coordinated Behavior

According to him (and we agree), this interaction model is too low-level because it only specifies communication *between two partners at a time* and this cannot be easily changed. A special class is needed to realize the 'coordinated behavior' reflecting the multi-collaboration.

In the left side, all objects communicate with all, behavior coordination is distributed. The other side has the TrafficLaws object abstracting the rules of interaction. It is a clear, justified demand. We comment that the collaboration laws are only the *boundaries* of behavior, showing the limits that must *not* be touched. With these laws no one is told what to do, where to go, but only what *not* to do. This is a goal-less coordination. CODE is different by capturing the *intended behavior itself* in Coordinators, through a complete definition of the event-responses.

In this context it is worth mentioning the Model-View-Controller (MVC) design approach, considered an architectural pattern in 'A System of Patterns' [Buschmann]. MVC was initially developed for the Smalltalk language (ex.: [Lalonde]), as a way of relating a document -the Model- with any format of View in user interface -by the means of a Controller. *Each kind of View has a corresponding of Controller* that facilitates the representation of same Model. Every needed Controller is registered with the Model so that when an update to the Model is done, each registered Controller updates its View. The specific Controller undertakes the GUI events, translates them in internal events, updates the Model or responds to user.

We learn from this approach that by using the Controller, a good decoupling occurs between the Model and the View. The interface logic has its own place, a distinct one, in the system and the designer can better clarify his intentions. The latter modifications in one part can be done with a minimal impact for the others. What we would like to see though is the way the responses and the behavior should be captured. The interaction diagrams which represent the messages between the three parts can only show a few message exchanges. For the systems we deal with, our quest is for a complete response design, starting with groups of objects.

The research on Coordination has now the trend to descend from multi-systems level to multi-processor applications and further down to multi-processes cases. Next natural step would be multi-object systems. CODE started viceversa, from simple embedded systems making its way upwards. We think that as we speak, a hand-shaking of both directions is on the way.

3.3 Existing CODE Applications

In our team from Ottawa University, the main research topic is the Coordination Oriented Design and its applications. There are comparative studies, enhancements and adaptations, as the following:

- Multiactivity as a Restricted Object Based Design Approach [Lemire]

In this work, 'Multiactivity' refers to the Multiactivity Paradigm, the initial approach for coordinator design. As mentioned, this was introduced in 'The Multiactivity Paradigm : an approach for the design of embedded systems by application specialists' [Krieger2].
The principal intent in the Restricted Object Based Design (ROBD) is to provide an *architecture* where the "*execution units*" are wrapped in restricted objects. These cannot communicate to each other but respond to calls from a manager object .This is a *structure driven design*, where objects are organized based on ROBD.

We propose in the present thesis a *behavior (event* response*) driven design*, where inside a Responsive Unit each response is completely designed and explicitly captured. The sequence of events and responses is also completely defined. The same is hoped between Responsive Units.

- Another direction of applying CODE was the domain of Component-Based Software Design. In 'Coordination-Based Design; Towards Component-Based Software Design' [Yang] it is presented the advantage of using a distinct coordinator component that will mange the requests to a group of components. It will provide for them the logic of interactions, which do not belong specifically to any one. The components will not need to know each other, as they receive action requests only from coordinator.

- The domain of Components Of The Shelf (COTS) is a specialization of the previous case in which the components are usual software packages. This case is studied in 'Coordination-Based Design for Tailorable Business Software : a Computerized Maintenance Management System Example' [Cox]. Complex applications where the parts are software-off-the-shelf can be related through a distinct coordinator that receives requests from an input and calls for execution in the respective packages. A visible major advantage comes from avoiding the direct connection between packages and using them only as 'executors'.

- In 'Coordination Oriented Architecture for Large Software Systems' [Coifan] several software architectures are analyzed and it is presented the need and the meaning of a 'coordination architecture'. For the 'design in the large' of some systems the organization around a coordinator may offer a structured design and the ability of easy change or evolution.

- An adaptation of COD for the *behavior modeling in OOD* is presented in this work . In actual methodologies the interactions between objects are partially defined. Our approach aims for the complete design of event responses, at least in groups of objects and eventually in the system. For this goal we adapt the CODE main concept -of using a coordinator separate from executors- and use it in the behavior modeling of object oriented system.

3.4 Synopsis

Coordination Oriented Development Environment is an engineering approach to systems design based on the *distinction between the concepts of coordination and execution.* In CODE Coordination is not only an abstract concept but it is implemented in an entity. This entity, the Coordinator, is an active manager that commands the Executors. CODE is a behavior driven development, where requests and responses are considered the primordial issue.

Some of the qualities of system designed with CODE are the following:

- There is a good relationship between requirements, design, implementation.
- Testing is related to requirements and is modular.
- The traceability is enhanced, as the main interaction logic is in Coordinators.
- Fault-tolerant design can be easier.
- Instrumentation can be done handy inside Coordinators.
- The system is easy to modify and upgrade.

CODE is intended to be an *integrated development environment* with clear phases. It was developed for event driven systems, but the approach may be applied for general system design.

Other coordination research is done mostly for the domain of multi systems applications, where the concept remains abstract, dispersed throughout systems and there is no trend to distinctly implement it. In this domain there are intensive debates about Coordination and the limits of the debates are pushed down towards multi-processes systems; moreover there is the suggestion to *implement* Coordination Processes. We wish to see the research reaching still a lower level, at 'multi-objects' systems; we noted a few ideas in the previous chapter.
Some distinctions on Coordination styles are the following:
From *localization* point of view, Coordination may be a

- *separate* component or
- *dispersed* throughout all components.

If *dispersed*, Coordination may be

- *implicit* or
- *explicitly* implemented in all components.

Concerning *dynamics*, it may be

- *active* or
- *passive* (as look-up data, separate or dispersed; synchronization is required)

From *role* point of view, Coordination can be

- *goal oriented* (for components which work together for a group goal)
- *without goal* (only 'protective', so that participants should not disturb each other)

The design of Coordination can be driven by:

- a *structural* demand or by
- the *behavior* intended for the system

In CODE coordination is: *separate* (thus *explicit*), *active, goal oriented* and *behavior driven.*
CODE came from the direction of embedded system, with ROBD touching the Objects domain. This thesis is concerned with applying CODE in the Behavior Modeling of Object Oriented Systems, domain where the ingeniosity of engineers is highly required since the logic of interactions is only partially defined.

Chapter 4

Introducing Dialog Coordination with its Subtypes:

Master-Slave, Team and Agreement Coordination

Introduction

This research sets from the beginning the focus on the dialog between components (fig. 4.1). At proper time notes were added about the *importance of the **dialog***, observing how it is treated and making some suggestions. Also, it was underlined that *wherever a dialog is in place, implicitly some sort of **coordination** is at work.*

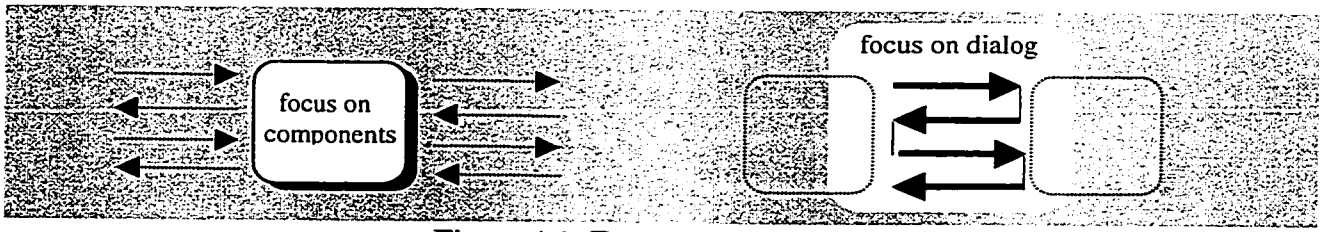


Figure 4.1 Focus on Dialog

In this work the term 'dialog' is used to express the logic of interactions between the components of a system. The term *'dialog'* is formed by two other Greek terms: *'dia'* that means *'through'* and *'logos'* that means *'logical sequence of words'* (mostly) or *'word'*. Thus, the term 'dialog' points to the *use of logical sequence of words (or logical sequence of all interactions).*
The problem that we want to address is that the behavior modeling of multi-objects systems is done in terms of partial responses, since interaction diagrams define partially the interactions. We believe that one *can improve the behavior modeling by firmly addressing the dialog and its coordination ; for this aim we consider that the software itself needs improvements* not only the documentation.

In this chapter we (1) develop the concepts that the *dialog is essential and implies coordination*, (2) consolidate the concept of *Dialog Coordinator as a distinct object*, explaining its role and (3) step in details, by discerning *three types of Dialog Coordination.*

4.1 Concepts

4.1.1 The Dialog is Essential - Being the Interaction Logic Between Components

With the focus on dialog, we studied the behavior modeling, addressing the state machines and the way Object Oriented methodologies consider the behavior. On the way, there have been made comments about the 'dialog', as well as for other issues. We extract here some representative ideas that should support the observation that 'dialog is essential'. Such basic observation is actually self sufficient, should not need an endorsement. However, usually the balance of attention is so much set on components, (on the touchable, visible, state, structure, objects) that it seems good to revise the importance of the dialog that relates them.

Starting with the behavioral definition of a system, we saw that the *sequence of inputs* is decisive. We underline that the 'user system' receives back a *sequence of inputs* as part of response and the *interleaved sequences in both directions* form *the dialog* intended by designer. In a similar way, state machines are constructed from the *sets of events* found in requirements. With the same nuance, we note that the mandatory 'user state machine' receives back a *set of events* and this *dialog* represents the intentions of the designer. Thus, an other observation is implied: at system level or at object state machine level, *the dialog, as intended at design time, shapes two sides.* The sides do not make sense individually ; together and with the dialog they form a higher level component.

The Object Oriented Development is by definition also a Dialog Oriented Development since the system is built through the dialog between objects. After structured design defeated the 'GOTO' statement, bringing a centralized view of systems (where dialog is minimal), the OOD camp won back localized responsibility. The dialog between objects was initially seen as a by-product [Constantine1], but *localized responsibility implies dialog for system responsibility.*
Observing how StateCharts deals with the interaction logic, we must mainly discuss the AND decomposition in orthogonal sub-machines. A dialog may be expected here, but these sub-machines do not communicate since the events come from outside without an address (are broadcasted). Therefore in UML the StateCharts model is *used only at the object level*, letting the dialog between objects to be defined with interaction diagrams. *According to the Use Case approach adopted in UML, the interactions need to be organized early in the design.* This trend in development strongly underlines the importance of interactions and this thesis pushes further in this direction by proposing a *complete* design of interaction logic *for each small group of objects.*

The dialog in SDL is clearly essential, as the independent state machines have to communicate. To minimize the complexity of an all-to-all dialog, SDL groups active objects in blocks according to the *layers of importance of the dialog.*

We underlined here that the *dialog is essential*, in the sense that a system is built through dialog & objects. However, we think that *essential* means more : *even the objects are built through dialog* . With this, we could say 'the dialog is primordial'. More on this last meaning in the subchapter

4.2.1 'Objects and Systems are Built Through Dialog'.

4.1.2 Dialog Implies Coordination

We consider the statement *'dialog implies coordination'* in the sense that a dialog only takes place if there is a certain kind of coordination. The components need to take in consideration some interaction rules, which are internally embedded or found in a public place.
There is also an observation in the reverse direction: it makes sense to speak of coordination only where there is a dialog. Logically, taking this proposition together with 'dialog implies coordination' one may even suggest that the concept of dialog is very tight to the concept of coordination.

The dialog coordination may have its logic *dispersed amongst objects* (implicitly defined or in distinct statements), or may be *implemented as a separate object.* Also, it may be active or passive.

In Object Oriented development the coordination is *usually dispersed* and rarely some part of it is *separately* defined. Between objects there are often complex relationships. During our study, we analyzed some proposals of dealing with the interactions between objects, each addressing a specific issue. We outline these ideas as they represent, in our opinion, a partial way of implementing *separately* an explicit dialog coordination.

At *Rumbaugh* we see a kind of separate coordination in the idea of a *'Control Object' [Rumbaugh].* This one cares about when or in which circumstances some actions have to be called ; it does not deal with the implementation of actions. This kind of object is not considered an application object but it is 'part of the language substrate'. It is seen as a support, not an expression of design and it is not essential to the methodology, as it is the dialog coordination that we suggest.

'The World in States' [Shlaer] considers the *'Relationships'* between objects as being dynamic and having a concrete state ; this can change during the 'lifecycle' (evolution) of relationships. This 'Relationship lifecycle' is implemented as a state machine in the body of an Assigner. This keeps track of the stage reached by the Relationship and assigns a new one, thus changing the relationships between objects. Here is a case of separate coordination that supervises the relationship ; we promote an object coordinating the interaction logic, the dialog itself.

'A Responsibility Driven Design' [Brook] defines layers of responsibility ; an object with higher responsibility explicitly coordinates objects with lower responsibility. This is a structural model only, as Wirfs-Brook does not develop a behavioral view with state machines.

The 'Use Case Driven Approach' [Jacobson], which is also the suggested way of system design in UML, directly addresses the interactions of the system with the user or between objects. In this methodology, the 'blocks' (one or more objects, even the system) may have several 'use cases' as required. These 'use cases' drive the design and the implementation of the interaction logic; usually the coordination of the dialog between blocks is *dispersed* in blocks. However, in some cases one of the usual objects may manage the others.

In Coordination Oriented Development the *interaction logic* that drives the executors to complete the event responses is compactly defined through a *public explicit coordination* . The method is aimed to be an improvement of system design.

Implicit or explicit, dispersed or in a public place, the coordination is implemented. If it is not explicit, by *perceiving* the dialog one can *think* however that *there is* coordination. We further look for a way of implementing the coordination that may improve the dialog definition.

4.2 Corollaries

4.2.1 Objects and Systems are Built Through Dialog

When we expressed the idea 'the dialog is essential', we said that it does not mean only 'cannot live without it' but much more: 'cannot *build* without it'. For the system level the idea is relatively obvious, since systems are built from objects and dialog, as was presented in 4.1.1. In system design, the dialog is reflected (partially though) in interaction diagrams.

In order to develop the part 'objects are built through dialog', we start with the observation that the event is an essential brick, a dynamic one. Inside an object, an event may be considered the name of a public method or a parameter brought by a general public method.[1]The event changes the state of the object ; it triggers a specific move (fig 4.2) We see it as a dynamic brick, and an essential one, as inside an object each single change is initiated by an event.

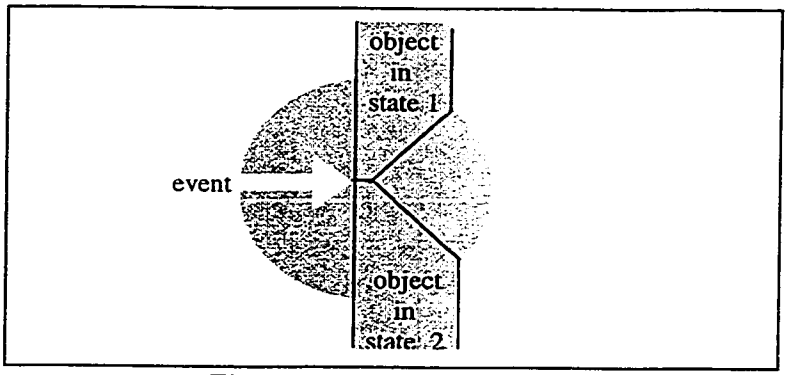


Figure 4.2 The Event

We would like if instead of 'event' we would use the name 'word-in' . It is not only reflecting an action at a specific time, but also has a unique name, based on which the system changes differently. The event in the opposite direction (a response message) could be named 'word-out'. Here we want to set the focus on 'event', when usually the focus is on 'state'. We do this because (1) we are interested in dynamics (not how an object stays) and (2) to enable the light on the following observation.
Typically objects are seen as the solid rocks of a system and the design is oriented on them. The interaction field, the dialog, is seen as an abstract element, which is described in a document.

[1] Since we addressed behavior intensive systems, here we refer to dynamic objects. The objects that are data oriented and less dynamic are considered here more like data structures than as dynamic objects.

However, shifting the focus on event (word-in), we can note that an event or *'word-in' for an object is a 'word-out'* from an other one and they have a *logical succession*. If we look only to one side we do not get the complete succession logic of words-in-out.

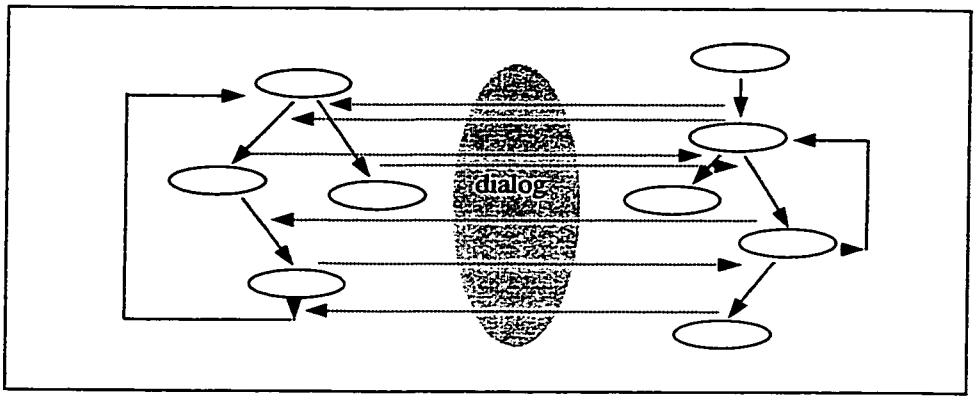


Figure NR:. 4.3 Dialog Shapes the Objects

At design time, the ***logical succession of 'words' or events*** *,* as intended by designer, *shapes the interacting objects.* (See figure 4.3) In other words, *objects are built through dialog.*

4.2.2 Dialog Coordination as an Object

The logical *succession* of coming and going messages, i.e., the dialog, has obviously a *meaning beyond the set* of individual messages. The messages are logically linked to each other and thus, this succession of messages is like a *compact entity*; it is self-standing and represents the intentions of designer about the future system.

How can this entity, that is the dialog, be defined?

- The designer describes it (completely or not) as a *compact entity* on some *pages* of document.
- In software usually it is *not a compact entity*, but the components take care of it, according to the usual mindset that components are substance and the interactions are an abstract layer.

We suggest to define the dialog as a *compact entity* also in software. The first advantage is that we can *comprehend* much better the logic of interactions if its *definition is in one module.* The comprehension of software is a major issue and we will see other reasons later.

Is it possible to define the dialog between components in a distinct object?
We just observed in the previous sub-chapter that the *objects are built through dialog.*
Without much effort, in software we can consider the dialog implemented as an object, (as presented at System Views in Ch. 2) that defines how the other objects interact. In building engineering the components are bricks and the interactions are ideas in a plan. The ideas cannot be captured in a brick . In software we can do it because both objects and interactions are built from logical sequences of words (or correlated words, as noted in Chapter 2).
Thus we *can* implement a dialog object, but a major question follows:

Why should we implement an object that defines the interactions logic?
The rationale we consider is *based on the coordination concept,* in the sense presented in the CODE chapter. Dialog implies coordination, as said, and coordination *can be goal-oriented* or *without defining group-goals.* Streets lights are an example of goal-less coordination, as they do not indicate to drivers where to go. A business manager would be a goal-oriented coordination, as he calls the workers in the right sequence to complete a business request. We discussed about the kind of coordination from CODE as being a *goal-oriented* one : the coordinator manages the executors to fulfill the system response; this includes error management.

The latest & most used OOD method, UML / Rational Rose, applies the use-case driven design observing the need to *organize interactions from the beginning* of design. Because only the main interactions are defined in diagrams, we go further looking for a detailed -eventually complete- interaction definition. To this end, we concentrate over a *group of objects* at a time and using a CODE type coordinator we attempt to localize and manage their interactions.

As said, we concentrate over a *group of objects and underline that each such group has a group-goal.* This goal represents all *group-level responses with their allowed relative sequences.* For this goal there are *goal-dependent values* that are not specific to any object from group, but are conceptually *public to the group* (not to the system). In the same way, there is some *logic specific to the goal,* again that should be public to that group. These *goal-logic and goal-values* may have a clear location -as a separate object- since they need to have a behavioral representation in design *and implementation. We build an object that contains the goal-logic and goal-values, for each group-request;* these will be public in the group. This object corresponds to the *object that defines the interaction logic* from our question, as objects interact for a goal. An object state can be explicitly defined.

The new object will be named Dialog Coordinator, as our intent is to organize the dialog for a group of objects.

A *specialized* type of coordination is used -as in CODE- rather than a *generic* type, in order to have a *complete* group-response design. Through 'generic type' we refer to coordination in OOD as presented (1)in 'A Use Case Approach' [Jacobson] or (2) in the 'Mediator' pattern [Patterns].

We wish to present and compare these coordination approaches in order to clarify our proposal.

1. Jacobson differentiates two kinds of structure for the interaction diagrams [Jacobson]: *centralized and decentralized.* He considers the difference 'fundamental'.

- An interaction diagram is *'centralized'* when, in a group of objects, there is a special one that 'controls the flow and operates on the other objects, and then it decides what to do.' This object contains the 'rules' for the necessary response at group level. In such a group, most of the behavior is found in the controlling object. The controller requires information from the other objects or commands them. The diagram is named *'fork diagram'*. (Figure 4.4 a)
- An interaction diagram is *'decentralized'* when 'each object only knows a few of the other objects and knows which object can help'. In this approach each object has separate tasks and uses others the complete a task. The type is named *'stair diagram'*. (Figure 4.4 b)

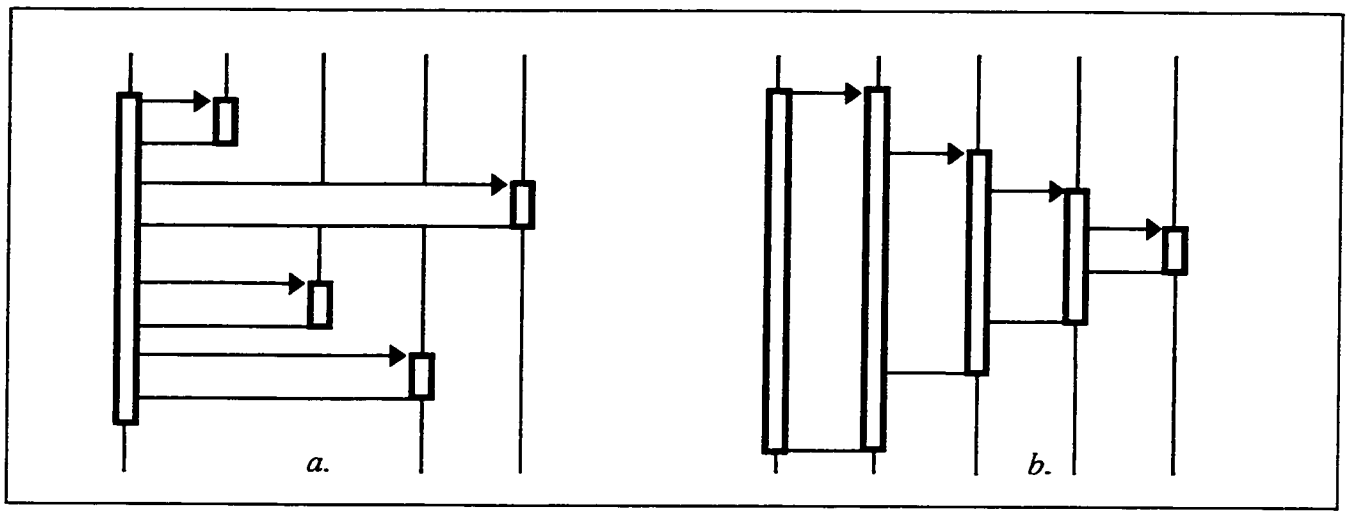


Figure 4.4 Fork Diagram (a) and Stair Diagram (b)

The *criteria for choosing* one type are based on the **'kinds of change'** that are expected. If we want to change *'the order of operations', a centralized structure* shall be chosen since 'changes will be local to one object'.
The 'principal rules' considered by Jacobson for this choice are:

- A centralized structure is preferred when
  - ⇒ the order of operations can change
  - ⇒ new operations may be often required
- A decentralized structure, when
  - ⇒ operation are done always in the same order
  - ⇒ they have a 'fix temporal relationship' or they form an information hierarchy.

In the systems we are addressing -event driven and intensive interactive- we observe that:
a). the order of events may change, as events are considered unpredictable.
b)'new operations may be often required' : the industries in which these systems are used (as telecommunication industry) require very often new features.

Thus, according to Jacobson in our kind of systems the centralized structure is preferable.
However, in our approach we will work with *groups* of centralized objects, not with *one* central controller per system.

2. In *'Design Patterns'* [Patterns], the OOD coordination is expressed by the 'Mediator' behavioral pattern. In this case, a central object is responsible for 'controlling and coordinating the interactions of a group of objects'. The objects do not relate to each other directly, but through the mediator. An interaction diagram shows how each object calls and receives calls only from the mediator.

Between this model and the fork diagram there is a delicate but interesting difference: the objects of the group can *call the mediator*, whereas in the fork diagram they are *only called* from the controller.

Again, as at Jacobson, we find the observation that 'it is difficult to change the system behavior' if the behavior is distributed. It is suggested to apply the pattern when :

- A group of objects have complex interactions, that however must be well defined.
- One needs customizable behavior (in our terms: system response design must change often)

Both these characteristics are present in the systems we address. *According to this approach, it makes sense to adopt a 'mediation' based design for our domain of interest.*

We consider that the previous approaches will increase the response design in our systems. However, both use interaction diagrams for behavior design. Each of these shows one execution path out of many possible, there are no decision points, no loops and therefore we may not get a full description of any event response for group.

Our initial quest was for a *complete* response design in a group of objects. The word 'complete' may be considered too demanding in a world of objects, where it is hard to completely design the interactions. But the word is not too demanding in an engineering world, where precision of response is a high request.

Therefore, we proceed for even a tighter approach -of CODE type- and we specialize and tailored it for our case - behavior modeling in OOD. In our proposal we concentrate over a group of related objects, a manageable number (5..10). As in CODE, we partition the responses in its smallest components, the *activities*. An activity is a sequence of operations that can be completely executed without waiting for supplementary data or resources. Thus, a response is a set of activities executed in a given precedence relation. This 'given precedence relation' , the coordination, is separated from 'activities' and defined in a different place.

- Each *activity* is implemented as a public method; this may call private methods that implement the operations of the activity. *Examples* of such activities may be computations, message changing or filtering, actions with hardware, data set / get / search / sort, user output.
- A *state machine coordinator* is built in a separate object, which calls the activity methods in the other objects. We use state machine as in OOD behavior modeling, while CODE uses sequence graphs to define responses. The *events at the group level* are treated based on the *state of the group*; thus, an event may get a *different response* in a different state. With this, the *order* in which events come is taken into consideration and precisely designed.

We see the coordinator as having two facets:

- The interactions with the *world outside the group*
- The *management of internal* activities.

Therefore we also distinguish in the *coordinator state machine* two areas :

- A part of FSM dealing with outside interactions : *the 'dialog machine'*,
- Several sub-machines, as the response managers from CODE: *'activity machines'*.
  These can advance by
  ⇒ repeating the event that provoked the entrance in sub-machine,
  ⇒ the return from an activity that provokes a sub-machine state change; as a result, a call for the next activity follows.

Another distinction may occur from layering the dialog machine itself based on :

- *major events*, for main requests to a group of objects
- *minor events*, for requests that logically may occur after a specific major event.

Two layers reduce the dialog complexity but we believe three layers complicates back a design.

The figure 4.5 shows the decomposition in sub-states (statecharts like) based on these distinctions.

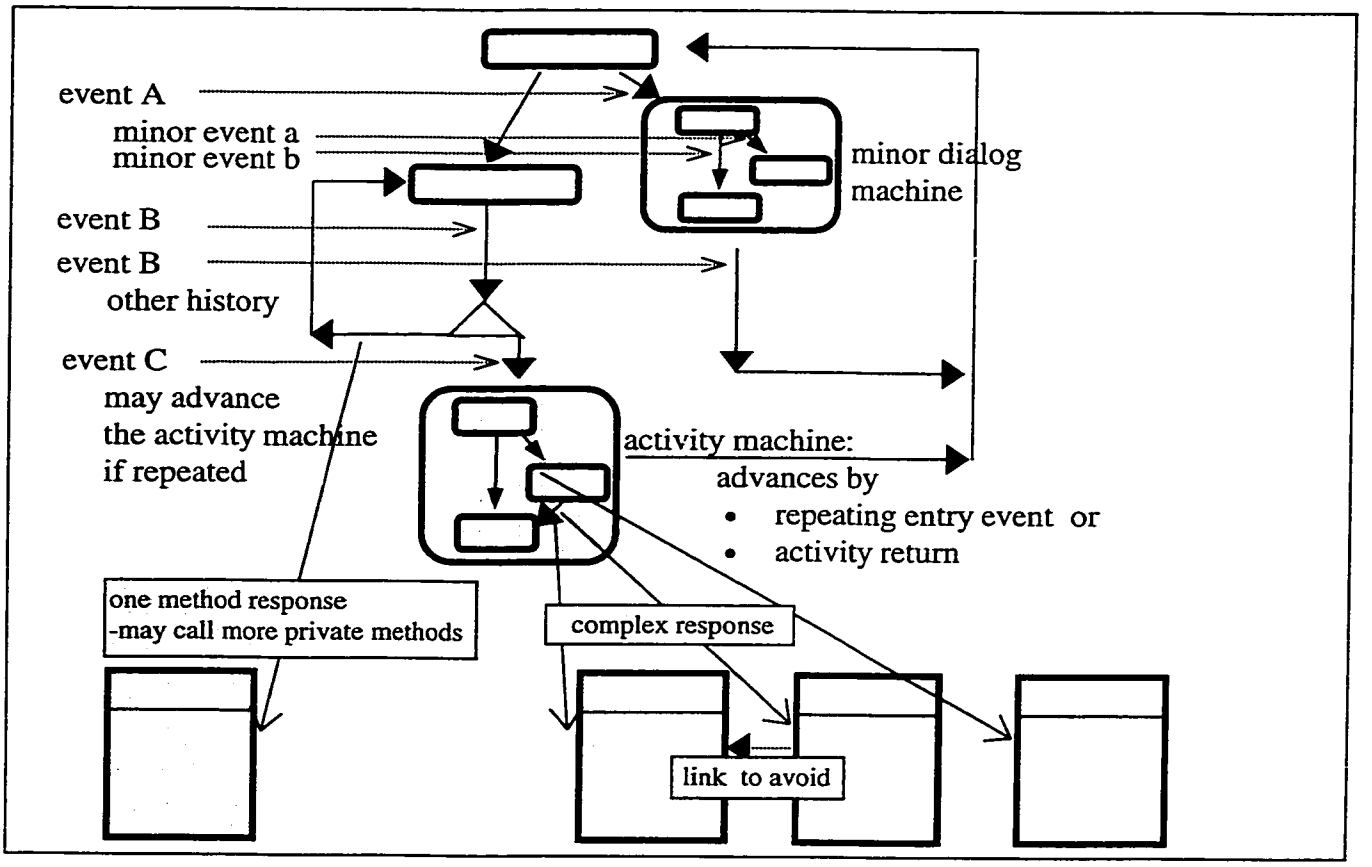


Figure 4.5 The dialog machine -major & minor- and an activity machine.

In this type of organization the behavior *document* defines the coordinator state machine and this one is *directly implemented* in the Dialog Coordinator. Thus, a group of objects with their coordinator can have a complete behavioral design and a unique corresponding implementation.

The *quantity* of objects, events and states must be not too big for a clear response design ; a designer should be able to capture and present them without complications. From this requirement it may result a larger interactions area between groups, as the choice is to keep the groups small enough to be definable. If the complexity of interactions between groups is high, we may look for a *next level of coordination.*

The Dialog Coordination concept is *scaleable beyond a group of objects* since it can be applied also between groups. Another coordination layer will lead, from the behavioral point of view, to a hierarchical state machine design. The paragraph 4.3, Dialog Coordination subtypes will expand on this issue.

After introducing our Dialog Coordination object with the given justifications, it makes sense to *observe some possible arguments* against its existence, as:

- *Maybe it brings an overhead* in messaging, speed, memory usage.
- *Conceptual issues against a 'centralized' view*, in a world of free objects.

For the first category, the overhead, we remember that the advance to high level languages from the assembly language had such opposition too. Also, at the introduction of the object oriented concept the argument was used against the new trend.Since then, the hardware engineers made progresses and the memory and speed are for sure not a major concern, with specific exceptions. For the sake of adding some order we can afford to consume some resources at object level. The facts speak actually in the favor of a higher software organization. In a complex system an *unorganized design is more bulky* and will consume more resources then an organized one. Also, a good behavior organization reduces the risks of discovering some unexpected behavioral overhead at testing time.

For the argument about *a 'centralized' system* we reiterate that the Dialog Coordinator reflects the goal-oriented logic of interactions. Such object is *not* a smarter, yet-an-other *problem domain* object. The Dialog Coordinator is *fundamentally dealing with the abstract, orthogonal issue of the dialog*. In a way it is like a theater-play script, which is an object but not a player-object.

A strong position -without a too strong argument- against coordination we found at Riel in 'Object Oriented Design Heuristics' [Riel]. He maintains that for a good 'action-oriented' design one needs to 'distribute system intelligence horizontally' (Heuristic 3.1) and to avoid objects like Driver or Manager (Heuristic 3.2). The main line of his motivation is that coordination leads to 'creation of many set and get data' and the behavior is not kept in one place with data. This may be true if we would deal with data processing systems, but in event driven systems there are many types of complex activities with complex behaviors that need to be managed. He does not comment on the ability (or inability) to fully define the interactions of a group of objects; also is not discussed how to deal with data & logic public to a group.

On the topic of freedom of objects, we can say that the Dialog Coordinator actually allows more independence to objects as it represents the designer's intentions concerning the dialog. The objects do not disturb or block each other as much as when the interaction rules are privatized in the Problem Domain Objects. If public rules are well defined in a public place, interactions are smoother. There is also an other kind of freedom. In this format the objects may not be called by any object anywhere in system (and when the web of interactions moves the object is hit). They are related in local groups instead of being the potential target of any relationship.

For a visual impact, we show in the figure 4.6 a design with very few requirements, responses and objects. The dialog is already complex to design, to understand or to expand as the goal-oriented logic of interactions has to be dispersed in the related objects. The responses are difficult to describe and also their relative influence is not obvious.

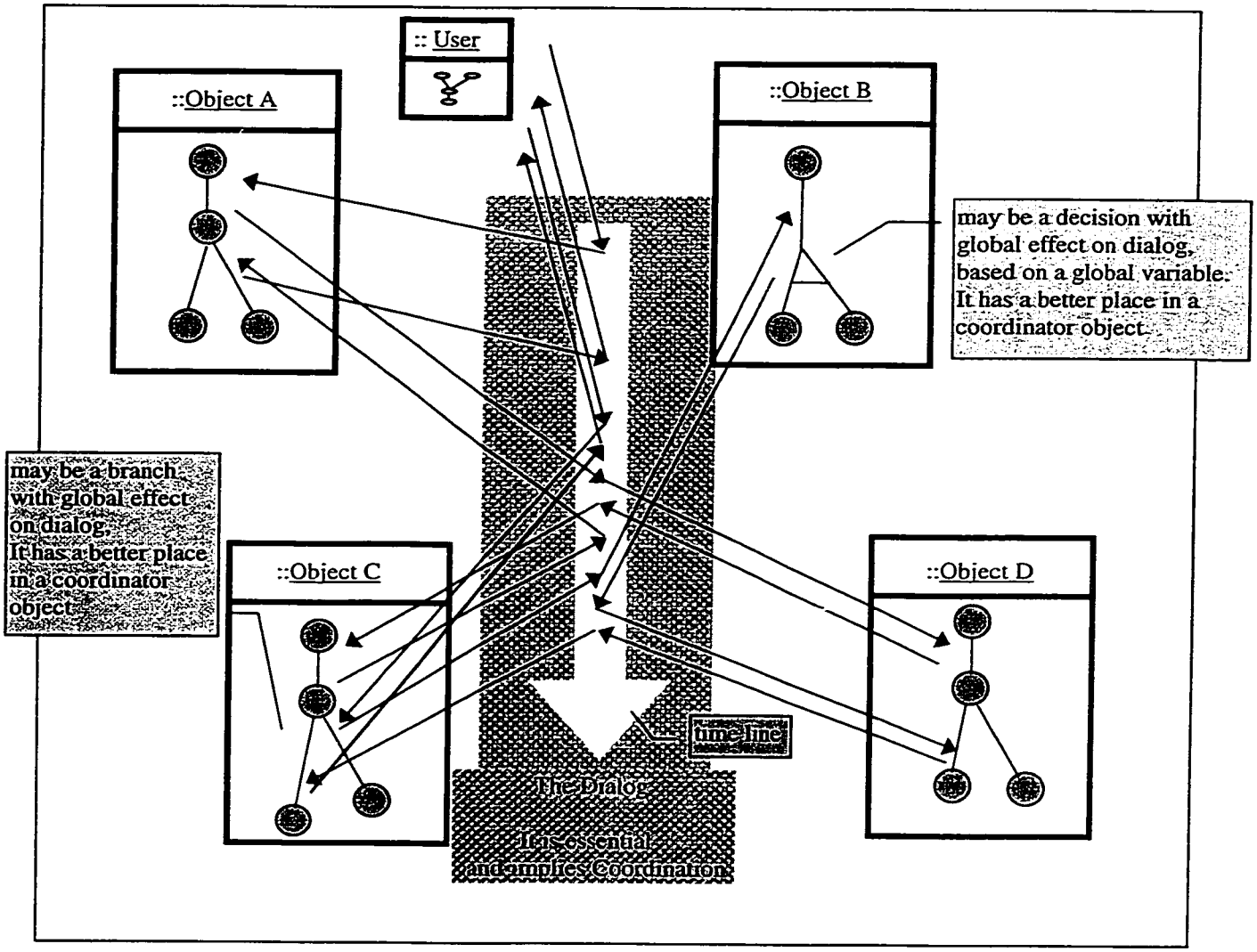


Figure 4.6 The Dialog for a Few Objects

There are notes in the drawing about decisions that belong to a group level, being theoretically public, but they have to be implemented in interacting objects. There is no public place in software to undertake the public logic or public values. Also, as the code is usually the only complete document, a new designer will hardly comprehend the interactions looking only at these objects. Trying to add a new object becomes (what is nicely called) a challenge. Statistics say and we know that much of designers' time is used to understand the previously written code. A similarly big problem is with testing : it is hard to cover all cases and bugs push the deadlines in an unpredictable way.

If developers did accept the Objects for their encapsulation quality, we could look for a next level of granularity: a Unit formed by a few objects related with a Dialog Coordinator object. The local logic will be clean and understandable.

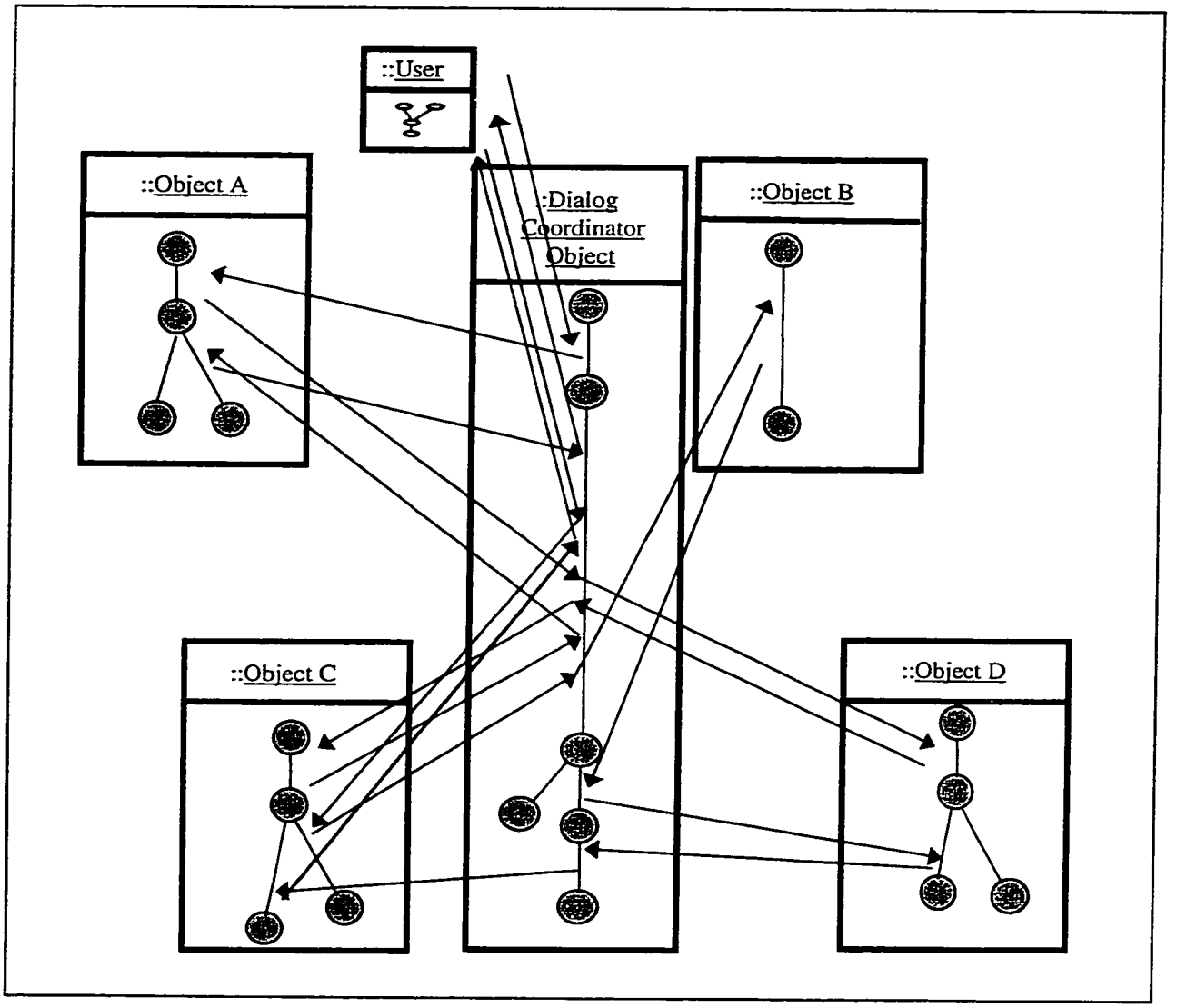


Figure 4.7 The Dialog Object for a Few Objects

Usually the dynamic logic of interactions is not visible, but is think-ible. In software we have the chance to write our thoughts in code and we should use the chance. Moreover, the existence of *public logic and public values requires a public object* in each group that shall undertake them. The group coordination is defined separately while the calls from coordinator to objects become calls for a specific activity, as in COD (Figure 4.7). A Dialog Coordinator may be seen as the dynamic script of what can happens in time with a dialog and thus the definition of the response to any request can be enhanced.

4.3 Dialog Coordination Subtypes: Master-Slave, Team, Agreement Coordination

The concept of Dialog Coordination is intended to cover any kind of dialog between objects. It is meaningful to distinguish some types of dialog for which the Dialog Coordination could specialize.

We consider that in a project the objects can be grouped according to the *kind of dialog* between them. We first distinguish the following *areas of dialog* :

a. *An area of strong interactions*, where a demand **must** be answered; inside this area, objects work together as a machine.

b. *An area of light interactions* where the dialog starts with a prologue as 'Please, **may** I ask you' and the epilogue does not necessary say 'Mission accomplished'.

The two areas, of light and strong interactions, appear in figure 4.8 where the shadowed surfaces cover the 'machinery' type of Objects' Groups.

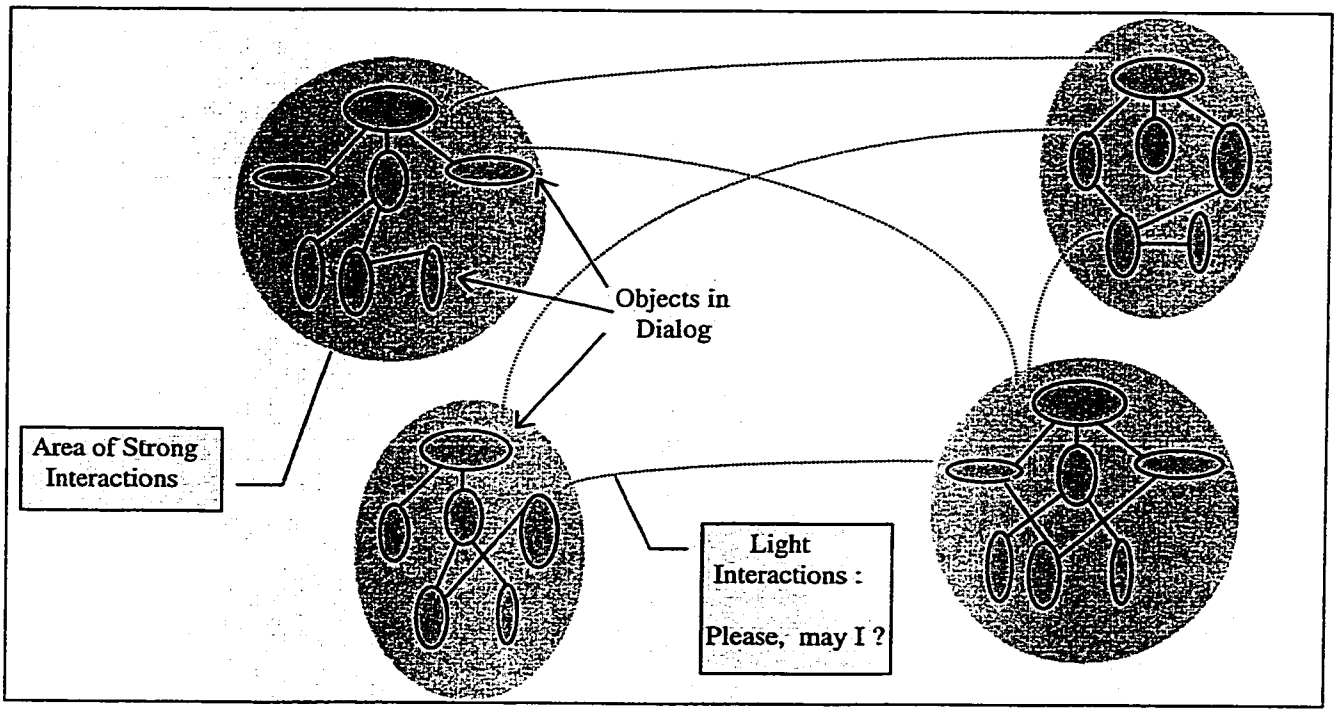


Figure 4.8 Light and Strong Interactions

1. Master-Slave Coordination

We introduce the first type of Dialog Coordination, the *Master-Slave type, as a coordination for the objects inside the areas of the strong interactions*. These objects **must** execute their part of the job, for the requests coming from outside the area. The boundary between the Master-Slave coordination and the next type, Team Coordination, is the limit between the *area of strong and the area of light interactions.*

2. Team Coordination

This Dialog Coordination covers an area of light interactions. More objects in strong interactions form a kind of unit, which is more responsive than the objects apart. Between these **units with more responsibility** the interactions are usually light ; the *Team Coordination relates several such units in teams.* A team will deal with certain higher level event-responses. The previous two types of Dialog Coordination are in *the same plane* working together directly to provide the right responses for the system requests.

We make a new distinction between the previous *plane* and another *plane* of dialog ; this dialog happens at system initialization for interaction agreements between teams and objects.

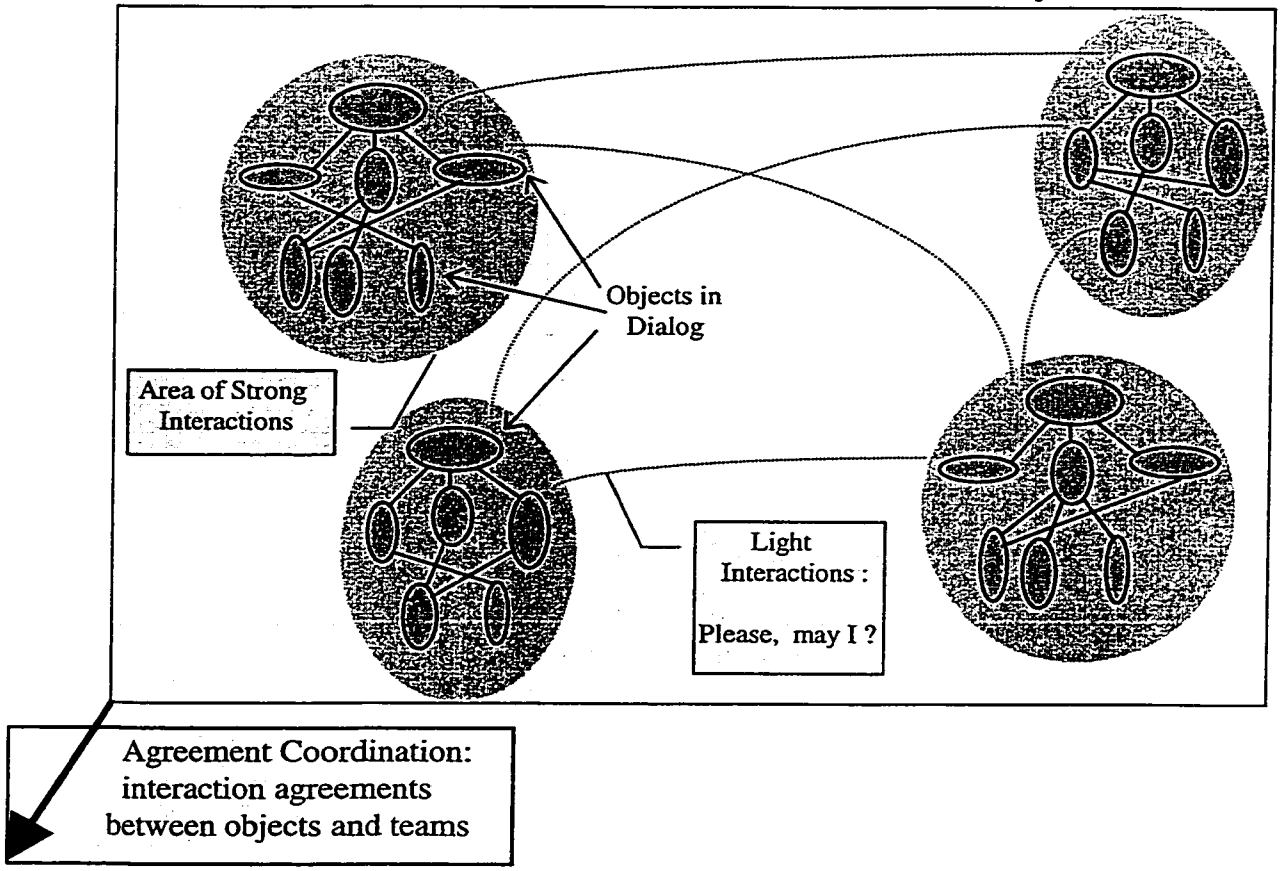


Figure 4.9 Agreement Coordination

3. Agreement Coordination

Because the Dialog Coordination is a generic concept covering different kinds of dialog, we may introduce the *Agreement Coordination* to cover the type of dialog just mentioned (Figure 4.9) At initialization time there is an exchange of messages between teams and objects for interaction agreements. This logic of interactions may have its specific dialog coordination. This may include systems *supervision, start-up, shut-down* and also agreements established at major system changes, in run-time not only at start-up. We underline that the *Agreement Coordinator* is in a *separate domain*, not in event responses domain, caring about the agreements between the

other entities. *Its behavior is NOT usually part of the definition of event-responses.* Exceptions are the management of major errors and the cases where objects are constructed and destroyed dynamically as part of response.

After introducing the three dialog coordination types, we have to make a comment concerning their necessity and sufficiency. The approach is a response to the author's experience with embedded event driven systems in which the interactions between (some tens of) objects are hard to capture, but there is a stringent necessity to do it.

In the response domain, driven by the need to capture all responses, we start from the bottom : we group few objects and coordinate them (and encapsulate them in Responsive Units, as presented further). Each group of objects is intended to have a clear and complete response design. A *Master-Slave Coordination type* is used, through the implementation of a state machine coordinator on top of 'activity' objects. Thus we build some mega-objects and a medium system can be covered with a relatively small number of those. In our kind of systems, we consider that for each group of related objects the *Master-Slave Coordination is necessary*, if a *complete* event response is desired for each group.

The interactions between groups (of objects) are much simpler than if we would deal openly with all objects ; still, if we use interaction diagrams between such groups, we may not get a total design for responses. If we intend to have a *complete system* response design, a *Team Coordination* may be necessary. A state machine will receive system events and, according to history, will call for actions. These are actually events towards groups. When it is mandatory to split the system in a number of teams, to reduce the complexity of Team Coordinators, a new coordination layer may be needed for Teams. One can *use the same type* of coordination for this case, the 'team coordination' type, since similar characteristics are expected. Therefore we consider that there is *no need* to add a new coordination *type* beyond teams.

The *Agreement Coordinator* is in a parallel world, not in the response domain. It takes away from the event response domain all the unrelated logic, as creation / deletion and major system issues. This logic may be complex and (1) is a system level issue that basically does not belong to 'response-oriented' objects, (2) complicates the response design when implemented disparate in objects. We consider that Agreement Coordination is needed to separate the design of the necessary sequences of event responses from the design of logic for set-up, shut-down and special cases. Thus, the author believes that the three types of coordination may be needed for a more visible response design and may be enough, as types, to cover an entire system. However, we reiterate that we address a certain class of systems, as presented; we hope though that these ideas may be useful in other domains too.

With the dialog coordination concept and with its three types we can proceed to our pragmatic suggestions about implementing them. The aim is towards no-nonsense and applicable solutions, that can be understood and used by developers and programmers.

Chapter 5

Responsive Units: Behavioral Elements

Based on Dialog Coordination

Introduction

This chapter completes the theory from the previous one with supplementary pragmatic ideas. The aim is to suggest a software environment that may enhance the behavior modeling. For this goal we organize the system software according to the Dialog Coordination concept and its sub-types. We do this since we believe that *focusing on dialog (interaction logic) and its coordination* is an interesting approach towards a *better interaction design.* The trend of UML to define interactions in the initial design stages is a good motivation for us to go even further: to look for a *complete design of interaction logic* for each group of objects.

In the previous chapter we presented generic justifications for our belief. In this chapter we first suggest that for behavior intensive systems the design decisions should be based on the behavior of system and of components. Then we propose the use of the Responsive Unit, which is an object composed of several predefined objects, having between them a coordinator object. The COD main concept -the distinction between coordination and execution- is applied. A state machine inside the coordinator explicitly defines the behavior of such a unit. More Units are organized in a team, around a Team Coordinator Unit. The system at large is maintained with an Agreement Coordinator that manages the relationship agreements between components.

5.1 Behavioral Design Driving the Structural View

As we discussed in previous chapters, the structural view and the behavioral view are representing independent views of a system. The first one is fundamentally static, the other deals with dynamics. Initially, in most Object Oriented methodologies the structural view was considered primordial, as objects were seen as primordial. However, in the later 'Use Case' methodology we see a *reconsideration of the behavioral view in OOD.* In this methodology, systems and internal blocks are built based on the way they should respond at requests in different 'use cases'. In other words the *design is based on how the system should behave* towards the users.

The Unified Modeling Language (UML) adopts this system design approach and with wish to continue in much more detail in this direction, towards a complete design of interaction logic. We considered also the Coordination Oriented Development Environment (CODE) paradigm, being the main research direction of our university team . CODE is fundamentally a behavioral design approach ; the system behavior is concentrated in coordinators, while executors must fulfill the required actions. The *structural view results from the behavioral separation of concerns.*

Our approach for behavior modeling in OOD follows this line of building the system structure based on behavioral considerations. We introduce the ***master-slave coordinator*** to *localize and manage the activities in a group of objects*; thus, in each such group we *completely* define the interaction logic between objects. Between groups, the ***team coordination*** is suggested for capturing the dialog at this level, with the hope / intent that all the interactions of the system may be completely designed. ***The agreement coordination*** undertakes all the logic for set-up, shut-down or major exceptions, clearing the other coordinators from such issues that are not on the event response path. This format is application independent, but each dialog coordination defines precisely the events and responses of one specific application. In this manner the initial design decisions are fast, backed by the dialog coordination theory and by the experience from other examples. It follows an elaboration of the approach, with development details.

5.2 Responsive Unit : an Object Composed of Coordinator, Activities, Data, Port Objects

In the previous chapter we noted that there are areas of strong and light interactions between objects. For the areas with strong interactions, the Dialog Coordinator that we consider is of the *master-slave* type ; the objects are more as organs in a body, they work very tight and they *must respond to requests.* We use the COD approach of making the *distinction between coordination and execution* and the coordinator is a *new object* in that sense. We build also a boundary object *corresponding to each area of strong interactions*, which will have in its composition *the coordinator* and the set of *strongly related objects.* This area-object makes the *boundary* between the *inner area of master-slave coordination* and the *outer area of light interactions.*

The o*bject composed by a coordinator object and the strong interacting objects will be named* ***Responsive Unit.*** This *may be seen as a higher level object* with enhanced abilities, when compared to usual objects. Therefore the calling mode for an inner object will always pass through the Responsive Unit object :

Object_Responsive_Unit_A -> Inner_Object

In this manner the names and pointers of internal objects are not accessible from anywhere at system level. An other advantage is at system set-up; when the new objects are built, a single instantiation of a Responsive Unit object will cascade internally in the instantiation of all objects. In general, building this object to encapsulate other objects has a similar impact as building an object that encapsulates data and methods.

The coordinator, which is the core of the Responsive Unit, has to manage several issues, like:

- The *dialog with other Responsive Units*,
- The logic of calling the *internal actions*, in order to build a complete response for any request coming to the Responsive Unit,
- The *internal, private data base,* to ensure that the right sequence of access is maintained,
- *The relationship between the previous three cases*, as they are related.

The figure 5.1 presents the areas of strong interactions, as considered in the previous chapter ; the *main object of the Responsive Unit exactly defines the boundary* of each area. This object has in composition the *objects with strong interactions and their coordinator*, as a distinct object.

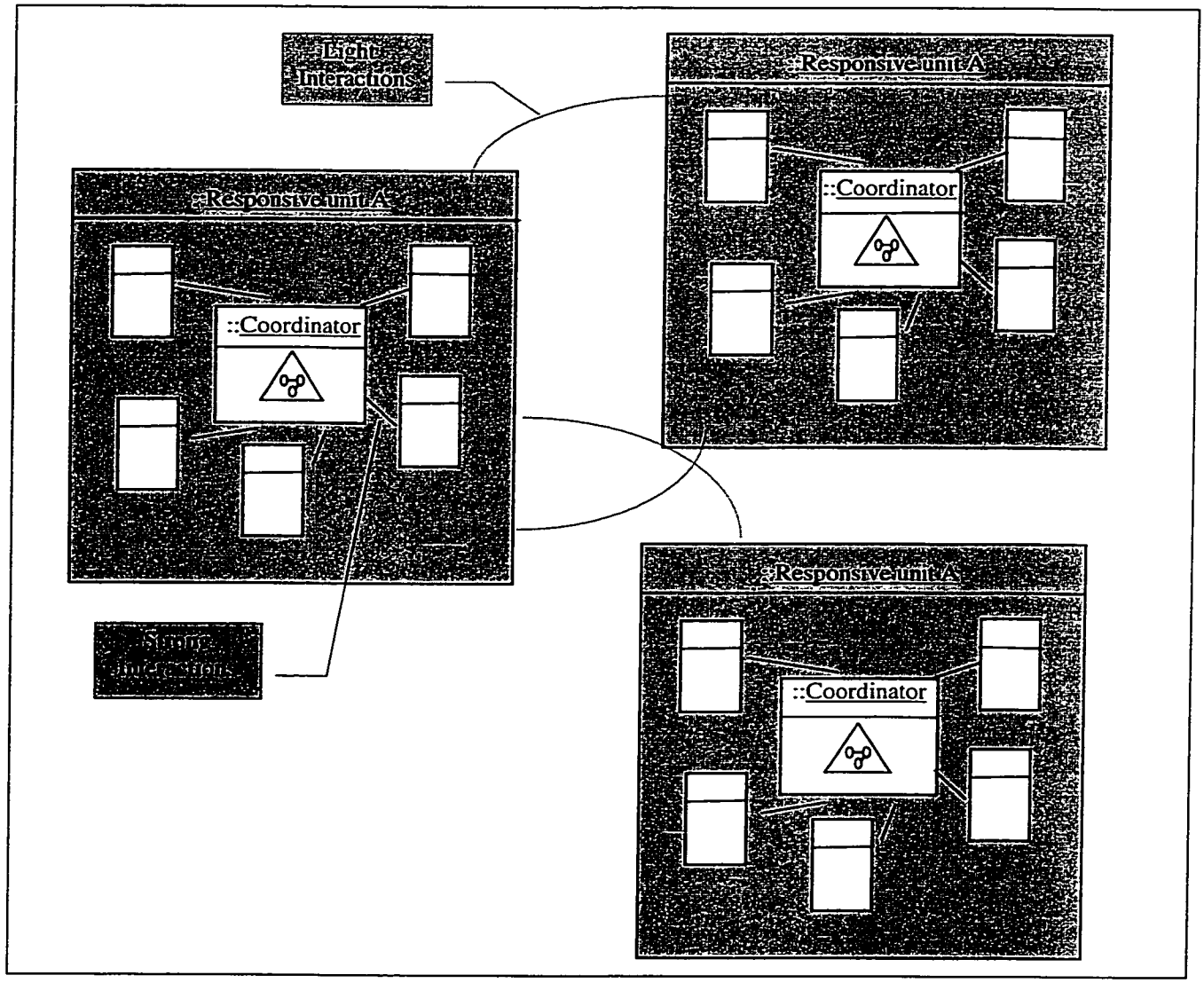


Figure 5.1 Responsive Units Around Strong Interactions

Responsive Units may be quite complex groups of objects and the interactions between units may take different complicated forms. Therefore each Responsive Unit may have a specialized **Port object** ; this can translate messages in the internal form, check and/or remove errors and even define a certain input logic. Such logic could be necessary when at the input a wide range of

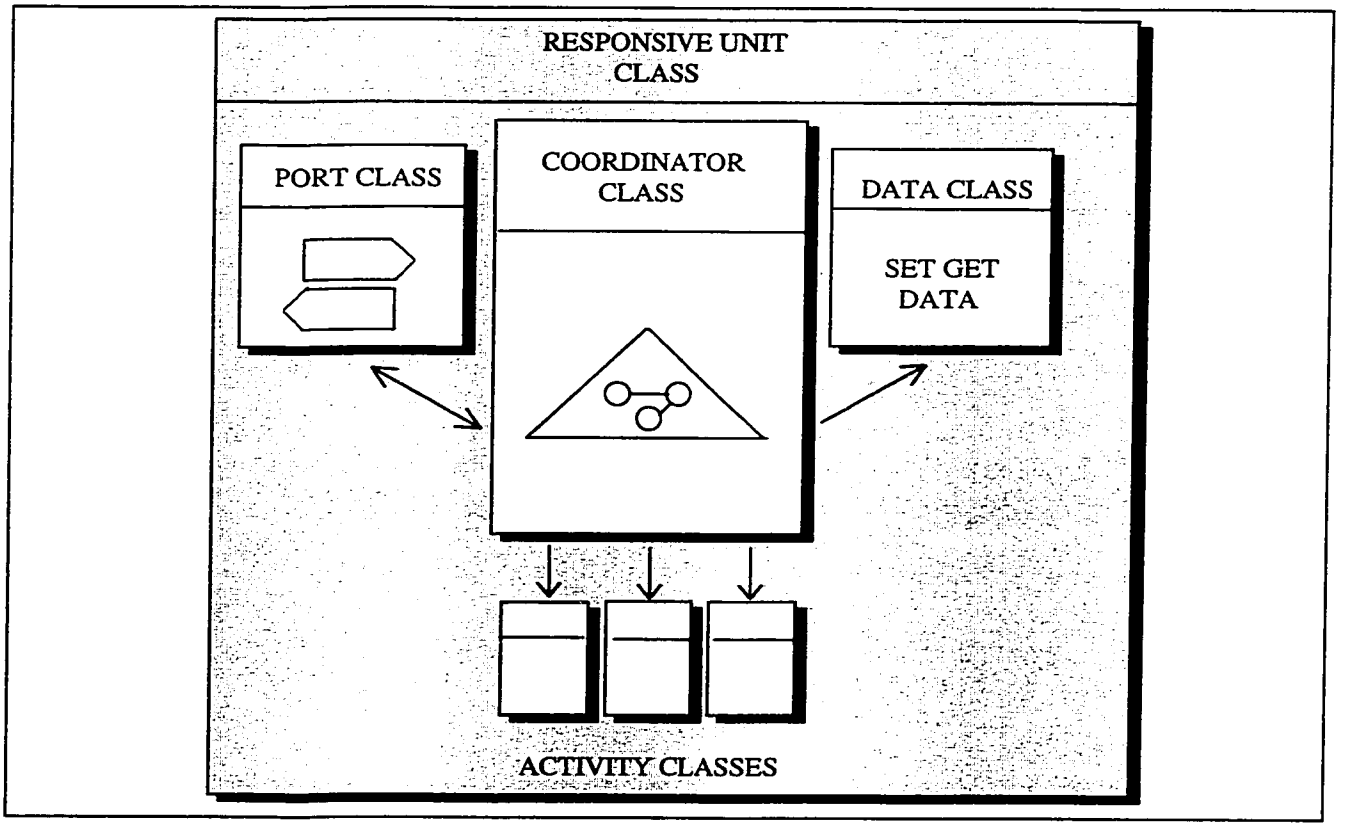
messages may come, eventually with a high error risk . For this problem, the Port object may have a filtering logic to let inside only the messages which also belong to an internally registered vocabulary. The coordinator will have to deal only with the pure interaction logic without even seeing the unreliable messages. Another object that may be typical for any Responsive Unit is a **Data object** ; this can gather all public data used for the public goals, as discussed in the previous chapter. For a simple case, this data can be kept in data members of the Responsive Unit ; however having a separate protected object specialized for data may enhance the quality of design. From the *behavioral point of view*, a Responsive Unit will be internally more like a mechanism of tightly interacting objects, with the *main behavior defined in the coordinator object* . We suggest that the *explicit implementation of a state machine* inside the coordinator is a good decision, based on the arguments from the previous chapter about the Dialog Coordination.

After all these considerations, we see an advantage in defining a template (in the usual meaning, not as the C++ keyword 'template') as some ready-written code for a Responsive Unit class. Such element can be cut and pasted in the code and adapted to each application. The typical objects will be in place with their relationships and pointers ; inside the coordinator a skeletal state machine will be waiting to be completed with events, actions, and states.

For such generic modeling, a **Responsive Unit** class (figure 5.2) will have in its composition :

- The **Dialog Coordinator class**, of master-slave type, that will organize the interactions.
- A number of **Action Oriented classes**, with internal computations, hardware actions etc.
- A **Port class**, to interact with other Responsive Units, linked to coordinator.
- A **Data class**, as a data warehouse, with set and get functions for data.

Figure 5.2 The Structure of a Responsive Unit



It is interesting that the Responsive Unit for a group of objects reflects the layout of an object:

- The *Port Object* mirrors the *public* methods
- The *Activity Objects* are like the *private* methods and
- The *Data Object* is like the *data* members.
  The *Coordinator does not have a counterpart in a single object.*

Even beyond Responsive Units the same layout can appear. The Units may specialize in similar directions, for :

- interface to users or to other groups of Units.
- complex activities
- data warehouse at group level
- for the coordination of the previous

Such a group of Responsive Units may define a team, as presented further.

5.2.1 Comparing the Responsive Unit approach with coordination related ideas from other methods

During the study on behavior modeling we have underlined several ideas that may be related to the generic concept of coordination. In most cases we already made some comments referring to our view and intentions for this work. Now we present them shortly again and compare them with the Responsive Unit approach.

For *Rumbaugh*, the *Dynamic Model* presents the *control aspects* of a system and the interactions between objects in the system [Rumbaugh]. There is no intent to 'control' the interactions. The *control aspects* mainly act inside objects. We retain that the *'control aspect' describes the sequences of operations that occur, regardless what the operations do internally.* This description is in the design but not necessarily in code. In a Responsive Unit the coordination is in a separate object while activities are methods of the other internal objects. More important, the design of a Responsive Unit is *driven by event responses, as we intend to clearly capture each of them* and their sequence. This is done by implementing in coordinator a state machine that calls for actions (methods in objects) as responses to requests.

At *Booch* we find design element named *'Mechanism'*, which is a 'decision about how collections of objects cooperate'[Booch]. The 'decision', described in the design document, should explain dynamic relationships between object. A Responsive Unit defines the 'decisions' inside the code, in a separate object. *'Decisions' are driven by event responses and define them completely.* Also, the 'collection of objects' is captured inside a container, which is the Unit itself.

In 'Use Cases' methodology [Jacobson]-and further in UML- [UML]there is a type of object named 'control object'. Such objects model 'functionality that is not naturally tied to any other object'. They operate on more entity objects, 'doing some computations and then returning the result to an interface object'.

The coordinator of a Responsive Unit is a kind of control object, but defined in terms of CODE. Inside Responsive Unit, in order to describe the event responses, we apply the distinction between coordination and execution. In each Responsive Unit there is one and only one coordinator and none is outside Units.

An Interaction Diagram can be, as Jacobson mentions [Jacobson], defined as a Fork Diagram; this is a centralized design where an object drives the others. The criteria for separating such an object are not further detailed. Also the diagram is still representing a unique trace of messages, without decision points. In Dialog Coordination chapter - 4.2.2 there is an expanded discussion. The state machine in the Responsive Unit' s coordinator shows all possible branching towards the internal 'activity' objects. The criteria used to build the 'driving object' are from the CODE concepts; again, the main idea is the distinction between coordination and execution.

In *'Modeling the World in States'*[Shlaer], the *Relationships* are considered worthy to be modeled with state machines. The relationships between objects have a lifecycle, as they evolve in time, have an instance and they are actively selecting the objects around. They act as an 'Assigner', linking objects to each other. In this model a *Relationship becomes an Object.*
In a Responsive Unit the dialog itself is objectualized, as the event responses with their sequence are captured in coordinator. Also, the group of objects becomes a Unit.

'Responsibility based design' [Brook] defines the objects based on their responsibilities.
In a subsystem, a distinct object is able to *respond* at major *requests*. The communication with the subsystem is through this object, which has higher responsibilities. The dynamic aspect is not captured and the layering is a *structural* one. The word 'responsibility' corresponds to one side of the pair request-response. Between objects, all such pairs represent their dialog; this is described in the coordinator of a Responsive Unit. A Unit cannot include other Unit; each one is a *'well-behaved'* entity.

About *real-time UML* [UML2], distinctly from original methodologies, we note the following:
A capsule may have several sub-capsules that collaborate; the eventual state machine of the capsule may play a controller role, with no detail how it influences the sub-capsules that collaborate freely. In ROOM [Room], which influenced real-time UML, there is the comment that more actors, i.e., capsules, may be organized in a coordinated, hierarchical *structure.*
In Responsive Unit the coordinator implements with its state machine all interactions between 'activity' objects. This organization is driven by behavioral requirements rather than structural decisions: 'what has to be done' is separate from 'when to do it' for response sake.
We note also that the use of ports is an attractive idea. As opposed to capsules, a Responsive Unit has only one port that relates the coordinator, to keep clear the response design.

The 'mediator' pattern [Patterns] *mediates* the relationships of other objects. Its interactions with the objects are defined in an interaction diagram; thus, as usually in these diagrams the response may remain only partially defined. It provides a 'room' for organizing the interactions.
In the Responsive Unit, the *coordinator defines each response and all allowable sequences* . It does *not provide only a 'room' for mediation* but *defines the complete interaction logic.* Also, it is an *active coordinator*, calling for actions, *not a 'facilitator'*.In 4.2.2. we analyze more details.

5.3 Team Coordinator : a Responsive Unit Managing a Team of Responsive Units

The visibility / clarity of event responses design is a major interest. The FSM in a coordinator may become too complicated if more than 5...8 objects are related. To avoid the state explosion problem in FSMs, the coordinators should be limited to a level that can be easily defined and understood. However, keeping them relatively simple means that the interaction logic between them remains complicate. In order to define the interactions properly, groups of coordinators can be further driven by another coordinator. If in a system there is a bigger number of Responsive Units, they may be grouped in Teams, to respond to higher level requests. The rationale of grouping them may be the generic layout observed in the previous paragraph.

Between Responsive Units there are light interactions, as the higher complexity and responsibility of Responsive Units require a more flexible dialog. For each specialized *Team of Units*, the type of dialog coordination used is *Team Coordination,* as introduced in chapter 5.

The *Team Coordinators are goal oriented*, implemented to act between Responsive Units that are working in groups. *Team Coordinators* are *on the **same domain*** (the *production* floor) as the Responsive Units. Their behavior is part of the definition of event-responses.

Team Coordinators will be similar to Responsive Units, as they may need the following:

- A Port to communicate with upper layers,
- A Data Object for the Team public data,
- A Coordinator which will maintain the *dialog between the Units* related in that team,
- Activities, which actually will be calls towards the coordinated Units.
  These activities are more as the signs made by an orchestra conductor, than working actions.

A *Team* has *specialized Responsive Units grouped together* for team goals, as mentioned.
As a generic observation, we note that components may be specialized as the following:

- Port oriented
- Data oriented
- Activity oriented
- Coordination oriented

A Team has the same layout we have inside Responsive Units, at the next level.

Again, the FSMs of coordinators need to be designed at a low complexity level, which can be understood. The result may be that the interactions between groups are complex and cannot be easily captured in interaction diagrams. Therefore, beyond coordinators of teams another coordinator (still of a 'team' type) can be added, as another Responsive Unit.

Each team is *specific* in its jobs, together covering the system requirements. The *specialized* Responsive Units may eventually be *reused*, as some may be *time-shared* between teams, to avoid duplication. Also, the Teams can be changed or reorganized for new sets of request-responses. These issues are not directly part of the event response domain; they belong to the *Agreement Coordinator*, which is presented in the following paragraph.

5.4 Agreement Coordinator :a Responsive Unit for Team Building and Maintenance

The third type of Dialog Coordination is in a *separate domain, not in the event-response domain*, as introduced in chapter 5. The *Agreement Coordination* does not deal with the production floor, but cares about the *agreements to be established between the other entities.*

An Agreement Coordinator may be instantiated as a Responsive Unit, or in more complicated cases as an entire Team. Usually we would consider one per system. It is defined for the *assignment, maintenance and change* of Responsive Units or teams ; may also be used for systems *supervision, start-up, shut-down and major error management.* More use details are following:

- For system *set-up*, with the necessary initialization test, resources check up, setting of hardware dependencies, or other needed actions.
- When Responsive Units are instantiated (and their internal objects in cascade) and when Units are connected in Teams.
- For the *reuse* of some specialized Responsive Units (in a *time-share* mode) when we do not want to have identical Responsive Units in different Teams.
- For a *dynamic reorganization* of Teams, the Agreement Coordinator can mediate the new Agreements. These can be renegotiated based on availability, priority or other criteria.
- For dealing with major *exceptions and errors* received from the Team level; a Team rebuilding may be required.
- For system *shut-down*, which may also need some kind of negotiation, like a closing Agreement. This has to reflect a specific sequence, dependent of the system state. The shut-down must be gentle enough to avoid surprises for the user or the hardware and to keep throughout the system a state that is optimal for the next start-up.

A Responsive Unit cannot contain other Responsive Units; each is a distinct, complete entity.

In figure 5.4 it is presented the Agreement Coordinator together with Teams of Responsive Units related through Team Coordinators. The view is a main view for the ideas presented, including that the one Responsive Unit can be time-shared.

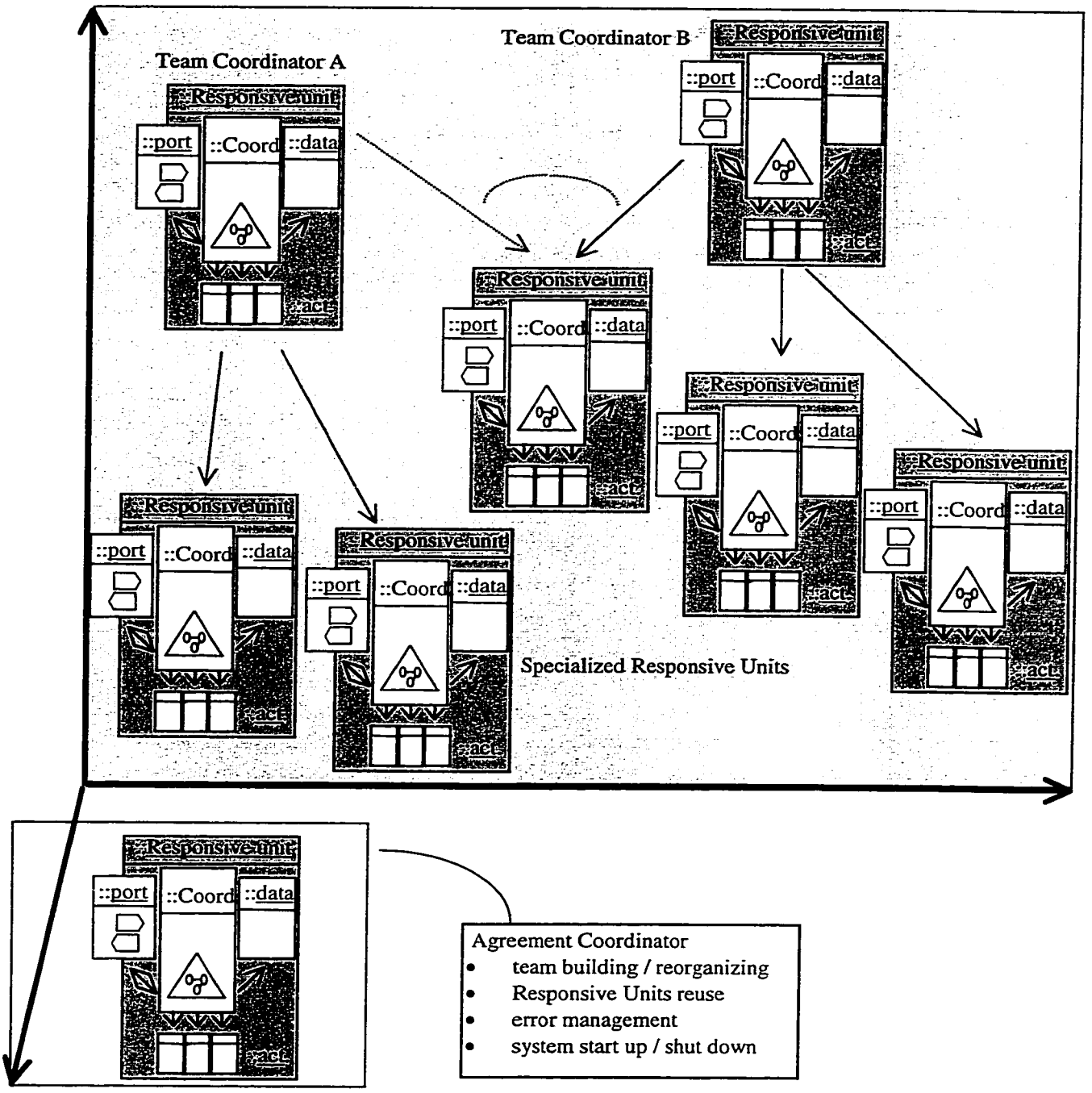


Figure 5.4 Teams of Responsive Units and the Agreement Coordinator

- The Master-Slave Coordinators are inside Responsive Units.
- The Team Coordinators are in the *Response oriented* area as they are goal oriented.
- The Agreement Coordinator is in the *set-up and maintenance* area.

This format is application independent and Responsive Units are similar for each utilization. Specific to the application are the definitions of state machines inside coordinators, the internal definition of each object and the grouping in teams.

5.5 The Behavior of the New Elements

For a Responsive Unit the behavior will mainly be specified in its coordinator since the other objects act mostly at its request. The explicit state machine helps for a clear relation with the design document. The internal dialog is thus very precise and the responses to events from outside the Responsive Unit are completely defined. The interaction logic may be well understood in code and corresponds with the design document.

If only Responsive Units would be used, without Teams and Agreement Coordinators, still the behavior modeling would be improved. This happens at least because the light interactions between Units are not as many as when one uses simple objects. Interaction diagrams have to cover fewer situations when the granularity of the basic entity is higher. A research points that a too small or a too big granularity leads to more problems [Hatton]

The behavior of a Team is represented by the state machines of the Units together with the state machine of the Team Coordinator. Actually this last one captures the logic of interactions between the other Responsive Units, that otherwise would be dispersed throughout the interacting Units. Without this Team Coordinator the Team behavior would be defined by the free interacting state machines, that define the Responsive Units. With it, the state machines of the Units are related through a higher level state machine, that receives the Team events. *This organization is a hierarchical state machine design, with the strong qualitative comment that the higher level machine only relates the others and does not do any work.*

Between Teams it may be necessary to have an other dialog coordinator, still as a Responsive Unit, at system level, that would define the dialog between Teams. Conceptually the behavior at this level is similar to the previous level.

The Agreement Coordinator defines mainly the initialization behavior, that is not normally in the event-response domain. Its events should be sent by start-functions and its actions are mainly initialization of Responsive Unit and Teams. This coordinator may be used in some cases also for dynamic call of constructors/destructors.

In figure 5.5 is shown the behavioral view of a team of Responsive Units with their Team Coordinator. We underline here the main points in this behavioral model:

- To completely define responses, the model differentiates between activities & coordination.
- Responsive Units localize the responses of a group of objects and define each in a coordinator state machine.
- Inside Responsive Units, activity objects do not ask each other for action.
- The system is divided in multiple state machines and each may have sub-machines.
- The actions of the Team Coordinator machine are events for the other Unit machines. These are NOT defined as sub-machines (as StateMate shows the system in ONE machine).

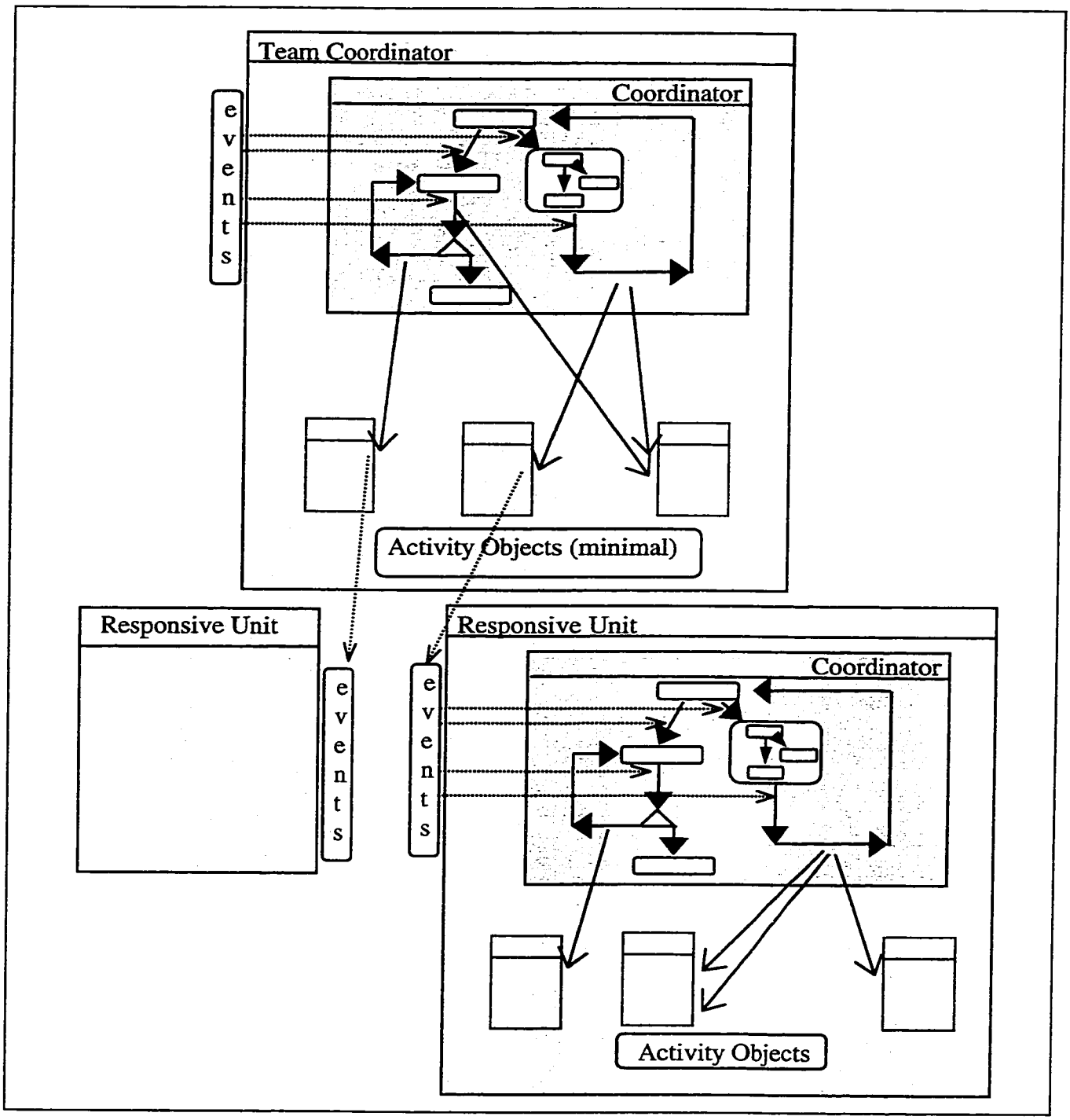


Figure 5.5 The Behavior of Responsive Units in Teams

The state machines in Responsive Units have to be *maintained in a proportion that allows a good understanding of event responses* -our engineering goal. Therefore several teams may be needed to define a system, eventually with a next level of coordination as explained before.

The behavioral requirements will drive the *development process* which follows the main lines from CODE, presented in Chapter 3. For any application there is already a general model as in figure 5.5 which provides an advanced step of design decision. The model becomes specific to a unique application in several steps:

- At requirement acquisition, major events (as introduced in Ch. 4) are first determined and their responses roughly described. The relative sequence between them is also clarified. At this point already a first functional diagram, in the form a state machine for the Team Coordinator can be issued. Its actions will appear as events to the Responsive Units of the Team. One can continue now with minor events (to see Ch. 4) or deal with them after a round of major design, as in spiral development.
- At architectural design, the major parts of the system are established and Responsive Units are determined for specific domains.
- The design phase defines the responses for each Responsive Unit by clarifying the 'atomic' activities needed for responses and by completing their coordination. The precise interaction sequence and history is captured in every Unit; this is a major interest of our engineering design.

The model can be directly and uniquely implemented as the state machine model has a clear *corespondent in code*. Also the code can be easily read back, if a good design proportion is maintained in Responsive Units and their state machines. For design changes, the localization of coordination and the separation of 'atomic' activities allows easy response modifications.

A clarification is needed concerning the relationship between Responsive Units and the class inheritance concept. There is much research referring to behavior inheritance, implied in some ways by class inheritance. The conclusions are very different as the idea of inheriting behavior proves to be complex, referring to ways of expanding or modifying state machines. The definition of their interactions is also a major issue. In our approach we prefer to clearly define the behavior of each Unit. The inheritance may be used for objects which are less interactive, as data objects, for user interfaces built from classes in libraries, etc.

We consider that the use of Dialog Coordinators (with sub-types), Responsive Units and Teams has only some drawbacks; these have been considered in the 'Dialog Coordination' chapter. The perception of an overhead (in timing or memory) would be the main issue; as discussed, it is not a major one. The quick hardware evolution is reducing the issue but we consider also that a system which is better organized has less overhead.

5.6 The new concepts and the multi-process environments

In this thesis the attention has been given to the dialog between objects, since we address recognized problem of interaction design in object oriented systems.

It is also believed that for the design of multi-process systems the Dialog Coordination concept and the Responsive Unit approach may help. From the strategy point of view, we review the position of Jacobson about such systems. He says that OOSE 'does not have *a priori* strong semantics' for them since '*various implementation environments provide various semantics*'. He would think about *methodology specializations for different environments.* The use of Responsive Units as mega-objects would follow a similar path, without details about semantics specific to environments. Basically the object oriented development itself relates well with the development with concurrent processes.

The thesis approach could be used by addressing the synchronization activities (as semaphore take or give) *inside* Team Coordinators, considering that the *synchronization* should not be done too deep in the objects of a team. The *communication* between tasks can also be built at the level of Team Coordinators, as these can define the necessary response-oriented interaction logic. In the cases where a Team of Units is not used, a Responsive Unit by itself still can help for synchronization and communication between threads. Its coordinator can keep track of the interaction logic, instead of letting activity objects to do it. A valuable book on this theme is 'Object-Oriented Multithreading Using C++'[Hughes]. It is a complex study from which we extract some 'final thoughts' : we should not 'leave the synchronization responsibilities of the object to the user of the object' since 'once an object's data can be accessed in a multithreaded environment, we have the potential for race conditions'. Within our Responsive Unit (a macro-object), there is even a defined location in the coordinator object where these 'synchronization responsibilities' can be placed.

Different systems employ different synchronization and communication mechanisms, but defining the response-driven logic of interactions in a clear location is a generic idea that may be applied in any case.

Another choice may be the building of a separate coordination task containing a Responsive Unit for the goal-oriented coordination of tasks. This idea relates to the thesis of Dellarocas [Del] that suggests the use of a coordination process as a mean of managing the relationships between processes. We note that he does not attempt to manage each event response, but only main process relationships. He covers quite exhaustively the possible coordination concepts (not specific responses) and recommends the design of a distinct process that contains them.

Such applications may be by themselves the subject of extensive research. We addressed in this thesis the general OOD behavior modeling, as for single-process systems, and we see the use of our approach in multitasking system more as a specialized implementation.

Chapter 6

Case Study for the New Concepts

Introduction

To apply the new concepts we wish to develop some systems having multiple objects with *relatively complex interaction logic* and also having a good amount of user interface. Since *modifiability* is an important quality of software, we develop three systems from the same family to explain the advantages of the suggested approach when changes are required.

The systems we chose are some *Microprocessor Simulators*, of the following types:

- **Single Accumulator Microprocessor**
- **RISC Microprocessor**
- **STACK Microprocessor.**

The three systems will be related through a separate user interface named '**School-of-Microprocessors**'.

The development language needs to be an Object Oriented one and to have Graphic User Interface abilities. Visual C++ and **Java** have these qualities and the author had industrial experience with them. The last one is chosen here for the flexible Graphic User Interface, for being a clear Object Oriented language and for the ability of designing the systems as applets, easy to present on web.

In this development, we will use the Dialog Coordinator with its three sub-types :

- Master-Slave Coordinator,
- Team Coordinator,
- Agreement Coordinator

to capture in code the desired interaction logic between objects and between user & system.

Responsive Units will be used as 'mega-objects', each one with a *Master-Slave Coordinator* object inside that defines the interactions between the other internal objects. The coordinator implements a StateCharts machine ; in this one , for each request that comes to the Responsive Unit, we define as response a corresponding sequence of actions.

The interaction logic between the Responsive Units of one system will be defined in a *Team Coordinator* (itself a Responsive Unit) which may also capture the user-system dialog. Inside

this Team Coordinator we will also implement Response Managers, one for each event. Inside one Response Manager its specific event may trigger different responses in different states. One response means a sequence of Actions that are executed in the various 'Problem Domain' Responsive Units. These Units receive the call for actions as events in their coordinator state machine.

A distinct Responsive Unit takes the role of an *Agreement Coordinator* that initializes and coordinates a user interface and the three simulators. It builds and controls button objects, sets a user text output, types out in this field appropriate information relative to each event and state, deals with the initialization of one simulator at a time in a certain order as designed and follows the deletion of each simulator to allow the next step. The paragraph 6.4. expands on this issues; here we underline that such global activities and their logic is separated from the three 'teams'. Thus, we can concentrate to define in each team the application specific responses.

It is meaningful to add a note concerning the number of coordination layers in this kind of application. As said, the agreement coordinator is not considered as a layer, being in an other plane than the event responses. Therefore, in response domain there are only two layers: the coordination in Responsive Units and the team coordination. The reasons of not having a third layer are (1) the simulator teams do not interact to each other in the response domain and (2) inside each team there are only three Responsive Units, that can be related with one coordinator. A new layer may be needed in systems with many Responsive Units if the interactions between teams are enough complicate, thus difficult to represent.

For development we will follow the Coordination Oriented Development *process* with its phases:

- *Architectural phase with*
    - *requirement acquisition and*
    - *architectural design*
- *Design phase*
- *Implementation phase*
- *Deployment phase*
- *Maintenance phase*

The last two are used for a commercial product, which is not the case here.

The development of the first Microprocessor, Single Accumulator, will be described in detail.

For the other two we will underline a major quality of the suggested model, the fact that we can *easily modify a system*, even if its interaction logic is quite complex.

6.1 Single Accumulator Microprocessor Simulator

6.1.1 Architectural phase

6.1.1.1 Requirement Acquisition

General view

We are interested to build a Microprocessor Simulator that may be used for a certain level of student training. The Simulator has to be presented through a Graphic User Interface, with action buttons and active data fields, to indicate the values of each element (for example a register) of the Microprocessor.
We need a kind of *Editor* for program input and also a way of saving the program.
The user has to be able to *execute* the program in one of the next modes:

- In micro steps, to visualize how *each register changes its value at each step*,
- in instruction step, to observe the result of *one instruction at a time*,
- No-stop run to the end.

The possibility to continue with an other mode from any point would be an advantage.

Refinement of Requirements

The first microprocessor will be a 'Single Accumulator' type.
We will need to display four main areas:
a) the microprocessor itself
b) the memory, for data and program
c) the editor
d) an area with all action buttons

a) *In the area for the Microprocessor*, we need to see:

- most major registers, as text fields (not write-able by user) :
  - Program Counter PC
  - Memory Address Register MAR
  - Memory Buffer MBR
  - Data Pointer DP
  - Instruction Register IR
  - Temporary Register TR
  - Accumulator ACC
  - Flags
  - Carry, Negative, Zero C N Z
- the Arithmetic Logic Circuit, as a drawing which indicates the circuit
- the Busses for interconnections between registers, ALC, memory.

b) *In the memory area* we have to show two different text fields or lists:

- data memory
  with an address numbering, editable
- program memory,
  where each line, as one memory location, has to keep an assembly language instruction and the needed data fields.

For the sake of avoiding input errors from user, these memory areas will be editable through an Edit Utility, not directly.

c) *The facility for editing* programs and for filling in the data memory shall be a list with the assembly language instructions chosen for the Simulator.
The user will not type the desired instruction but will choose it from the ready written list.
Thus the user can quickly fill in, and we avoid extensive spell checking.
A separate window will show this area, activated from a menu.

d) *The action buttons* will be essentially the *Events received* from the user, as *requests* for certain *responses.*
These Requests will be one part of the main Dialog, for which we will build a Dialog Coordinator.
The *Dialog in the other direction*, towards the user, happens through the

- registers' text fields
- highlighting line in the memory lists
- a 'Feedback' text field, with information on the specific step.

Now we clarify the required event responses, in the manner presented in chapter 5, page 73.
The *Requests from the user*, which are the system events, take place through the Action Buttons that will be:

- An EDIT button with six related buttons:

  For Program Memory:

  - APPEND to append a new line at the end of program list
  - INSERT to insert after the user-highlighted line
  - DELETE to delete the user-highlighted line

  For Data Memory:

  - APPEND DATA to append data at specified address
  - DELETE DATA to delete data at the user-highlighted line

- A LOAD button, for loading the program

- An EXECUTE button with three related buttons:
  - RUN for starting a free-run of the program
  - INSTRUCTION STEP to execute one assembly line at a time
  - MICRO STEP to execute one Register change at a time
- A SAVE button , to save the program in the view
- A RELOAD button , to reload the previously saved program.

The interaction logic between user & system will be captured in a Team Coordinator, which will also relate the other internal Responsive Units. Thus the user dialog will not be directed towards any 'problem domain' Responsive Unit, but towards this Team Coordinator. The definition of the complete response at each request will be accomplished with the help of Response Managers.

The Action Buttons send the Requests to the Team Coordinator. This one sends back a message and /or tells a Response Manager to call the sequence of actions that form the complete Response.

Figure 6.1 presents the Requirements Acquisition for the main Dialog.

After EDIT and EXECUTE we get theoretically in substates.
The Events as APPEND, INSERT, etc. are succeeding EDIT,
and RUN, INSTR-STEP,MICRO-STEP are succeeding EXECUTE.
Response Managers or *simple Actions* will be called on these events.

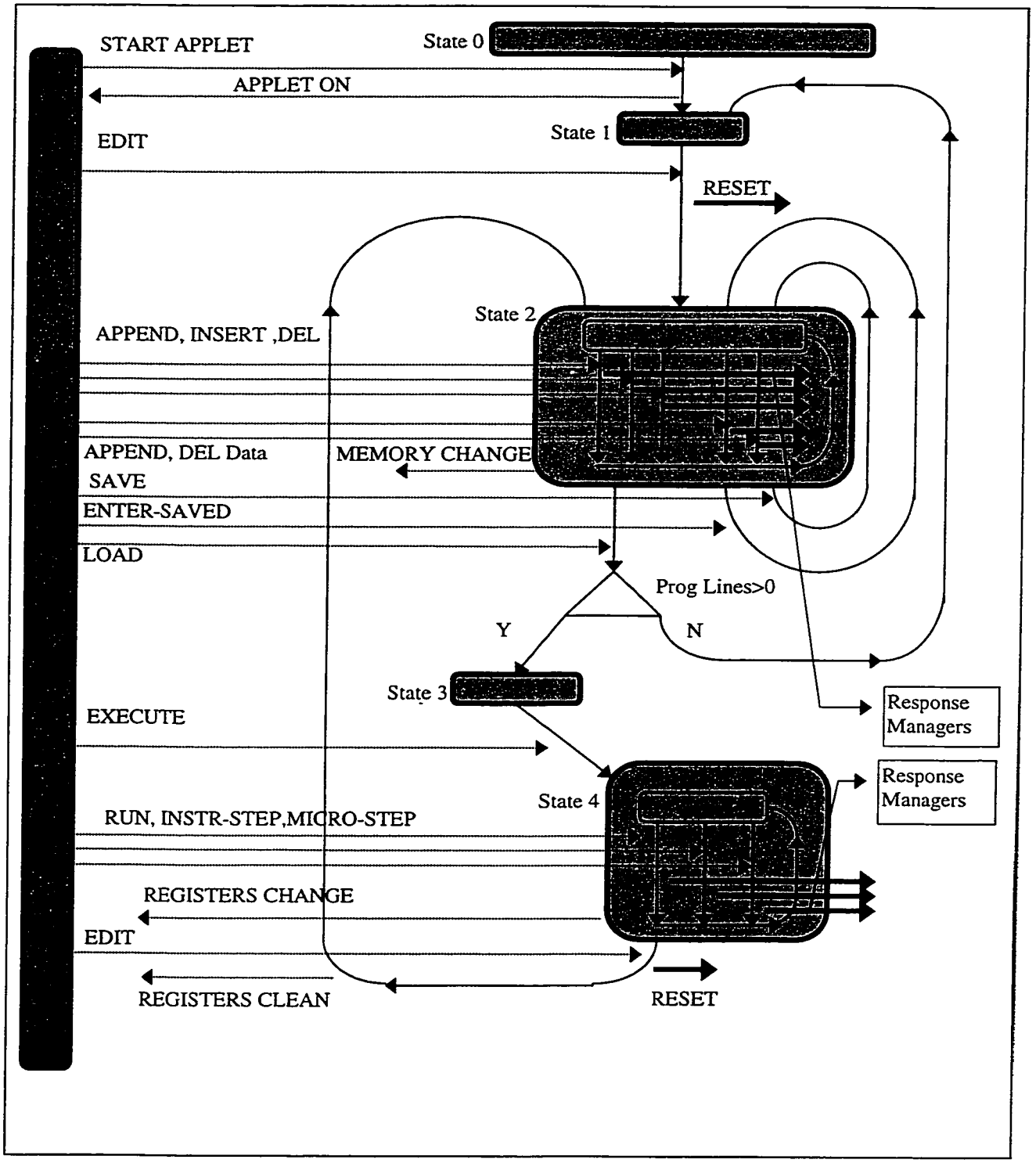


Figure 6.1 The USER-SYSTEM Dialog : StateChart in the Team Coordinator

As part of Requirements Acquisition we need to present the choice of assembly language functions and addressing modes considered representative for the Microprocessor.

	Addressing Modes		
	immediate	*direct*	*indirect*
add_imm	x		
add_dir		x	
add_ind			x
adc_imm	x		
adc_dir		x	
adc_ind			x
sub_imm	x		
sub_dir		x	
sub_ind			x
sbb_imm	x		
sbb_dir		x	
sbb_ind			x
st__dir		x	
st__ind			x
ld_imm	x		
ld__dir		x	
ld__ind			x
jmp_dir		x	
jmp_ind			x
joc_dir		x	
joc_ind			x
jon_dir		x	
jon_ind			x
joz_dir		x	
joz_ind			x

Instructions without data :
clr
neg
inc
dec

For this Requirement Acquisition phase we define the Response at the level of main Team Coordinator. The next level of details, internal to the system, will be part of the Design Phase. Also as part of Requirement Acquisition, we should look for important system limitations as the following constraint: Java Applets are not allowed to write in files. Because of this constraint the saved program is kept during one session, until the user exits the applet.

6.1.1.2 Architectural Design

We will follow the style from Coordination Oriented Development Environment (CODE) complemented with the concepts introduced in this thesis - Team Coordination, Master-Slave Coordination and Responsive Units. The Agreement Coordination is in the 'School of Microprocesors' applet, besides the simulators. We follow further the development process outlined in chapter 5, page 73. As explained in theoretical chapters, we want to *instantiate the logic of interaction, the dialog, in special objects. Again, these are NOT problem domain objects.*

One Response Unit (like a mega-object) is going to be the *main Dialog Coordinator* considered *'Team Coordinator'*, with links to all other 'Problem Domain' Responsive Units.

The Team Coordinator has to implement

1. the Main User-System Dialog, (as the StateChart in Requirements Acquisition)
2. the Response Managers, (as StateCharts to be described in Design Phase) which will call the other Responsive Units for action.

The User Interface is built in a class inheriting the 'Frame' class from Java.
It creates the interface objects and sends *Events* to the *Team Coordinator.*

We have distinguished *three 'Problem Domain' Responsive Units:*

1.The Memory Responsive Unit through which we get access to memory.

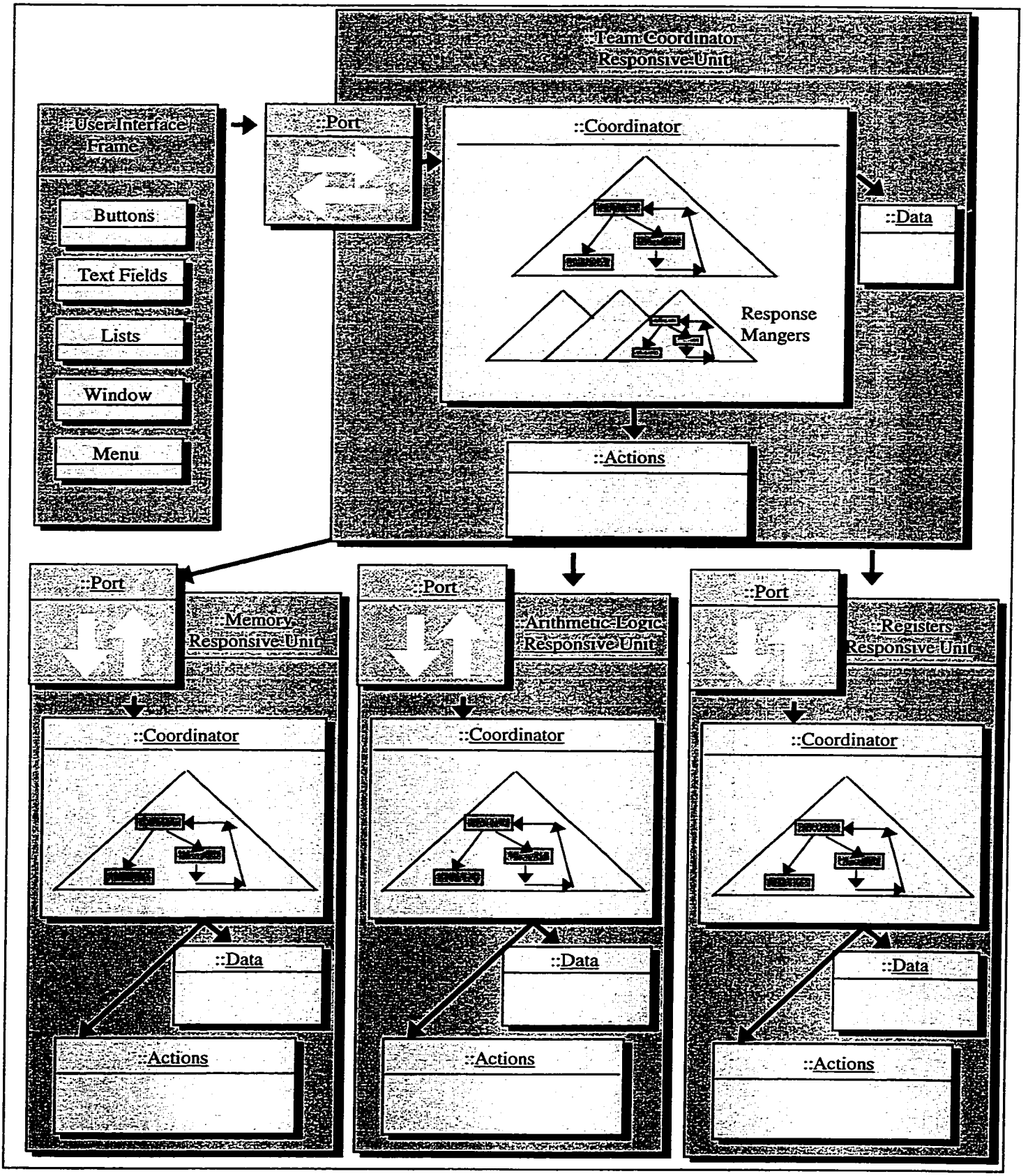
Its Coordinator Object implements the interaction logic for related activities.
The Data Object relates to the memory locations.
The Port Object relates the Unit to the external world.

2.The Arithmetic-Logic Responsive Unit where we implement the required operations of the Microprocessor and the related logic.

3.The Registers Responsive Unit that contains the registers.
(In the Stack_Machine Microprocessor case, we will have special interaction logic between registers.)

The figure 6.2 presents these elements and their relationship in the architectural design.

Figure 6.2 Architectural Design



The map represents objects with connection arrows as associations. The objects inside an other one, which is container-like, are instantiated and accessible through this container.

The Team Coordinator implements all relationships between Units. The symbols of triangles with state machines inside suggest that inside any coordinator object we have at least a method that implements a StateChart machine. The method will receive the events in their sequence, according to the design of the interaction logic, and will call sets of activities as responses.

After this image, *we should reiterate our major attempts* for enhancing the *Behavior Modeling*:

1. *Dialog Coordinators implement the most of interaction logic.*
In other methodologies we find Control Objects, Mechanisms, Relationship• Lifecycles as behavior modeling elements beyond basic objects, but none of these are directly concerned to capture the flow of interactions between objects.
2. *Responsive Units* are used as 'mega-objects' with a *Coordinator of Master-Slave* type inside. This coordinator *implements the order* in which the public methods from the Port may be called and *the order* of execution of private actions.
3.The behavior is defined with *multiple StateCharts Machines in a hierarchical organization.* The existent choices, which usually lead to an unmanageable level of complexity as previously discussed, are:

- One mega StateChart per System,
- Parallel, interacting state machines on a single layer as in SDL.

With the new approach we manage:

- *To contain the complexity* of states and interactions in Responsive Units.
- *To restrict* the initial attention to the main Team Coordinator, representing it as a *State Machine* (or better said *'Dialog Machine').*
- *To define the design detail later* in 'problem domain' Responsive Units representing them with StateCharts.

The behavior of a system is defined through *the behaviors of some SIMPLE Units at which one adds the behavior of a Coordination Unit. This last one defines the dialog between Units and also with the user.*

In the Architectural Phase is also the place for the definition of the *User Interface.* We will do it at the level of detail relevant to the thesis.

An Applet starts from a web page and has to display the required areas:

a) the Microprocessor itself
b) the memory, for data and program
c) the editor
d) an area with all action buttons

For space reasons we build a *separate* window to show the Editor Utilities, where the user chooses ready-written assembly instructions from a list. We do this to avoid direct writing, which is error prone and would require an extensive error management.

On the main view we set elements as in figure 6.3:

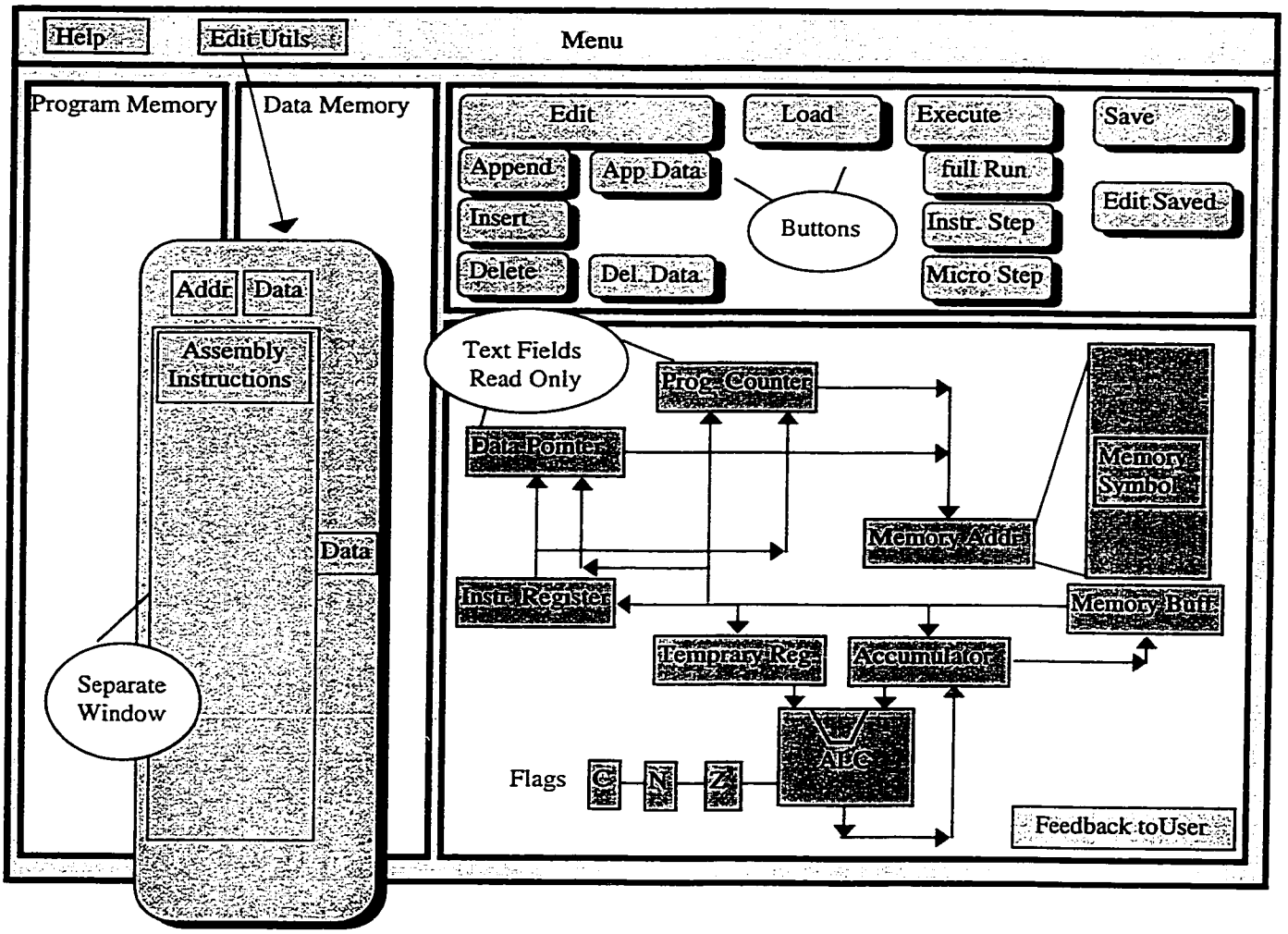


Figure 6.3 User Interface

In Edit Utils

- fields labeled Addr, Data are the input for Data Memory and
- the Data field near Assembly Instruction gets the value used with an instruction.

Requests, or events, are sent in through Action Buttons. The responses to requests are visible mainly in the change of registers and in the text field named 'Feedback to User'.

6.1.2 Design phase

This phase defines the internal details of the system, that are not visible to the user or customer. We continue the development process presented in chapter 5, page 73. In the Requirement Acquisition document we captured the interactions with the user. Now we have to *detail the Response Managers and the 'Problem Domain' Units.*

Design of Response Managers

Response Managers belong to the Team Coordinator Unit and are connected to StateChart defined at Requirement Acquisition as in figure 6.4 (that shows the relevant a part of the named StateChart).

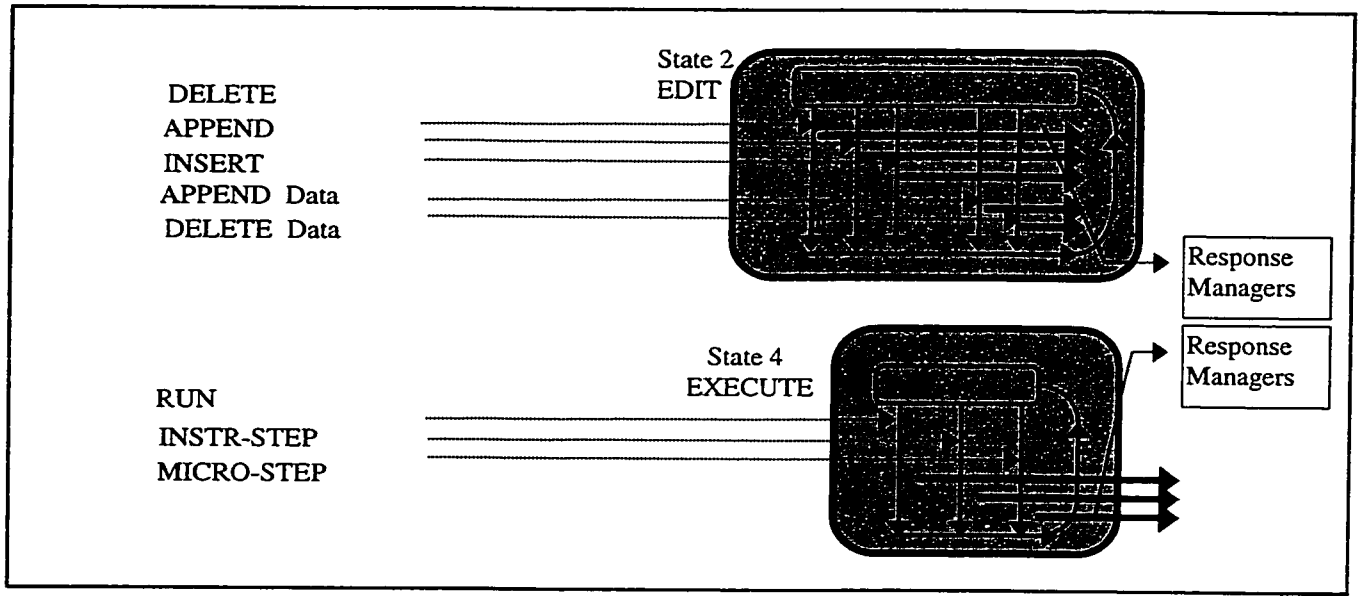


Figure 6.4 Response Managers

The Response Managers are activated by one specific event. In different system-states the same event can produce a different response. Each response is defined as a certain sequence of actions that need to be completed. The Response Managers define thus a state machine driven by a single event.

6.1.2.1 Response Managers from State 2 , after the event 'EDIT'

State 2 is reached after the user clicks on Edit. The Response Managers from this state act either for the Program Memory or for the Data Memory.

6.1.2.1.1 Edit the Program Memory:

In Edit-Utils supplementary window, the user selects an assembly instruction and eventually writes a value in the Data Field.

After pushing 'Edit', the user may choose 'Append' or 'Insert' (figure 6.5). To delete, he selects the line.

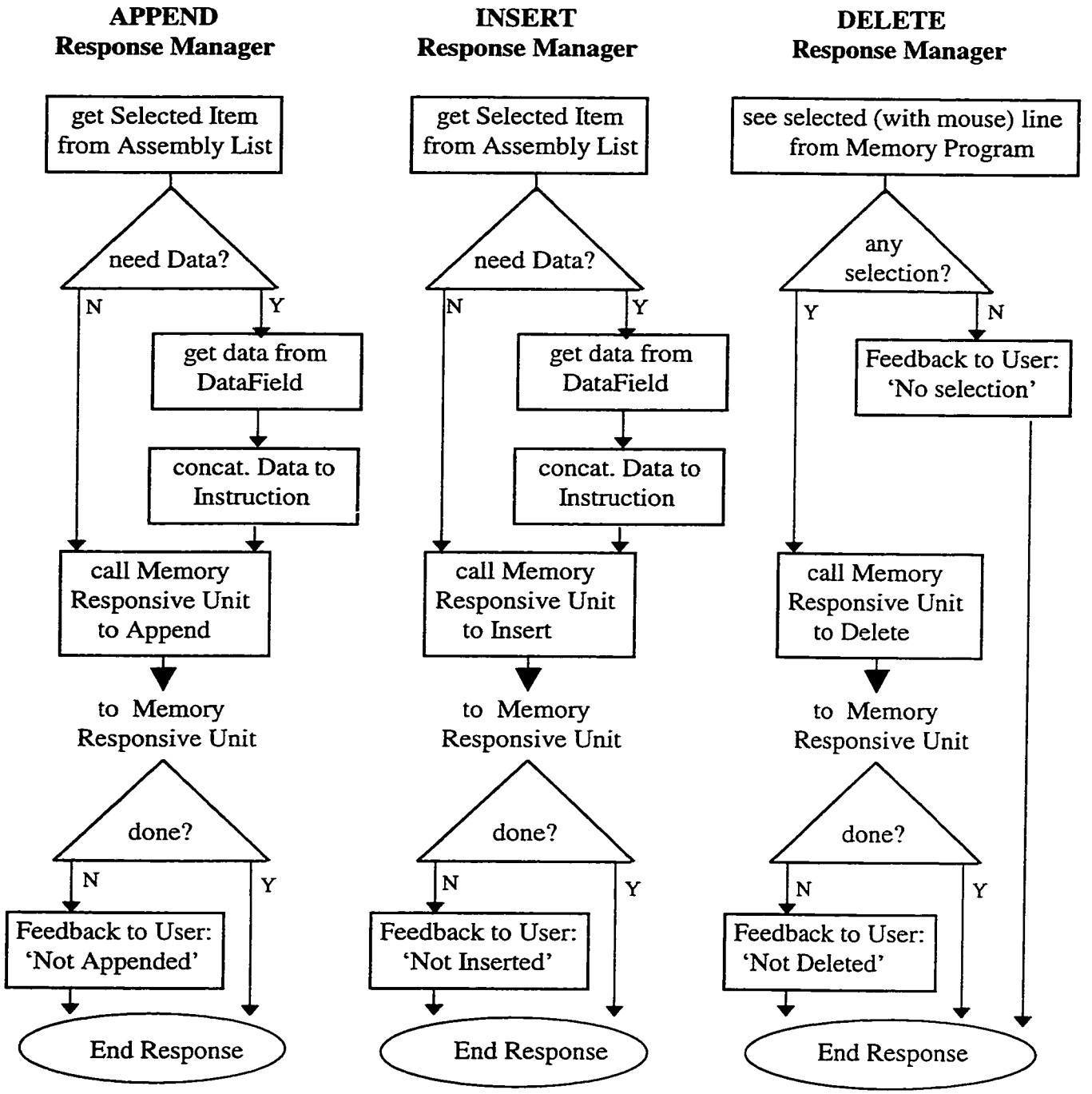


Figure 6.5 Internal Actions in Response Managers

6.1.2.1.2 Edit the Data Memory:

To Append Data:

- The user fills Address and Data in Edit-Utils supplementary window,
- clicks 'Edit' in main Frame,
- clicks Append Data.

To Delete Data:

- The user selects a line with the mouse in Data Memory,
- clicks Delete Data.

The internal actions for these response managers are as in figure 6.6.

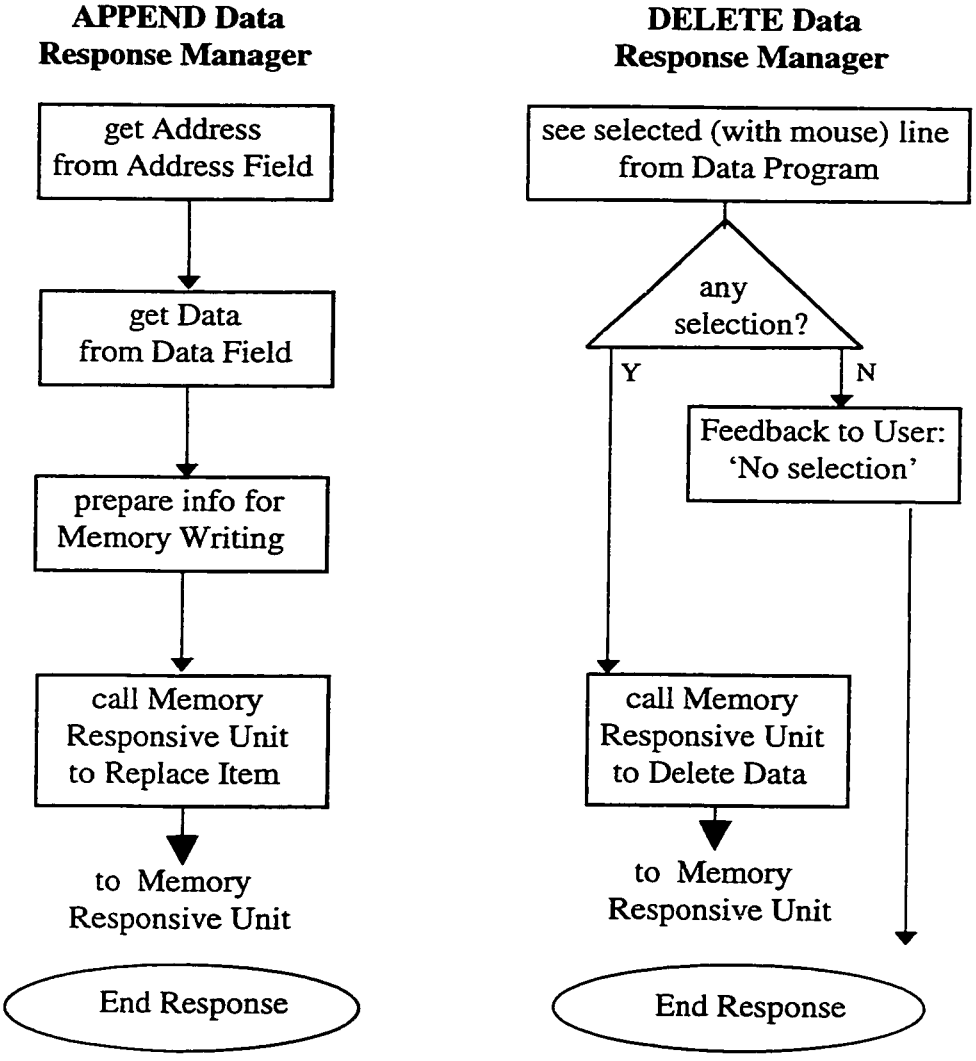


Figure 6.6 Append & Delete Response Managers

The Memory Responsive Unit is called in all previous cases. Its 'Port' object has five methods corresponding to each call. Inside the Memory Responsive Unit its 'Coordinator' object implements the interaction logic for the access to memory. This part will be described distinctly from the Team Coordinator Unit, when we will describe the 'Problem Domain' Responsive Units. This separation of concerns improves all layers of development and maintenance.

6.1.2.2 Response Managers from State 4 , after the event 'EXECUTE'

RUN Response Manager: calls one instruction at a time till the end of program.

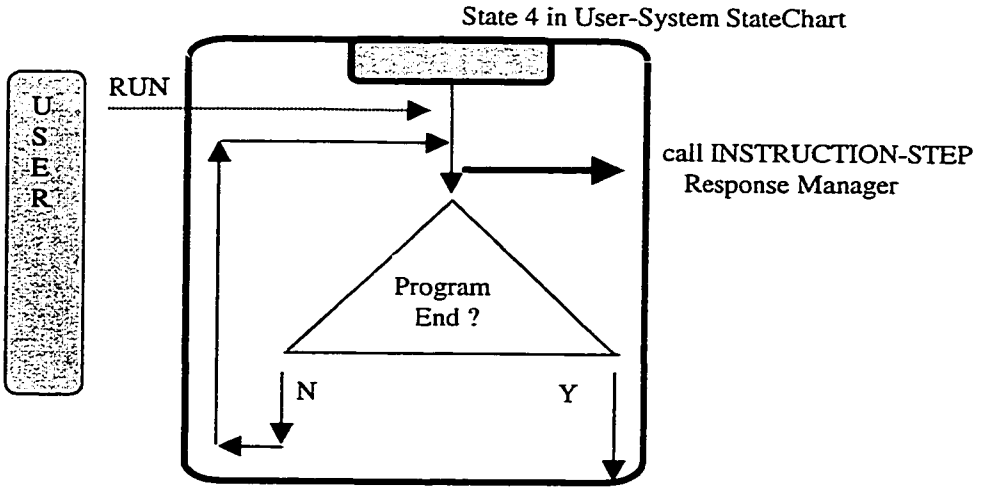


Figure 6.7 RUN Response Manager

INSTRUCTION-STEP Response Manager
It calls one micro-step at a time till the end of instruction.

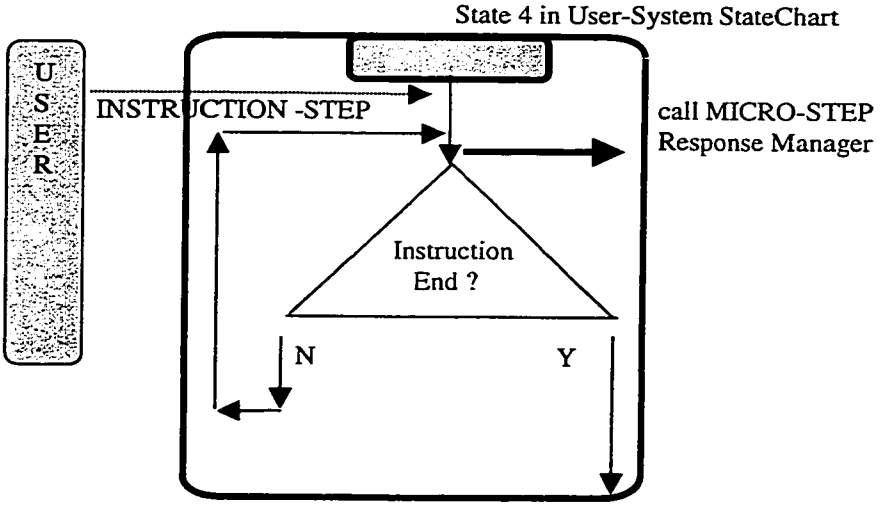


Figure 6.8 INSTRUCTION-STEP Response Manager

The MICRO-STEP Response Manager is presented in figures 6.9, 6.10, 6.11, 6.12

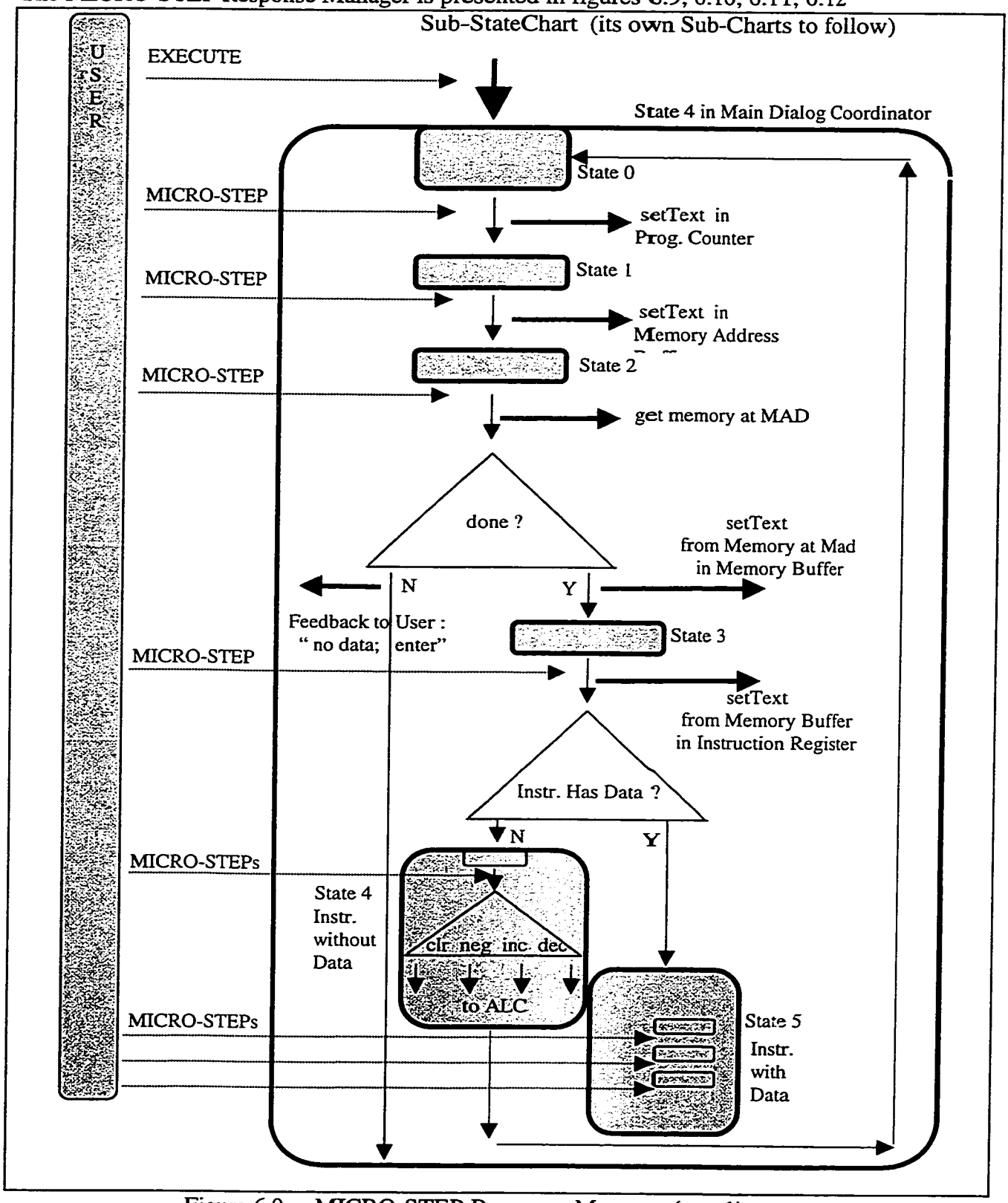


Figure 6.9 MICRO-STEP Response Manager (part 1)

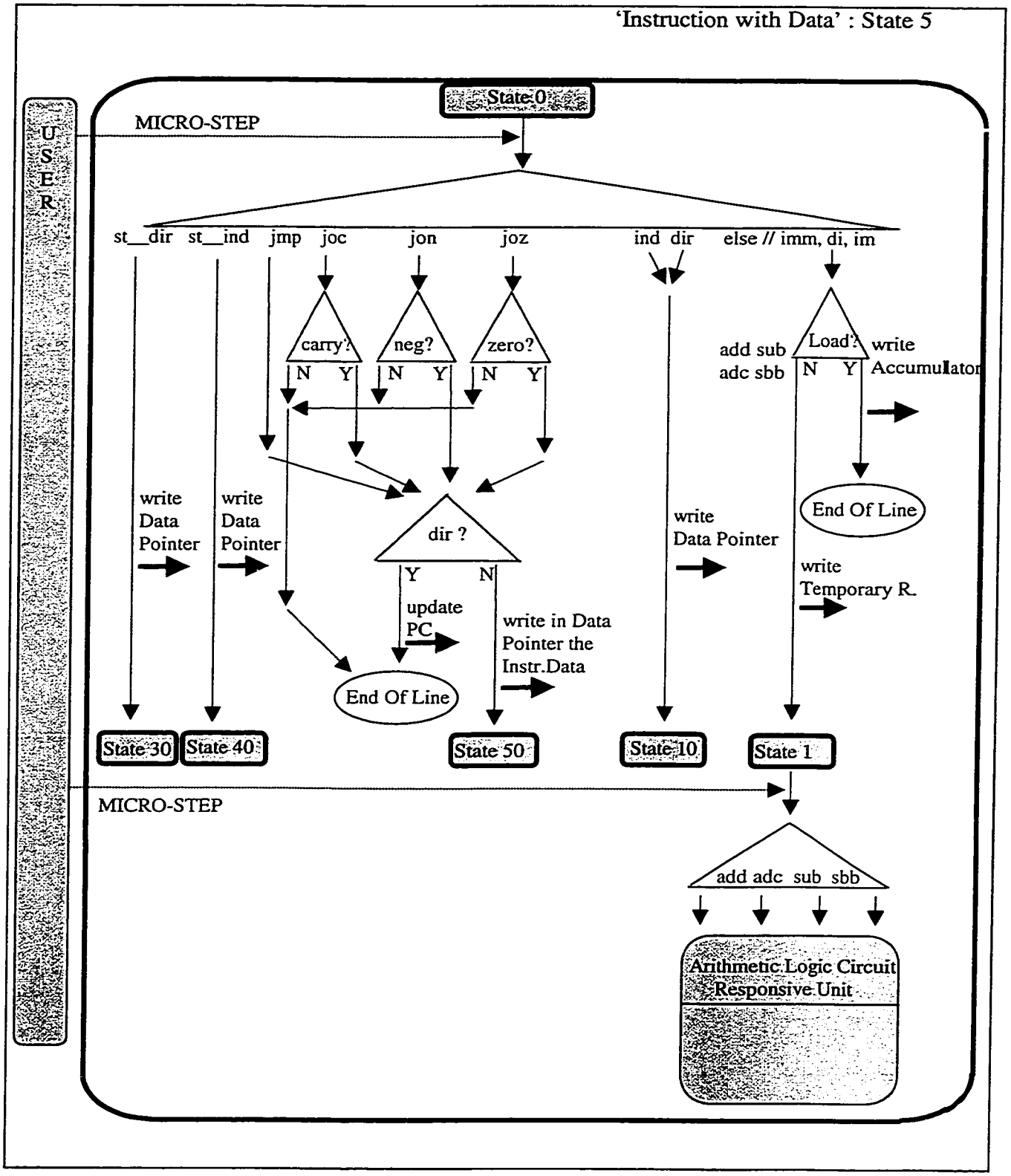


Figure 6.10 MICRO-STEP Response Manager (part 2)

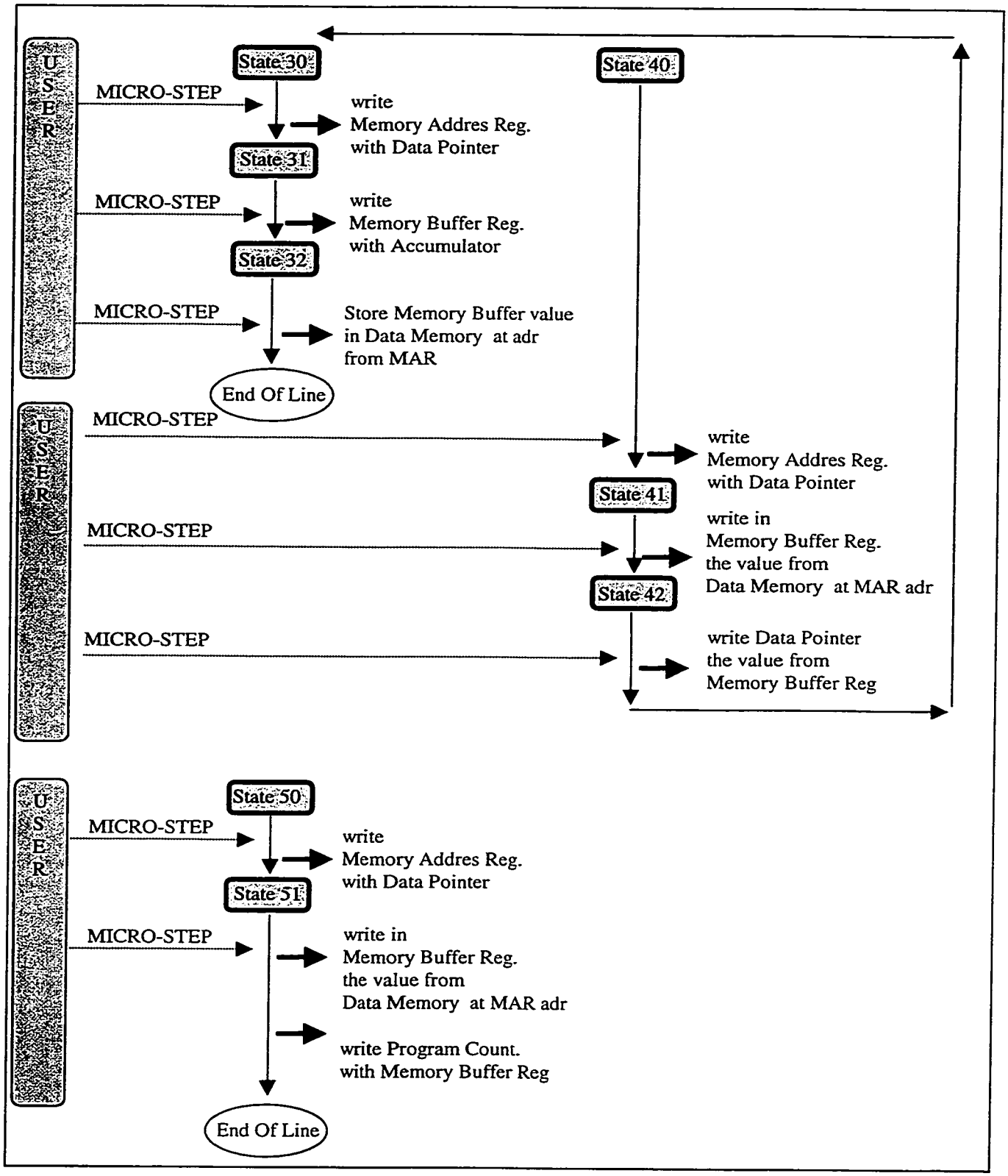


Figure 6.11 MICRO-STEP Response Manager (part 3)

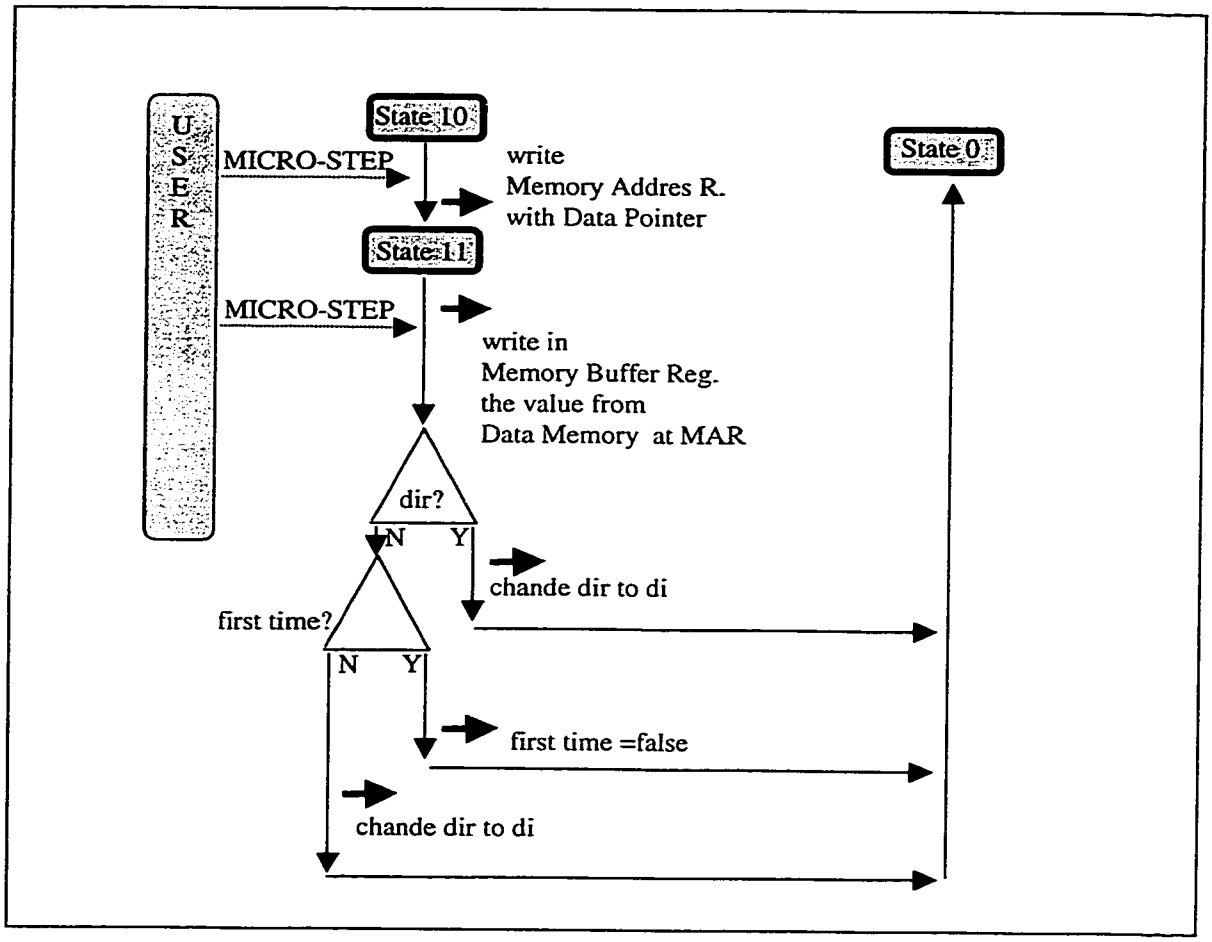


Figure 6.12 MICRO-STEP Response Manager (part 4)

The Arithmetic Logic Circuit Responsive Unit is called from :

1. the State 'Instruction without Data' for the instructions
   clr neg inc dec
2. the State 'Instruction with Data' , subState 1 for the instructions
   add adc sub sbb

These instructions can have the access to data immediate, direct or indirect.
At the time of calling the *ALC Responsive Unit* the data is ready to use.

To complete the Response for each of these instructions we will pass at the next step, the design of the Responsive Units.

Design Phase continuated:

Arithmetic Logic Circuit Responsive Unit

The assembly language instructions we mentioned on previous page correspond to Requests that are sent to this Responsive Unit to be solved.
The Coordinator Object receives them and based on the respective event (request) asks the Activity Object to perform the respective Action.

In this specific situation, the Requests are completely independent, as there is no demand for any relative order between them. Also each Action is uniquely related to one Request, i.e., in our case we do not have interrelated Actions.

For these reasons the state machine of the coordinator is simple : one state with a case statement for each event (figure 6.13).

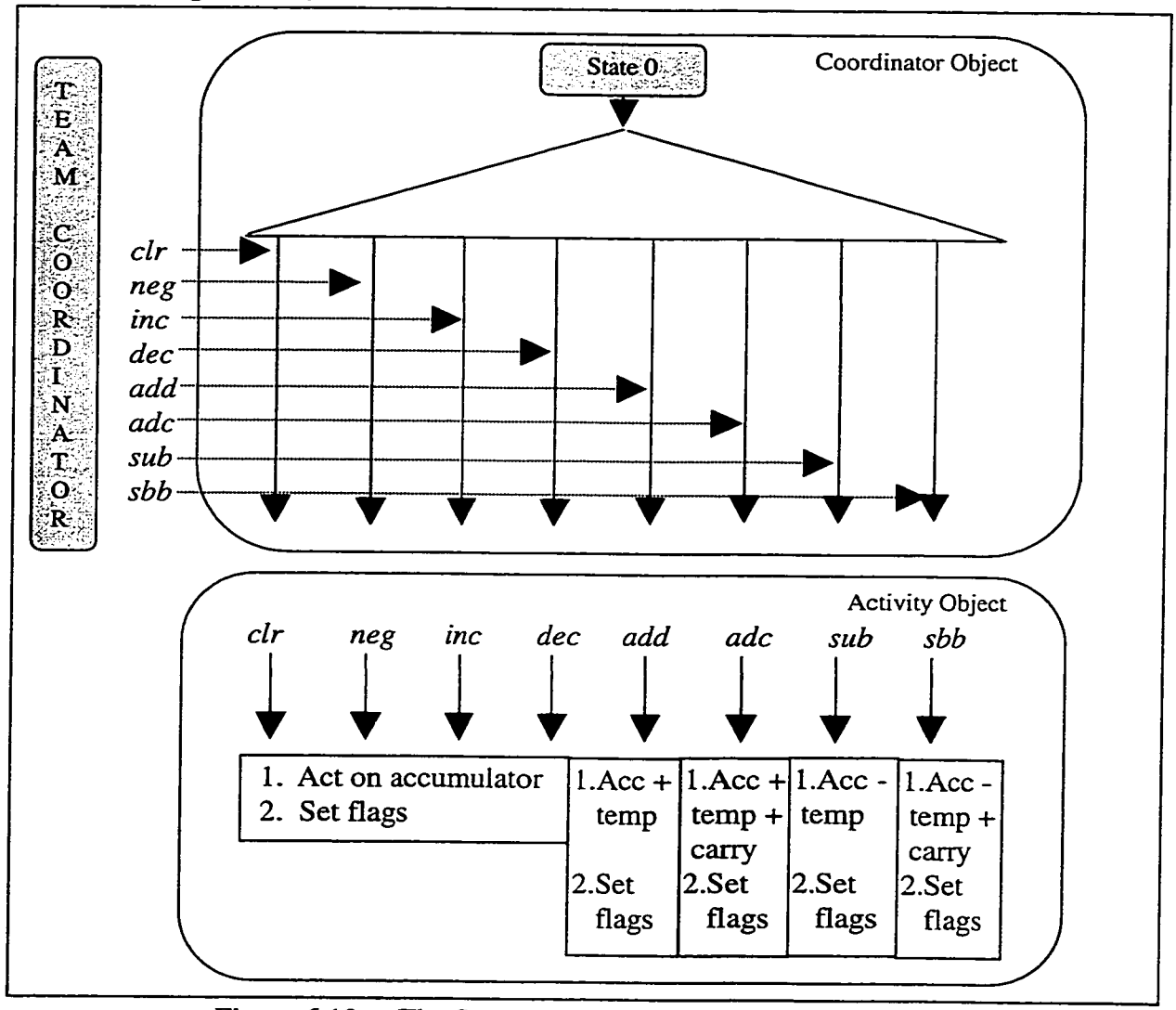


Figure 6.13 The State Machine of ALU Coordinator

Memory Responsive Unit

In the case of Memory Responsive Unit the incoming Requests have a certain dependency one to an other, relative to the *status of memory*. Each Request is translated as a public method in the Port Object. Than the *name* of the method is sent as an event to the Coordinator Object, which implements the interaction logic (figure 6.14). The action is taken or rejected based on the outcome from the Coordinator state machine.

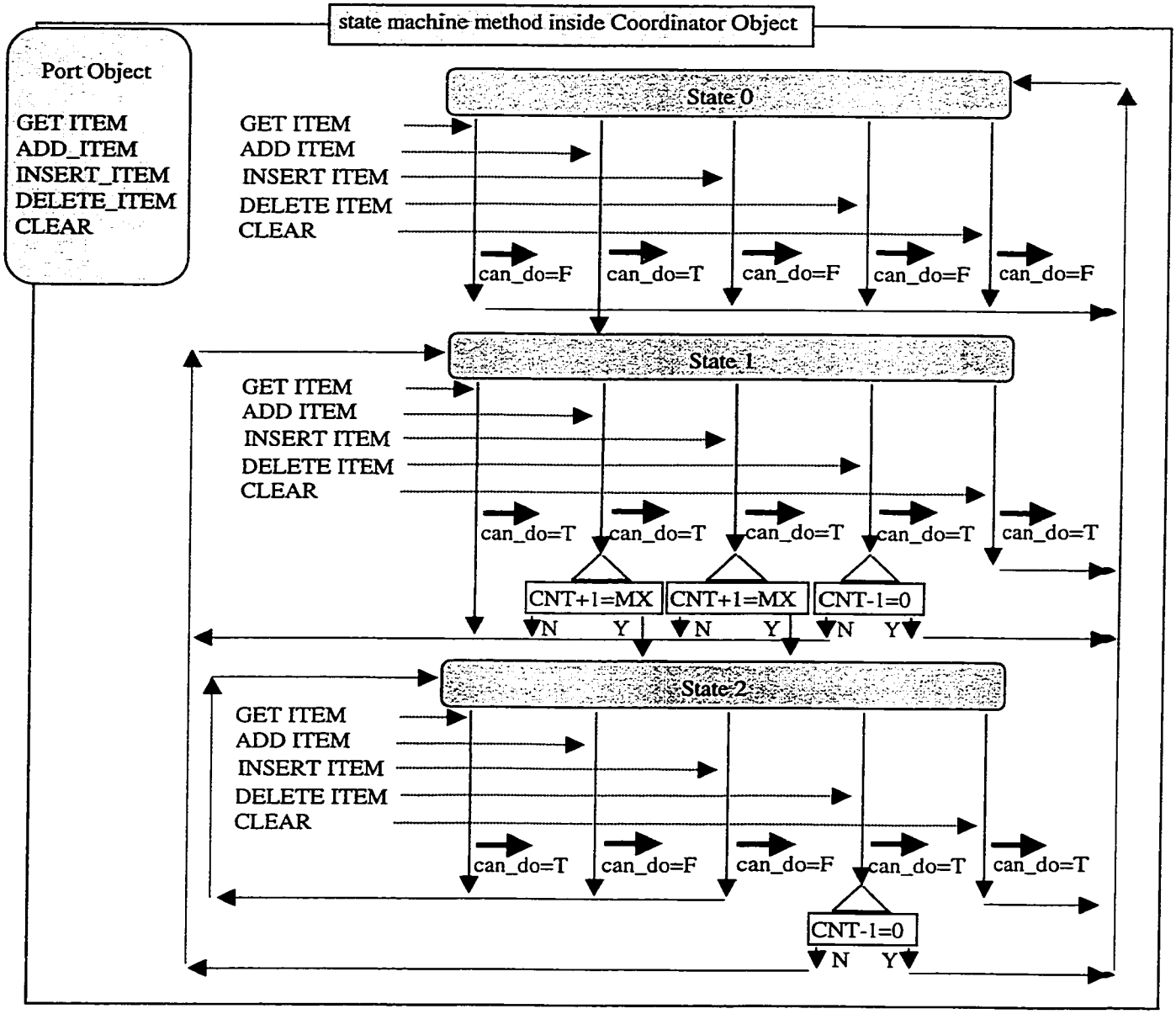


Figure 6.14 The State Machine of Memory Responsive Unit

We have here a case where the Coordinator does not look to the values passed with the methods. It establishes if a Request (or incoming method) is allowed or not by considering the history of interactions and the boundaries of the memory.

Registers Responsive Unit

In this example the use of a Responsive Unit is minimal. We chose however to have it for the sake of being prepared for the next processors that might have more register logic, which is also a way of presenting the 'Responsive Unit' technique. In other microprocessors the registers may be related and the requests to act on them may be correlated.

In this case we only set or get data in/from registers and they are not related ; therefore the actions do not interfere with one another. For these reasons the things we need are pointers to each register. It is easy to cut-and-paste one Responsive Unit in the beginning and if it is not of much use to shrink it during the design. Is it better to shrink the design of an area that initially was considered necessary, than to find ourselves later under pressure to extend the design with more objects. For the proof of our concept we used the Memory and ALC Responsive Units. This last one may be taken as a proof of non rigidity of our design approach; this idea actually is good to be underlined. Developers avoid methods requesting a solid format. Our model suggests to start with Responsive Units demanding more space, to finish eventually with less. This may be a happier end than the opposite approach, where designers have to add more and more objects with less and less defined interactions.

6.1.3 Implementation phase

We *separated the Logic of Interactions from Activities* ; Coordinator Objects contain this dialog, as it is necessary in different layers.
Therefore for implementation we have to consider the next major elements:

- The User Interface
- Coordinators, where we enter the data from Event-Response definitions,
- Activities, where we directly write code to implement simple requirements.

The code can be in a one-to-one relationship with the main User-System StateChart that belongs to the Team Coordinator, with Response Managers and with the Coordinators inside the Responsive Units.

The implementation of Responsive Units can be done using the suggested ready-written code, which will be completed and refined according to each case. As for the Architectural design, it may be considered quite typical. When we introduced the concept of Team Coordination, we suggested that Teams of Units can have a typical format : around a Coordinator Unit one may have Activity, Data and Port Units. Our case fits in this format : in the same order we have the Team Coordinator, the Arithmetic Logic Unit, the Memory & Registers Units and the User Frame (a special entity in Java).

With this format the exercise of analysis, design and implementation is quick and one cannot have many surprises. *The User* is able to see in a *language-independent format the Requirements* and we can convince him that *what we implement is what he sees*. The stepwise testing can start

from the implementation of the first layer, the User-System StateChart in the Team Coordinator. One can log each state, event and action and follow thus the evolution in time of the system. Such log is excellent for the

- Proof of Rapid Prototyping,
- User demo and feedback,
- Proof of a Correct Design,
- Record of normal use.

In our case we consider a permanent Text Field as a 'Feedback to User' that may take the log role, since Java applets are not allowed to write on disk.

We will follow the next steps for Implementation, as suggested in our theory:

1. Initially we cut and paste (from a service file) the Responsive Units for:
   the Team Coordinator Responsive Unit
   the Arithmetic Logic Responsive Unit
   the Memory Responsive Unit
   the Registers Responsive Unit.
   Inside Responsive Units the ready-written code will include:
   the Coordinator object, with a skeletal state machine,
   the Activity object,
   the Port object,
   the Data object.

2. We fill in the elements for the User Interface in the Java Frame:
   Text Fields, Buttons, Lists, Menus, Windows.
   Between them we define no interaction logic, since it will be defined in the Team Coordinator.
   The Events will be sent to the Main Responsive Unit, the Team Coordinator.

3. The user-system dialog can be written now in the Coordinator Object belonging to the Team Coordinator.
   We build the user-system dialog in a public method using the skeletal state machine from the ready-written code.
   In the *Team Coordinator* we add the Response Managers as private methods with state machines clearly implemented, driven as said by one specific event.

4. The completion of responses is done in the 'Problem Domain' Responsive Units. Between the Units there should be no interaction, as these are captured inside the *Team Coordinator*. Inside these Units we have to build the state machines of Coordinator Objects and the Activities.

In next presentation of essential implementation we concentrate on elements relevant to this thesis. Where we skip details we add the " *// ... details* " comment. The complete code may be found in the Code Annex.

1.Responsive Units

Inside a Responsive Unit class we put the internal objects and in the constructor we instantiate them.

The main Dialog Coordinator follows this rule.
Supplementary to other Responsive Units, it has pointers to all entities

```
class RU_DCoordinator
{
        OneAccMicro    Frame_mp;                    |  Pointer to the User Interface

        RU_RandomAccsessMemory  RU_RAM_p ;          |
        RU_Registers            RU_Regs_p;          |  Pointers to the other
        RU_ArithmeticLogicUnit  RU_ALU_p ;          |                Responsive Units

        AM_Coordinator          AM_Coor_p;          |  Pointers to the internal
        AM_Port                 AM_Port_p;          |                Objects
        AM_InfoBase             AM_InfoBase_p;      |
        AM_Act                  AM_Act_p;           |

        Event event;
        int eventArea;
        RU_DCoordinator(OneAccMicro OneAccMicro_p )
        {
                Frame_mp        = OneAccMicro_p;the RU_DCoordinator is
                                                instantiated in main
                                                Frame,i.e. the Java User Interface.
                                                Its pointer is received by the constructor.

                AM_Coor_p       = new AM_Coordinator(this);
                AM_Port_p       = new AM_Port(this);
                AM_InfoBase_p   = new AM_InfoBase(this);
                AM_Act_p        = new AM_Act(this);
        }
        //...details
}
```

The Memory Responsive Unit,
the Arithmetic Logic Responsive Unit and
the Registers Responsive Unit
follow this model. *Note that there are no direct pointers between them.*

We show here only the first one:

```
class RU_RandomAccsessMemory
{
        OneAccMicro Frame_mp;
        RU_DCoordinator    RU_DCoordinator_p ;
        RAM_Coordinator RAM_Coor_p ;
        RAM_Port       RAM_Port_p      ;
        RAM_InfoBase   RAM_Info_p      ;
        RAM_Act        RAM_Act_p       ;

        RU_RandomAccsessMemory( OneAccMicro OneAccMicro_p )
        {
                Frame_mp = OneAccMicro_p;
                RU_DCoordinator_p = Frame_mp.RU_DCoordinator_p; // to main Coordinator
                RAM_Info_p = new RAM_InfoBase(this);
                RAM_Coor_p = new RAM_Coordinator(this);
                RAM_Port_p = new RAM_Port(this);
                RAM_Act_p  = new RAM_Act(this);
        }
}
```

2. The User Interface

In Java we have the class Frame that builds the main Window. We inherit it for our frame.

```
public class OneAccMicro extends Frame {

        RU_DCoordinator      RU_DCoordinator_p;   |  pointers to all Responsive Units
        RU_RandomAccsessMemory RU_RAM_p     ;     |  since they get instantiated in Frame.
        RU_Registers         RU_Regs_p   ;       |
        RU_ArithmeticLogicUnit RU_ALU_p     ;    |

        Button Edit     ;
        TextField progCounter;
        // ...details : more buttons ,TextFields   and all User Interface objects.
        public OneAccMicro() {  //constructor
                // ...details
                public boolean action(Event event, Object arg) {
                                // ...details
                                //  passes Events to RU_DCoordinator:
                        RU_DCoordinator_p.AM_Port_p.action( event);
                }
        }
}
```

3. The user-system dialog

Inside main Dialog Coordinator Responsive Unit (playing the role of a Team Coordinator) we have the AM_Port class through which events are passed and prepared for the class AM_Coordinator. In this last class we call the StateAuto() public method that implements a state machine, reflecting the main User-System Dialog.

```
class AM_Coordinator
{
        //...details
        void StateAuto()
        {
                event = Manager_mp.eventArea; // manager is the RU_DCoordinator
                switch( state )
                {
                case 1:
                        switch(event)
                        {
                        case EDIT:
                                Manager_mp.AM_Port_p.say( " EDIT accepted" ); //feedback
                                Manager_mp.AM_Act_p.EnterProg();       // call Action
                                reset();
                                state=2;
                                break;
                        }
                        break;
                case 2:
                        switch(event)
                        {
                        case APPEND:
                                Manager_mp.AM_Act_p.Append();
                                state=2;
                                break;
                        // similar for  INSERT,          DELETE,
                        //              APPEND_DATA,  DELETE _DATA,
                        //              SAVE_PROG,       ENTER_SAVED

                        case LOAD_PROG:
                                listCount=actMemory.countItems();

                                if ( listCount> 0)      state=3;
                                else                    state=1;     // nothing to load, need enter
                                break;
                        }
                        break;
```

```
        case 3:
                switch(event) {
                case EXECUTE:   // we take the licence here to reduce the State
                                // number from the Design, in a practical manner
                        break;
                case AUTORUN:
                        ResponseManager_AutoRun();
                        state=3;
                        break;
                // similar for INSTRSTEP, MICROSTEP, EDIT
                }
                break;
        }
}
```

For all Edit-related events the Response Managers are Actions implemented as methods in the Act Object, since no state machine is needed to implement them.
They are a packet of sequential functions as seen at Design Phase.
The Memory Responsive Unit will be implied for completing the responses to these events.

From the group of Execute-related events, the first two are simple functions with 'while loops'.

The MICROSTEP is a very complex state machine.
We build it in the same *'double switch' manner as the previous state machine* . It relates very easily, in a *direct one-to-one relationship with the Design model.*

For this very reason, which is a major advantage of the method, we do not need to present it here as there are no other important ideas to add to the design.

Again we mention that the decision of implementing the interaction logic as state machines in the Dialog Coordinator gave us this excellent Design-Implementation relationship.

4. Response completion in the 'Problem Domain' Responsive Units

The **Memory Responsive Unit** receives the events

GET_ITEM
ADD_ITEM
INSERT_ITEM
DELETE_ITEM
CLEAR

The Coordinator Object has as main public method the state machine that directly implements the Design Model:

```
public boolean StateAuto( int mem_event )
{
        boolean retToDo = false;
        switch( mem_state )
        {
        case 0:
                // similar with case 1,  based on design phase
        case 1:
                switch( mem_event )
                {
                case GET_ITEM:
                        retToDo = true;
                        break;
                case ADD_ITEM:
                        retToDo = true;
                        if( RAM_Info_p.programList.countItems()+1 ==MAX_MEM)
                                mem_state = 2;
                        else
                                mem_state = 1;
                        break;
                // similar for
                // INSERT_ITEM
                // DELETE_ITEM
                // CLEAR
                }
                break;
        case 2:
                // similar with case 1,  based on design phase
        }
        return retToDo ;
}
```

The events are actually the public methods from the *Port Object*.
We chose an example showing how the state machine from Coordinator is called, with the respective event. The *return* value from state machine tells us if we can do the action.

```
public void addItem( String sToAdd)
{
        if(  RAM_Coor_p.StateAuto( ADD_ITEM ) ) {
                memActed = true;
                programList.addItem( sToAdd );
        } else  {       //set warning
                memActed = false;
        }
}
```

We finish the action with *programList.addItem (sToAdd)* and we do not call the Action Object since the case is solved directly. We are happy to shrink the initial design instead of facing some need of expansion.

The **Registers Responsive Unit** implements directly the registers and as there is no correlation between them (as opposed to the STACK Microprocessor) we do not build a complete Coordinator for it.

The **Arithmetic Logic Responsive Unit** has a simple coordinator that calls the Activities from Activity Object, which are some of the assembly language instructions described at design phase. The model is similar to the other Responsive Units. Since there is no supplementary idea for the proof of concept, we stop here the implementation phase even if there is a certain quantity of code in the Activities.

Observing the code with its details we get a positive feeling about the idea of making a distinction between the interaction logic implemented in Coordinators and Activities. Also the use of Responsive Units gives us an initial place where we can put each element. After this exercise of development, in the Conclusions Chapter we will consider more the benefits of using the new concepts.

6.2 RISC Microprocessor Simulator

An important issue in software engineering is the ability to modify a system for an other use or for an extended one. Code read-ability is a highly desired quality when we need a change.
A major part of read-ability is the understanding of the interaction logic between components.
We speak of 'legacy' software when the code is a labyrinth hard to touch. There is the tendency to think that older systems are like that but as we speak much code is written in a spaghetti Object Oriented mode, becoming 'legacy' at birth.

We believe that with the Dialog Coordination and Responsive Units a change is easier to be done. The read-ability is enhanced by the compact description of interaction logic between components, but it is something more than that. A change often implies new objects and new interactions. If the interaction logic is localized, the change is easier articulated and risks are minimized. We like to think about the Dialog Coordination design that it is fundamentally a 'design for change'.

With the following microprocessor simulators we wish to show that the method allows easy and clear modifications. The presentation will concentrate on *differences* in requirements and on the manner of implementing the changes.

6.2.1 Architectural phase (changes)

6.2.1.a Requirement Acquisition

General view

RISC Microprocessors have several characteristics from which we select a major one : the use of a bank of registers instead of temporary and accumulator registers.
The registers in bank can be used in any order. They need to be specified with the assembly instruction that uses them.
This machine is a 'load-store' type as the access to the memory happens only through *load* and *store* instructions, all others acting on data from specified registers.

Refinement of Requirements (changes)

In the list of registers we remove the temporary and the accumulator registers and add a *Bank of 8 Registers:*

- Counter PC
- Memory Address Register MAR
- Memory Buffer MBR
- Data Pointer DP
- Instruction Register IR
- Bank of 8 Registers REGS
- Flags:Carry, Negative, Zero C N Z

In the list of assembly functions there are new Instructions without data :

add adc sub sbb .

They take values from designated registers (as add r2 r3 :r6).

The User Interface changes (figure 6.15) are in the registers area and in Edit Utils window where we place Registers I, J and K.

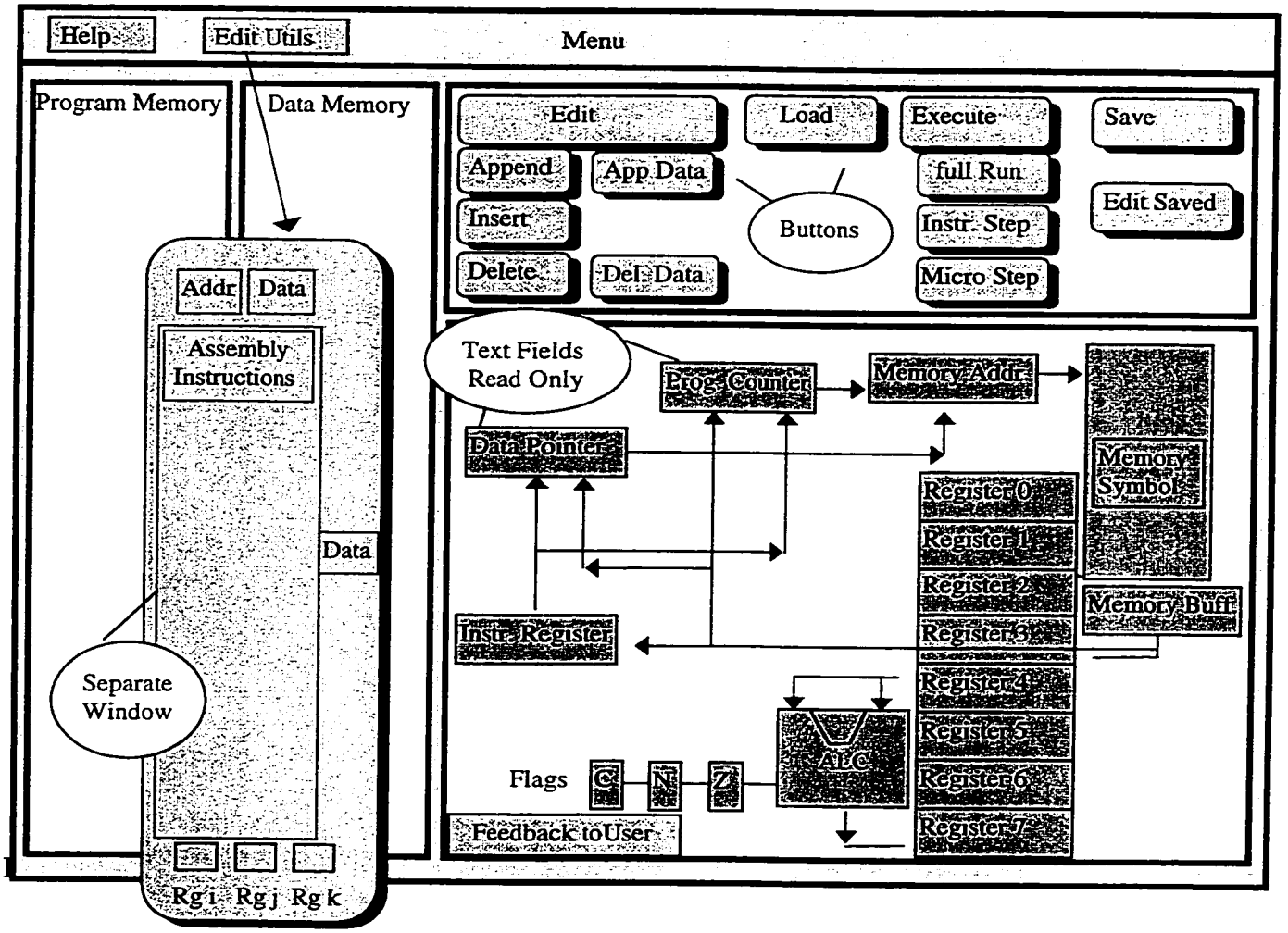


Figure 6.15 User Interface for RISC

Rg_i and rg_j are used for two operand instructions and rg_k is used for the result of operations and load / store. Their values need to be written near the instruction in the Program Memory as needed. At the time for Append or Insert the software has to decide, looking to the instruction selected in Edit Utils, what registers have to be considered.

6.2.2 Design Phase (changes)

To solve the issue with registers from previous paragraph we add at **Response Managers for APPEND and INSERT** a new action, before the call to the Memory Responsive Unit.

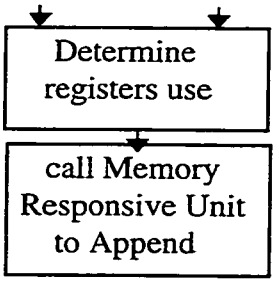


Figure 6.16

In the **MICRO_STEP Response Manager** we change State 4 by adding four more Instructions without data, **add, adc, sub, sbb.**
They will take their data from registers in the ALC Responsive Unit in the Activity Object.

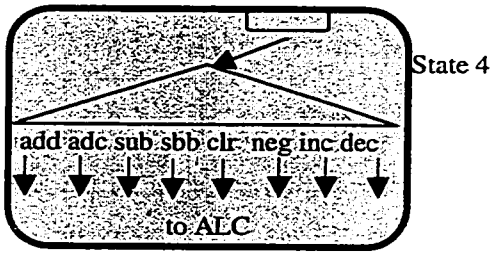


Figure 6.17

MICRO_STEP Response Manager continues with State 5 , 'Instruction with Data' where in the last case we

- take out the **add, adc, sub, sbb** choice and
- change at *load* instruction the 'write accumulator' action with 'write Reg_k'.

Store instructions remain the same in this sub-state 0 of State 5 , 'Instruction with Data'(fig 6.18)

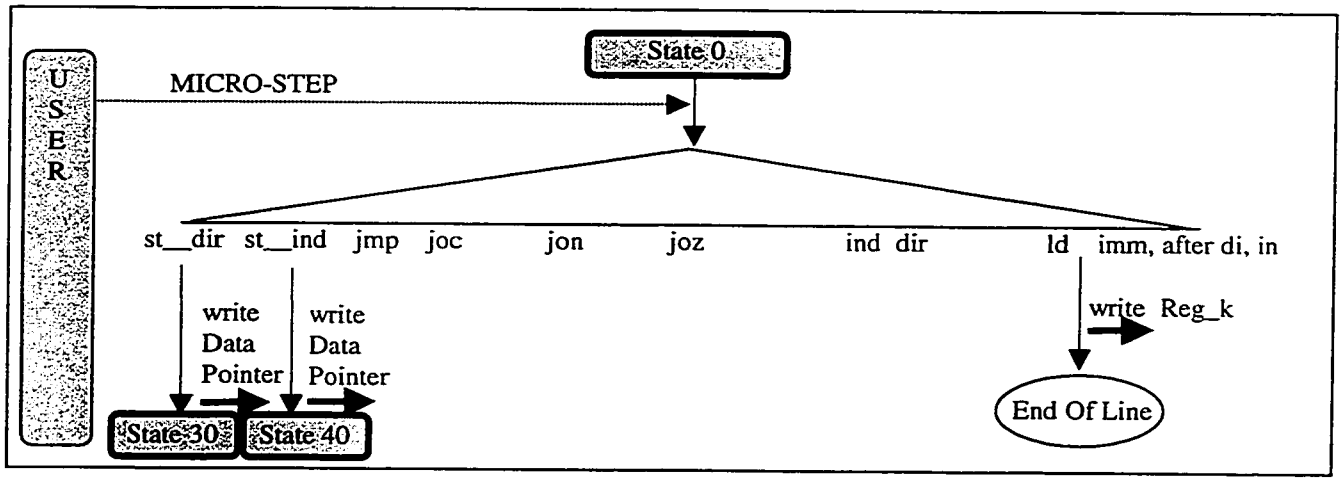


Figure 6.18 Changes for Instruction with Data

In State31, for *Store* instructions we change 'accumulator' with **'Reg_k'** in the action.

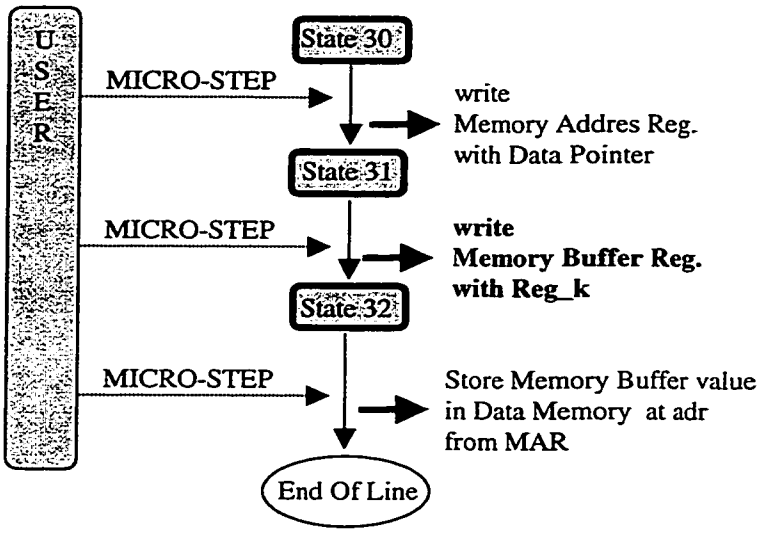


Figure 6.19

The **ALC Responsive Unit** needs to be changed in the Activity Object (Figure 6.20) so that instead of using the Accumulator and Temporary registers we act on the Registers Bank. Registers numbers are taken from the instruction line in the Memory Program, as introduced at Edit time.

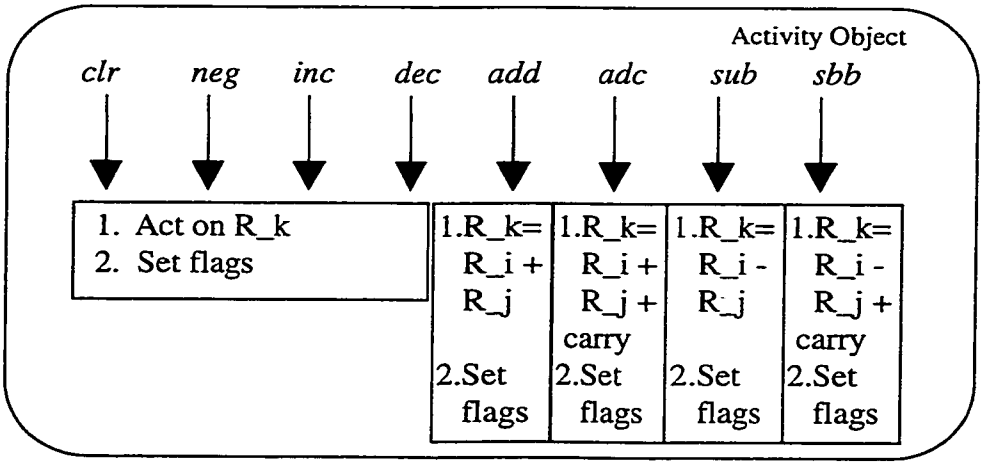


Figure 6.20 ALC changes in the Activity Object

These are the main design changes implied by the requirements. As one can see, they are precisely localized, being either Interaction Logic or Activity. The design is read-able and the implementation is as a direct copy, therefore the changes in implementation are easy to make.

6.2.3 Implementation phase (changes)

In **the User Interface** :

- add new TextFields for *Registers i,j,k* in the Edit Utils window.
- remove accumulator, temporary registers
- add *Registers Bank*

In **Response Managers for APPEND and INSERT** a new method is built and called to determine the *use of registers*, before the Memory Responsive Unit is reached.

The **MICRO_STEP Response Manager** :

State 4 : 'Instructions without data' : -add cases of **for add, adc, sub, sbb.**
State 5 , 'Instructions with data' :
- remove **add, adc, sub, sbb**
- change 'accumulator' with **'Reg_k'** at *load* instruction.

State 31: - change 'accumulator' with **'Reg_k'**.

The **ALC Responsive Unit**, in the Activity Object:
change Accumulator and Temporary registers with
- **Reg_i , Reg_j** for operands
- **Reg_k** for the result of operation.

6.3 STACK Microprocessor Simulator

In order to build the Stack Microprocessor we start from Risk Microprocessor Simulator.

6.3.1 Architectural phase (changes)

6.3.1.1 Requirement Acquisition

These machines have a Stack of Registers, as the Registers Bank from Risk Machines.
Specific is the fact that the operations have no address, since:

load and *store* operations act ever at the StackTop,
forcing a *push* or respectively a *pull* of data in the Stack,
'one operand' instructions act on the StackTop
'two operand' instructions act on the StackTop and StackTop-1.

Therefore there is no need for the Text Fields Reg_i , Reg_j, Reg_k in the Edit Utils window.

6.3.2 Design Phase (changes)

In **Response Managers for APPEND and INSERT**

we *remove the method* used to *determine registers use.*

In **MICRO_STEP Response Manager** inside State 5 , 'Instruction with Data' (Figure 6.21)

- add at *load* a *PUSH STACK action.* It calls the Registers Responsive Unit.
- add a *check for PUSH DONE* . If TRUE do the next 'write' action.
  If FALSE inform the User.
- change at *load* the 'write accumulator' action with 'write **StackTop**.

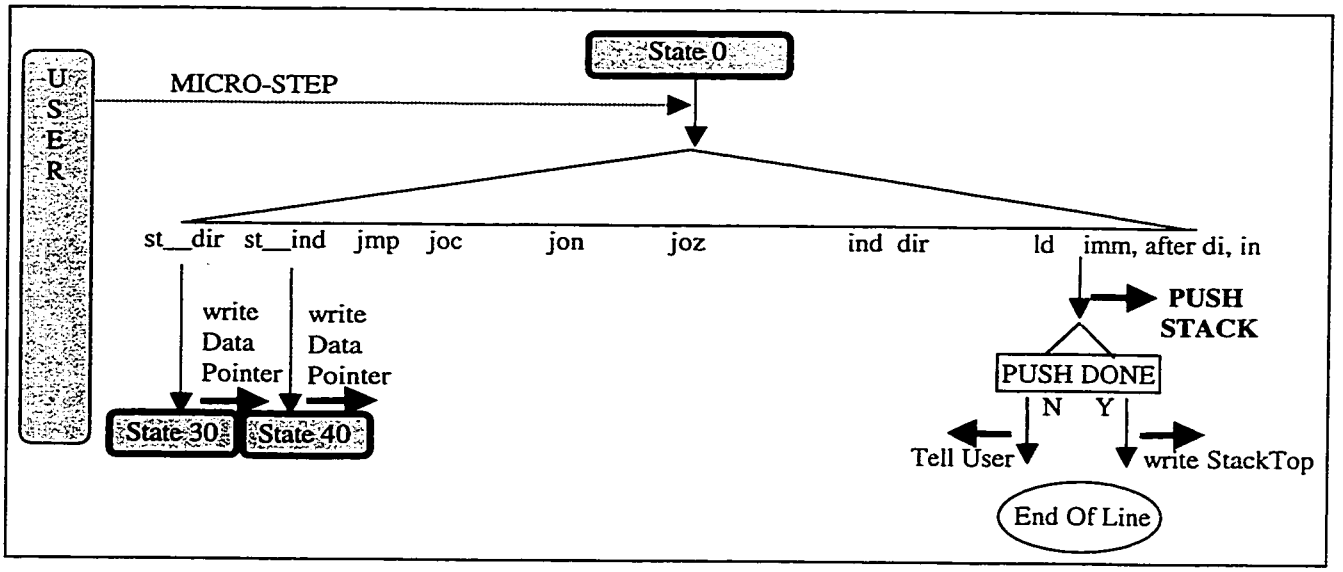


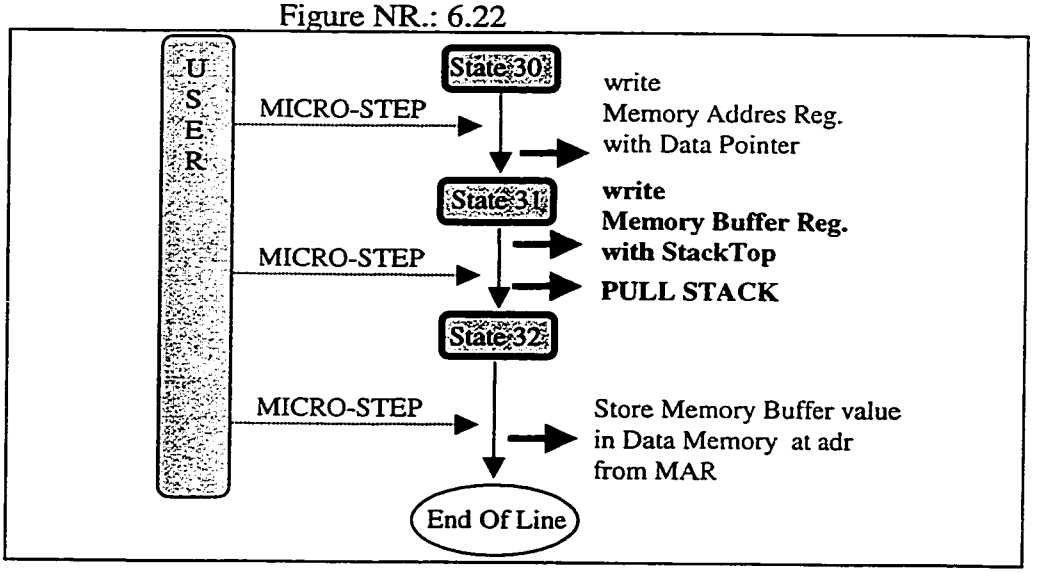
Figure 6.21 Changes for Instruction with Data

Still in **MICRO_STEP Response Manager** inside State 5:

in subState31:

- at *store*, change 'accumulator' with *'StackTop'* in the action.
- add *PULL STACK* action. It calls the Registers Responsive Unit.

Figure NR.: 6.22



The **ALC Responsive Unit** needs to be changed in the Activity Object:
Replace in all cases

R_k with *StackTop,*
R_i with *StackTop,*
R_j with *StackTop-1* .

Registers Responsive Unit

In the previous Microprocessors, the registers were independent and so were the actions on them. Therefore no Coordination was necessary in this Unit. Now the Stack concept relates the registers as they exchange values on PUSH and PULL actions.

At PUSH the values in registers move from Reg_0, StackTop, to the StackBottom
the new value is loaded in Reg_0 .
At PULL the value from Reg_0, StackTop, is stored in memory
the values in registers move from StackBottom to the Reg_0, StackTop
Reg[StackBottom - index] = Reg[StackBottom - index -1]

For the proof of concept we build a state machine in the Coordinator Object of the *Registers Responsive Unit*, even if in this case a minimization can be done. However, if we think of future extensions, minimization should not be preferred.
The Coordinator will represent the interaction logic of the public methods. We have three 'events' : PULL, PUSH, RESET coming from the Team Coordinator through the Port Object as public methods. Their *name* becomes the name of the events passed to the state machine of the internal Coordinator Object.

The Actions happen if we get 'can_do =True' after calling this state machine; the values of registers are moved down or up in the Stack (Figure 6.23).
For other simple applications cases where we do not need a complete Responsive Unit, we can still add inside an object a method that implements a state machine as a Coordinator.

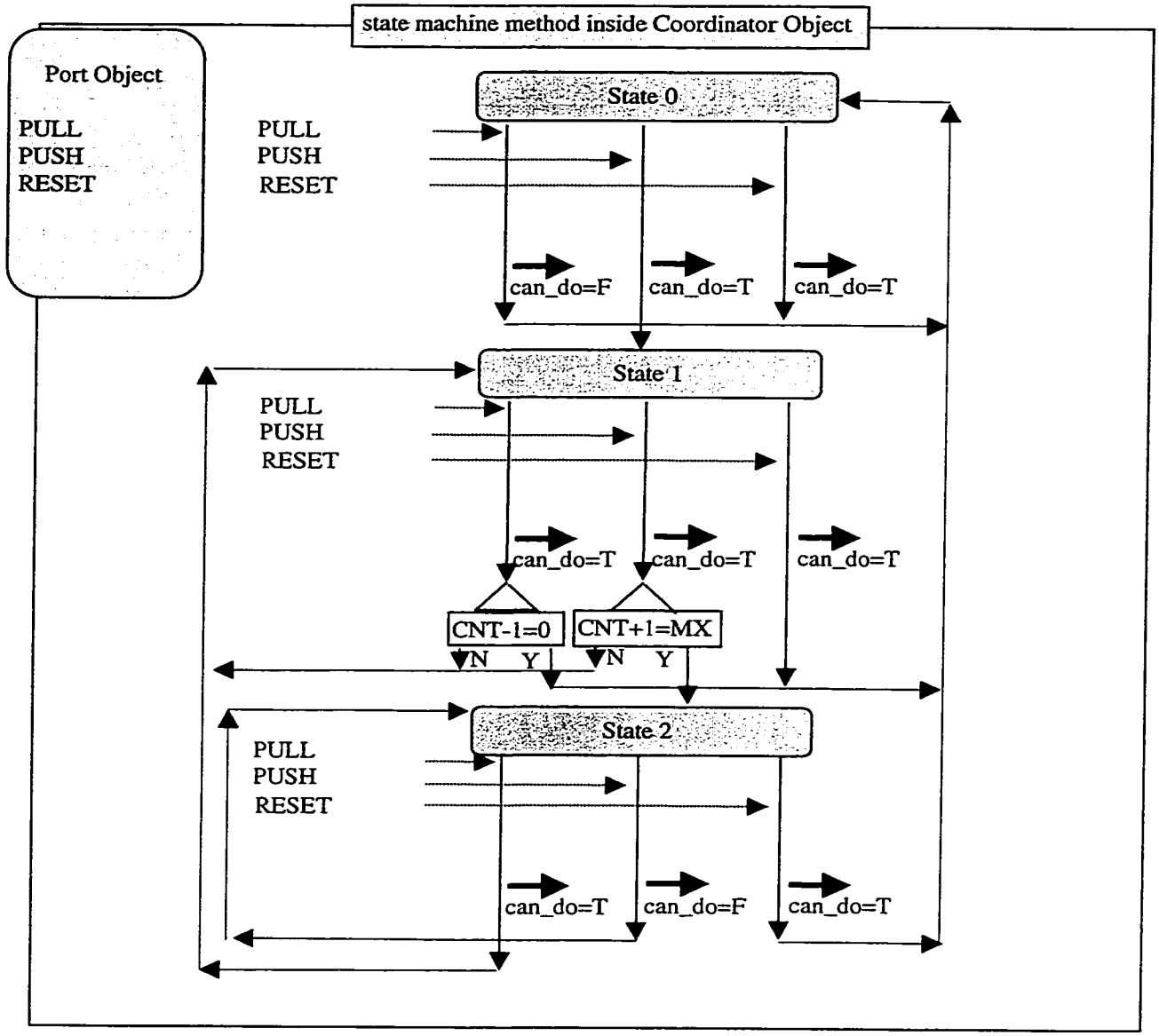


Figure 6.23 The State Machine of Registers Responsive Unit

6.3.3 Implementation phase (changes)

The implementation is here also a direct copy of the design. To describe the changes in words means a repetition of the design phase. Testing for the new elements is precise and Regression Testing can be minimized as the impact is well contained.

The 'design for change' as seen in this exercises appears to be better done if we consider the change mainly as an undefined & unused interaction with new objects. Therefore we prepare for change if we compactly define the existent interactions in a Dialog Coordinator object. In the Conclusions Chapter we will expand and make more such comments.

6.4 School-of-Microprocessors

The place from which the three simulators can be started is a web-page that has an applet named 'School of Microprocessors'. We have here a supplementary 'core microprocessor' simulator (built also with Responsive Units), an applet for absolute beginners. The four buttons for simulators and a reset button can be called in a specific order. In this applet is also added a suggestion for a way of teaching at distance, described a step further.

A Responsive Unit is used as an *Agreement Coordinator* that deals with these issues. It is a place that includes the needed interaction logic and which can accommodate further changes, having the format of a Unit (an object composed of a Coordinator, Activities, Port and Data object). It may communicate with the Teams of Units forming the Microprocessors and when needed the *Agreement Coordinator* can become an interaction bridge between them. The design of this Unit is done directly on the top of the typical format for Responsive Units and the implementation follows. The predefined concept of distinguishing the coordinator from the other objects facilitates quick design decisions and keeps the door open for new elements.

Specifically, the School of Microprocessors deals in this application with the following:

◊ Defines, initiates and coordinates a user interface in which are contained
  - Button objects for the start-up of each microprocessor simulator.
  - A reset button for restarting the 'school' from any point.
  - A text field for a message to the user referring to the momentan state and event of the 'school' coordinator.
  - A text field that can display the last event from a simulator.
  - A text field that can receive a user input with an event equivalent with a GUI event. The last two text fields can be used for distance learning, as presented further.

◊ Initializes and allows a specific order of execution for the simulators, based on the order in which a student may better understand microprocessors.
  It keeps track of the deletion of simulators to maintain the explained sequence.

Without this agreement coordinator the sequence of execution had to be maintained through each simulator 'team' and the 'global' user interface had to belong to one of 'teams'. This 'coordination-less' procedure is often considered a better approach since it saves a class [Riel]. In our exercises we found that by adding an agreement coordinator, one adds more clarity/visibility; besides, even the coding quantity may decrease, since the needed logic is not dispersed. This argument is generic for any coordination type and we made more comments about it in the 'Dialog Coordination' chapter.

We further propose a supplementary feature which is facilitated by the coordination approach. Each event from GUI is transformed in a word and then applied to a Unit and thus the GUI may be replaced by a text input. This made us think about a way of teaching at distance, where one

user is a teacher while others are students. For this, in the 'School of Microprocessors' applet we placed, as previously mentioned, two text fields:

- A text output, where appears as a word the last event activated by the teacher in his GUI.
- A text input, where students enter a word-event that activates the simulator without the GUI.

The teacher can cut-and-paste the event-word in a chat page and students can pick it from there. An extended version could link internally a chat page with the 'School of Microprocessors' applet to have an automated transmission of events. It would require a CGI script on an ISP server. We consider that for such applications future research can be done to observe the benefits of Dialog Coordination concept and of using Responsive Units.

The Case-Study we developed was intended to present the concepts introduced in the thesis and to observe their advantages and drawbacks. Therefore we built an Agreement Coordinator (the 'School' applet) as a Responsive Unit that instantiates three teams of Units.
Each of those Teams (the microprocessor simulators) has a Team Coordinator Unit and three other Responsive Units. We followed two major issues : the OOD for certain event-driven systems and the ability of modifying a system. The outcome of this exercise is presented in the 'Conclusions' chapter.

Chapter 7

Conclusions and Future Research

7.1 Focus on Dialog Suggesting a Dialog Coordination Object

After this research, we consider that setting the **focus on the dialog between objects** was a reasonable thing to do in our quest to better capture / define the event responses.

The problem that triggered our interest for a better organized dialog was the fact that the response-oriented interactions between objects may be described only partially. Since the behavior of objects may be defined with state machines, we observed that describing the interactions of these machines is a complex theory in itself. Such theory is difficult to follow in a typical development environment.

Independent of this problem, we perceived that the inter-objects **dialog is essential** for the design of objects and systems ; thus, a good dialog design should be a purpose in itself. The associated thought that **dialog implies coordination** (implicit or distinctly implemented) proved to be a good motivation for our approach. The choice of distinctly implementing the dialog coordination as a public component was sustained by the theoretical and experimental background from Coordination Oriented Design. That philosophy, of **implementing the coordination separately from execution tasks**, led us to have a similar vision for the inter-objects behavior modeling.

In our view, the **response-oriented interaction logic between the objects of a group is useful to be implemented in a distinct object**. The behavior of this object is directly represented by a state machine and thus we use a hierarchical state machine design for the system behavior. We could consider that a group of objects, that work together for some group event-responses, are related through such an interaction object. Inside this one at design time we could place the intended interaction logic between the group objects, to offer the right response to each group event. We can do this because, as said, in software **everything is built by a kind of words, correlated** in some way ; *so are built the objects and so are built the interactions*. Therefore the interactions with their logic *can* be defined in an object. If this can be done, it seems reasonable to use the possibility and to define the **dynamic dialog in a module by a dynamic language**. The other choice is to explain the dialog only in the static language of a document.

We further perceived a reason that even *demands* the definition of the interaction logic in a distinct module. The reason was that the objects of a group work together for **a goal**. A *goal oriented coordination* is better implemented if there is a *location* (as a new object in the group) for the **goal oriented values** and for the **goal oriented interaction logic** between the group objects. The group state and the requests to the group are also demanding such an object. Thus, **a dialog coordinator can facilitate the design of a goal oriented coordination** for the objects in a group. The behavior of the coordinator is coded as state machine, which calls for 'activities' in the objects of the group. The implementation of this state machine is a direct reflection of the state machine from the design document, which defines the main behavior of the group.

7.2 Dialog Coordination Types Lead to Responsive Units and Teams

The three types of dialog coordination, namely
- Master-Slave coordination
- Team coordination
- Agreement coordination

represent a theoretical distinction with a helpful practical outcome, as further outlined.

The domain of objects that deal directly with event-responses was separated in two areas: *one of light interactions and one of strong interactions.*

A group of objects related through strong interactions and their *Master-Slave coordinator* suggested the idea of a *Responsive Unit.* More Units related through light interactions and their *Team coordinator* were grouped in a Team. An *Agreement coordinator* may be used to establish relationship agreements between Responsive Units, at system initialization or reorganization. Since it does not belong to the event responses domain, we conclude that its design is a distinct issue and it should be distinctly implemented.

Even if we put the emphasis on the dialog between objects, *the outcome of this research has some solid elements, as the dialog coordinators and the Responsive Units.* By using such a Unit as an object composed of a number of objects and their coordinator, one can build the structure of the system based on the desired behavior of the object groups. *The behavior inside a Responsive Unit can be well defined by the state machine of the coordinator*, while the other objects are mainly executors. Thus, one can precisely define the event responses for each Unit. The behavior between Units, that builds the system behavior, *can be described in the state machine of the Team Coordinator* (a Responsive Unit itself). The dialog between Units becomes *much simpler* than the possible dialog between all system objects. Thus, the dialog coordination for the Team(s) of Units may be accurately defined since it refers only to this higher level dialog.

Inside each Responsive Unit we considered that beside a coordinator and activity objects one may typically add *a data object* (containing the public data of the Unit) and *a port object* (a Unit interface). This model can really provide from the start of the design clear locations for distinct issues, related by the dialog coordinator.

One can use some ready-written code (as a template, but not in the C++ special meaning) that describes the Responsive Unit object, having in composition a coordinator object, several activity objects, a port object and a data object. The skeletal state machine added inside the coordinator helps since it can be easily filled with events and actions, specific to each Unit.

As the object concept has been a good step for the design organization, so *these Units may be used as a next step* for a quick and clear event-driven design. The Responsive Unit is in a way a mega-object with the interaction logic distinctly defined in the coordinator object. We observed that it helps for a good response definition, for a fast design and for easy modifications. The Units could be specialized in a Team in the same manner the objects specialize inside a Responsive Unit. *A Team Coordination Unit defines the dialog between Activity Units and eventually a Port Unit and a Data Unit.* This behavior driven organization separates well the design concerns, offering a generic design approach. The application specific design is done by completing the state machines of the dialog coordinators and by further specialization of objects.

7.3 Remarks on the Development Process

In the **development and implementation of the three microprocessor simulators** we could apply these concepts directly and in a clear fashion. As stated in the beginning, we wanted to introduce a useful enhancement for the interaction logic modeling. It is believed that the dialog coordinators and Responsive Units can be understood *without a long learning curve* and can be implemented *without much overhead.* These impediments are typical for the tools for event-driven system design, which often are simulators based on some sort of state machines. The simulation is supposed to help for the interaction design and to prove its correctness. Because of this simulation necessity the tools are very complex and add much overhead, this being a disadvantage for embedded systems.

The *definition of the response-oriented interaction logic inside the dialog coordinators simplifies the understanding* of the system. It also relates the behavior design using state machines to the implementation, increasing thus the traceability in the development. A first *'simulation'* can be done on the new *system itself* when the Team Coordination *Unit is ready,* before other Units are built. The Team Coordinator state machine receives events and its actions are events for state machines of the Units. A *second exercise can take place when the coordinators* from the other Responsive Units are built, before all the other objects are finished. Such 'simulation' is actually *identical with a rapid prototyping and also corresponds directly with the testing* of the system.

The simulation is tightly related with **testing** in this approach, since the *simulation is done on the product.* For detailed testing, in the dialog coordinators we could temporary add logging facilities for events, states and actions. This way one can quickly delimit a problem at the level of a Responsive Unit and further at the object level.

It is difficult to compare **the speed of development** for different design approaches, but according to the author's previous experience this method offered a fast design environment.

Key advantages were

- The definition inside dialog coordinators of the goal-oriented dialog between objects.
- The distinction between light and strong interactions leading to the use of Responsive Units,
- The use of predefined Responsive Units, with predefined generic objects,
- Last but very important is the use of *a state machine to coordinate the dialog between interacting state machines*, as applied in Team Coordinators.

Since *testing is part of development*, the advantages of testing play also their part in measure of speed. In our case study the easiness of simulation and testing helped a lot for the overall timing.

The use of dialog coordinators with Responsive Units also facilitates the **modifications or additions** necessary for system evolution. A *'Design for Change'* concept should follow the *'Design for Reuse'* concept, that was promoted by OOD. The concept of system dynamics should be extended to the ability of the system to accept new elements or to accept modifications. For such a definition, the importance of goal-oriented dialog coordinators would be even greater. As changes require new interactions, the definition of the response-oriented interaction logic in an object offers clarity and localization of change. The risk of disturbing other interactions is also much reduced, with implications in quality and testing.

The second and third microprocessor designs started with a precise location of the new interaction logic required for the new responses. Dialog coordinators helped to clearly express the changed dialog involved by the modified request-responses and by the eventual new components. We could thus quickly apply the modified requirements. The use of Responsive Units implied a behavioral localization for the changes, minimizing ripple effects in other areas. The new interaction requirements could be translated directly in new events, states and actions in the respective state machines. The master-slave coordinator inside Units ensures a complete definition for event responses at Unit level and the changes can also be completely defined. When a response change implies a change in the Team dialog, the Team Coordinator can accommodate its description.

Also the third processor design, starting from the second one, could be done in the same easy way. In some other designs, often a change inherits the troubles introduced by a previous change and a *modification after another modification is even more difficult and risky*. In our case study, the second processor design maintained the qualities of the first one ; the third processor design followed smoothly. We had such a result because the response-oriented interaction logic was localized in the dialog coordinators of Responsive Units. The **simulation** had to be done only for the modified Responsive Units, with exact steps defined by the new dialog coordinators. **Regression testing** was minimized and no risks were involved. In many systems where the dialog is less explained, regression testing must be very extensive to cover the cases of interactions that are less understood. In those systems the testing is supposed to complement the interaction design, as a 'proof of change' concept. In our case study the need for such a back-end proof was avoided, this being another advantage of the method.

About the **speed of changing**, we noticed that *understanding the object dialog* with its responses from the first project was an important speed factor for the next design. *Dialog coordinators offer*

good visibility for event-responses and offer the trust that the change is well done. The author, as every software engineer, experiences in the daily design/change activity that valuable time is spent for understanding of interactions. A compact definition of the goal-oriented interaction logic in a dialog coordinator significantly increases the code readability. Also the speed is increased by rapid prototyping : the new dialog for new responses may be verified in coordinators before any eventual new components are built. Another advantage for speed of changing was seen in the quick delimitation of the change in the necessary Responsive Unit. Inside a Unit, a modification means a dialog coordinator change and/or a specific object change. This format offers a rapid pinning of the new dialog and the new object. Yet another advantage, a major one, was seen in the definition of the interactions between Units state machines by using the Team Coordinator. This has its own behavior defined as a state machine through which the Team responses are defined. To define new or different responses, the changes of the interactions between Units state machines could be easily done in the Team Coordinator. It also relates (in our projects) to the user interface and therefore the changes in interface requirements could be placed in its state machine.

As a generic conclusion about changes, we note that they are done by modifying the **interaction logic in order change the responses** and eventually to connect new elements. Thus, such a design *with Responsive Units and Team Coordinators can be considered as being 'design for change'*.

The **overhead that seems to be added** by using dialog coordinators and Responsive Units is *compensated at system level and at modification time*. Often in other designs the lack of dialog coordination allows the system to grow in unexpected ways as the design evolves. Modifications further complicate the system, especially for event-driven designs when event responses are only partially defined. Sometimes in the beginning of design we had the perception that using the dialog coordinator is a (short) waist of time and resources. When the system grew, the perception was that we gain time and coding space by a good organization ; the new elements and their interactions could be nicely connected through dialog coordinators. This is a general phenomenon: for small villages a parliament may be an overhead, but a country cannot miss it.
The issue of code and resources overhead was raised against objects too. Now even medium size embedded systems use them, as the design proved to be easier and hardware resources became less restrictive. Responsive Units look like bigger-objects but the overall organized dialog is believed to reduce the system complexity. Considering also the 'design for change' requirement, the expectations of reducing complexity with this method are even higher.

Throughout the thesis we considered object oriented systems with no reference to multi-tasking cases. For multi-tasking cases we note a few ideas in the 'Future Research' section.

7.4 Future Research

As mentioned in chapter 5, much research effort can be put in the study of multitasking applications for the thesis approach. Different environments may be considered separately. Even if the Responsive Unit, as a mega-object, is a generic element as any object, various synchronization and communication semantics can lead to interesting specializations. This would be an attempt to fill the gap between the goal-oriented coordination of the components in *one* system (as in CODE and this work) and the studies of the coordination between separate systems. This last kind of studies are abounding (as said in chapter 3) and they reached down to the level of processes [Del]; but they did not come too much lower, towards interacting objects.

A good opportunity to study these ideas will be the development of a new wireless modem for a wideband CDMA basestation, in which the author is involved. It has to be an object oriented multi-tasking system, intensively interactive. Dialog Coordinators and Responsive Units may easily find their place for the definition of inter-object dialog. As for the dialog between tasks, it will be more a matter of research, since there are many designers involved and also there is a certain level (~50%) of legacy code. The study will be hopefully ready for the Nortel Design Forum the next spring. The Responsive Unit concept for inter-object level was already used in a commercial project (as explained in CODE applications paragraph) and presented at a departmental information session.

There is another possible extension for the use of dialog coordinators and Responsive Units. Since our simulators are built in Java as web applets, they can run from any computer. In Java, a user interface event is internally transformed in a word, implemented as a string. An internal event-word (seen in a text output) can be sent to other users using a chat-page or any other way. The receivers can put the event-word in a text input of their simulator, bypassing the normal user interface. Thus, the simulator is activated directly by the event-word. At this level we made an implementation where one user can drive the others. For such system interactions we may want to observe the value of the dialog coordination. Obviously, the dialog between users can not be defined in one place, as in our normal theory. However, we can have one dialog coordination *class* that becomes in each machine a dialog coordination *object,* identical in all machines. We believe that this replication maintains some of the advantages of defining the dialog in one object. The idea falls in the complex domain of interacting systems and requires much research. Here we only suggest that the dialog coordination theory, in the sense introduced in this work, may eventually be extended beyond the edge of a physical system.

References

[Rumbaugh] Object Oriented Modeling and Design, J.Rumbaugh, Prentice Hall 1991

[Booch] Object Oriented Analysis and Design, G.Booch, Benjamin/Cummings 1994

[CY] Object Oriented Analysis, P.Coad, E.Yourdon, Prentice Hall 1990

[Jacobson] Object Oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley 1992

[ShMellor] Object Lifecycles, Modeling the World in States, S.Shlaer, S.Mellor Prentice Hall 1992

[Brook] Designing Object Oriented Software, A Responsibility Driven Approach, R.Wirfs-Brook Prentice Hall 1990

[UML] Unified Modeling Language, Rational Software Corp., www.rational.com/uml

[Fowler] UML Distilled, M. Fowler, Addison-Wesley 1997

[Powel] Real-Time UML, B. Powel, Addison-Wesley 1998

[UML2] Using UML for Modeling Complex Real-Time Systems, B.Selic, J.Rumbaugh, www.objecttime.com 1998

[UML3] UML for Real-Time Overview, A.Lyons, www.objecttime.com, 1998

[Room] Real-Time Object Oriented Modeling, B. Selic, G.Gullekson, P.Ward, Wiley&Sons, 1994

[Patterns] Design Patterns, Gamma & all, Addison-Wesley 1995

[Buschmann] A System of Patterns, F. Buschmann & all, John Wiley & Sons, 1996

[Lalonde] Dicovering Smalltalk, Wilf Lalonde, Benjamin / Cummings,1994

[Riel] 'Object Oriented Design Heuristics', G.Riel, 1996.

[Krieger1] Multiactivity Paradigm for the Design and Coordination of FMS's -M. Krieger, Computer-Integrated Manufacturing Systems, Vol. 6, No. 3, p195-203, August 1993.

[Krieger2] The Multiactivity Paradigm : An Approach for the Design of Embedded System by Applications Specialists -M. Krieger, Embedded Systems lectures, University of Ottawa

[Krieger3] Coordination Oriented Development Environment (CODE): An Engineering Approach to Computer Based Systems (Software) Development -M. Krieger, research paper, University of Ottawa

[Krieger4] Restricted Object Based Design of Event Driven Commercial Software, CASCOM'94 Integrated Solution Toronto CD-ROM, Ont., Nov. 1994

[Krieger5] Coordination Based Design/Development, M. Krieger, "ObjectTime Workshop on Research in Real-Time Object Oriented Modeling" Jan 13, 1998

[Krieger6] Digital Computer Organization, M. Krieger, lectures, University of Ottawa, 1997

[Jackson] System Development, M. Jackson, Prentice-Hall 1983

[Budgen] Software Design, D. Budgen, Addison-Wesley 1993

[Dijkstra] Structured Programming, J.Dahl, E. Dijkstra, R.Hoare, Academic Press, 1992

[Papadopoulos] Coordination Models and Languages, G. Papadopoulos, Third Int'l Conference on Coordination Models and Languages, COORDINATION'99, www.cs.unibo.it/~coord99

[Malone] The Interdisciplinary Study of Coordination. T.Malone, ACM Computing Surveys Vol. 26, No. 1. March 1994

[Del] A Coordination Perspective on Software Architecture, C.Dellarocas, http://ccs.mit.edu/CCSWP193/title.html

[Bergmans] Obstacles in Object Oriented Software Development, L.Bergmans, OOPSLA'92 p341

[Carriero] Coordination Languages and their Significance, D.Gelernter, N.Carriero, Communications of the ACM, Feb. 1992, Vol.35

[Hughes] Object-Oriented Multithreading Using C++, C. Hughes, J.Wiley&Sons, 1997

[Harel1] StateMate: A Working Environment for the Development of Complex Reactive Systems D. Harel, IEEE Transactions on Software Engineering, Vol. 16, No.4, April 1990

[Harel2] StateCharts: A Visual Formalism for Complex Systems, D. Harel, Science of Computer Programming 8, 1987 North-Holland

[Harel3] Executable Object Modeling with StateCharts, D. Harel, IEEE, July 1997

[Hoare] Communicating Sequential Processes, C.Hoare, Prentice Hall 1985

[SDL1] SDL with Applications, F.Belina, D.Hogrefe, A.Sarma, Prentice Hall 1991

[Bjor] Real Time Modeling Marries UML, SDL-2000 , M. Bjorkander, Electronic Engineering Times, Sep. 27, 1999

[Constantine1] From Events to Objects: The Heresy of Event-Orientation in a World of Objects, L. Constantine (panel chair), OOPSLA 1992

[Hatton] Does OO Sync With What We Think? IEEE Software, L. Hatton, May 1998

[Lemire] Multiactivity as a Restricted Object Based Design Approach, S.Lemire, M.Sc, Carleton U- June 1993

[Yang] Coordination-Based Design; Towards Component-Based Software Design Master Project, R.Au-Yang, Carleton U-Jan.1998

[Cox] Coordination-Based Design for Tailorable Business Software : a Computerized Maintenance Management System Example, C.Cox , M.A.Sc, Ottawa U-Nov 1998

[Coifan] Coordination Oriented Architecture for Large Software Systems, G.Coifan, M.A.Sc, Ottawa U-Oct.1999

[Leue] Research on Message Sequence Charts and Message Flow Graphs, S.Leue, 1998, http://www.fee.uwaterloo.ca/~sleue/msc.html

[Colleman] Introducing ObjectCharts or How to Use Statecharts in Object Oriented Design D. Colleman, IEEE Transactions on Software Engineering, Vol. 18, No.1, Jan. 1992

[Rizman] Application Development through Object Composition by Means of Event and Object Environment , K.Rizman, 1992 IEEE Int'l Conference on Computer System and Software Eng.

Appendix A

Coding Format for Dialog Coordinators, Responsive Units

A.1 Suggested Development Style in Java
A.2 Suggested Development Style in C++

We have seen how from the concept of Dialog Coordination we derived several modeling elements, generic enough to be easy applicable and clear enough to be defined in a programming language. We suggest implementation modes for Responsive Units with their internal Coordinator, Port, Data and Activities objects, and also for Team Coordinators and Agreement Coordinators. In order to distinguish concepts from particular semantic styles, we have chosen to present implementations in two object oriented languages, Java and C++.

A.1 Suggested Development Style in Java

Java has much similarity in semantics with C++, the differences addressing issues of networking, security, multi platform. The definitions of methods are in Java as for 'inline' methods in C++. The function pointers are not implemented in Java. They are useful in C++ as we can apply them for table implementation of State Machines. In Java we only use Switch statements.

A generic Responsive Unit is a class that may be instantiated at start-up time by the Main Frame, which inherits a ready made class, the 'Frame'. This is the first object built in Java; it is also the area through which we have access to the user interface, and where we can built our gadgets for input-output to the screen.

A Responsive Unit has pointers to:

- the parent, which in Java is the Main Frame, a class that inherits (extends) the Frame Class, ready made in Java.
- its Team Coordinator and
- the internal Objects :
    - Coordinator,
    - Port,
    - Data Base,
    - Activities.

At Construction time the internal Objects are built and all pointers established.

```
class ResponsiveUnit_1_Class
{
        // the Main frame which instantiates this class
        MainManagementFrame   Frame_p;
        // the Team Coordinator of this Responsive Unit
        TeamCoordinator_RUnit  TeamCoordinator_RUnit_p ;
        // the internal Objects
        RU1_Coordinator_Class   RU1_Coordinator_p ;
        RU1_Port_Class          RU1_Port_p        ;
        RU1_DataBase_Class      RU1_DataBase_p    ;
        RU1_Activity1_Class     RU1_Activity1_p   ;
        // the Constructor  receives pointer of parent
        //                  reads the pointer of its Team Coordinator
        //                  instantiates internal Objects
        ResponsiveUnit_1_Class(MainManagementFrame MainManagementFrame_p)
        {
                Frame_p                 = MainManagementFrame_p;
                TeamCoordinator_RUnit = Frame_p.TeamCoordinator_RUnit_p;

                RU1_Coordinator_p     = new RU1_Coordinator_Class( this );
                RU1_Port_p            = new RU1_Port_Class( this );
                RU1_DataBase_p        = new RU1_DataBase_Class( this );
                RU1_Activity1_p       = new RU1_Activity1_Class( this );
        }
}
```

The Java-type Main-Frame instantiates all Responsive Units. Each reads the parent's pointer.

```
public class MainManagementFrame extends Frame
{
        TeamCoordinator_RUnit  TeamCoordinator_RUnit_pointer
                                        =new TeamCoordinator_RUnit(this);
        DataBase_RUnit          DataBase_RUnit_pointer
                                        = new DataBase_RUnit(this);
        Specialized1_RUnit      Specialized1_RUnit_pointer
                                        = new Specialized1_RUnit(this);.....
}
```

A Team Coordinator Responsive Unit is specialized by having pointers to the coordinated Responsive Units. Several Team Coordinators may exist in a system, each related also to the system level Agreement Coordinator.

```
class TeamCoordinator_RUnit
{
        // the Main frame which instantiates this class
        MainManagementFrame   Frame_p;
        // the coordinated Responsive Units
        DataBase_RUnit      DataBase_RUnit_p ;
        Specialized1_RUnit   Specialized1_RUnit_p;
        Specialized2_RUnit   Specialized2_RUnit_p;
        // the internal Objects
        TC1_Coordinator_Class  TC1_Coordinator_p;
        TC1_Port_Class         TC1_Port_p;
        TC1_DataBase_Class     TC1_DataBase_p;
        TC1_Activity1_Class    TC1_Activity1_p;
        // the Constructor receives pointer of parent and instantiates internal Objects
        TeamCoordinator_RUnit(MainManagementFrame MainManagementFrame_p)
        {
                Frame_p              = MainManagementFrame_p;
                TC1_Coordinator_p  = new TC1_Coordinator_Class ( this );
                TC1_Port_p           = new TC1_Port_Class ( this );
                TC1_DataBase_p     = new TC1_DataBase_Class ( this );
                TC1_Activity1_p    = new TC1_Activity1_Class ( this );
        }
        // method called in parent AFTER all Responsive Units are instantiated
        public void init()
        {
                DataBase_RUnit_p     = Frame_p.DataBase_RUnit_pointer ;
                Specialized1_RUnit_p = Frame_p.Specialized1_RUnit_pointer;
                Specialized2_RUnit_p = Frame_p.Specialized2_RUnit_pointer;
        }
}
```

Next is presented an example of a Coordinator Class which belongs to a Team Coordinator Responsive Unit. It is instantiating a State Machine that describes the sequence of interactions, i.e., the responses to the requests made from outside the Team.

The Coordinator is calling 'problem domain' Responsive Units with their internal actions. Exceptions can be also solved as part of the state machine, and upper layer informed.

```
class TC1_Coordinator_Class
{
        protected int state;                          // values of state and event
        protected int event;
        TeamCoordinator_RUnit TC_RUnit_p ;      // pointer to the parent
        // constructor reads the pointer to the parent and  sets  the initial state
```

```
TC1_Coordinator_Class(TeamCoordinator_RUnit TeamCoordinator_RUnit_p)
{
        TC_RUnit_p = TeamCoordinator_RUnit_p ;
        state = 0;  //Start
}
void StateAuto()    // the main method is a State Machine implementation
{
        // event as in parent
        event = TC_RUnit_p.event;
        switch( state )
        {
        case 0: //Start
                switch(event)
                {
                case 0:
                        TC_RUnit_p.TC1_Activity1_Class.computeA();// Call an activity
                        state = 1;                                // set the next state
                        break;
                case 1:
                        TC_RUnit_p.TC1_DataBase_Class.getDataA();
                        state = 1;
                        break;
                default:
                        TC_RUnit_p.Frame_p.ToUser.setText("Not good ");// exceptions
                        break;
                }
                break;
        case 1:
                switch(event)
                {
                case 2:
                        TC_RUnit_p.TC1_Activity2_Class.computeC();
                        state = 0;
                        break;
                default:
                        TC_RUnit_p.Frame_p.ToUser.setText("Not good ");
                        break;
                }
                break;
        } // end switch( state )
    }       // end StateAuto()
}           // end class TC1_Coordinator_Class
```

We have chosen to implement the State Machine as a double switch, on state and on event. In C++ we will show a Table style of State Machine design.

A.2 Suggested Development Style in C++

In the case of C++ language (not in Java) we can use function pointers to refer to the called activities. They are arrayed in tables, with columns referred by state numbers and rows referred by event numbers. We chose such table oriented case here. The implementation of a switch statement is similar to Java.

A Responsive Unit is built as a class, instantiated at initialization time. It is associated with its internal objects as it contains their pointers.

```
class ResponsiveUnit_Class
{
        public:
                ResponsiveUnit_Class ();                    //constructor
               ~ResponsiveUnit_Class ();                    //destructor
        private:
               // pointers to internal objects
               Coordinator_Class   *Coordinator_Pointer;
               Port_Class          *Port_Pointer;
               DataBase_Class      *DataBase_Pointer;
               Activities_Class    *Activities_Pointer;
}
```

When the constructor is called, the internal objects get also instantiated.
Each receives the pointer to the parent object.

```
ResponsiveUnit_Class :: ResponsiveUnit_Class ()
{
               Coordinator_Pointer = new Coordinator_Class(this);
               Port_Pointer        = new Port_Class(this);
               DataBase_Pointer    = new DataBase_Class(this);
               Activities_Pointer  = new Activities_Class(this);
}
```

The Coordinator_Class is built to maintain the rules and the sequence of the external dialog and of the calls to interior activities.
As explained previously it has (inherits) a generic StateMachine_Class.

The implementation of a State Machine may be as a Table with states on rows and events on columns. Each cell contains an activity, implemented as a function pointer. The Switch/case style is hard to apply for a bigger number of events/states.

```
Class StateMachine_Class
{
        public:
                  StateMachine_Class();                          //constructor
                ~ StateMachine_Class();                          //destructor
                void SetEvent ( int event ) { event_m  = event  ;}
                void SetState  ( int state  ) { state_m  = state  ;}
                int GetState ( )             { return state_m    ;}
                int getNrEvents()            { return nrEvents_m;}
                // For cases when the decision is based on a condition
                void SetCondition ( bool condition ) { condition_m  = condition }
                unsigned char  * StateEventTable_Pointer;  // pointer to the Sate - Event table
                // pointer to a table of strings with all acceptable Words-in– optional
                char **WordsIn_Pointer;
        private:
                int state_m;   // the actual state and events;
                int event_m;
                //flags hows if the decision for the next state is taken from the table or is based
                //on an internal value
                bool condition_m;
}
```

A coordinator inherits the State Machine class as the mode of representing the sequence of requests-responses. We repeat that the name Dialog Machine would better reflect the meaning of the machine in this context.

```
Class Coordinator_Class : public StateMachine
{
        public:
                 Coordinator_Class ( ResponsiveUnit_Class* ); // constructor knows its parent
                ~Coordinator_Class();                          // destructor
                CoordinatorMain();                             // calls the internal State Machine
        private:
                // table keeps function-pointers to activities for each state-event pair
                static Co_Function_Pointer  Co_Activities [Co_state][Co_event];
                // activities
                Activity_1();
                Activity_2();
                // If complex, the Coordinator may have nested State Machines
                // here are methods for calling the necessary sub-machine.
                // These are called as activities, through function -pointers table.
                void SetNestedStateMachine_1();
                void SetNestedStateMachine_2();
}
```

The acting method is the CoordinatorMain(). It decides the action and the next state based on state, event and condition. It is called at the appropriate time to give the full response to each request.

```
Class Coordinator_Class :: CoordinatorMain()
{
// Call the activity through its function pointer , from the table at the // actual state/event pair.
// Check first if the pointer is not null , for cases when no action is  expected.
        if( Co_Function_Pointer [ state_m ][ event_m ] != 0 )
        {
                ( *Co_Function_Pointer [ state_m ][ event_m ] )();
        }
        // In usual situations the next state depends of actual pair state/event.
        // Check if there is no condition , for such usual case.
        if ( !condition_m  )
        {
                state_m = StateEventTable_Pointer [ state_m ][ event_m ];
        }
        // If we get an internal condition, the next state will be set in the activity.
        // Need to care about releasing the condition flag after it is used.
        else
        {
                condition_m = False;
        }
}
```

A nested State Machine may be built, as in the StateCharts model, if the complexity requires. A switch to the nested Machine is implemented by a call from a certain state of the main Machine.

```
Class NestedStateMachine_1: public StateMachine
{
        public:
                //constructor , calls also the StateMachine constructor
                NestedStateMachine_1();
                // main method calling the internal State Machine
        Main_NestedStateMachine_1();
        private:
                // table keeps function-pointers to activities for each state-event pair
                static NS1_Func_Pointer NS1_Activities [NS1_state][ NS1_event];
                // activities called at this level of nesting
                NS1Activity_1();
                NS1Activity_2();
```

/The 'parent' State Machine will call the Switch when the next step will be executed inside the nested State Machine.

Appendix B

Essential Code for Design Cases

B.1 Single Accumulator Microprocessor Simulator

B.2 RISC Microprocessor Simulator (main differences)

B.3 STACK Microprocessor Simulator (main differences)

B.1 Single Accumulator Microprocessor Simulator

```
import java.awt.*;
import java.util.*;

public class OneAccMicro extends Frame {
    boolean inAnApplet = true;

      RU_DCoordinator        RU_DCoordinator_p;
      RU_RandomAccsessMemory RU_RAM_p      ;
      RU_Registers           RU_Regs_p     ;
      RU_ArithmeticLogicUnit RU_ALU_p      ;

      Button Edit      ;
      Button LoadProg  ;
      Button Execute   ;

      Button AutoRun   ;
      Button InstrStep ;
      Button MicroStep ;

      Button Append    ;
      Button Insert    ;
      Button Delete    ;

      Button AppendData;
      Button InsertData;
      Button DeleteData;

      Button SaveProg;
      Button EnterSaved;

      TextField progCounter;
      TextField accumulator;
      TextField temporary  ;

      TextField instruction;
      TextField memAddress ;
      TextField memBuffer  ;

      TextField dataPointer;

      TextField carry      ;
      TextField negative   ;
      TextField zero       ;

      TextField ToUser     ;

   TextField InsertAt;
   TextField DeleteAt;

     Choice opCode ;
   Choice addrMode ;

     TextField dataField ;
   List programList ;
   List programMem ;
     List dataList ;
```

```
    TextArea DataAddr;
    //EditDialog m_Edit;
    MenuBar mb;
    Menu help;
    Menu EditUtils;

    Dialog m_EditD;

    Panel wPanel ;
    Panel ePanel ;

    List  choseList;
    TextField funcData;

    TextField   dataMemAdd;
    TextField   utilData  ;

    boolean have_m_EditD;

  public OneAccMicro() {


    int up   =50;
    int low  =200;
    int right=175;
    int left =300;
    have_m_EditD = false;

    setTitle("Single Accumulator Processor ");

    mb = new MenuBar();

  help  = new Menu("Help");
  help.add(new MenuItem("help"));
  help.add(new MenuItem("about"));
  mb.add(help);

  EditUtils  = new Menu("EditUtils");
  EditUtils.add(new MenuItem("EditUtils"));
  mb.add(EditUtils);

  setMenuBar(mb);

     Edit      = new Button("Enter Prog");
     LoadProg  = new Button("Load Prog");
     Execute   = new Button("Execute");

     AutoRun   = new Button("Auto Run");
     InstrStep = new Button("Instr Step");
     MicroStep = new Button("Micro Step");

     Append    = new Button("App");
     Insert    = new Button("Ins");
     Delete    = new Button("Del");

     AppendData= new Button("App D");
     InsertData= new Button("Ins D");
```

```
    DeleteData= new Button("Del D");

   SaveProg   = new Button("Save Prog") ;
   EnterSaved = new Button("Enter Saved") ;


   Edit.    reshape(right+80 ,up+10 ,75,25);
   LoadProg.reshape(right+162,up+10 ,70,25);
   Execute. reshape(right+240,up+10 ,70,25);

   AutoRun  .reshape(right+250,up+ 40,60,25);
   InstrStep.reshape(right+250,up+ 70,60,25);
   MicroStep.reshape(right+250,up+100,60,25);

   Append.reshape(right+80 ,up+40  ,35,25);
   Insert.reshape(right+80 ,up+70  ,35,25);
   Delete.reshape(right+80 ,up+100 ,35,25);

AppendData.reshape(right+120 ,up+40  ,35,25);
InsertData.reshape(right+120 ,up+70  ,35,25);
  DeleteData.reshape(right+120 ,up+100 ,35,25);

SaveProg  .reshape(right+330,up+10 ,65,25);
EnterSaved.reshape(right+330,up+40 ,65,25);

  add(Edit);
  add(LoadProg);
  add(Execute);

  add(AutoRun);
  add(InstrStep);
  add(MicroStep);

  add(Append);
  add(Insert);
  add(Delete);

  add(AppendData);
  add(InsertData);
  add(DeleteData);

  add(SaveProg);
  add(EnterSaved);

   progCounter= new TextField("0");
   accumulator= new TextField("0");
   temporary  = new TextField("0");

   instruction= new TextField("function");
   memAddress = new TextField("0");
   memBuffer  = new TextField("0");

   dataPointer= new TextField("0");

   carry      = new TextField("0");
   negative   = new TextField("0");
   zero       = new TextField("0");
```

```
    ToUser      = new TextField("Feedback");

    DataAddr    = new TextArea();

   progCounter.reshape(right+70 ,low+30,60,25);
   accumulator.reshape(right+260,low+100,60,25);
   temporary.  reshape(right+140,low+100,60,25);

   instruction.reshape(right+70       ,low+140,75,25);
   memAddress. reshape(right+left     ,low+30 ,60,25);
   memBuffer.  reshape(right+left+85 ,low+80 ,75,25);

   dataPointer.reshape(right+140      ,low-10,60,25);

   carry.    reshape(right+100,low+200,25,25);
   negative.reshape(right+125,low+200,25,25);
   zero.     reshape(right+150,low+200,25,25);

   ToUser  .reshape(right+70,  440,290,25);

   DataAddr.reshape(right+left+170,up+40, 60,170);

   progCounter.setEditable(false);
   accumulator.setEditable(false);
   temporary.setEditable(false);

   instruction.setEditable(false);
   memAddress.setEditable(false);
   memBuffer.setEditable(false);

   dataPointer.setEditable(false);

   carry.setEditable(false);
   negative.setEditable(false);
   zero.setEditable(false);

ToUser.setEditable(false);

  DataAddr.setEditable(true);

add(progCounter);
add(accumulator);
add(temporary);

add(instruction);
add(memAddress );
add(memBuffer);

add(dataPointer);

add(carry);
add(negative);
add(zero);

  add( ToUser );
  ToUser.selectAll();
```

```
        //add( DataAddr );

        InsertAt  = new TextField("0");
        InsertAt.reshape(right+150 ,up+70 ,25,25);
        InsertAt.setEditable(true);
      //add(InsertAt);

      DeleteAt  = new TextField("0");
        DeleteAt.reshape(right+150 ,up+100 ,25,25);
        DeleteAt.setEditable(true);
      //add(DeleteAt);


        setLayout(new BorderLayout());
      Panel topPanel  = new Panel();
      Panel westPanel = new Panel();
      Panel eastPanel = new Panel();


        dataField  = new TextField("0");
        dataField.reshape(20,420,270,25);
        dataField.setEditable(true);
      topPanel.add(dataField);

      programList = new List(28, false);

      programMem  = new List(28, false);

        dataList    = new List(28, false);

        for (int id1=50;id1<100;id1++)
              dataList.addItem(""+id1);

      westPanel.add(programList);
//      westPanel.add(programMem);
      westPanel.add(dataList);
      add("West", westPanel);

              MyCanvas canvasMain = new     MyCanvas();
              canvasMain.resize(640,470);
              canvasMain.move(0,0);
              canvasMain.setBackground(Color.blue);
        add( canvasMain );

        /////////new Responsive Units
              RU_DCoordinator_p = new RU_DCoordinator(this);

              RU_RAM_p       = new RU_RandomAccsessMemory(this);
              RU_Regs_p      = new RU_Registers(this);
              RU_ALU_p       = new RU_ArithmeticLogicUnit(this);

        /////////new Responsive Units done

              RU_DCoordinator_p.init();
              RU_ALU_p.init();
```

```
    }

    public void reset()
    {
    }

public boolean action(Event event, Object arg) {

    if (event.target instanceof MenuItem) {
       if ( ((String)arg).equals("EditUtils") && (!have_m_EditD) ) {

                m_EditD = new Dialog( this,false    );
                m_EditD.resize(160, 450);
                m_EditD.show();
                have_m_EditD=true;
                wPanel = new Panel();

                choseList= new List(27, false);
                choseList.addItem("add_imm");
                choseList.addItem("adc_imm");
                choseList.addItem("sub_imm");
                choseList.addItem("sbb_imm");
                choseList.addItem("ld__imm");

                choseList.addItem("add_dir");
                choseList.addItem("adc_dir");
                choseList.addItem("sub_dir");
                choseList.addItem("sbb_dir");
                choseList.addItem("ld__dir");
                choseList.addItem("st__dir");
                choseList.addItem("jmp_dir");
                choseList.addItem("joc_dir");
                choseList.addItem("jon_dir");
                choseList.addItem("joz_dir");

                choseList.addItem("add_ind");
                choseList.addItem("adc_ind");
                choseList.addItem("sub_ind");
                choseList.addItem("sbb_ind");
                choseList.addItem("ld__ind");
                choseList.addItem("st__ind");
                choseList.addItem("jmp_ind");
                choseList.addItem("joc_ind");
                choseList.addItem("jon_ind");
                choseList.addItem("joz_ind");

                choseList.addItem("clr");
                choseList.addItem("neg");
                choseList.addItem("inc");
                choseList.addItem("dec");
                choseList.addItem("halt");

                wPanel.add(choseList);

                funcData = new TextField("0");
                funcData.setEditable(true);
```

```
                    wPanel.add(funcData);

                    ePanel = new Panel();

                    dataMemAdd = new TextField("0");
                    dataMemAdd.setEditable(true);

                    ePanel.add(dataMemAdd);

                    utilData = new TextField("0");
                    utilData.setEditable(true);

                    ePanel.add(utilData);

                    m_EditD.add("North", ePanel);
                    m_EditD.add("West" , wPanel);

            }
          }
            else
             RU_DCoordinator_p.AM_Port_p.action( event);


          return true;
      }

      public boolean handleEvent(Event event) {

          if (event.id == Event.WINDOW_DESTROY) {
             if (inAnApplet) {
                    if (event.target instanceof Dialog)
                          {
                                choseList.clear();
                                m_EditD.dispose();
                                have_m_EditD=false;
                          }
                          else
                                dispose();
              } else {
                  System.exit(0);
              }

          }
          return super.handleEvent(event);
      }
}


class MyCanvas extends Canvas {

           int u  =50;
             int l  =200;
             int r  =175;
             int lf =300;
             int acc=r+185;

             public void paint(Graphics g) {
```

```
        int w = size().width;
        int h = size().height;


        g.drawRect(0, 0, w - 1, h - 1);
        g.setFont(new Font("Helvetica",  Font.BOLD, 12));

            g.setColor(Color.red);
            g.fillRect(acc,340,95,80);

        g.drawLine(r+100,l+ 55,r+100    ,l+140);
        g.drawLine(r+170,l+ 92,r+170    ,l+100);
        g.drawLine(r+290,l+ 92,r+290    ,l+100);
        g.drawLine(r+130,l+ 42,r+ lf    ,l+ 42);
        g.drawLine(r+100,l+ 92,r+lf+120 ,l+ 92);
            //acu tmp
        g.drawLine(acc+5  ,l+ 125  ,acc +5,340);
            //acu alu
        g.drawLine(acc+90  ,l+ 122,acc+ 90,340);
            //alu acu 1
        g.drawLine(acc+45  ,420    ,acc+ 45,435);
            //alu acu 2
        g.drawLine(acc+45  ,435    ,r+35 0    ,435);
            //alu acu 3
        g.drawLine(r+350,l+ 115,r+350    ,435);
            //acu r
        g.drawLine(r+350,l+ 115,r+320    ,l+ 115);
            //acu mbuf h
        g.drawLine(r+320  ,l+ 110,acc+2-45 ,l+ 110);
            //acu mbuf ver
        g.drawLine(acc+245,l+ 110,acc+2-45 ,l+ 90);

 //flg
            g.drawLine(r+175,l+ 210,acc ,l+ 210);
 //instr alu
            g.drawLine(r+130,l+ 152,acc ,l+ 152);

            g.drawLine(r+170  ,l-10    ,r+170  ,l+125 );//dp to tmp
            g.drawLine(r+200  ,l       ,r+l+130,l     );//dp to up mar
            g.drawLine(r+l+130,l       ,r+l+130,l+25  );//up mar to mar

            g.setColor(Color.orange);
//madd m u
            g.drawLine(r+lf+60,l+ 42,acc+235,u+40);
 //madd m lw
            g.drawLine(r+lf+60,l+ 42,acc+235,u+40+170);

            g.fillRect(acc+215,u+40,60,170);
            g.setColor(Color.red);
            g.drawString("Memory" , acc+225, u+40+80);

            g.setColor(Color.yellow);
            g.drawString("Prog Count" ,  r+70 , l+30-5);
            g.drawString("Accumulator",  r+260, l+100-10);
            g.drawString("Temporary"   ,  r+140, l+100-10);
```

```
            g.drawString("Instr Reg", r+70     , l+140-5);
            g.drawString("Mem Addr ", r+1f     , l+30-5);
            g.drawString("Mem Buff ", r+1f+100, l+80-5);

            g.drawString("Data Ptr ", r+140, l-15);

            g.drawString("Flag Reg" , r+100, l+200-5);
            g.drawString("  C     N     Z", r+100, l+200+35);

            g.drawString("ALC" , acc+35, 340+40);

    }

    public Dimension minimumSize() {
        return new Dimension(150,130);
    }

    public Dimension preferredSize() {
        return minimumSize();
    }
}


class RU_DCoordinator
{
      OneAccMicro    Frame_mp;

      RU_RandomAccsessMemory RU_RAM_p ;
      RU_Registers           RU_Regs_p;
      RU_ArithmeticLogicUnit RU_ALU_p ;

      AM_Coordinator AM_Coor_p;
      AM_Port              AM_Port_p;
      AM_InfoBase    AM_InfoBase_p;
      AM_Act         AM_Act_p;

      Event event;
      int eventArea;

      RU_DCoordinator(OneAccMicro OneAccMicro_p )
      {
            Frame_mp           = OneAccMicro_p;

            AM_Coor_p      = new AM_Coordinator(this);
            AM_Port_p      = new AM_Port(this);
            AM_InfoBase_p = new AM_InfoBase(this);
            AM_Act_p       = new AM_Act(this);
      }

      public void init()
      {
            RU_RAM_p       = Frame_mp.RU_RAM_p      ;
            RU_Regs_p      = Frame_mp.RU_Regs_p     ;
            RU_ALU_p       = Frame_mp.RU_ALU_p      ;
            AM_Coor_p.init();
            AM_Act_p .init();
      }
```

```
}


class RU_RandomAccsessMemory
{
      OneAccMicro Frame_mp;
      RU_DCoordinator   RU_DCoordinator_p ;

      RAM_Coordinator RAM_Coor_p ;
      RAM_Port        RAM_Port_p        ;
      RAM_InfoBase    RAM_Info_p        ;
      RAM_Act         RAM_Act_p         ;

      RU_RandomAccsessMemory( OneAccMicro OneAccMicro_p )
      {
            Frame_mp = OneAccMicro_p;
            RU_DCoordinator_p = Frame_mp.RU_DCoordinator_p;

            RAM_Info_p = new RAM_InfoBase(this);
            RAM_Coor_p = new RAM_Coordinator(this);
            RAM_Port_p = new RAM_Port(this);
            RAM_Act_p  = new RAM_Act(this);
      }

}

class RU_Registers
{
      OneAccMicro Frame_mp;
      RU_DCoordinator   RU_DCoordinator_p ;

      Regs_Coordinator Regs_Coor_p;
      Regs_Port        Regs_Port_p;
      Regs_InfoBase    Regs_Info_p;
      Regs_Act         Regs_Act_p ;

      RU_Registers( OneAccMicro OneAccMicro_p )
      {
            Frame_mp = OneAccMicro_p;
            RU_DCoordinator_p = Frame_mp.RU_DCoordinator_p;

            Regs_Coor_p = new Regs_Coordinator(this);
            Regs_Port_p = new Regs_Port(this);
            Regs_Info_p = new Regs_InfoBase(this);
            Regs_Act_p  = new Regs_Act(this);
      }
}


class RU_ArithmeticLogicUnit
{
      OneAccMicro Frame_mp;
      RU_DCoordinator   RU_DCoordinator_p ;

      ALU_Coordinator ALU_Coor_p;
```

```
        ALU_Port        ALU_Port_p;
        ALU_InfoBase    ALU_Info_p;
        ALU_Act         ALU_Act_p;

        RU_ArithmeticLogicUnit( OneAccMicro OneAccMicro_p )
        {
              Frame_mp = OneAccMicro_p;
              RU_DCoordinator_p = Frame_mp.RU_DCoordinator_p;

              ALU_Coor_p     =new ALU_Coordinator(this);
              ALU_Port_p     =new ALU_Port(this);
              ALU_Info_p     =new ALU_InfoBase(this);
              ALU_Act_p      =new ALU_Act(this);
        }

        public void init()
        {
              ALU_Act_p.init();
        }
}



//internals to Responsive Units

class AAStateMachine
{
      protected int state;
      protected int event;
}



class AM_Coordinator extends AAStateMachine
{

      RU_DCoordinator Manager_mp;
      int PC=0;
      int listCount=0;
      boolean followsInstruction = true ;
      boolean endOfProgram       = false;
      boolean endOfLine          = false;
      String instrField="";

      TextField progCounter;
      TextField accumulator;
      TextField temporary  ;

      TextField instruction;
      TextField memAddress ;
      TextField memBuffer  ;

      TextField dataPointer;

      TextField carry      ;
      TextField negative   ;
```

```
TextField zero       ;

AM_Coordinator( RU_DCoordinator RU_DCoordinator_p)
{
      Manager_mp = RU_DCoordinator_p;
      state = 1;//Start

      //local pointers to text regs
      progCounter=Manager_mp.Frame_mp.progCounter;
      accumulator=Manager_mp.Frame_mp.accumulator;
      temporary  =Manager_mp.Frame_mp.temporary;

      instruction=Manager_mp.Frame_mp.instruction;
      memAddress =Manager_mp.Frame_mp.memAddress;
      memBuffer  =Manager_mp.Frame_mp.memBuffer;

      dataPointer=Manager_mp.Frame_mp.dataPointer;

      carry      =Manager_mp.Frame_mp.carry;
      negative   =Manager_mp.Frame_mp.negative;
      zero       =Manager_mp.Frame_mp.zero;

}

private RAM_Port actMemory;

public void init()
{
      actMemory =Manager_mp.RU_RAM_p.RAM_Port_p;
}

public final static int EDIT=1;
public final static int LOAD_PROG=2;
public final static int APPEND=7;
public final static int INSERT=8;
public final static int DELETE=9;
public final static int APPEND_DATA=10;
public final static int INSERT_DATA=11;
public final static int DELETE_DATA=12;
public final static int EXECUTE=3;
public final static int AUTORUN=4;
public final static int INSTRSTEP=5;
public final static int MICROSTEP=6;

public final static int SAVE_PROG  =13;
public final static int ENTER_SAVED=14;

void StateAuto()
{
      event = Manager_mp.eventArea;

      switch( state )
      {
      case 1:
            switch(event)
            {
            case EDIT://
```

```
                        Manager_mp.AM_Port_p.say( "EnterProg accepted" );
                        Manager_mp.AM_Act_p.EnterProg();
                        reset();
                  state=2;
                        break;
                    default:
                        Manager_mp.AM_Port_p.say( "Not accepted" );
                        break;
                    }
                    break;
              case 2:
                    switch(event)
                    {
                    case LOAD_PROG:
                        listCount=actMemory.countItems();
                  if ( listCount> 0)
                        {
                              Manager_mp.AM_Port_p.say( "LoadProg  accepted > do
Execute " );
                              //Manager_mp.AM_Act_p.Load();
                              state=3;
                        }
                        else
                        {
                              Manager_mp.AM_Port_p.say( "LoadProg not accepted >
do EnterProg" );
                              state=1;
                        }
                        break;
                     case APPEND://7
                        Manager_mp.AM_Port_p.say( "Append accepted" );
                        Manager_mp.AM_Act_p.Append();
                  state=2;
                        break;
                    case INSERT://8
                        Manager_mp.AM_Port_p.say( "Insert accepted" );
                        Manager_mp.AM_Act_p.Insert();
                        state=2;
                        break;
                    case DELETE://9
                        Manager_mp.AM_Port_p.say( "Delete accepted" );
                        Manager_mp.AM_Act_p.Delete();
                        state=2;
                        break;
                    case APPEND_DATA://10
                        Manager_mp.AM_Port_p.say( "AppendData accepted" );
                        Manager_mp.AM_Act_p.AppendData();
                  state=2;
                        break;
                    case INSERT_DATA://11
                        Manager_mp.AM_Port_p.say( "InsertData accepted" );
                        Manager_mp.AM_Act_p.InsertData();
                        state=2;
                        break;
                    case DELETE_DATA://12
                        Manager_mp.AM_Port_p.say( "DeleteData accepted" );
                        Manager_mp.AM_Act_p.DeleteData();
```

```
                state=2;
                break;
        case SAVE_PROG://13
                Manager_mp.AM_Port_p.say( "SaveProg accepted" );
                Manager_mp.AM_Act_p.SaveProg();
                state=2;
                break;
        case ENTER_SAVED://14
                Manager_mp.AM_Port_p.say( "EditSaved accepted" );
                Manager_mp.AM_Act_p.EditSaved();
                state=2;
                break;

        default:
                Manager_mp.AM_Port_p.say( "Not accepted" );
                break;
        }
        break;

case 3:
        switch(event)
        {
        case EDIT://1
                Manager_mp.AM_Port_p.say( "EnterProg accepted" );
                Manager_mp.AM_Act_p.EnterProg();
                reset();
    state=2;
                break;
        case EXECUTE://3
                Manager_mp.AM_Port_p.say( "Execute accepted" );
    state=3;
                break;
        case AUTORUN://4
                Manager_mp.AM_Port_p.say( "AutoRun accepted" );
                ResponseManager_AutoRun();
    state=3;
                break;
        case INSTRSTEP://5
                Manager_mp.AM_Port_p.say( "InstrStep accepted" );
                ResponseManager_InstrStep();
                state=3;
                break;
        case MICROSTEP://6
                Manager_mp.AM_Port_p.say( "MicroStep accepted" );
                ResponseManager_MicroStep();
    state=3;
                break;
        default:
                Manager_mp.AM_Port_p.say( "Not accepted" );
                break;
        }
        break;

default:
        break;

}
```

```
}

void ResponseManager_AutoRun()
{
	endOfProgram  =false;//start from anywhere

	while(endOfProgram == false)
	{
		ResponseManager_InstrStep();
	}

	endOfProgram = false;

}
void ResponseManager_InstrStep()
{

	endOfLine =false;//start from anywhere

	while(endOfLine == false)
	{
		ResponseManager_MicroStep();
	}

	endOfLine = false;
}

int stateMS= 0;
String memAtMad="";

void ResponseManager_MicroStep()
{
	switch( stateMS )
	{
	case 0:
		//write pc
		progCounter.setText(""+PC);
		stateMS =1;
		break;
	case 1:
		//write mad
		memAddress.setText( progCounter.getText() );
		actMemory.select(PC);
		stateMS =2;
		break;
	case 2:
		//get mem at mad =PC
		memAtMad=actMemory.getItem(PC);
		if(actMemory.memActed == false)
		{
			Manager_mp.AM_Port_p.say("no data: enter ");
		}
		//write mbr
		memBuffer.setText(memAtMad);
		//next state:
		stateMS =3;
		break;
```

```
        case 3:
              // write ir
              instruction.setText(memBuffer.getText());
              instrField=instruction.getText();

              if( instrHasData() )
              {
                    stateMS =5;
              }
              else //instr has no data
              {
                    stateMS =4;
              }
              break;
        case 4:
              subM_InstrNoData();
              stateMS =0;
              break;
        case 5:
              subM_InstrWithData();
              //stateMS =0;in subm
              break;

        default:
              break;
        }
}


int stateSubMInstr = 0;
StringTokenizer            TokObj;
String                           Token;

private void subM_InstrNoData()
{
      switch(stateSubMInstr)
      {
      case 0:

            if( instrField.regionMatches(0,"clr",0,3 ))
            {
                  Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(0);
            }else
            if( instrField.regionMatches(0,"neg",0,3 ))
            {
                  Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(1);

            }else
            if( instrField.regionMatches(0,"inc",0,3 ))
            {
                  Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(2);

            }else
            if( instrField.regionMatches(0,"dec",0,3 ))
            {
                  Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(3);
```

```
                }

                break;
        default:
                break;

        }
        endOfLine();//incl PC++;

}

int stateSubMData = 0;
int dataMemAdr    = 0;
int mbr = 0;
int mar = 0;
boolean  firstTime=true;

String instrText   ="";
String dataMemValue="";
//      storeStr    ="";

StringTokenizer   instrTextTok;
String InstrTok="";
String DataTok ="";

String dataToMemBuf="_ ";
String _Str="_";

private void subM_InstrWithData()
{
        switch(stateSubMData)
        {
        case 0:
                //st imm dir else
                instrText=instruction.getText();

                //instrTextTok = new StringTokenizer(instrText);

                //InstrTok = instrTextTok.nextToken();
                //DataTok  = instrTextTok.nextToken();//was at no _
                DataTok  = getDataTok();

                if(instrText.regionMatches(0,"st__dir",0,7 ))
                {
                        dataPointer.setText(DataTok);
                        stateSubMData=30;
                }
                else
                if(instrText.regionMatches(0,"st__ind",0,7 ))
                {
                        dataPointer.setText(DataTok);
                        stateSubMData=40;
                }
                else if(instrText.regionMatches(0,"jmp",0,3))
                {
                                if(instrText.regionMatches(4,"dir",0,3 ))
                                {
```

```
                        endOfLine();
                        PC=Manager_mp.AM_Act_p.strToInt(DataTok);
                    }
                    else//ind
                    {
                        dataPointer.setText(DataTok);
                        stateSubMData=50;
                    }

                }
                else if(instrText.regionMatches(0,"joc",0,3))
                {
                    if(Manager_mp.AM_Act_p.strToInt(carry.getText())==1)//pc
                    {
                        if(instrText.regionMatches(4,"dir",0,3 ))
                        {
                            endOfLine();
                            PC=Manager_mp.AM_Act_p.strToInt(DataTok);
                        }
                        else//ind
                        {
                            dataPointer.setText(DataTok);
                            stateSubMData=50;
                        }
                    }
                    else
                    {
                        endOfLine();
                    }
                }
                else if(instrText.regionMatches(0,"jon",0,3))
                {

if(Manager_mp.AM_Act_p.strToInt(negative.getText())==1)//pc
                    {
                        if(instrText.regionMatches(4,"dir",0,3 ))
                        {
                            endOfLine();
                            PC=Manager_mp.AM_Act_p.strToInt(DataTok);
                        }
                        else//ind
                        {
                            dataPointer.setText(DataTok);
                            stateSubMData=50;
                        }
                    }
                    else
                    {
                        endOfLine();
                    }
                }
                else if(instrText.regionMatches(0,"joz",0,3))
                {
                    if(Manager_mp.AM_Act_p.strToInt(zero.getText())==1)//pc
                    {
                        if(instrText.regionMatches(4,"dir",0,3 ))
                        {
```

```
                        endOfLine();
                        PC=Manager_mp.AM_Act_p.strToInt(DataTok);
                    }
                    else//ind
                    {
                        dataPointer.setText(DataTok);
                        stateSubMData=50;
                    }
                }
                else
                {
                    endOfLine();
                }
            }
            else
            if(instrText.regionMatches(4,"ind",0,3 ))
            {
                dataPointer.setText(DataTok);
                stateSubMData=10;
            }
            else
            if(instrText.regionMatches(4,"dir",0,3 ))
            {
                dataPointer.setText(DataTok);
                stateSubMData=10;
            }
            else //imm Idir Iimd
            if(instrText.regionMatches(4,"imm",0,3 ) ||
               instrText.regionMatches(0,"_"  ,0,1 )    )//after ind,dir

            {
                if( instrText.regionMatches(0,"ld",0,2 )||
                    instrText.regionMatches(1,"ld",0,2 )   )
                {
                    accumulator.setText(DataTok);
                    endOfLine();
                }
                else//ad sub c b
                {
                    temporary.setText(DataTok);
                    stateSubMData=1;
                }
            }
            break;
    case 1:
            //instrText=instruction.getText();
            //Manager_mp.AM_Port_p.say( instrText );
            if(instrText.regionMatches(0,"add",0,3  )||
       instrText.regionMatches(0,"_add",0,4 )     )//_add
            {
                Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(4);
                //Manager_mp.AM_Port_p.say( "add" );

            }else
            if(instrText.regionMatches(0,"adc",0,3 )||
       instrText.regionMatches(0,"_adc",0,4 )     )
            {
```

```
                        Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(5);
                    }else
                    if(instrText.regionMatches(0,"sub",0,3 )||
                instrText.regionMatches(0,"_sub",0,4 )      )
                    {
                        Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(6);
                    }else
                    if(instrText.regionMatches(0,"sbb",0,3 )||
                instrText.regionMatches(0,"_sbb",0,4 )      )
                    {
                        Manager_mp.RU_ALU_p.ALU_Coor_p.StateAuto(7);
                    }
                    //Manager_mp.AM_Port_p.say( " eol" );

                    endOfLine();
                    break;

            case 10:
                    memAddress.setText( dataPointer.getText() );
                    stateSubMData=11;
                    break;
            case 11:
                    dataMemAdr    =Manager_mp.AM_Act_p.strToInt(
memAddress.getText() );
                    dataMemValue
=Manager_mp.RU_RAM_p.RAM_Info_p.dataList.getItem(dataMemAdr-50);
                    //tok 2
                    TokObj=new StringTokenizer(dataMemValue);
                    Token=TokObj.nextToken();
                    Token=TokObj.nextToken();

                    dataToMemBuf="_       ";  //need to ad a tok;rewrite each time
                    dataToMemBuf= dataToMemBuf.concat(Token);
                    //write mbr
                    memBuffer.setText(dataToMemBuf);

                    //write mbr
                    //memBuffer.setText(Token);//was with no _

                    Manager_mp.RU_RAM_p.RAM_Info_p.dataList.select(dataMemAdr-50);

                    if(instrText.regionMatches(4,"dir",0,3 ))
                    {
                          //dir=di
                          //instrText=instrText.substring(0,6);
                          //instruction.setText(instrText);

                          //add _ at begining of instrText
                          instrText = _Str.concat(instrText);
                          instruction.setText(instrText);

                    }else//ind
                    {
                          if(firstTime)
                          {
                                firstTime=false;
                          }else
```

```
                        {
                                //ind=in
                                //instrText=instrText.substring(0,6);
                                //instruction.setText(instrText);
                                //add _ at begining of instrText
                                instrText = _Str.concat(instrText);
                                instruction.setText(instrText);
                                firstTime=true;
                        }
                    }
                    stateSubMData=0;
                    break;

            case 30://st dir
                    memAddress.setText( dataPointer.getText() );
                    stateSubMData=31;
                    break;
            case 31:
                    memBuffer.setText(accumulator.getText());
                    stateSubMData=32;
                    break;
            case 32:
                    //st mb in dm at adr_in_mar
                    mar =Manager_mp.AM_Act_p.strToInt(memAddress.getText());
                    String storeStr=""+mar;
                    storeStr=storeStr.concat(" ");
                    storeStr=storeStr.concat(memBuffer.getText());

      Manager_mp.RU_RAM_p.RAM_Info_p.dataList.replaceItem(storeStr,mar-50);
                    endOfLine();
                    break;

            case 40://st indir
                    memAddress.setText( dataPointer.getText() );
                    stateSubMData=41;
                    break;
            case 41:
                    //mbr= dm(mar)
                    dataMemAdr  =Manager_mp.AM_Act_p.strToInt(
memAddress.getText() );

      dataMemValue=Manager_mp.RU_RAM_p.RAM_Info_p.dataList.getItem(dataMemAdr-
50);
                    //tok 2
                    TokObj=new StringTokenizer(dataMemValue);
                    Token=TokObj.nextToken();//adr
                    Token=TokObj.nextToken();//data

                    //write mbr
                    memBuffer.setText(Token);
                    stateSubMData=42;
                    break;
            case 42:
                    //dp=mb
                    dataPointer.setText(memBuffer.getText());
                    stateSubMData=30;
                    break;
```

```
            case 50:// j ind
                  memAddress.setText( dataPointer.getText() );
                  stateSubMData=51;
                  break;
            case 51:
                  //mbr= dm(mar)
                  dataMemAdr  =Manager_mp.AM_Act_p.strToInt(
memAddress.getText() );

      dataMemValue=Manager_mp.RU_RAM_p.RAM_Info_p.dataList.getItem(dataMemAdr-
50);
                  //tok 2
                  TokObj=new StringTokenizer(dataMemValue);
                  Token=TokObj.nextToken();//adr
                  Token=TokObj.nextToken();//data

                  //write mbr
                  memBuffer.setText(Token);

                  endOfLine();//PC++
                  PC=Manager_mp.AM_Act_p.strToInt(memBuffer.getText());
                  break;

            default:
                  break;
            }
      }

      private String getDataTok()
      {
            instrTextTok = new StringTokenizer(memBuffer.getText());

            InstrTok = instrTextTok.nextToken();
            return     instrTextTok.nextToken();
      }

      private void endOfLine()
      {
            stateSubMData=0;
            //sm func  has one st
            stateMS=0;
            PC++;
            endOfLine  =true;

            //check for end of list

            if( PC > (listCount-1) )
            {
                  Manager_mp.AM_Port_p.say( "End of program:push Enter to reset"
);
                  endOfProgram=true;
                  PC=0;
            }
      }
      private void reset()
      {
```

```
            stateSubMData=0;
            //sm func  has one st
            stateMS=0;
            PC=0;

            progCounter.setText("0");
            accumulator.setText("0");
            temporary  .setText("0");

            instruction.setText("0");
            memAddress .setText("0");
            memBuffer  .setText("0");

            dataPointer.setText("0");

            carry      .setText("0");;
            negative   .setText("0");;
            zero       .setText("0");;

      }

      private boolean instrHasData()
      {
            if( instrField.regionMatches(0,"clr",0,3 )) return false;
            if( instrField.regionMatches(0,"neg",0,3 )) return false;
            if( instrField.regionMatches(0,"inc",0,3 )) return false;
            if( instrField.regionMatches(0,"dec",0,3 )) return false;

            else return true;
      }

}//class

class AM_Port
{
      OneAccMicro Frame_mp;
      RU_DCoordinator Manager_mp;

      AM_Port( RU_DCoordinator RU_DCoordinator_p)
      {
            Manager_mp = RU_DCoordinator_p;
            Frame_mp   = RU_DCoordinator_p.Frame_mp;
      }

      public void action( Event event_new )
      {
            Manager_mp.event = event_new;
            SeeInput();
            Manager_mp.AM_Coor_p.StateAuto();
      }
      private void SeeInput()
      {
            Object target = Manager_mp.event.target;

            if (target instanceof TextField)
            {
                  TextField a_textField = ( TextField )target;
```

```
            }
            else

            if (target instanceof Button)
            {
                  Button button = (Button)target;

                        if(button.equals(Frame_mp.Edit) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.Edit;}
                  else if(button.equals(Frame_mp.LoadProg) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.LoadProg;}
                  else if(button.equals(Frame_mp.Execute) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.Execute;}
                  else if(button.equals(Frame_mp.AutoRun) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.AutoRun;}
                  else if(button.equals(Frame_mp.InstrStep) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.InstrStep;}
                  else if(button.equals(Frame_mp.MicroStep) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.MicroStep;}
                  else if(button.equals(Frame_mp.Append) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.Append;}
                  else if(button.equals(Frame_mp.Insert) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.Insert;}
                  else if(button.equals(Frame_mp.Delete) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.Delete;}
                  else if(button.equals(Frame_mp.AppendData) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.AppendData;}
                  else if(button.equals(Frame_mp.InsertData) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.InsertData;}
                  else if(button.equals(Frame_mp.DeleteData) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.DeleteData;}
                  else if(button.equals(Frame_mp.SaveProg) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.SaveProg;}
                  else if(button.equals(Frame_mp.EnterSaved) == true )
                  { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.EnterSaved;}

            }//if
            else
            { Manager_mp.eventArea = Manager_mp.AM_InfoBase_p.unused;}

      }

      public void say( String toSay)
      {
            Frame_mp.ToUser.setText( toSay );
      }
}

class AM_InfoBase
{
      RU_DCoordinator Manager_mp;

      AM_InfoBase( RU_DCoordinator RU_DCoordinator_p)
      {
            Manager_mp = RU_DCoordinator_p;
      }
      int unused   =0;
```

```
        int Edit     =1 ;
        int LoadProg =2 ;
        int Execute  =3 ;
        int AutoRun  =4 ;
        int InstrStep=5 ;
        int MicroStep=6 ;
        int Append   =7 ;
        int Insert   =8 ;
        int Delete   =9 ;
        int AppendData=10;
        int InsertData=11;
        int DeleteData=12;
        int SaveProg=13;
        int EnterSaved=14;
}

class AM_Act
{
      private RU_DCoordinator Manager_mp;

      AM_Act( RU_DCoordinator RU_DCoordinator_p)
      {
            Manager_mp = RU_DCoordinator_p;
      }

      private RAM_Port actMemory;

      public void init()
      {
            actMemory =Manager_mp.RU_RAM_p.RAM_Port_p;
      }

      int nrItems=0;
      void Append()
      {
          String choseListStr =Manager_mp.Frame_mp.choseList.getSelectedItem();
            String funcDataStr  =Manager_mp.Frame_mp.funcData .getText();

          int choseListIndex  =Manager_mp.Frame_mp.choseList.getSelectedIndex();

            if(choseListIndex<25)
            {
                  choseListStr=choseListStr.concat(" ");
                  choseListStr=choseListStr.concat(funcDataStr);
            }

            actMemory.addItem(choseListStr);
            if(actMemory.memActed == false)
            {
                  Manager_mp.AM_Port_p.say("no add ");
            }

      }

      void Insert()
      {
          String choseListStr =Manager_mp.Frame_mp.choseList.getSelectedItem();
```

```
            String funcDataStr  =Manager_mp.Frame_mp.funcData .getText();

          int choseListIndex  =Manager_mp.Frame_mp.choseList.getSelectedIndex();

            if(choseListIndex<25)
            {
                  choseListStr=choseListStr.concat(" ");
                  choseListStr=choseListStr.concat(funcDataStr);
            }

            int insertIx=actMemory.getSelectedIndex();
            if(insertIx >= 0)
            {
                  actMemory.insertItem(choseListStr,insertIx);

                  if(actMemory.memActed == false)
                  {
                        Manager_mp.AM_Port_p.say("no insert ");
                  }
            }
            else
            {
                  Manager_mp.AM_Port_p.say("No choice, no insert ");
            }
      }
      int toDel=0;
      void Delete()
      {
      //get nr
            int slix = actMemory.getSelectedIndex();
            if( slix >= 0 )
            {
                  actMemory.delItem(slix);
                  if(actMemory.memActed == false)
                  {
                        Manager_mp.AM_Port_p.say("no delete");
                  }
            }
            else
            {
                 Manager_mp.AM_Port_p.say("No choice, no delete ");
            }
      }

      void AppendData()
      {
            String dataMemAddStr  =Manager_mp.Frame_mp.dataMemAdd.getText();
            String utilDataStr    =Manager_mp.Frame_mp.utilData  .getText();

            int dataMemAddInt = strToInt(dataMemAddStr);

            dataMemAddStr=dataMemAddStr.concat(" ");
            dataMemAddStr=dataMemAddStr.concat(utilDataStr);

            Manager_mp.Frame_mp.dataList.replaceItem(dataMemAddStr,
dataMemAddInt-50);
            Manager_mp.Frame_mp.dataList.select(dataMemAddInt-50);
```

```
}
void InsertData()
{
}
int toDelData=0;
void DeleteData()
{
//get nr
      int slix = Manager_mp.Frame_mp.dataList.getSelectedIndex();
      slix+=50;
      Manager_mp.Frame_mp.dataList.replaceItem(""+slix,slix-50);
      for(int l=0;l<1000000;l++);
}

void Load()
{
//    Manager_mp.RU_RAM_p.RAM_Act_p.Load();
}
void EnterProg()
{
//  Manager_mp.RU_RAM_p.RAM_Act_p.DeletePrev();
}

int nrIt=0;
String LStr="";

void SaveProg()
{
      Manager_mp.Frame_mp.programMem.clear();

      nrIt =actMemory.countItems();

      for (int pix=0;pix<nrIt;pix++)
      {
            //prM=prL
            LStr=actMemory.getItem(pix);
                  Manager_mp.Frame_mp.programMem .addItem(LStr);
      }
}
void EditSaved()
{
      actMemory.clear();

      nrIt =Manager_mp.Frame_mp.programMem.countItems();

      for (int pix=0;pix<nrIt;pix++)
      {
            //prL=prM
            LStr=Manager_mp.Frame_mp.programMem .getItem(pix);
                  actMemory.addItem(LStr);
      }

}

int strToInt( String sToConvert)
{
      return Manager_mp.RU_ALU_p.ALU_Act_p.strToI(sToConvert);
```

```
	}

}


class RAM_InfoBase
{

	RU_RandomAccsessMemory RU_RAM_p;
	List programList;
	List dataList;
	List programMem;

	RAM_InfoBase( RU_RandomAccsessMemory RU_RAM_p_in )
	{
		RU_RAM_p = RU_RAM_p_in;
		programList = RU_RAM_p.Frame_mp.programList;
		dataList    = RU_RAM_p.Frame_mp.dataList;
		programMem  = RU_RAM_p.Frame_mp.programMem;
	}


}

class RAM_Coordinator extends AAStateMachine
{
	RU_RandomAccsessMemory RU_RAM_p;

	public  final static int GET_ITEM    = 20;
	public  final static int ADD_ITEM    = 21;
	public  final static int INSERT_ITEM = 22;
	public  final static int DELETE_ITEM = 23;
	public  final static int CLEAR       = 24;

	private final static int MAX_MEM = 50;

	RAM_Coordinator( RU_RandomAccsessMemory RU_RAM_p_in )
	{
		RU_RAM_p = RU_RAM_p_in;
	}

	int mem_state = 0;

	public boolean StateAuto( int mem_event )
	{
		boolean retToDo = false;

		switch( mem_state )
		{
		case 0:
			switch( mem_event )
			{
			case GET_ITEM:
				retToDo = false;
				break;
			case ADD_ITEM:
				retToDo = true;
```

```
                        mem_state = 1;
                        break;
                  case INSERT_ITEM:
                        retToDo = false;
                        break;
                  case DELETE_ITEM:
                        retToDo = false;
                        break;
                  case CLEAR:
                        retToDo = false;
                        break;
                  default:
                        break;
                  }
                  break;
            case 1:
                  switch( mem_event )
                  {
                  case GET_ITEM:
                        retToDo = true;
                        break;
                  case ADD_ITEM:
                        retToDo = true;
                        if(RU_RAM_p.RAM_Info_p.programList.countItems()+1
==MAX_MEM)
                              //cntList+1(1 for this next add) = maxmem :reached
limit
                              mem_state = 2;
                        else
                              mem_state = 1;
                        retToDo = true;
                        break;
                  case INSERT_ITEM:
                        retToDo = true;
                        if(RU_RAM_p.RAM_Info_p.programList.countItems()+1
==MAX_MEM)
                              //cntList+1(1 for this next add) = maxmem :reached
limit
                              mem_state = 2;
                        else
                              mem_state = 1;
                        retToDo = true;
                        break;
                  case DELETE_ITEM:
                        retToDo = true;
                        if( (RU_RAM_p.RAM_Info_p.programList.countItems()-1)
==0)
                              //cntList-1(1 for this next del) = 0:reached limit
                              mem_state = 0;
                        else
                              mem_state = 1;
                        retToDo = true;
                        break;
                  case CLEAR:
                        retToDo = true;
                        mem_state=0;
                        break;
```

```
                default:
                        break;
                }
                break;
        case 2:
                switch( mem_event )
                {
                case GET_ITEM:
                        retToDo = true;
                        break;
                case ADD_ITEM:
                        retToDo = false;
                        break;
                case INSERT_ITEM:
                        retToDo = false;
                        break;
                case DELETE_ITEM:
                        retToDo = true;
                        if( (RU_RAM_p.RAM_Info_p.programList.countItems()-1) ==0
)
                                //cntList-1(1 for this next del) = 0:reached limit
                                mem_state = 0;
                        else
                                mem_state = 1;//not at max mem anymore
                        break;
                case CLEAR:
                        retToDo = true;
                        mem_state=0;
                        break;
                default:
                        break;
                }
                break;
        default:
                break;
        }
        return retToDo ;

    }


}
class RAM_Port
{
    RU_RandomAccsessMemory RU_RAM_p;
    List programList;

    RAM_Port( RU_RandomAccsessMemory RU_RAM_p_in )
    {
        RU_RAM_p = RU_RAM_p_in;
        programList = RU_RAM_p.RAM_Info_p.programList;
    }

    boolean memActed = true;

    public String getItem( int ix )
    {
```

```
            if( RU_RAM_p.RAM_Coor_p.StateAuto( RU_RAM_p.RAM_Coor_p.GET_ITEM ) )
            {
                  memActed = true;
                  return programList.getItem( ix );
            }
            else
            {
                  //set warning
                  memActed = false;
                  return "";
            }
      }

      public void addItem( String sToAdd)
      {
            if( RU_RAM_p.RAM_Coor_p.StateAuto( RU_RAM_p.RAM_Coor_p.ADD_ITEM ) )
            {
                  memActed = true;
                  programList.addItem( sToAdd );
            }
            else
            {
                  //set warning
                  memActed = false;
            }
      }

      public void insertItem( String sToAdd , int ix)
      {
            if( RU_RAM_p.RAM_Coor_p.StateAuto( RU_RAM_p.RAM_Coor_p.INSERT_ITEM )
)
            {
                  memActed = true;
                  programList.addItem( sToAdd , ix);
            }
            else
            {
                  //set warning
                  memActed = false;
            }

      }

      public void delItem( int ix )
      {
            if( RU_RAM_p.RAM_Coor_p.StateAuto( RU_RAM_p.RAM_Coor_p.DELETE_ITEM )
)
            {
                  memActed = true;
                  programList.delItem( ix );
                  for(int l=0;l<1000000;l++);
            }
            else
            {
                  //set warning
                  memActed = false;
            }
```

```
	}

	public void clear()
	{
		if( RU_RAM_p.RAM_Coor_p.StateAuto( RU_RAM_p.RAM_Coor_p.CLEAR ) )
		{
			memActed = true;
			programList.clear();
		}
		else
		{
			//set warning
			memActed = false;
		}
	}
	public void select( int ix)
	{
		programList.select( ix );
	}
	public int countItems()
	{
		return programList.countItems();
	}
	public int getSelectedIndex()
	{
		return programList.getSelectedIndex();
	}


}
class RAM_Act
{

	RU_RandomAccsessMemory RU_RAM_p;
	List programMem;

	String                  memListLine;
	StringTokenizer             TokObj;
	String                          Token;

	int function ;
	int   addrMode ;
	int   workData ;
	int lix=0;
	int programMemCount=0;

	RAM_Act( RU_RandomAccsessMemory RU_RAM_p_in )
	{
		RU_RAM_p = RU_RAM_p_in;
	}

	void DeletePrev()
	{
	//    RU_RAM_p.RU_DCoordinator_p.Frame_mp.programList.clear();
	}
```

```
}

class Regs_Coordinator extends AAStateMachine
{

	RU_Registers RU_Regs_p;
	Regs_Coordinator( RU_Registers RU_Regs_p_in )
	{
		RU_Regs_p = RU_Regs_p_in;
		state = 0;
	}

	public void StateAuto()
	{
	}
}

class Regs_Port
{
	RU_Registers RU_Regs_p;

	Regs_Port( RU_Registers RU_Regs_p_in )
	{
		RU_Regs_p = RU_Regs_p_in;
	}

	public void action()
	{
		Event event = RU_Regs_p.RU_DCoordinator_p.event;
	}
}

class Regs_Act
{
	RU_Registers RU_Regs_p;

	Regs_Act( RU_Registers RU_Regs_p_in )
	{
		RU_Regs_p = RU_Regs_p_in;
	}
}

class Regs_InfoBase
{
	RU_Registers RU_Regs_p;

	int progCounter=0;
	int accumulator=0;
	int temporary  =0;

	int instruction=0;
	int memAddress =0;
	int memBuffer  =0;

	int carry      =0;
	int negative   =0;
	int zero       =0;
```

```
        Regs_InfoBase( RU_Registers RU_Regs_p_in )
        {
              RU_Regs_p = RU_Regs_p_in;
        }
        public void reset()
        {
        }
}


class ALU_Port
{
        RU_ArithmeticLogicUnit RU_ALU_p;

        ALU_Port( RU_ArithmeticLogicUnit RU_ALU_p_in )
        {
              RU_ALU_p = RU_ALU_p_in;
        }

        public void action()
        {
        }

}

class ALU_InfoBase
{
        RU_ArithmeticLogicUnit RU_ALU_p;


        ALU_InfoBase( RU_ArithmeticLogicUnit RU_ALU_p_in )
        {
              RU_ALU_p = RU_ALU_p_in;

        }

        public void reset()
        {
        }
}

class ALU_Coordinator extends AAStateMachine
{

        RU_ArithmeticLogicUnit RU_ALU_p;
        boolean endOfProgram=false;
        int Line=0;
        boolean endOfLine = false;

        ALU_Coordinator( RU_ArithmeticLogicUnit RU_ALU_p_in )
        {
              RU_ALU_p = RU_ALU_p_in;
              state = 0;
        }
```

```
        public void StateAuto(int event)
        {
              switch(event)
              {
              case 0://clear
                    RU_ALU_p.ALU_Act_p.clr();
                    break;
              case 1://neg
                    RU_ALU_p.ALU_Act_p.neg();
                    break;
              case 2://inc
                    RU_ALU_p.ALU_Act_p.inc();
                    break;
              case 3://dec
                    RU_ALU_p.ALU_Act_p.dec();
                    break;

              case 4:
                    RU_ALU_p.ALU_Act_p.add();
                    break;
              case 5:
                    RU_ALU_p.ALU_Act_p.adc();
                    break;
              case 6:
                    RU_ALU_p.ALU_Act_p.sub();
                    break;
              case 7:
                    RU_ALU_p.ALU_Act_p.sbb();
                    break;
              default:
                    break;
              }
        }
}

class ALU_Act
{
        RU_ArithmeticLogicUnit RU_ALU_p;
        RAM_InfoBase                RAM_Info_p;
        Regs_InfoBase               Regs_Info_p;

        int func=0;
        int toAdd=0;


        TextField accumulator;
        TextField temporary  ;

        TextField carry      ;
        TextField negative   ;
        TextField zero       ;

        String minusStr="-";

        ALU_Act( RU_ArithmeticLogicUnit RU_ALU_p_in )
        {
              RU_ALU_p   = RU_ALU_p_in;
```

```
        accumulator= RU_ALU_p.Frame_mp.accumulator;
        temporary  = RU_ALU_p.Frame_mp.temporary;

        carry      = RU_ALU_p.Frame_mp.carry;
        negative   = RU_ALU_p.Frame_mp.negative;
        zero       = RU_ALU_p.Frame_mp.zero;

}

public void init()
{
        RAM_Info_p = RU_ALU_p.RU_DCoordinator_p.RU_RAM_p.RAM_Info_p;
        Regs_Info_p= RU_ALU_p.RU_DCoordinator_p.RU_Regs_p.Regs_Info_p;
}


public void clr()
{
        accumulator.setText("0");
        setFlags(0,0,1);//CNZ
}
public void neg()
{
        if( (accumulator.getText() ).charAt(0) == 0x2d )//already neg
        {
                accumulator.setText( (accumulator.getText()).substring(1) );
                setFlags(0,0,0);//CNZ
        }
        else
        if( accumulator.getText().equals("0") )
        {
                setFlags(0,0,1);//CNZ
                //do noth.
        }
        else
        {
                accumulator.setText( minusStr.concat(accumulator.getText() ));
                setFlags(0,1,0);//CNZ
        }
}
public void inc()
{
        int acc= strToI( accumulator.getText() );
        acc++;
//      acc = setFlagsAcc(acc);

        accumulator.setText(""+acc);
}
public void dec()
{
        int acc= strToI( accumulator.getText() );
        acc--;
//      acc = setFlagsAcc(acc);

        accumulator.setText(""+acc);
}
```

```
public void add()
{
	int acc= strToI( accumulator.getText() );
	int tem= strToI( temporary  .getText() );
	acc = acc + tem;
	acc = setFlagsAcc(acc);

	accumulator.setText(""+acc);
}

public void adc()
{
	int acc= strToI( accumulator.getText() );
	int tem= strToI( temporary  .getText() );

	acc = acc + tem + getCarry();
	acc = setFlagsAcc(acc);

	accumulator.setText(""+acc);
}

public void sub()
{
	int acc= strToI( accumulator.getText() );
	int tem= strToI( temporary  .getText() );
	acc = acc  - tem ;
	acc = setFlagsAcc(acc);

	accumulator.setText(""+acc);

}
public void sbb()
{
	int acc= strToI( accumulator.getText() );
	int tem= strToI( temporary  .getText() );
	acc = acc  - tem + getCarry();
	acc = setFlagsAcc(acc);

	accumulator.setText(""+acc);
}

public int strToI( String sToConvert)
{
	int sIsInt=0;
	boolean negative=false;
	int Nr=0;

	sToConvert=sToConvert.trim();
	char Char = sToConvert.charAt(0);
	int length= sToConvert.length();

	if( Char == 0x2d)//minus -
	{
		negative=true;
		//clear it
		if( length >0 )
```

```
                {
                        sToConvert=sToConvert.substring(1);
                        length--;
                }
                else
                {
                        return 1000;
                }
        }
        for(int ix=0;ix<length;ix++)
        {
                Nr = sToConvert.charAt(ix)-0x30 ;
                for(int pw=0;pw<length-ix-1;pw++)
                {
                        Nr*=10;
                }
                sIsInt += Nr;
        }
        if(negative)
                sIsInt=-sIsInt;

        return sIsInt;
  }

private int setFlagsAcc( int accF)
  {
             if(accF > 100) setFlags(1,0,0);//CNZ
        else if(accF== 100) setFlags(1,0,1);//CNZ
        else if(accF< -100) setFlags(1,1,0);//CNZ
        else if(accF==-100)     setFlags(1,1,1);//CNZ
        else if(accF== 0  ) setFlags(0,0,1);//CNZ
        else if(accF<  0  ) setFlags(0,1,0);//CNZ
        else if(accF>  0  ) setFlags(0,0,0);//CNZ

        if (accF>=  100)  accF -= 100;
        if (accF<= -100)  accF += 100;

        return accF;

  }

  private void setFlags(int Carry, int Negative, int Zero)
  {
        if(Carry   ==0) carry   .setText("0");
        else            carry   .setText("1");
        if(Negative==0) negative.setText("0");
        else            negative.setText("1");
        if(Zero    ==0) zero    .setText("0");
        else            zero    .setText("1");

  }

  private int getCarry()
  {
        int isCarry=0;
        if( ( carry.getText() ).charAt(0) == 0x31 )//C=1
        {
```

```
                if( ( negative.getText() ).charAt(0) == 0x31 )//N=1
                        isCarry = -100;
                else//N=0
                        isCarry =  100;
        }
        return isCarry;
    }

}
```

B.2 RISC Microprocessor Simulator (main differences)

```
.................
class R_AM_Coordinator extends R_AAStateMachine
{

      R_RU_DCoordinator Manager_mp;
      int PC=0;
      int listCount=0;
      boolean endOfProgram       = false;
      boolean endOfLine          = false;
..............
      R_AM_Coordinator( R_RU_DCoordinator R_RU_DCoordinator_p)
      {
            Manager_mp = R_RU_DCoordinator_p;
            state = 1;//Start
.............
      }

      private R_RAM_Port actMemory;
..............

      void ResponseManager_MicroStep()
      {
            switch( stateMS )
            {
            case 0:
                  //write pc
                  progCounter.setText(""+PC);
                  stateMS =1;
                  break;
            case 1:
                  //write mad
                  memAddress.setText( progCounter.getText() );
                  actMemory.select(PC);
                  stateMS =2;
                  break;
            case 2:
                  //get mem at mad =PC
                  memAtMad=actMemory.getItem(PC);
                  if(actMemory.memActed == false)
                  {
                        Manager_mp.R_AM_Port_p.say("no data: enter ");
                  }
                  //write mbr
                  memBuffer.setText(memAtMad);
                  //next state:
                  stateMS =3;
                  break;
            case 3:
                  // write ir
                  instruction.setText(memBuffer.getText());
                  instrText=instruction.getText();

                  if( instrHasData() )
                  {
                        stateMS =5;
                  }
                  else //instr has no data
```

```
            {
                  stateMS =4;
            }
            break;
      case 4:
            subM_InstrNoData();
            stateMS =0;
            break;
      case 5:
            subM_InstrWithData();
            //stateMS =0;in subm
            break;

      default:
            break;
      }
}


int stateSubMInstr = 0;
StringTokenizer            TokObj;
String                           Token;

private void subM_InstrNoData()
{
      switch(stateSubMInstr)
      {
      case 0:

            if( instrText.regionMatches(0,"clr",0,3 ))
            {
                  Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(0);
            }else
            if( instrText.regionMatches(0,"neg",0,3 ))
            {
                  Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(1);

            }else
            if( instrText.regionMatches(0,"inc",0,3 ))
            {
                  Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(2);

            }else
            if( instrText.regionMatches(0,"dec",0,3 ))
            {
                  Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(3);

            }else
            if(instrText.regionMatches(0,"add",0,3 ))
            {
                  Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(4);
            }else
            if(instrText.regionMatches(0,"adc",0,3 ))
            {
                  Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(5);
            }else
            if(instrText.regionMatches(0,"sub",0,3 ))
```

```
                {
                        Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(6);
                }else
                if(instrText.regionMatches(0,"sbb",0,3 ))
                {
                        Manager_mp.R_RU_ALU_p.R_ALU_Coor_p.StateAuto(7);
                }
                break;
          default:
                break;

          }
          endOfLine();//incl PC++;

    }

.................
    private void subM_InstrWithData()
    {
          switch(stateSubMData)
          {
          case 0:

                DataTok  = getDataTok();

                if(instrText.regionMatches(0,"st__dir",0,7 ))
                {
                      dataPointer.setText(DataTok);
                      stateSubMData=30;
                }
                else
...............
                if(instrText.regionMatches(4,"imm",0,3 ) ||
                   instrText.regionMatches(0,"_"   ,0,1 )    )//after ind,dir

                {
                      if( instrText.regionMatches(0,"ld",0,2 ) ||
                            instrText.regionMatches(1,"ld",0,2 )   )//after
ind,dir
                      {
                            //need third tok

    reg_k_int=Manager_mp.R_AM_Act_p.strToInt(getThirdTok());

    Manager_mp.Frame_mp.register[reg_k_int].setText(DataTok);
                            endOfLine();
                      }
                }
                break;
.............

          case 30://st dir
                memAddress.setText( dataPointer.getText() );
                stateSubMData=31;
                break;
          case 31:
                //need third tok
```

```
                    reg_k_int=Manager_mp.R_AM_Act_p.strToInt(getThirdTok());
                    toStore=Manager_mp.Frame_mp.register[reg_k_int].getText();

                    memBuffer.setText(toStore);
                    stateSubMData=32;
                    break;
              case 32:
                    //st mb in dm at adr_in_mar
                    mar =Manager_mp.R_AM_Act_p.strToInt(memAddress.getText());
                    String storeStr=""+mar;
                    storeStr=storeStr.concat(" ");
                    storeStr=storeStr.concat(memBuffer.getText());

        Manager_mp.R_RU_RAM_p.R_RAM_Info_p.dataList.replaceItem(storeStr,mar-50);
                    endOfLine();
                    break;

              default:
                    break;
              }
        }
.....................
}

class R_AM_Act
{
        private R_RU_DCoordinator Manager_mp;
        private risc Fr;

        public R_AM_Act( R_RU_DCoordinator R_RU_DCoordinator_p)
        {
              Manager_mp= R_RU_DCoordinator_p;
              Fr        = R_RU_DCoordinator_p.Frame_mp;
        }

        private R_RAM_Port actMemory;

        public void init()
        {
              actMemory =Manager_mp.R_RU_RAM_p.R_RAM_Port_p;
        }

...................
        public void Append()
        {
              prepareAdd();
              actMemory.addItem(choseListStr);
              if(actMemory.memActed == false)
              {
                    Manager_mp.R_AM_Port_p.say("no add ");
              }
        }
        public void Insert()
        {
              prepareAdd();
              int insertIx=actMemory.getSelectedIndex();
              if(insertIx >= 0)
```

```
            {
                  actMemory.insertItem(choseListStr,insertIx);

                  if(actMemory.memActed == false)
                  {
                        Manager_mp.R_AM_Port_p.say("no insert ");
                  }
            }
            else
            {
                  Manager_mp.R_AM_Port_p.say("No choice, no insert ");
            }
      }
      int toDel=0;
      void Delete()
      {
      //get nr
            int slix = actMemory.getSelectedIndex();
            if( slix >= 0 )
            {
                  actMemory.delItem(slix);
                  if(actMemory.memActed == false)
                  {
                        Manager_mp.R_AM_Port_p.say("no delete");
                  }
            }
            else
            {
                  Manager_mp.R_AM_Port_p.say("No choice, no delete ");
            }
      }

.....................
}
.....................
```

B.3 STACK Microprocessor Simulator (main differences)

```
.....

    void ResponseManager_MicroStep()
    {
          switch( stateMS )
          {
          case 0:
                //write pc
                progCounter.setText(""+PC);
                stateMS =1;
                break;
          case 1:
                //write mad
                memAddress.setText( progCounter.getText() );
                actMemory.select(PC);
                stateMS =2;
                break;
          case 2:
                //get mem at mad =PC
                memAtMad=actMemory.getItem(PC);
                if(actMemory.memActed == false)
                {
                      Manager_mp.S_AM_Port_p.say("no data: enter ");
                }
                //write mbr
                memBuffer.setText(memAtMad);
                //next state:
                stateMS =3;
                break;
          case 3:
                // write ir
                instruction.setText(memBuffer.getText());
                instrText=instruction.getText();

                if( instrHasData() )
                {
                      stateMS =5;
                }
                else //instr has no data
                {
                      stateMS =4;
                }
                break;
          case 4:
                subM_InstrNoData();
                stateMS =0;
                break;
          case 5:
                subM_InstrWithData();
                //stateMS =0;in subm
                break;

          default:
                break;
          }
    }
```

```
.................
      private void subM_InstrWithData()
      {
            switch(stateSubMData)
            {
            case 0:

                  DataTok  = getDataTok();

                  if(instrText.regionMatches(0,"st__dir",0,7 ))
                  {
                        dataPointer.setText(DataTok);
                        stateSubMData=30;
                  }
....              else
                  if(instrText.regionMatches(4,"imm",0,3 ) ||
                     instrText.regionMatches(0,"_"  ,0,1 )    )//after ind,dir

                  {
                        if( instrText.regionMatches(0,"ld",0,2 ) ||
                              instrText.regionMatches(1,"ld",0,2 )   )//after
ind,dir
                        {
                              Manager_mp.RU_S_Regs_p.S_Regs_Coor_p.push( DataTok
);
                              if(Manager_mp.RU_S_Regs_p.S_Regs_Coor_p.actStack
== false)
                                    Manager_mp.S_AM_Port_p.say( "ld not
done:stack full" );
                              endOfLine();
                        }
                  }
                  break;
.......
            case 30://st dir
                  memAddress.setText( dataPointer.getText() );
                  stateSubMData=31;
                  break;
            case 31:
                  toStore=Manager_mp.RU_S_Regs_p.S_Regs_Coor_p.pull();
                  memBuffer.setText(toStore);
                  stateSubMData=32;
                  break;
            case 32:
                  //st mb in dm at adr_in_mar
                  mar =Manager_mp.S_AM_Act_p.strToInt(memAddress.getText());
                  String storeStr=""+mar;
                  storeStr=storeStr.concat(" ");
                  storeStr=storeStr.concat(memBuffer.getText());

      Manager_mp.RU_S_RAM_p.S_RAM_Info_p.dataList.replaceItem(storeStr,mar-50);
                  endOfLine();
                  break;

.........         default:
                  break;
            }
```

```
        }

.................
class S_AM_Act
{
        private S_RU_DCoordinator Manager_mp;
        private stack Fr;

        public S_AM_Act( S_RU_DCoordinator RU_DCoordinator_p)
        {
                Manager_mp= RU_DCoordinator_p;
                Fr        = RU_DCoordinator_p.Frame_mp;
        }

        private S_RAM_Port actMemory;

        public void init()
        {
                actMemory =Manager_mp.RU_S_RAM_p.S_RAM_Port_p;
        }

        private int nrItems=0;
        private String choseListStr="";
        private int choseListIndex  =0;

        private void prepareAdd()
        {
            choseListStr  =Fr.choseList.getSelectedItem();
            choseListIndex=Fr.choseList.getSelectedIndex();

              if(choseListIndex<5)//ld st
              {
                      choseListStr=choseListStr.concat(" ");
                      choseListStr=choseListStr.concat(Fr.funcData.getText());
              }
              else
              if(choseListIndex<9)//ad ...
              {
              }
              else
              if(choseListIndex<17)//j
              {
                      choseListStr=choseListStr.concat(" ");
                      choseListStr=choseListStr.concat(Fr.funcData.getText());
              }
              else //cl ..
              {
              }
              //ch 0 7
        }

        public void Append()
        {
                prepareAdd();
                actMemory.addItem(choseListStr);
                if(actMemory.memActed == false)
                {
```

```
                Manager_mp.S_AM_Port_p.say("no add ");
            }
        }
        public void Insert()
        {
            prepareAdd();
            int insertIx=actMemory.getSelectedIndex();
            if(insertIx >= 0)
            {
                actMemory.insertItem(choseListStr,insertIx);

                if(actMemory.memActed == false)
                {
                    Manager_mp.S_AM_Port_p.say("no insert ");
                }
            }
            else
            {
                Manager_mp.S_AM_Port_p.say("No choice, no insert ");
            }
        }
.................
}


.................
class S_Regs_Coordinator extends S_AAStateMachine
{

        S_RU_Registers RU_S_Regs_p;
        S_Regs_Coordinator( S_RU_Registers RU_S_Regs_p_in )
        {
            RU_S_Regs_p = RU_S_Regs_p_in;
            state = 0;
        }

        public boolean actStack = false;// act can be checked from exterior

        private int nrOfFilledRegisters = 0;

      public String pull()
        {
            String valueOut = "0";

            actStack = StateAuto(0);// event = pull
            if( actStack )
            {

                valueOut=RU_S_Regs_p.S_Regs_Info_p.register[0].getText();

                String regVal;

                for(int rg=1 ; rg < nrOfFilledRegisters ; rg++ )
                {
                    //reg[rg] -> reg[rg-1]
                    regVal=RU_S_Regs_p.S_Regs_Info_p.register[rg].getText();
```

```
                        RU_S_Regs_p.S_Regs_Info_p.register[rg-
1].setText(regVal);

                        //clear unused reg
                        RU_S_Regs_p.S_Regs_Info_p.register[rg].setText("0");
                  }

                  nrOfFilledRegisters--;
            }

            return valueOut;
      }
      public void push( String valueIn )
      {
            actStack = StateAuto(1);// event = push
            if( actStack )
            {
                  String regVal;

                  for(int rg=nrOfFilledRegisters ; rg>0 ;rg-- )
                  {
                        //reg[rg-1] -> reg[rg]
                        regVal=RU_S_Regs_p.S_Regs_Info_p.register[rg-
1].getText();

                        RU_S_Regs_p.S_Regs_Info_p.register[rg].setText(regVal);
                  }

                  RU_S_Regs_p.S_Regs_Info_p.register[0].setText( valueIn );

                  nrOfFilledRegisters++;
            }
      }
      public void reset()
      {
            actStack = StateAuto(2);// event = reset
            if( actStack )
            {
                  for(int regNr=0;regNr<8;regNr++)
                  {
                        RU_S_Regs_p.S_Regs_Info_p.register[regNr].setText("0");
                  }
                  nrOfFilledRegisters=0;
            }
      }

      private boolean StateAuto(int event)
      {
            boolean retToDo = false;

            switch(state)
            {
            case 0: //stack is empty
                  switch(event)
                  {
                  case 0: //pull
                        retToDo = false;
```

```
                break;
        case 1: //push
                retToDo = true;
                state = 1;
                break;
        case 2: //reset
                retToDo = true;
                state = 0;
                break;

        default:
                break;
        }

        break;

case 1:
        switch(event)
        {
        case 0: //pull
                retToDo = true;
                if( (nrOfFilledRegisters-1) == 0) state = 0;
                else                              state = 1;

                break;
        case 1: //push
                retToDo = true;
                if( (nrOfFilledRegisters+1) == 8) state = 2;
                else                              state = 1;

                break;
        case 2: //reset
                retToDo = true;
                state = 0;

                break;

        default:
                break;
        }

        break;

case 2: // stack is full
        switch(event)
        {
        case 0: //pull
                retToDo = true;
                state = 1;
                break;
        case 1: //push
                retToDo = false;
                break;
        case 2: //reset
                retToDo = true;
                state = 0;
                break;
```

```
                default:
                        break;
                }

                break;

        default:
                break;
        }
        return retToDo;
    }
}
```