An Algebra Of Fixpoints For Characterizing Interactive Behavior Of Information Systems

An Algebra Of Fixpoints For Characterizing Interactive Behavior Of Information Systems

Der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus
vorgelegte Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
von
Diplom-Ing. Srinath Srinivasa
geboren am 09. Mai 1973 in Bangalore (Indien)

Gutachter: Prof. Dr. Klaus Jantke
Gutachter: Prof. Dr. Bernhard Thalheim
Gutachter: Prof. Dr. Myra Spiliopoulou

Tag der mündlichen Prüfung: 26. April 2001

Abstract

The dynamics of an information system (IS) is characterized not only by its computational behavior, but also by its interactive behavior. Interactive dynamics forms an integral part of most information systems. Despite this, an understanding of the interactive nature of an IS is still low.

Interaction impacts expressiveness of an IS at such fundamental levels that Wegner [Weg97, WG99a] came with a contention saying interactive behavior cannot be modeled by Turing Machines (TMs). A TM is considered the foundational model of computation. It models computable functions that map between problem and solution domains. However, a TM models only non-interactive mappings. A mapping between a problem and a solution domain that is interactive in nature can change its direction of computation resulting from intermediate interactions. Based on this contention, Wegner proposes interaction (rather than computation) as the fundamental framework for IS modeling [WG99].

In this thesis, we address Wegner's contention and the nature of interactive dynamics. An information system is modeled as a collection of *semantic processes* or *Problem Solving Processes (PSPs).* If these PSPs are interactive in nature, they are called *open* systems; and if they are non-interactive, such an IS is called a *closed* system. Intuitively, open system dynamics are known to be richer than closed system dynamics.

We make this distinction precise in this thesis. Interaction is shown to be made up of three properties: *computation*, *persistence of state across computations*, and *channel sensitivity.* Persistence of state and channel sensitivity each contribute to richer behavioral semantics than just computation. This is shown by introducing a concept called the *solution space* of a semantic process. A solution space is the abstract domain characterized by the process dynamics. Interactive solution spaces are found to be richer than algorithmic solution spaces and also interactive solution spaces require at least a three-valued system of logic for their characterization.

The earlier question of interactive behavior as applied to IS design is then revisited. Interactive dynamics of an IS characterize the IS functionality. We call the solution space of interactive IS behavior as its *interaction space.* The interaction space of an IS is contrasted with the *object space* of the IS which is concerned with the IS structure and state maintenance dynamics. The interaction space has a degree of autonomy with respect to the object space. This aspect is often not acknowledged in IS design, resulting in the intermixing of structural and functionality concerns. Separating these concerns can avoid certain conflicting problems in IS design, as well as provide better maintainability. We call this the "dual" nature of open systems.

Based on this insight we propose an IS design paradigm called *dualism*, where an IS model is made up of an *object schema*, characterizing the IS structure and an *interaction schema*, characterizing the IS functionality. The interaction schema is characterized by a three-valued system of logic, representing a set of *obligated* (or liveness) behavior, *permitted* (or possible) behavior and *forbidden* behavior. The system *should* perform the obligated behavior to be termed functional; it *may* perform any of the permitted behavior and it *may not* perform forbidden behavior. An analysis of the dynamics of any real world system can make these three-valued characteristics apparent.

Domain theory is used to propose solution space concept, and deontic logic is used to represent the three modalities of interactive IS behavior.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

What is laid down, ordered, factual is never enough to embrace the whole truth: life always spills over the rim of every cup. *–Boris Pasternak*

1.1 Open Systems and Interaction Space

Information System (IS) design has evolved through various paradigms over the years. They address dynamic systems which essentially have two dimensions of concern – the static structure and dynamic behavior.

In early paradigms of IS design, static and dynamic aspects were modeled separately. Static aspects were addressed by data and concept modeling paradigms like hierarchical, networked, relational, ER, etc. Similarly, dynamic aspects were addressed by process modeling paradigms like structured programming, modular design, data flow, etc.

With the proposal of object orientation, there were some claims that the above two aspects of system design have been unified. This is because of the encapsulation of structure and behavior into a single abstraction called the object. Contentions were made that a system is just a complex object, and system dynamics are simply the dynamics defined in the object's methods. However despite such a claim, current modeling paradigms can be placed along two complementary streams called the "object centric" and "process centric" streams respectively. In the object centric stream, a system is built from domain entities (or objects, or actors [AMST92], or agents [AR96, Wei99]) which interact with one another to achieve specific objectives. Interaction is defined "on top" of the domain entities. In the process centric stream, a system is built from processes which achieve specific goals, and which affect different shared variables of the system. Domain entities like objects or actors exist in order to facilitate process execution. Some examples are use cases [UML], workflows [Law97], process patterns [Cop98], etc.

Figure 1.1 depicts these streams schematically. A discernible split exists between object

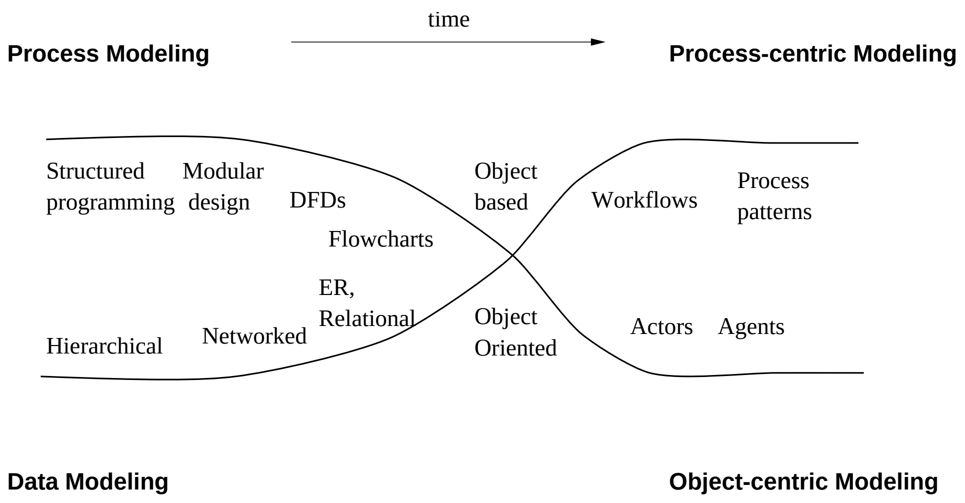


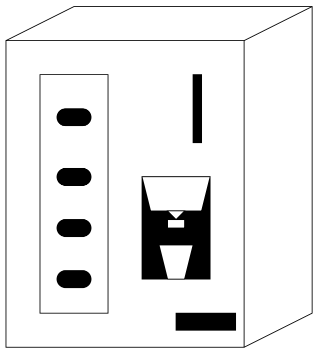
Figure 1.1: Evolution of IS Modeling Paradigms

centric and process centric streams despite the contention that a system is after all, just an object.

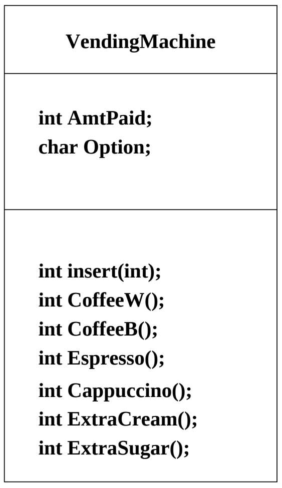
The existence of such a split can be attributed to the *open* nature of IS dynamics. Open systems are systems whose dynamics are defined not only by computations within the system, but also by *interactions* with one or more environments. It is not only necessary to specify and model the computational semantics, but the interactive semantics are also important. Process centric streams regard interactive dynamics as fundamental, while the object centric streams consider computational dynamics to be more fundamental than interactive dynamics.

To better understand the nature of open system dynamics consider a simple example of modeling a coffee vending machine. This is shown in Figure 1.2. The machine vends different flavors of coffee and has a few other options like extra cream and extra sugar. Each flavor of coffee and options come with their own cost. The task of the IS designer is to create a software model that works according to the vending machine specifications.

A typical method of addressing such a problem would be to model the vending machine as an object with its buttons and coin slots as the methods. This is shown in Figure 1.2(b). The behavior of the vending machine is now said to be defined in its methods. However, the *functionality* of the vending machine is not only dependent on its method behaviors, but also on *sequences of method invocation.* For successful usage of the vending machine, it is not only necessary to define method behaviors, it is also necessary to define correct sequences of method invocations. Figure 1.2(c) shows some interaction sequences in bold that would result in successful functioning of the machine, and few other sequences which would not be successful. This interactive aspect makes it insufficient to represent a system as an object, and its dynamics as the methods. We call the space of all such sequences of interaction as the *interaction space* of the IS. While the example of a vending machine is simple, the interaction space of any fairly large IS would be very complex. In fact, an understanding of what constitutes an interaction space is still largely unclear in the IS community. For this, we need to first understand the components that make up an interactive activity, before being able to model an interaction space.



(a)



(b)

```
insert(50) --> insert(50) --> CoffeeW();
insert(50) --> insert(100) --> ExtraSugar() --> CoffeeB();
insert(100) --> insert(100) --> Espresso();

insert(50) --> insert(50) --> Espresso();
insert(100) --> ExtraCream() --> CoffeeW();
insert(50) --> CoffeeB();
```

(c)

Figure 1.2: Vending Machine Example

Most real world information systems are interactive in nature. Interactive dynamics that are "open" produce richer behavior compared to "closed" algorithmic dynamics. Interaction is also a necessary ingredient for "intelligent" characteristics of information systems like emergent behavioral properties, reflection, learning and evolution.

1.2 Interaction Modeling

Open systems are common in real life. Most information systems model open systems having complex interaction spaces. Despite this, an understanding of the nature of open systems is still low. Indeed, the term "open system" is itself defined with various connotations. It has been used to signify different properties like extensible design [Nut92], evolvability [AMST92], multiple interfaces [Awa93], and irreducibility to functions [Mil96, WG99, WG99a, WG99b].

We formalize the notion of an open system by defining it as a system whose semantic processes are interactive in nature. Open systems display complex interactive dynamics in addition to computational dynamics.

In order to understand the nature of open system dynamics, it is necessary to understand what constitutes interactive behavior, and address interaction modeling in a domain-independent fashion. Wegner [Weg97, WG99, WG99a] was the one of the first to propose domain independent models of interaction in the form of *Interaction Machines (IMs).* Interaction is claimed to be more powerful than algorithms, and interactive computing is claimed to be irreducible to Turing Machines. IMs are proposed as an alternative abstraction for open systems and "empirical computer science."

In this thesis, we address Wegner's contention, and domain independent models of interaction to identify the essential ingredients that make up interaction. Some preliminary ideas proposed by Wegner are extended and formalized to show that interaction is made up of three ingredients: *computation*, *persistence of state* and *channel sensitivity.* By introducing a concept called the *solution space* of a semantic process, we show that persistence of state and channel sensitivity each characterize progressively richer domains than just computation.

1.3 The Dual Nature of Open Systems

With an insight into what constitutes interactive behavior, we return to the question of modeling the interaction space of an IS. Open systems (most IS) that have interactive dynamics are shown to have two kinds of behaviors. They may be termed the "database" behavior and the "interactive" behavior respectively [GST00].

The "database" behavior of an IS concerns maintenance and updation of the system state. The pertinent issues here include characterization of the system state, integrity constraints on the IS structure, and reliable transitions between system states. Database dynamics require ACID (atomicity, consistency, isolation, durability) properties.

On the other hand, an IS also has a functionality or "interaction" dimension to its behavior. The interaction aspect of an IS concerns how semantic processes can be executed by interacting with the IS. It has to address issues like dealing with divergent interaction paths, coordination among multiple interactive processes and maintenance of the system's dynamic integrity. Interactive dynamics are not necessarily amenable to properties of atomicity and isolation.

Interactive dynamics are autonomous to some degree with respect to database dynamics. Different architectures for maintaining system state can adopt the same functionality;

and different functionalities can be manifested on a single structure. For example, an interactive process for vending train tickets can be manifested in different ways: over the counter where a human being represents the IS, over the WWW where a user interacts with a WWW server, over telephone, etc. The different manifestations represent the same *functionality* (vending tickets) and are bound by the same kind of integrity constraints (*Ex.* no reservation before 60 days of planned departure, no refund if cancellation is done later than 30 minutes before departure time, etc.) In different implementations, the system state may be maintained in different ways like – a relational database, a flat file system, or maybe simply as paper files in a cupboard. The "database" behavior of the IS in this case concerns issues like how to efficiently maintain the reservation data; how to ensure integrity of the data in case of updates and deletions; etc. The "functionality" aspect of the IS concerns issues like designing different interaction protocols; enforcing policy decisions regarding reservations and cancellations; coordinating between multiple requests; optimizing the interactive process; etc.

We call this the "dual" nature of information systems. If the dual nature of IS concerns are not recognized and treated separately, the IS designer may encounter problems that pose contradictory requirements. Some examples are given below:

- Transaction processing in database design define transactions as "semantic processes." Traditionally, transaction processing addressed maintenance of database consistency in the form of ACID properties. However when transactions are extended to long duration transactions, the "virtue" of ACID properties becomes a "vice" in the form of rollback cost [GR93, Kim95]. We contend that this problem occurs because a long duration transaction belongs to the functionality behavior of an IS, rather than to the database behavior. Typical long duration transactions are interactive in nature and address issues like coordination and collaboration. A transaction is implicitly expected to exhibit algorithmic properties. However, a semantic process can be interactive in nature where non conformance to atomicity and isolation properties is not a vice, but could well be a requirement.

- The field of agent computing [AR96, ES99, SBD+00] has the problem of resolving between agent autonomy vs system integrity. With the use of agent autonomy, complex behavioral properties may be manifested using simple rules. However, arbitrary behavior on the part of a system of autonomous agents cannot ensure integrity of the system as a whole. This brings out the need for explicitly characterizing integrity of system dynamics.

In this thesis, we propose a modeling paradigm called "dualism" that models the dual nature of information systems. An IS is said to have two "spaces" of concern – the *object space* represents the structural properties, and its dynamics alter the system state. The *interaction space* represents interactive properties, and its dynamics manifest the system's functionality. The dynamics of an interaction space is shown to be characteristically different from that of an object space. Interaction space dynamics require at least a three-valued system of logic for their characterization. These are respectively called: *obligated* or liveness behavior, *permitted* or possible behavior and *forbidden* behavior. An IS that adheres to its dynamic integrity *should* perform the obligated behavior; *may* perform one or more permitted behaviors and *may not* perform forbidden behaviors.

Theoretical underpinnings for the solution space of an interactive system are based on domain theory, and logical underpinnings for the interaction space in a dualism model is based on deontic logic. A process of specification, modeling and verification of a dualism model is also proposed.

1.4 Contributions of the thesis

The contributions of the thesis may be enumerated as follows:

- Open systems are formalized by defining them as systems having interactive dynamics
- Interactive behavior is identified to be made up of three properties: *computation*, *persistence of state* and *channel sensitivity*
- A concept called the "solution space" of a problem solving process is proposed that shows the rich nature of an interactive process as against an algorithmic process
- Interactive solution spaces are shown to require at least a three-valued logic for their characterization
- Deontic constructs and assertions are proposed to characterize open system dynamics and to reason about open system dynamics respectively
- An open system is shown to have a dualism of properties which have a degree of autonomy with respect to the other. A modeling paradigm called *dualism* that models an IS as a set of two schemata: an object schema modeling the IS structure and an *interaction schema* modeling the IS functionality.
- A notion of inheritance of dynamics is proposed and contrasted with the conventional notion of inheritance
- Paradigms for validating and limited verification of dynamic integrity are proposed.

1.5 Organization of the thesis

This thesis is organized into two parts. Part I of the thesis addresses the concept of interactive behavior. The driving force behind our approach is based on Wegner's contention and proposal of Interaction Machines. Interaction is shown to be made up of three properties, each providing greater expressiveness over the other. Different existing approaches to model interactive behavior are considered and contrasted based on how they address these three ingredients of interaction. A concept of solution space is also introduced to formally characterize the behavior of interactive systems.

Part II of the thesis concerns modeling the interaction space of an information system. The concept of a solution space introduced in Part I is used to propose a modeling paradigm called dualism. The dualism paradigm models IS functionality in the form of an *interaction schema*. Building blocks of an interaction schema called dialogs and constrained associations are introduced. A process framework addressing different stages of interaction schema design like specification, modeling and implementation is also proposed.

Part I

Interaction Modeling

Understanding what is interaction

Chapter 2

Models of Interaction

In computer science, important concepts usually come with a plethora of alternative characterizations. – *Christos Papadimitriou*

Information systems and computation are considered to be tools for "problem solving." The concept of computability addresses solvability of classes of problems by computation. Problem solving is said to be computably tractable if there exists an algorithm for the problem solving process.

In a natural way, concepts of problem solving and computability are extended to systems of computational processes as in IS design. However, as observed by Wegner [Weg97], the behavior of an IS seems to be richer than the behavior of algorithms. The phrase "no silver bullet" coined by Brooks [Bro95] to denote IS design complexity, has been interpreted by Wegner to mean IS processes cannot be reduced to algorithms. This is claimed to be because of the *interactive* nature of IS processes.

This chapter addresses this claim. It provides an overview of various models of interaction that have been proposed in related literature and compares interactive behavior against algorithms.

2.1 On the Notion of Interaction

Bob and Alice are attendees of the twentieth reunion of their old batch of engineering classmates. One of the attendees hears the following two sets of conversation that involves Bob and Alice.

Conversation 1:

Bob: Hi, how are you doing these days?

Alice: No, rain was predicted for today; but it didn't rain.

Bob: And what happened to your plans of changing jobs?

Alice: Yes, even though it isn't raining, there is a lot of humidity.

Bob: Oh, that sounds good. All the best in your new job.

Alice: Oh yes! That was very unexpected, I didn't even have an umbrella.

Conversation 2:

Bob: Hi, how are you doing these days?

Alice: Well, nothing spectacular, but good enough.

Bob: And what happened to your plans of changing jobs?

Alice: I did change my job. And it has already been a month in my new job.

Bob: Oh, that sounds good. All the best in your new job.

Alice: Thanks. And tell me about yourself..

Even though both sets of conversations involve words spoken by Bob and Alice, the listener would easily conclude that Bob and Alice were *interacting* in the second conversation, and not in the first. The second conversation involves a shared subject or *state* that evolves throughout the interaction. Without the common evolving state, interaction cannot take place. Not every sequence of activity (speech in this case) constitutes interaction. Interaction requires a collective shared state which evolves through the process.

Consider now, Paul joining Bob and Alice. And two sets of conversation are again overheard involving the three.

Conversation 1:

Bob: Hi, how're you all doing?

Paul: Just great!

Alice: Same here...

Bob: Isn't it great to be meeting after so long?

Alice: Yes, and I am having difficulties recognizing who is who.

Paul: Laura met me near the doorway, and believe it or not, I could not remember her name as long as we spoke!

Conversation 2:

Bob: Hi, how're you all doing?

Paul: Just great!

Bob: And you Alice?

Alice: I'm fine too, thanks!

Bob: Isn't it great to be meeting after so long?

Alice: Yes, and you have changed quite a bit Bob; but Paul still looks the same.

Bob: Really? (*whispers to Paul*) Oh my, I thought no one would notice my paunch.

Paul: And you seem to be getting younger Alice..

Alice: Now, now. Flattery will get you nowhere!

In the above two conversations, both constitute interaction since there is a shared evolving state. However, the first conversation is distinctly different from the second. In the second, there is an element of *intended recipient* in the statements. In the first conversation, everything was said to everybody. It didn't matter who answered a question posed and who received a particular statement. The second conversation however is *channel sensitive*. Even though all three are involved in the same conversation, there are some specifics regarding the recepient of particular statements.

When everything is said to everybody, the shared state can be modeled as a set of information that is duplicated with each participant, or as a broadcast media accessible to all participants. All participants share the same set of information as far as this interaction is concerned. But with a notion of intended recipient, modeling channels and their interconnectivity also becomes important. Here, even though each participant is involved in the same interactive process, the information that is maintained in each of their shared states is different. One can think of such a model of interaction as a shared, evolving state that is *individualized* for each recipient or channel. Figure 2.1 schematically sketches the two images of interaction discussed above. In the first case, the state is shared among all participants and every participant has access to every information in the shared state. In the second case, the shared state is individualized among the participants. From a database vocabulary, the participants in the second conversation maintain their own *views* of the shared interaction state. The interacting participants may or may not be aware of the global shared state that their individualized state is part of. And changes in the global shared state affects the individualized state of the participants. Chapter 4 shows how dynamics of an IS can also be modeled by using above two paradigms.

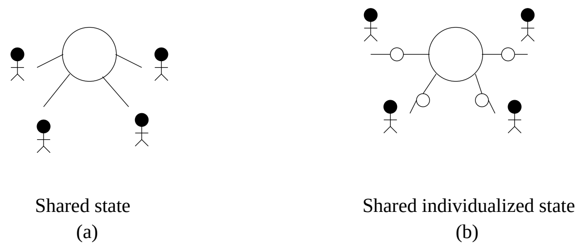


Figure 2.1: Images of Interaction

Of course, other factors like distribution in space and time also affect the interactive *process*. However they do not affect *what* can be effectively achieved by interaction. Time can affect what can be achieved by an interaction, if time is one of the determinant factors of interaction, as in real-time interaction. However, we do not consider real-time interaction here, and restrict the subject matter of dynamic processes to temporal precedence relationships. The emphasis is on modeling interaction sequences which can be temporally ordered using any logical time rather than real time.

This chapter formalizes the above concepts of persistence of state and channel sensitivity which determine interaction. Interaction is assumed to be taking place "here and now" so that distribution in space and time is abstracted away. The next chapter introduces a concept called "solution space" of a dynamic process, and shows how each of the above factors contribute to a richer domain than what can be achieved by computation alone.

2.2 On the Notion of Problem Solving

2.2.1 Connotations of problem solving

With a tenet that information systems are meant for "problem solving" we have to look at what is meant by problem solving, to understand how interactive problem solving is different from algorithmic problem solving. A notion of problem solving can be defined along at least two different connotations:

Connotation 1: Provide an output "answer" or "solution" to an input "question" or "problem"

An example of this connotation is a problem statement like "*What is the square-root of 81?*" Problem solving of this connotation can be modeled by a function between two domains, called the problem domain and the solution domain. The function takes an input parameter (81 in this case) and maps it to the output result from the solution domain. They are mathematically represented as $f : I \rightarrow O$ where I is the input problem domain and O is the output solution domain.

Connotation 2: Find a means to change the state of a given system from the current "problem" state to a desired "solution" state, under a set of constraints.

In this connotation, a system is given and the configuration of the system has to be altered. The desired configuration is called the solution state, and the configuration where the process starts is called the problem state. An example of the second connotation is a problem statement like the following:

"**Towers of Hanoi.** *The towers of Hanoi consist of three poles, where the first pole has a set of n discs on it. The discs are arranged such that a disc of smaller diameter is above a disc of larger diameter. The task is to move the set of discs to the third pole with the following constraints: (a). only one disc may be moved at one time, and* *(b). at any point in time no disc of larger diameter should be above a disc of a smaller diameter on any pole."*

In this example, the system consists of n discs and three poles. To represent the state of the system, consider that $n = 3$. Let the three discs be on the first pole, and let the other two poles be empty. The system state can be represented as: $[(3, 2, 1)00]$, which shows the three discs on the first pole and none on the rest. It also shows the order in which the three discs are on the pole. A disc with a bigger number (say 3) is assumed to be bigger than a disc with a smaller number (say 1). And the top of the pole is assumed to be the rightmost element in a list of disc numbers on a pole. Hence the representation shows three discs on the first pole, with the largest disc being the lowest one and the smallest disc being the highest one.

Problem solving now is to change the configuration to $[00(3, 2, 1)]$, governed by a set of constraints. The constraints say that (a). any state of the form $[(1, 2)(3)0]$ which shows a smaller disc below a larger disc is illegal; and (b). any transition of the form $[(3, 2, 1)00] \rightarrow [(3)0(2, 1)]$ which shows more than one disc has moved simultaneously, is illegal.

Mathematically, if S is the set of all states of the system, problem solving of this connotation is a mapping of the form $f : S \rightarrow S$. But this is inadequate since it does not show the process of problem solving, and all the constraints that govern intermediate state transitions.

An other means of representation is to associate a label with each state transition. Thus $S_0 \xrightarrow{a} S_1 \xrightarrow{b} S_2$ says that the transition from S_0 to S_1 is labeled a and the transition from S_1 to S_2 is labeled b. The set of all such transitions is called the "action" set A. We can now redefine problem solving of this connotation as follows: given a set of input parameters, find a sequence of actions $s \in A^*$ that can take the process to a desired solution state.

This would reduce the second connotation of problem solving to the first. The problem solving now is a function $f : I \rightarrow A^*$, where I is the domain of input parameters, and A^* is the set of action strings that is the solution to the problem. In the above Towers of Hanoi example, the input parameter is n the number of discs, and the output string is the sequence that shows how discs are moved between poles.

Most real world problem solving that involve information systems, take on the second connotation. For example, vending coffee from a machine, booking a flight ticket, managing supply chains, etc. all involve problem solving processes that change the state of a system from a given "problem" state to a desired "solution" state under a set of constraints.

2.2.2 Algorithmic and interactive problem solving

Having defined a concise notion of what is problem solving, we can now define an information system formally as a collection of problem solving processes. Each problem solving process (PSP) defines specific functionality provided by the information system. In the next chapter, we shall provide a more formal representation for a PSP which would further formalize the following definition of an IS.

Definition 2.1: An Information System (IS) is a collection of semantic processes called Problem Solving Processes (PSPs) each of which maps between a given problem state of the system to a desired solution state. □

Now, in trying to define what is problem solving, there have however been some implicit assumptions. Firstly, is it assumed that the state of the system does not change arbitrarily unless changed by the process itself. Secondly it is assumed that the problem statement and its governing constraints do not change midway during the process. Thirdly, it is assumed that the desired solution state is known at the start of the process. For example, in the Towers of Hanoi problem it is implicitly assumed that the discs don't move by themselves unless moved by the problem solving process itself; or that the user will not change constraints midway, or choose a different state as the solution state. Specifically, *it is assumed that the transition from the problem state to the solution state takes place in a closed atomic operation that is isolated from anything else happening in the system.* Hence, if the system is in state $p \in S$ when the problem solving began; a desired solution state $s \in S$ is fixed before the problem solving begins. The problem solving process starts at p and shuts off the world until it reaches s. Once the process is over, the system will be either in state p or state s.

Here is an example quotation from Gray and Reuter [GR93] about the expected behavior of functions.

> "Consider, for example, an SQRT function invoked from a Pascal program. The invocation is synchronous from the program's point of view; that is, it waits until the result has been computed. The function will either return the right value of the square root, or it will return an error code, but it will not change any data structures or parameters in an unpredictable way. Whether it does the computation using an iterative algorithm, a table lookup, or by asking a number-crunching friend is irrelevant to the caller; under no circumstances will it produce "partial" square roots or otherwise incorrect results. It will also not return somebody else's square root if the function is invoked by many programs at the same time."

The database community adopts the above assumptions in the form of ACID transactions on databases. A *transaction* over a database, models a problem solving process that changes the state of the database from a given problem state to a desired solution state [GR93, Kim95, RG00]. A transaction is expected to behave like a function having the above properties. In order to maintain correctness of behavior in the face of multiple transactions acting concurrently on the system, a transaction is designed to display the following properties: *atomicity*, *consistency*, *isolation* and *durability*.

Atomicity requires that the transition between problem and solution states take place as an atomic operation with respect to the outside world. A transaction should either leave the database in the problem state or the solution state and not in any intermediate states. Consistency requires that the solution state of the database satisfies integrity constraints of the database. Isolation mandates that given any set of concurrent transactions, the net effect of running those transactions will be the same as running those transactions in some order. That is, each transaction should run in semantic isolation. Durability requires that the effects of a transaction are "durable" – that is, the system has effectively changed states after the transaction is complete. These four properties, collectively called the ACID properties are adopted by database applications involving transaction processing.

But real world problem solving need not always have the above constraints in mapping between problem and solution states. In a semantic process that changes between problem and solution states, the problem solving process may sometimes interact with the user for intermediate inputs or guidance regarding the next step. This makes each state transition dependent on the external intermediate input. The external intermediate input may in turn depend on the intermediate output provided by the problem solving process. In addition, when problem solving involves coordination between multiple processes, the system state can get changed by other processes acting on the system. The problem solving process may also wilfully interact with multiple environments without one having knowledge about the other.

The inadequacy of a closed, atomic paradigm of problem solving is apparent when "long duration" transactions are considered. Long duration transactions are semantic processes which can run for several seconds to several days or months. They might involve other "high level" operations like coordination and collaboration [Kim95].

Traditional transaction processing uses locks to ensure atomicity and isolation properties of a transaction. In long duration transactions, the "virtue" of locks becomes a "vice" because locks have to be held for long durations of time. In addition, the cost of a rollback in the case of a failed transaction is high. Many approaches towards long duration transactions have hence relaxed the ACID criteria. They have instead concentrated on modeling and maintaining correctness at a semantic level [Beh99, Elm92, KR96, TS94]. Semantic processes are modeled as "transactional workflows" [SR93], consisting of a number of steps corresponding to database transactions. Correctness in the face of concurrent workflows is maintained by logical dependencies between steps of different workflows.

However the field of workflows still lacks sound conceptual underpinnings. It is generally agreed that a set of workflows cannot adequately model the complex nature of actual business processes. As a result issues like exception handling pose significant design complexities [CP99].

A PSP that maps between problem and solution states in a closed atomic fashion is said to be algorithmic in nature. In contrast, if a PSP is not amenable to properties of atomicity and isolation, it is said to be interactive in nature. An IS which contains only algorithmic PSPs is said to be a "closed" system, and one that contains interactive PSPs is said to be an "open" system.

Definition 2.2: A "closed" system is a collection of algorithmic PSPs, while an "open" system is a collection of PSPs where at least one of them is interactive. □

An open system can be of two kinds. A *single-stream* open system that interacts one only one stream; and a *multi-stream* open system that can interact over more than one streams. The difference between single-stream interaction and multi-stream interaction has been first noted by Wegner [Weg97, WG99a]. In the following sections, we describe each of the above three kinds of systems (closed, single-stream open, multi-strem open) in detail.

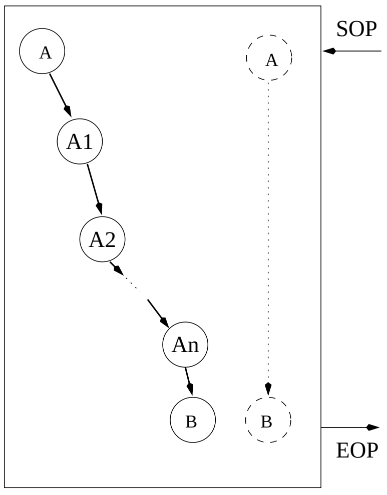
2.3 Algorithms

An algorithm is the most fundamental form of interactive behavior. They model computable functions of the form $f : I \rightarrow O$ that map between I and O in a closed fashion. They have exactly one interaction with their environment[1].

The behavior of an algorithm is schematically shown in Figure 2.2. It accepts an input set of parameters and computes until it reaches the desired solution. If the algorithm terminates, the output would be the result of the computation or an error code. A problem solving process (PSP) that is algorithmic in nature starts by accepting a set of input parameters. The PSP ends by outputting the solution. In the above figure, the PSP begins the algorithmic process at state A and ends at state B. The start of problem solving is denoted by the symbol SOP and the end of problem solving is denoted by the symbol EOP.

Definition 2.3: An *algorithmic problem solving process (APSP)* is defined as a tuple,

[1]In common discourse, the term "algorithm" is loosely used to also include interactive protocols. However we use the term algorithms to mean a step by step procedure that models computable functions. This entails mapping a given element from the input "problem" domain to an element from the output "solution" domain in a finite number of steps – essentially non interactive once the process has begun. Also, from the definition of a problem solving process, we in effect neglect algorithms that don't terminate.



Algorithmic Computation

Computable function f: I --> O

SOP: Start of Problem solving, EOP: End of Problem solving

Figure 2.2: Algorithmic computation

$APSP = \langle I, O, \delta \rangle$, where I is the "problem" domain depicting the set of possible inputs, O is the "solution" domain depicting the set of possible outputs, and δ is a computable function of the form $\delta : I \rightarrow O$, that maps input "problems" to output "solutions." $\square$

2.3.1 Turing Machines

The mathematical model of an algorithm is a Turing Machine (TM) (c.f. [HU79]). The structure of a TM is as follows:

Definition 2.4: A Turing Machine (TM) is defined as $TM = \langle S, T, s_0, \delta, H \rangle$, where

- S is the set of TM *states*,
- T is a set of *tape symbols*,
- s_0 is the *start* state
- δ is a mapping of the form $\delta : S \times T \rightarrow S \times T \times \{L, R\}$, and
- $H \subset S$ is the set of halt states.

$\square$

TMs read input from a tape of infinite length, on which the input is provided in the form of a finite sequence of tape symbols. They begin computation from an internal start state s_0 and at each computational step map the current input symbol $x \in T$ to an output symbol $y \in T$, and moves one step to the left or right on the input tape. In addition, the internal state of the TM is altered by each computation step. The TM halts when it reaches any halting state $h \in H$.

A TM is the most expressive model among a class of models representing computable mappings. Some of these other models are: Deterministic finite automata, Nondeterministic finite automata, Pushdown automata, Context-free grammars, etc.

A tape in a Turing Machine is the working memory of the TM. Expressiveness of computational models depend on their representation of the working memory and in their internal state transition primitives. For instance, a pushdown automaton maintains memory in the form of a stack where symbols can be placed only at the top and only the topmost symbol can be read at any given instant. A TM, which is the most expressive, is said to represent all mathematical processes of computation.

Despite the different kinds of computational models, all of them share a common feature. This is explained in more detail below.

Postulate 2.1: Models for computational mappings have a generic form: $M_C = \langle s_0, \delta \rangle$, where s_0 is the "starting point," and δ is a set of dynamics beginning from the starting point. Computation always begins at s_0 and proceeds based on the input parameters and δ. $\square$

Computation always begins from a well known "start state" s_0. Depending on the particular model, the dynamics δ can have different levels of complexity. It could be either monotonic or non monotonic, deterministic or non deterministic, memoryless or may maintain memory or have any other property. However, the important aspect here is that of the start state. This is important in order to distinguish algorithmic dynamics from interactive dynamics.

2.3.2 Algorithmic systems

Functional programming: A system consisting of a collection of algorithms itself behaves as an algorithm. This forms the basis for functional programming paradigms [Hug89, RL99, Ros82, Sab98]. Functional programming is a paradigm where computation is expressed declaratively in terms of functions. There are no side effects to a function's evaluation and a function will always evaluate to the same value for the same input. There are no assignment statements or any other imperative constructs that specify actual procedures in evaluating an expression.

An entire program can be considered to be a function, whose starting point is determined by some form of a "main" function. Examples of functional programming languages include ML, Haskell and Miranda. The logical formalism for functional programming is based on pure λ-calculus. Pure λ-calculus consists of only functions without other constructs like data types or constants. Imperative programming languages like Pascal and C are also algorithmic paradigms that are based on the typed λ-calculus with constants.

λ-calculus: λ-calculus is a branch of mathematical logic developed by Alonzo Church, that deals with the application of functions to their arguments. The pure λ-calculus contains only functions and no constants or types. Computation as well as data types are expressed as functions.

Functions in λ-calculus are represented by λ-abstractions. A λ-abstraction is of the form $\lambda x.f(x)$. Here x is a bound variable and $f(x)$ is a λ-expression. A λ-expression can be either a λ-abstraction, a variable or a constant.

The λ-expression $f(x)$ is called the body of the λ-abstraction. It denotes the expression that is returned as an evaluation of the function. For example, a λ-abstraction of the form $\lambda x.(x * x)$ denotes a function, with x as its bound variable, and which returns x^2 as the result of the function.

Functions having multiple input parameters can be reduced to nested functions having single parameters. Thus a λ-abstraction of the form $\lambda x.\lambda y.(x * y)$ can be reduced to a function having two input parameters x and y. This can be rewritten as $\lambda xy.(x * y)$. The process of representing a function of the form $(a, b) \rightarrow c$ in an alternative form as $a \rightarrow b \rightarrow c$ is called "currying." The reverse process illustrated above is a process of uncurrying.

A detailed exploration into functional programming is beyond the focus of this work. The emphasis here is on interactive behavior which displays properties that cannot be directly reduced to a function.

2.3.3 Inductive domains

Recursively enumerable sets: An alternative representation of an algorithm is to represent the *set* of all elements that can be generated by the algorithm. Hence, if an algorithm models a function $f : I \rightarrow O$, the set representation of the algorithm would be the set of all pairs of the form (i, o) where $i \in I$ and $o \in O$. Such a set, whose elements are generatable by a TM is called a *recursively enumerable* set.

Definition 2.5: A set is said to be *recursively enumerable* (r.e.) if there exists a Turing Machine that generates all elements of the set. □

A set that is r.e. has a one-one mapping to the set of natural numbers. However, even if a set is r.e., it does not mean that the membership of a given element is decidable with respect to the set. For any set S that is r.e. and that contains elements from a universe U, the TM that generates elements of the set may fail to halt in determining membership for some element $u \in U, u \notin S$ [HU79].

Not all sets are r.e. Given that some functions are not computable, it follows that there could be set representations whose elements cannot be generated by a TM [DeV98]. An example is the set of all real numbers.

Induction: Mechanisms for recursively generating the elements of a set, adopt a principle of *induction*. A process of induction to generate elements of a set S from a universe of elements U involves the following steps:

1. State a set of *atoms* $\in U$ that belong to S,
2. State a set of *operators* that apply to elements of S and which select an element from U as belonging to S
3. Recursively keep applying the operators to elements of S to generate further elements of S

For example, the set of whole numbers W can be generated from the set of integers I by the following pair: $(0, succ())$, where $succ(n) = n + 1$. Such a structure is also called an *algebra*. Algebra is a well established part of mathematics and also of computer science in abstract data type theory. An algebra is defined as follows:

Definition 2.6: An *algebra* is defined as $Q = \langle A, A^*, \delta \rangle$, where A is the set of atoms, A^* is the carrier set or the set of unfolded strings, and $\delta : A \times A^* \rightarrow A^*$ is a set of production rules. □

A process of induction that generates a given abstract domain has the following properties:

1. **Initiality:** A set of elements are introduced as *atoms* of the algebra.
2. **Iteration:** A set of *production rules* determine how strings of the algebra are generated from the atoms and other generated strings of the algebra. The set of all strings generated by an algebra is called the *carrier set* of the algebra.
3. **Minimality:** No other string, other than those generated by the algebra belongs to the carrier set of the algebra.

The "constructivist" nature of an algebra make them *minimalist* in nature. A minimalist model is described by "initiality." An *initial* algebra is the minimal set of elements and production rules that are required to generate the entire set. Alternatively, an initial algebra of a particular type is that subset that is necessarily present in every algebra of that type.

$$
\begin{aligned}
&String \sqsubset [\backslash x20 - \backslash x7F]^* \\
&\{\backslash 0, addchar()\} \\
&addchar(s,c) \;\; = \;\; cs, \quad s \in String, \;\; c \in [\backslash x20 - \backslash x7F]
\end{aligned}
$$

Figure 2.3: An algebraic declaration of data type String

Figure 2.3 shows an example declaration of the data type `String` using an algebra. Here the universe of discourse is all the ASCII characters between 32 and 127 (denoted by their hex equivalent). A `String` is a sequence of characters terminated by a null (\0) character. Hence the null character forms the set of atoms for the algebra. The algebra has a production rule called $addchar()$ that adds a character to a previously generated `String`. The set representing all strings generated in this way forms the carrier set of the algebra.

Computational processes are also inductive in nature. As seen earlier, a computational process can be represented as $\langle s_0, \delta \rangle$. Here, the set of actions A may be considered to be the set of *atoms*, s_0 is the start production rule and A^* is the carrier set of the process.

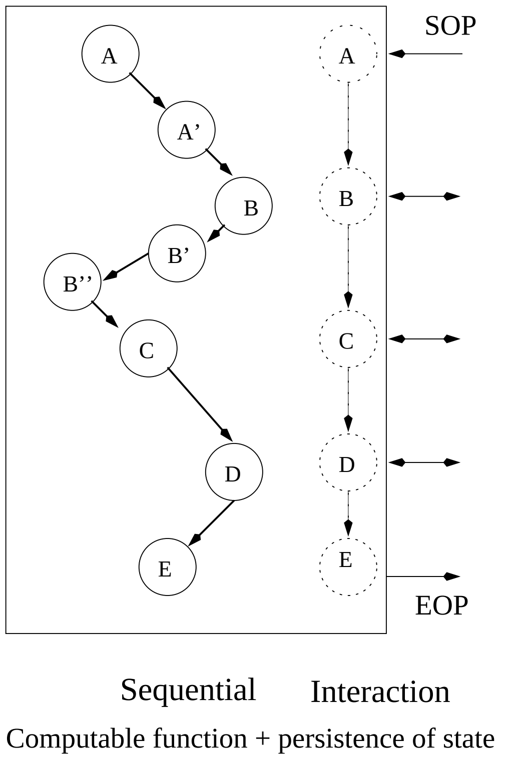
2.4 Single-stream Interaction

In an interactive process, there are one or more intermediate interactions with an outside environment in mapping between problem and solution states of a PSP.

Figure 2.4 depicts sequential, or single-stream interaction. The PSP is the transition $SOP \rightarrow EOP$. It begins at state A, and ends at state E. However, this transition involves a number of intermediate interactions with the environment. The behavior of the environment at any intermediate interaction is unknown at the start of the process. Also, intermediate inputs from the environment may be coupled with previous intermediate outputs from the computing machine. This makes it infeasible to provide all inputs from the environment at the start of the process itself. In addition, the start state A may be the result of earlier interactions, and would be different every time a new interactive process begins.

Single-stream interaction is common in practice. For example, Figure 2.4 may represent a user session in a database application. The state at which a user session starts is the current state of the database, which is a function of interaction history. When a user starts a session, the database cannot determine what the state would be at the end of the session, since it may be affected by intermediate inputs.

Traditional transaction processing in databases performs the entire set of operations between SOP and EOP as a single atomic operation. Computations and intermediate interactions with the user are not committed to the database until it is sure that the end



SOP: Start of Problem solving, EOP: End of Problem solving

Figure 2.4: Single-stream Interaction

of the process has been reached. The entire changes are then committed to the database in one go. Hence, after the process is completed, the database is either in state A or in state E.

However, this need not always be the case in real world processes involving interaction. For example, if the first interaction involves actual physical operation (launch missile, open dam sluice, destroy buildings, etc.) and subsequent interactions denote progressive control of the operation; the semantic operation cannot be atomic. It may not be possible to provide the second intermediate input until the first operation is done; and an operation performed cannot be rolled back. Similarly, if the semantic process is spread over a period of days, it is not possible to lock the contents of the database from changing over the whole duration of the process. The lack of atomicity here is not a shortcoming of the information system, but a part of the problem domain requirements.

In an interactive process, the computation between any two consecutive interactions is algorithmic in nature. This mapping is described by a *partial function* of the form $(s_k, i_k) \rightarrow (s_{k+1}, o_k)$, where s_k is the state of the system at the k^{th} interaction, i_k is the input provided at the k^{th} interaction, o_k is the output obtained after the k^{th} interaction; and s_{k+1} is the new system state. The behavior of the function is not only dependent on the input, but also on the current state. On termination, the function not only produces an output but also changes the system state. Such a partial function may be represented as $\delta : S \times I \rightarrow S \times O$, where S is the state space of the system, I and O are the input and output domains respectively.

Definition 2.7: A *Sequential Interactive Process (SIP)* is defined as $SIP = \langle S, I, O, \delta \rangle$, where S is the state space of the system, I and O are input and output domains respectively, δ is a computable partial function of the form $\delta : S \times I \rightarrow S \times O$. $\square$

Postulate 2.2: The generic form of a sequential interactive process is represented as $\langle S, \delta \rangle$, where S is a set of system states and δ is the computational behavior defined from each state $s \in S$ that is executed when the system interacts with the environment from state S. $\square$

Note the difference between the generic forms of sequential interaction and algorithmic systems. Algorithmic systems have a unique start state and behavior defined from that state. Sequential interaction has computational behavior defined from a number of states on which interaction is possible. Later in this section we call this the "maximalist" nature of interactive systems.

It is sometimes confusing to note that interactive processes like the above are also called "concurrent" processes (Ex. [Mil89]). This is because, an interactive process maintains a *coroutine* or a concurrent relationship with its environment, as opposed to a *subroutine* or sequential relationship maintained by an algorithmic process. However, in our case, we term this as a single-stream or a sequential interactive process to denote that the interactive stream of inputs and outputs takes place in a sequential manner.

Sequential interaction is characterized by *persistence of state* in addition to computation. Sequential interaction is made possible by a persistent system state across computations. Persistence of state is necessary for properties like history sensitive behavior, learning, evolution, etc. It makes the behavior of an interactive process more expressive than just a function. A function always produces the same output for the same input. But the behavior of an interactive process is history sensitive.

Some models of sequential interaction in related literature are explored in the following subsections.

2.4.1 Monads

Monads are a mechanism for incorporating persistent state into functional programming. They have been used to incorporate a number of features like continuations, input-output and exceptions into functional programming languages. They have also been used to model interactions [Wad97].

A Monad consists of three parts [How93] $\langle M, unitM, bindM \rangle$. M is some function on types. $unitM$ converts a value of some type into monadic form of M. And $bindM$ applies a function to a monadic value. The monadic value models persistent state.

2.4.2 SIM and PTM

To represent sequential interaction, an abstraction called Single-stream Interaction Machine (SIM) was proposed by Wegner [WG99a]. A SIM interacts with a single environment and its behavior is dependent on interaction history. A SIM models the behavior of an object as against an algorithm.

Later on Goldin [GW98, Gol00] proposed an extension to TMs called the Persistent Turing Machine (PTM) as a mathematical model for SIMs. A PTM is defined as follows:

Definition 2.8: A *Persistent Turing Machine* is of the form $PTM = (W, TM)$, where W is a worktape whose contents are persistent across computations. The TM part of a PTM begins each computation from a state that is determined by the contents of W rather than the same start state for every computation. □

Different kinds of PTMs are then defined: *amnesic PTMs* lose the contents of W after each computation and are equivalent to TMs; *finite-memory PTMs* have a bound on the number of internal states that the TM can have; and *finite-state PTMs* have a bound on the size of W. Finite-memory PTMs are shown to have the same expressiveness as finite-state PTMs.

2.4.3 Labeled Transition Systems

Reactive systems [JKSS90, MP92] address single-stream interaction. They represent systems whose goal is to maintain an ongoing interaction with an environment. The fundamental model of a reactive system (as well as of infinite state systems in general [May97]) is a Labeled Transition System (LTS) which is defined as follows:

Definition 2.9: A *Labeled Transition System* (LTS) is of the form $\langle S, A \rangle$, where S is the set of states and A is a set of "actions. " At each state $s \in S$, an action $a \in A$ takes the LTS to another state $s' \in S$. This is denoted by $s \xrightarrow{a} s'$. □

In some definitions, an LTS is defined to also have a start state s_0 from where it begins computation. However, more generic models make away with the start state and define behaviors from each state explicitly. The LTS would then be able to start from any state, and the behavior of an LTS at any given instant would be determined from its current state.

2.4.4 Streams

An other way of modeling persistent state in sequential interaction, is to consider the system state as *interaction history.* Each interaction is of the form (i, o), where an output is returned for an input. Hence interaction history or system state can be modeled as a *stream* of the form $(i, o)^*$. Broy [Bro97] adopts such an approach to model interactive behavior. Some pertinent definitions from this model are as follows:

Definition 2.10: A *stream* over a set M is a finite or an infinite sequence of elements from M. The set of finite sequences are denoted by M^* and the set of infinite sequences are denoted by M^∞. The empty sequence is denoted by $\langle\rangle$. $\square$

Definition 2.11: The behavior of an interactive component is modeled by a *timed stream.* A timed stream is an infinite stream of finite streams. Formally a timed sequence $M^\kappa =_{def} (M^*)^\infty$. $\square$

Each finite sequence of input-output represents an interaction session, and the infinite sequence of all sessions represents the component's behavior.

2.4.5 Process calculus

The calculus of communicating systems (CCS) or process calculus [Mil89] models inputs, outputs, states and composition of communicating processes. A typical definition in CCS is shown below:

$$C =_{def} in(x).C'$$
$$C' =_{def} out(x).C$$

This denotes an interactive process where at state C, an input of x takes the system to state C' and an output of x at C' takes the system to state C. Processes can be combined by connecting input and output channels of different processes. A combination of processes can be abstracted as a single compound process. In such processes the system state may change because of internal interactions, even without interactions with the outside environment. Such state change transitions are called τ-transitions. A τ-transition is not observable by the environment.

Expressiveness of two or more systems is established by a process of *bisimulation.* Bisimulation is based on a concept of observational equivalence of inputs and outputs. Two systems S_A and S_B are said to be *bisimilar until* R if there exists a relation $R \subseteq I \times O$ of inputs and outputs, such that for each input element in R, both S_A and S_B return the same output.

2.4.6 Coalgebras and coinduction

A notion dual to the concepts of induction and algebras is gaining popularity in modeling dynamic systems. This is the concept of a coalgebra and the associated proof principles based on coinduction.

Coalgebras incorporate concepts from all the paradigms seen above, into a common unifying framework. They can be used to specify, model and reason about (single-stream) interactive behavior. A concise tutorial on coalgebras as applied to interactive IS behavior is provided here. A more detailed tutorial on the concept of coalgebras is available by Jacobs and Rutten [JR97].

Fundamentally, a coalgebra is the dual concept of an algebra. In an algebra the carrier set is generated from a set of axioms and a set of production rules. In a coalgebra, the set of axioms is generated from the carrier set using a set of *abduction* rules. Abduction implies inference based on observation. While induction models the concept of *construction* (of the carrier set from the set of axioms), coinduction models the concept of *observation* (observing the carrier set and deducing its axioms).

An inductive process is *minimalist* in nature while a coinductive process is *maximalist* in nature. As noted by Wegner [WG99a], in an inductive process "everything is forbidden (to be included in the carrier set) other than what is allowed (by the algebra)." In contrast, in a coinductive process "everything is allowed (to be included in the set of possible observations) other than what is forbidden."

The formal definition of a coalgebra is as follows:

Definition 2.12: A *coalgebra* is a tuple $\mathcal{Q} = \langle A, A^*, \delta \rangle$, where A is the set of *observed axioms*, A^* is the carrier set and δ is a mapping of the form $A^* \rightarrow A \times A^*$. $\square$

Note the difference in the definition of δ. In an algebra, δ was a mapping *into* the carrier set A^*, and in a coalgebra, δ is a mapping *from* the carrier set A^*. This change in the definition of δ allows for modeling interactive paradigms like persistent state and lazy binding of intermediate inputs [WG99b].

Coalgebras have also been attempted for describing reactive systems [Kie97]. Rutten [Rut96] characterizes dynamical systems using coalgebras and goes on to claim that coalgebras and dynamical systems are equivalent. The definition of a dynamical system in the form of a coalgebra is as follows:

Definition 2.13: A *F-coalgebra* or an *F-system* is a tuple $\langle S, \alpha_S \rangle$, where S is the state space of the system and α_S is the set of system *dynamics*. α_S is of the form $S \rightarrow F(S)$, which maps each state S to a functor $F(S)$ that defines dynamics from any given state. $\square$

Depending on the definition of $F(S)$ a coalgebra models different kinds of systems. For example if $F(S) = S$, then system dynamics are modeled by $S \to S$ and the coalgebra represents a deterministic state machine. If $F(S) = \mathcal{P}(A \times S)$, then the coalgebra models an LTS.

Lemma 2.1: The definition of a coalgebra as $\langle S, \alpha_S \rangle$ is equivalent to the definition of a coalgebra as $\langle A, A^*, \delta \rangle$.

Proof: The equivalence of the two definitions may be established by the fact that a state in a dynamical system models interaction history or an element of the carrier set. Hence a mapping of the form $S \to F(S)$ is equivalent to a mapping of the form $A^* \to A \times A^*$. $\square$

In modeling a dynamical system, the essential difference between an inductively defined model and a coinductively defined model is shown in Figure 2.5.

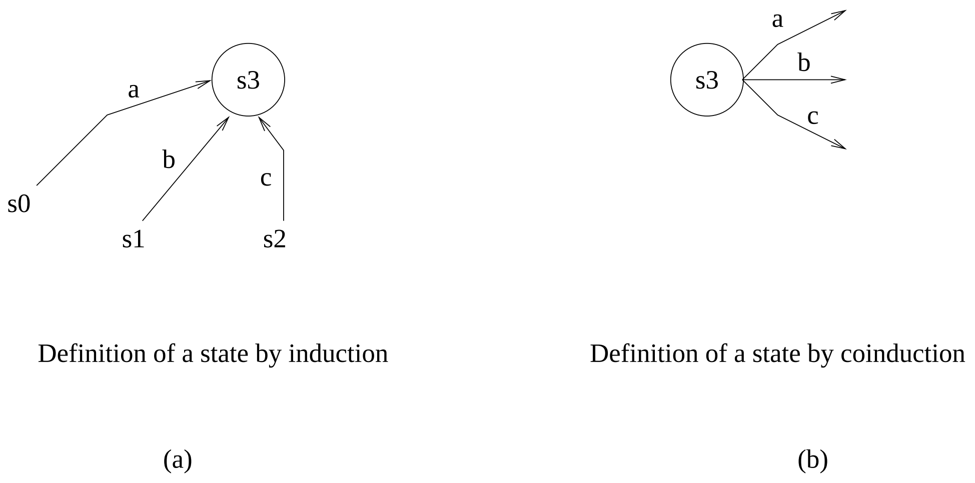

s3
a
b
c
s0
s1
s2
s3
a
b
c
Definition of a state by induction
Definition of a state by coinduction
(a)
(b)

Figure 2.5: Inductive and coinductive modeling

An inductive definition models state transitions as $\delta : S \times A \to S$, while a coinductive definition models state transitions as $\delta : S \to A \times S$.

In other words, a state in an inductive model is defined based on all the paths leading to it. Each state is characterized by an action from an other state. A state that does not require such a characterization would form the start state. Inductive characterization is schematically depicted in Figure 2.5(a).

In contrast, a coinductive definition characterizes a state by the set of all behaviors leading from it. The definition of a state is not dependent on another state, and this obviates the need for a start state. In fact, a coalgebraic system can be considered to be a collection of behaviors, each of which is abstracted by a state. *A state in a coalgebra is an abstraction that represents behavior from that point.*

Coinductive modeling in practice: Inductive modeling is based on a notion of "intended functionality." An algebra for an interactive process represents a mechanism for generating all valid interaction sequences.

Consider an example where a user interacts with an automatic teller machine (ATM) to withdraw money. The set of intended functionalities of an ATM include the following elements:

Vending money: achieved by the following activity sequence: {(insert_card, insert_PIN, specify_amount)}.

Cancelling a transaction: achieved by the following set of sequences: {(insert_card, cancel), (insert_card, insert_PIN, cancel)}.

handling erroneous inputs: The machine may be said to go into an error state if the user enters a wrong PIN number thrice, or enters an invalid amount for the required money.

The interactive process can be represented by a grammar whose atoms denote user actions. The set of atoms are as follows: $\{C = insert\_card$,
$P = insert\_PIN$,
$N = insert\_invalidPIN$,
$A = specify\_amount$,
$I = specify\_invalidAmount$,
$X = cancel\}$.

The interactive process shown as a grammar is as follows:

```
<Start> ::= (null) | <Done>
<Done> ::= <Success> | <Fail> | <Cancel>
<Success> ::= <PIN> A
<PIN> ::= <Card> P | <N1> P | <N2> P
<N2> ::= <N1> N
<N1> ::= <Card> N
<Card> ::= <Start> C
<Fail> ::= <N2> N | <PIN> I
<Cancel> ::= <Card> X | <N1> X | <N2> X | <PIN> X
```

The grammar represents a mechanism for generating all the valid interaction sequences that make up the process. Some of the valid strings that are possible from the specification are – CX, CNPI, CPX, CPA, . . .

A grammar can be translated into a state machine representation by making all the non terminal symbols like ⟨Card⟩, as states that are reached by following the input sequence of operations. Hence ⟨Start⟩ forms the start state, where an action of inserting a card takes the interactive process to the ⟨Card⟩ state. The state machine is shown in Figure 2.6. In the diagram τ-transitions denote transitions between states without any observable action taking place. It can be seen that, each state is defined based on the paths that lead to it. For example, ⟨`PIN`⟩ is reached by action `P` from states ⟨`Card`⟩, ⟨`N1`⟩, or ⟨`N2`⟩. This is as shown in Figure 2.5(a).

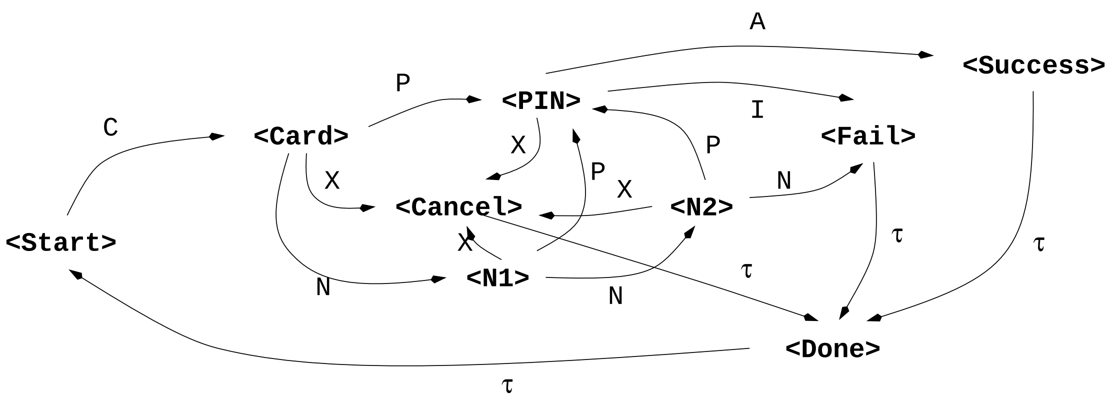


Figure 2.6: Inductive modeling of ATM interaction

In contrast to inductive modeling, coinductive modeling is based on a concept of *observable behavior*. Observations are carried out by a set of *observer functions* which determine what outputs are generated for each input in each observed state. A coinductive definition of a function f is generated by specifying the values of all observer functions for all outcomes $f(x)$ [JR97].

Coinductive modeling of the ATM is carried out by defining the following observer functions:

$c()$ corresponds to observing what happens when the ATM card is inserted;

$p()$ corresponds to observing what happens if the PIN number is entered;

$n()$ which corresponds to observing what happens if an invalid PIN is inserted;

$a()$ observing what happens when the required amount is specified;

$i()$ observing what happens if an invalid amount is specified and

$x()$ observing what happens when cancel is pressed.

From each state, the behavior of each observer function is defined which takes the machine to another state. The modeling process ends when there is no undefined state from any observer function and no undefined observer function from any state.

The resulting model is an LTS where each state has a mapping to other reachable states following each kind of input. The LTS is partially shown in Figure 2.7. The set of observations from the *Start* and the *Card* states would be as follows:

$Start = \{$
$c(Start) = (\texttt{PIN\_entry\_field}, Card)$
$p(Start) = (\texttt{error\_msg}, Start)$
$n(Start) = (\texttt{error\_msg}, Start)$
$a(Start) = (\texttt{error\_msg}, Start)$
$i(Start) = (\texttt{error\_msg}, Start)$
$x(Start) = (\phi, Start)$
$\}$
$Card = \{$
$c(Card) = (\phi, Card)$
$p(Card) = (\texttt{Amount\_menu}, PIN)$
$n(Card) = (\texttt{error\_msg}, N1)$
$a(Card) = (\texttt{error\_msg}, Card)$
$i(Card) = (\texttt{error\_msg}, Card)$
$x(Card) = (\texttt{ATM\_card}, Start)$
$\}$

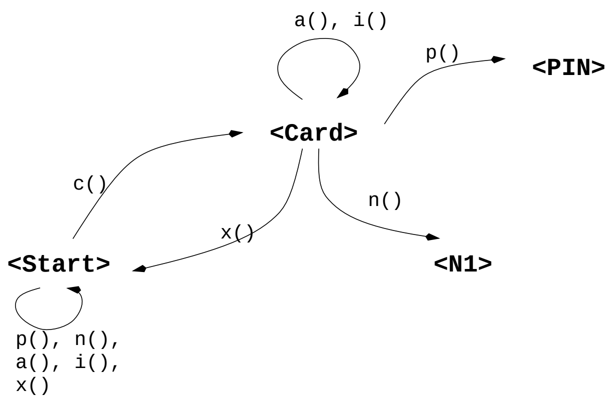


Figure 2.7: Coinductive modeling of ATM interaction

In the above, an observer function of the form $x(Card) = (\texttt{ATM\_card}, Start)$, indicates that in the state *Card*, pressing cancel would return back the ATM card, and the machine would go to the *Start* state. Semantically, rather than specifying how a particular state is reached, coinduction specifies for each discerned state, the output behavior and the resultant state following each type of input. This is as shown in Figure 2.5(b).

Canonical coalgebras: A coalgebra is said to embody a paradigm of *maximal fixpoints* as against an algebra which embodies a paradigm of *minimal fixpoints* [Jac96]. This is evident from the process of coinduction that describes abduction, in contrast with induction that describes construction. A process of coinduction is as follows:

Inductive modeling	**Coinductive modeling**
The advantages and limitations of an inductive modeling are as follows:	The advantages and limitations of a coinductive modeling are as follows:
Because of a recursive specification based on intended functionality, the state space is compactly described.	Observational closure ensures that no observable behavior is left unspecified. As in the earlier example, when the ⟨`Lang`⟩ state is introduced, the model may be made observationally closed by ensuring that no observer function is left undefined in any state. The dependency of ⟨`Cancel`⟩ on ⟨`Lang`⟩, can be discovered when accounting for $x(Lang)$.
However, induction represents a closed world, by its criteria of minimality. Any change in specifications has to be handled by changing the axioms. Consider a new state called ⟨`Lang`⟩ (language selection) that needs to be introduced after the ⟨`Start`⟩ state. This would require a modification of the specification of the ⟨`Card`⟩ and ⟨`Cancel`⟩ states. While the modification of the ⟨`Card`⟩ state is evident, the modification requirement for the ⟨`Cancel`⟩ state is implicit. It is based on the notion that the user may cancel after selecting the language. Since each state is defined by the paths leading to it, handling changes is difficult since there is no algorithmic method to determine implicit dependencies among states.	However, a limitation of this process is that it is iterative in nature, and there would be an explosion in the number of paths to be explored when the number of observers and states are high. Without a notion of intended functionality, having a user specify observable behavior in an iterative fashion would be impractical.

Table 2.1: Inductive vs coinductive modeling

Definition 2.14: A process of coinduction describes generation of a set of axioms or states A by observation over a carrier set A^*. A set of observer functions δ is used for observation. The coinductive process is described by the following steps:

1. **Iteration:** Start with any state x. Add x to A and describe the behavior of an observer function $\delta_i \in \delta$ from x whose behavior is as yet undefined for x. Let x' be the new state obtained after the process of observation using δ_i. Add x' to A.
2. **Maximality:** Continue the process of iteration until every $\delta_i \in \delta$ is defined for every $x \in A$.

$\square$

Definition 2.15: A coalgebra $\langle S, \alpha_S \rangle$ is said to be *canonical* or *final* if for any $s_1, s_2 \in S$, $\alpha_S(s_1) = \alpha_S(s_2) \Rightarrow s_1 = s_2$. $\square$

Canonical coalgebras can model interactive processes in a natural way. An interactive process begins from any interaction state, and each interaction involves computation that takes the process to another state.

A final coalgebra is in contrast to an *initial* algebra. An initial algebra represents the smallest set of axioms that can generate the carrier set. A final coalgebra represents the largest set of axioms required to account for every string in the carrier set.

Bisimulation: Equivalence of coalgebras are established by a process of bisimulation. This is defined as follows:

Definition 2.16: Two coalgebras $\langle S, \alpha_S \rangle$ and $\langle T, \alpha_T \rangle$ are said to be *bisimilar until* R, if there exists a coalgebra $\langle R, \alpha_R \rangle$, such that the following homomorphisms exist: $R \rightarrow S$, $R \rightarrow T$, $\alpha_R \rightarrow \alpha_S$ and $\alpha_R \rightarrow \alpha_T$. □

The similarity between the above definition and the definition of bisimulation in CCS is apparent by comparing definitions of Rutten [Rut96] and Milner [Mil89]. Based on these concepts Rutten proposes "universal coalgebra" as a generic formalism for dynamical systems.

Algebras	**Coalgebras**
construction paradigm	observation paradigm
$\delta : A \times A^* \rightarrow A^*$	$\delta : A^* \rightarrow A \times A^*$
inductive process of construction	coinductive process of observation
Induction: initiality, iteration, minimality	Coinduction: iteration, maximality
minimal fixpoints (algorithms)	maximal fixpoints (interaction)
congruence of algebras	bisimulation of coalgebras
initial algebras	final coalgebras

Table 2.2: Algebras vs coalgebras

For every concept in an algebra its dual is present in a coalgebra. This is shown in Table 2.2.

However, despite the comprehensive nature of coalgebras, they are still insufficient to model IS behavior. Dynamical systems not only have persistent state but are also channel sensitive. Channel sensitivity is a necessary ingredient for many kinds of IS processes like coordination and collaboration. This is explored in more detail in the next section.

2.5 Multi-stream Interaction

When interactions take place over multiple streams, its behavior is not reducible to a single-stream interaction. In addition to history dependent behavior resulting from persistent state, open systems also exhibit a property of *channel sensitivity*.

Figure 2.8 depicts a multi-stream interactive process. Here, interaction takes place on more than one stream simultaneously. The figure shows two processes being executed by the machine concurrently. They share the same state space of the machine, and need not be isolated from one another. While the internal state of the machine changes according to the sequence $ABCDEFG$, the stream on the left hand side (SOP0 – EOP0) perceives the change of system state to be $BDEFG$ and the right hand side (SOP1 – EOP1) perceives the change to be ACG respectively.

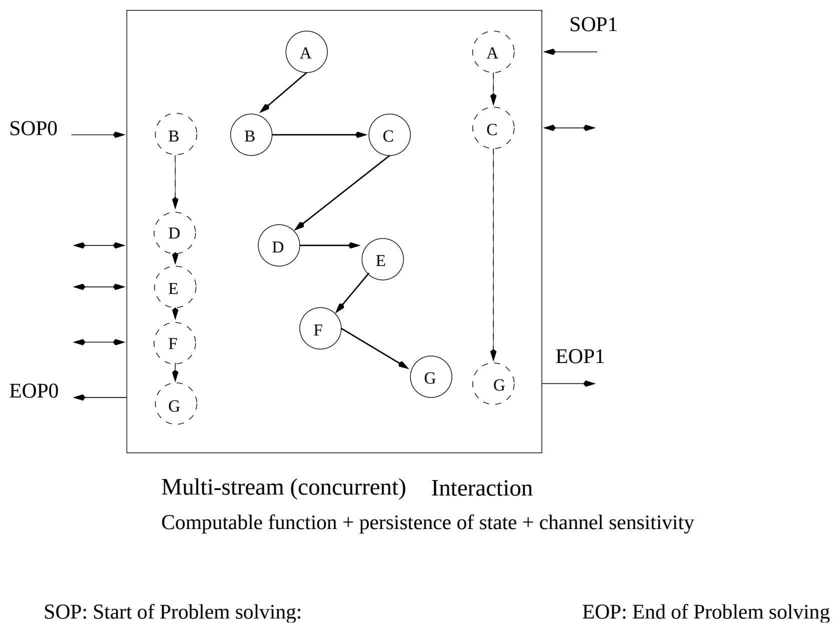

SOP1
SOP0
A
B
C
D
E
F
G
EOP1
EOP0
Multi-stream (concurrent) Interaction
Computable function + persistence of state + channel sensitivity
SOP: Start of Problem solving:
EOP: End of Problem solving

Figure 2.8: Multi-stream Interaction

Multi-stream interactive processes are very common in practice. Tasks like coordination, resource sharing, multi-user applications, process management and workflows, all address processes that interact with multiple streams simultaneously. However, multi-stream interaction has received far less attention than is necessary, by the IS community. To the best of our knowledge, multi-stream interaction has not been explicitly addressed as an issue in its own right in IS modeling. Issues related to multi-stream interactions have surfaced in domains like coordination models, process management and resource sharing. They have been mainly addressed as contention resolution problems or consistency preservation problems. Some examples are multiplexing protocols like token ring in networking, locks in databases and semaphores in operating systems. However, multi-stream interaction goes beyond just contention resolution and consistency preservation. It may consist of other issues like liveness, completeness, dynamic integrity, etc. A separate inquiry into the nature of multi-stream interaction is required to comprehensively identify these issues. A formal definition of multi-stream interaction is as follows:

Definition 2.17: A *Multi-stream Interactive Process (MIP)* is defined as a 5-tuple $\langle S, I, O, \mathcal{E}, \delta \rangle$, where S is the state space of the system; I and O are input and output domains respectively; $\mathcal{E} = \{E_1, E_2, \ldots E_n\}$ is a set of environments that are simultaneously interacting with the MIP, and δ is a computable partial function of the form $S \times \mathcal{E} \times I \rightarrow S \times \mathcal{E} \times O$; that, given an input from a particular environment and the present state of the process, maps to an output on possibly another environment and possibly changes the system state. □

The distinguishing factor between multi-stream and single-stream interactions is *channel sensitivity* of inputs and outputs. It is not sufficient to read inputs and determine outputs for each interaction; it is also necessary to know on which channel input is to be read from and to which channel the output is written.

In order to represent multi-stream interaction, Wegner and Goldin [WG99a] introduce an abstraction called Multi-stream Interaction Machine (MIM). MIM is an extension of SIM in that it interacts on multiple streams simultaneously. In contrast to the reducibility of Multi-tape TMs to single-tape TMs, MIMs cannot be reduced to SIMs. However, unlike the PTM which models a SIM, there is no mathematical model proposed for a MIM.

Theoretically, a MIM is a foundational model of an information system. Every IS represents an interaction machine that is interacting over multiple streams simultaneously. A formalism that defines a MIM would also form the underpinnings of activities like coordination and collaboration (by considering each collaborating actor as an interaction stream). But the MIM abstraction still lacks a complete formalism that can define all capable behaviors of a MIM.

Some of the pertinent questions that need to be answered for finding theoretical models for a MIM include the following:

1. What characterizes a MIM computation?
2. How can MIMs be compared? (*Reactive systems use a concept of observational equivalence to compare interactive systems. However, since MIM behavior can be affected by unknown adversaries on other interaction channels, observational equivalence may not be reliable*),
3. Are there functions which are not computable by an algorithm but are computable by a MIM? (*MIMs can model Turing Machines with Oracles which are shown to be more expressive than Turing Machines*)
4. How can the complexity of MIM behavior be caliberated?
5. What are the boundaries of MIM computation?
6. What kind of systems can be built with MIMs? Are they defined by any specific set of characteristics?

Channel sensitivity is not new to the IS modeling community. Many interactive models assign labels to their interaction channels to explicitly involve them as part of the model. Some examples are CCS [Mil89] and Broy's components [Bro97]. Labeling interaction channels do not model true channel sensitivity because they change the input and output domains. An input at time k and channel p can be represented as i_k^p. But this would place channels as part of the input-output domain and not as part of the computation itself. If the input and output domains are I and O, and the set of channels is denoted by $\mathcal{E}$, then interacting over labeled channels amounts to mapping between $I \times \mathcal{E}$ and $O \times \mathcal{E}$, rather than I and O. How channel sensitivity affects computation itself is apparent when MIM computation is represented as a domain. As seen in the previous chapter, channel sensitivity introduces individualized views from each channel over an interactive process. This is explored in the next chapter.

2.6 On the Irreducibility of Interaction to Algorithms

Wegner's work on interaction machines attracted a lot of attention because of its claim that interaction cannot be modeled by a Turing Machine. Since a TM is considered as a foundational model of everything that a computer can compute, such a claim has been sought to be refuted.

Prasse and Rittgen [PR98] argue that interaction machines are not more expressive than algorithms for a simple reason that they cannot compute something that an algorithm cannot. For example, interaction machines cannot solve the halting problem which is known to be an uncomputable function.

Single-stream interaction models partial functions while a TM models functions that are r.e. Prasse and Rittgen place models described by TMs and IMs as shown in Figure 2.9. The domain characterized by IMs is shown to encompass areas not covered by a TM; however, the authors suggest that this area might be "potentially empty."

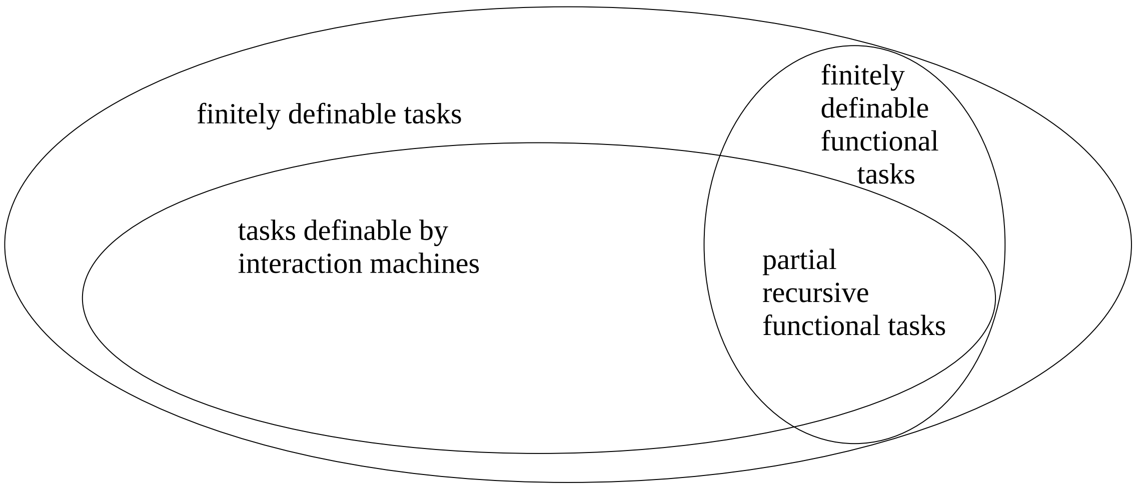

finitely definable tasks
finitely definable functional tasks
tasks definable by interaction machines
partial recursive functional tasks

Figure 2.9: Domains modeled by TMs and IMs

An alternative mechanism to reduce IM computations to that of a TM is to note the mappings computed by a SIM and a MIM. While a TM depicts a computable function between domains I and O, a SIM depicts a computable function between $S \times I$ and $S \times O$; and a MIM depicts a computable function between $S \times \mathcal{E} \times I$ and $S \times \mathcal{E} \times O$. However, considering $S \times I$ or $S \times \mathcal{E} \times I$ as input domains would mean that the state of the system (and the environment identifier, in the case of a MIM) have to be explicitly passed as input from the environment during each interaction.

While this would reduce IS behavior to algorithmic behavior; it does not reflect how IS behavior proceeds in reality. It would still be interesting to model interactive behavior as they occur in practice – where persistence of state and channel sensitivity are part of the problem solving system model and not outside of it.

For example, while it is not possible for an interaction machine to compute a non-computable function, it is possible for a MIM to provide greater expressiveness than a TM. MIMs can model Turing Machines with Oracles; and if the solution to a non-computable function is available from the Oracle, a MIM would have *effectively* computed the function.

Persistence of state and channel sensitivity are properties that are distinct from computation. Hence trying to reduce IMs to TMs would mean trying to reduce persistence of state and channel sensitivity to functions. Our contention is that TMs, SIMs and MIMs can be considered as foundational models for algorithms, single-stream interactive systems and information systems respectively. While a TM defines what a computer can compute a MIM defines what can be performed by an information system that not only consists of computation, but persistent state and channel sensitivity.

In the next chapter we show that interaction characterizes a richer domain as compared to algorithms. We call the domain characterized by a dynamic process as the *solution space* of the process. We also propose an algebra based on fixpoints for characterizing MIM solution spaces. This forms the formal underpinning for the dialogs model that is proposed for modeling IS dynamics.

2.7 A Taxonomy of Well-known Models of Computation

After having divided IS processes into three kinds namely, algorithmic, single-stream interactive and multi-stream interactive, it is pertinent to visit some well known models of computation and place them is the above taxonomy.

Subroutines and procedures: Subroutines and procedures are algorithmic in nature. They are not history sensitive or channel sensitive.

Functions with static variables: Static variables in a programming language like C are variables which are stored in the process memory and not on the execution stack. This makes static variables retain their values even after the function finishes its

TMs	SIMs	MIMs
functions, subroutines, procedures	functions with static variables	
	objects	group of objects sharing class variables
read-only databases	read-write single user databases	read-write multi-user databases
read-only transactions	read-write serializable transactions	read-write non-serializable transactions
	tuple-space	coordination applications using tuple space
Trained neural network	Neural network under training	
Rule based reasoning	(Possibly non-monotonic) reasoning with rule updates	

Table 2.3: A taxonomy of some well-known computational models

execution. A function that uses static variables hence models persistence of state. Its behavior is equivalent to that of a SIM.

Objects: Objects in object-orientation are history sensitive in their behavior. They have persistent state, but they are not channel sensitive. The methods of an object cannot distinguish between two or more processes that calls them. They are hence modeled by SIMs. However, a group of objects belonging to the same class which has one or more class variables behave collectively as a MIM. This is because, the behavior of an object is not only dependent on its interaction history but also on changes in class variables happening due to interactions on other interaction streams.

DBMSs: A DBMS behaves like an algorithm if the database is read-only. If the database is read-write, and the DBMS does not maintain user-profiles and/or distinguish between calling processes in any way, its behavior would be modeled by a SIM. Most contemporary databases are much more complicated than that and provide support for activities like coordination and collaboration which requires channel sensitivity. Hence they are modeled by a MIM.

Database transactions: A series of (read-write) database transactions that are serializable in nature are modeled by a SIM. If the set of transactions are not serializable, then they are modeled by a MIM.

Tuple space: The Linda [CG89] paradigm for coordination models coordination with the concept of a "tuple space." Processes can write tuples of data into the tuple space, and any process can read and/or remove tuples from the tuple space by providing a query in the form of a tuple template. Fundamentally, a tuple space is not channel sensitive; however it models persistent state. Hence it can be represented by a SIM. Some coordination models built on top of Linda tuple spaces incorporate means for authorizing tuples to be read and written by specific processes. This makes a tuple space channel sensitive modeled by a MIM.

Neural Networks Neural networks (Ex. [Lip87]) are computational models that mimic the process of learning by the human brain. An artificial neural network (ANN) can learn the correspondence between two domains after encountering sufficient exemplars of correspondence, and a suitable strategy of rewards and punishments by a teaching algorithm. The behavior of a neural network during the learning process is history sensitive. More learning results in richer behavior. Such a model requires persistent state and is equivalent to a SIM. Once a neural network has finished learning, and assuming that it does not learn anymore, its behavior becomes algorithmic.

Rule based systems: Expert systems that reason based on a set of rules can model different kinds of behavior. If the set of rules is static and does not change with usage, the behavior of such a system is algorithmic. On the other hand, if the set of rules evolves over time with interaction, the expert system models a SIM behavior.

Table 2.3 summarizes the above paradigms along three models of interaction.

Chapter 3

The Solution Space of an Interactive System

The difference between art and science is that science is what we understand well enough to explain to a computer. Art is everything else. –*Donald Knuth*

For modeling interactive behavior in a domain independent fashion, we introduce a concept called the "solution space" of a semantic process. A solution space is intuitively the abstract domain characterized by a dynamic process. We show that interactive solution spaces are richer than algorithmic solution spaces. Solution spaces are defined using domain theory.

We also show that interactive solution spaces require at least a three-valued system of logic for their characterization. The concept of a solution space and the three valued system of interactive dynamics is very important in designing interactive IS dynamics. They provide a domain independent fashion to conceptually represent interactive IS behavior. They also provide a mechanism for representing and verifying integrity constraints on IS functionality independent from the IS implementation.

3.1 Background Concepts

The underlying concept in this work is to represent interactive behavior in a domain independent fashion. To show that interactive behavior is more expressive than algorithmic behavior, we show that interactive behavior characterizes a richer domain than algorithmic behavior. This is done by introducing a concept called the solution space of a dynamic process. The concepts presented here are based on domain theory. The domains representing solution spaces are modeled as lattices which have to be traversed for performing any semantic process. From this perspective, algorithmic domains are shown to have a characteristic pointed structure. Interactive domains on the other hand, have no characteristic structure. They can also contain circular relationship chains of partial orderings without any least elements. From a set theoretic perspective, such domains cannot be represented by conventional sets that adhere to the axiom of foundation. They require stronger models presented by non-wellfounded sets or Hypersets [Acz88, Len98].

In addition, we show that interactive solution spaces require at least a three valued system of logic for their characterization. Conventionally, system dynamics are characterized by rules that describe their liveness. However, interactive dynamics can have a large amount of *possible* behaviors which are not decided by the program logic, but by the environment. This requires categorization of system dynamics into at least three kinds of behaviors. We find that deontic logics [MW93] provide intuitive mechanisms for modeling such a domain.

3.1.1 Domains

Fundamental concepts of lattices and domain theory are reviewed here.

Definition 3.1: A *domain* or a *lattice* is defined as $\langle X, \sqsubseteq \rangle$, where X is a set of elements (or "points") and $\sqsubseteq$ is a partial order among elements of S. In a lattice, any finite subset $S \subseteq X$ is characterized by a *least upper bound* and a *greatest lower bound.* $\square$

Definition 3.2: The *least upper bound* (lub) of any two elements a, b is an element c such that $a \sqsubseteq c$, $b \sqsubseteq c$; and if there exists any other upper bound c', then $c \sqsubseteq c'$. $\square$

Definition 3.3: The *greatest lower bound* (glb) of any two elements a, b is an element c such that $c \sqsubseteq a$, $c \sqsubseteq b$; and if there exists any other lower bound c', then $c' \sqsubseteq c$. $\square$

A domain X is said to be *complete* if the lub and glb requirements hold for any infinite subset of X. A domain is said to be *pointed* if it has an element $\perp$ such that for any element x of the lattice $\perp \sqsubseteq x$. The element $\perp$ is said to be the "bottom" element of the domain.

The behavior of algorithms can be described using domains. This is called the *denotational semantics* of an algorithm [How93]. Recall that the behavior of an algorithm can be represented as a computable function $f : I \rightarrow O$, where I and O are input and output domains. An algorithm may be made of many functions and objects. In pure λ-calculus everything is represented as functions. A domain representation of a set of functions of the form $I \rightarrow O$ is a set $\langle D, \sqsubseteq \rangle$, such that for any two functions f and g, $f \sqsubseteq g$ iff for all $x \in I$, $f(x) \sqsubseteq g(x)$. The meaning of the algorithm is said to be the solution of the equation $D = D \rightarrow D$, which states that the domain D is isomorphic to some function space from D to itself.

3.1.2 Hypersets

A domain can be represented as a set of elements such that, every point p in the domain is a set whose elements are all other points x of the domain such that $x \sqsubset p$.

Consider a set X of sets. Let the relation $\subset$ be defined over X. For any $x_1, x_2 \in X$, define $x_1 \subset_* x_2$ if there exists a chain $x_1 \subset x' \subset x'' \subset \ldots x_2$. The set X is said to be *wellfounded* if there is no such $x \in X$ such that $x \subset_* x$.

A wellfounded set adheres to the Axiom of foundation [How93] of the Zermelo-Fränkel set theory. The Axiom of foundation prevents circular structures in definitions of sets to eliminate paradoxes in set definitions. The axiom of foundation was developed as a response to the Russel's paradox that tries to define a set X as $X = \{x \mid x \notin x\}$. If X contains itself, then by definition of X it should not contain itself; while if X does not contain itself, by definition it should contain itself.

A paradox essentially has a circular structure. By making it mandatory for every set to be distinct from its elements, the axiom of foundation forbids circular definitions.

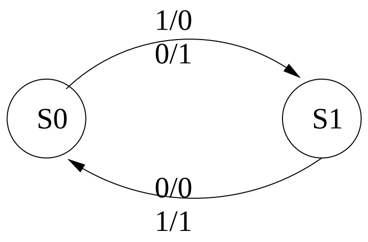
However, not every circular structure is a paradox. A paradox is a circular structure *that evaluates to a contradiction.* Hence, if a set X were to be defined as the set $X = \{x \mid x \in x\}$; X can be termed to be an element of itself and there is no paradox. Because of the elimination of the contradiction it forms a circular structure but not a paradox.

A set representation of a consistent circular structure is said to be a *non-wellfounded* set, or a *Hyperset.* They were first proposed by Aczel [Acz88]. Circular structures and circular phenomena are very ubiquitous. Barwise and Moss [BM96] and Lenisa [Len98] explore circular structures in computer science, philosophy, dynamical systems, natural processes and other disciplines.

Circular structures in computer science usually occur in the form of infinite data types like streams. A stream has the form $s = (a, s)$ where s is a stream, and a is the "head" of the stream. A stream is defined as a head element followed by a stream.

Non-wellfounded sets can be defined in a natural fashion using coinduction. A definition of a set by coinduction defines the behavior of a set of "observer functions" for all sequences of observation [Jac96, JR97]. Consider the coinductive definition of a stream on the basis of the behavior of two observer functions *head*() and *tail*(). The coinductive definition of a stream s is given as:
$s = \{head(s) = a;$
$tail(s) = s\}$,
where $a \in A$ is an element from the set of possible elements of the stream.

Milner [Mil89], Lenisa [Len98] and Wegner [WG99] have independently proposed the use of hypersets to describe interactive processes. Figure 3.1 shows an example of an



{
{(1,0)(0,0)(0,1)(0,0)(1,0)(1,1)(0,1)...}
{(0,0)(1,0)(0,0)(0,1)(1,1)(0,1)(1,1)...}
}

(a) (b)

Figure 3.1: Interactive streams as non-wellfounded sets

interactive process represented as a circular phenomenon. The diagram in Figure 3.1(a) describes a reactive machine in a switching circuit that alternates between two states and accepts a bit as input and provides another bit as output. In state S_0, the machine negates the input, and in state S_1 the machine reflects the input. Figure 3.1(b) depicts the machine as two interaction streams – one of which begins when the machine is in state S_0 and the other begins when the machine is in state S_1. Since it is possible for an interaction stream to begin from any state, both states can be considered as start states. Defining the system states S_0 and S_1 we obtain a circular definition: $S_1 = (S_0, a \to \neg a)$ and $S_0 = (S_1, a \to a)$ where $a \in \{0, 1\}$. The interaction state S_0 is defined in terms of S_1, and S_1 is defined in terms of S_0.

3.1.3 Deontic logics

System dynamics of most real world systems have a number of modalities like preferences, priorities, obligations, permissions, authorities, etc. In order to define and reason about such dynamics richer models of logic are required. Deontic logic is a branch of modal logic that is concerned with describing *normative* behavior.

Deontic logic was first proposed in 1926, by the Austrian philosopher Ernst Mally who introduced a modality "ought to be" towards a proposition (cf. [MW93]). Hence the term Op denoted p ought to hold. Unfortunately, with a single norm, Mally's system was reducible to propositional logic by the simple assertion that $Op \to p$. Although, Mally's system is no longer used, various other flavors of deontic logic have been proposed and applied in different domains.

Normative behavior concerns "the way things should be." It describes modalities of behavioral policies or administrative decisions that make up real world systems. Deontic logic has been used in domains like ethics, law, fault tolerance, computer security, etc. [MW93], and more recently, in specification of agent behavior [ES99, SBD+00].

A system based on deontic logic is made of modalities of the form *obligations*,

authority, *permissions*, *entitlement*, etc. Deontic axioms relate the different modalities and define deduction rules. Some example deontic axioms are given below:
$Op \Rightarrow \neg P \neg p$ (obligation to p means no permission to not p)
$p \rightarrow q \Rightarrow Op \rightarrow \neg Fq$ (if p leads to q then if p is obligated q should not be forbidden)
$O(p \wedge q) \Rightarrow Op \wedge Oq$ (obligation to p and q implies obligation to p and obligation to q)
$\neg(Op \wedge O\neg p)$ (one cannot be obligated to do conflicting action)

In this work we use deontic logic to describe the behavior of multi-stream interaction. We show that multi-stream interaction requires (at least) a three valued system of logic that describes its behavior. Deontic logic with three norms called *obligation*, *permission* and *prohibition* is used for this purpose.

3.2 The Solution Space of a PSP

A dynamic process can be seen as characterizing an abstract space of concern. This domain represents transitions from problems to solutions. Characterizing these domains mathematically shows how certain classes of processes are richer than other classes of processes.

In an intuitive sense, it is easy to show that interactive spaces are richer than algorithmic spaces. Consider a system consisting of a set of algorithms or functions. Functions maintain a subroutine relationship with their environment. They execute synchronously with respect to their environment. As a result, a system of functions can be organized in a hierarchical fashion where higher granule functions call lower granule functions. The hierarchy is well defined and different levels of hierarchy can be maintained using stacks. Figure 3.2(a) schematically depicts an algorithmic solution space.

An interactive process on the other hand, maintains a *coroutine* relationship with its environment. A system of coroutines calling one another can no longer be expressed as a well defined hierarchy. Figure 3.2(b) shows a system of coroutines interacting on a single stream with the environment; and Figure 3.2(c) shows a system of multi-stream interactive processes. As is evident here, incorporation of persistent state and channel sensitivity breaks down hierarchical structures. In fact, there is no characteristic structure of multi-stream interactive spaces.

Perhaps, this might be cited as the reason for the emergence of paradigms like design patterns and frameworks [CNM95, GHJV95, Pre95], that seek to identify and document recurrent characteristic structures of IS behavior.

Formal definition of a solution space: As seen in the earlier chapter, a TM, PTM and MIM can be considered as foundational models for functional programs, single-stream interactive systems and information systems respectively. The characteristic feature that distinguishes an algorithmic and interactive system is the nature of their *semantic pro-*

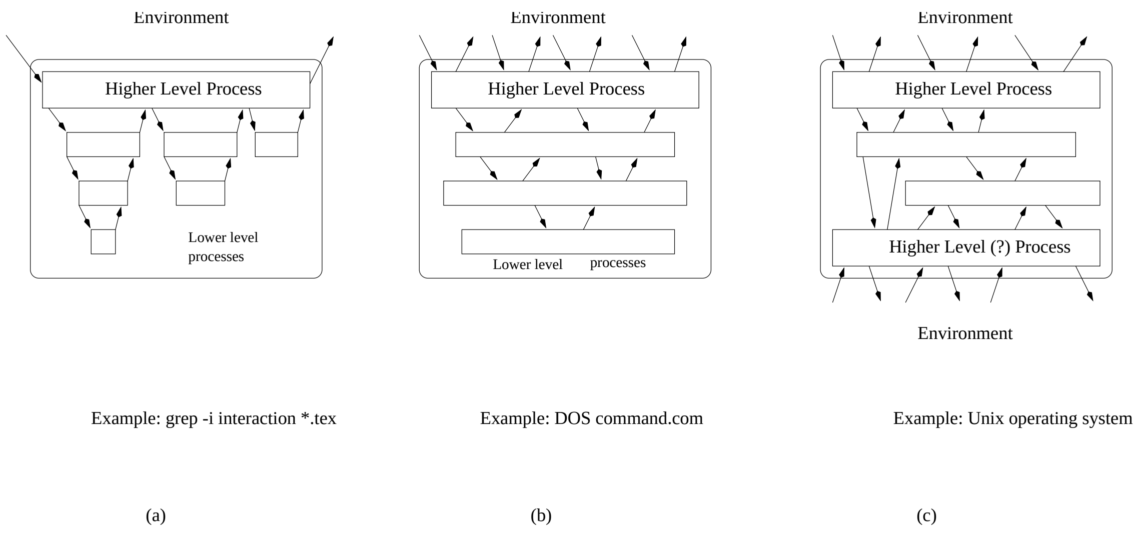


Figure 3.2: Algorithmic and Interactive Spaces

cesses. The semantic processes that make up an algorithmic system (for example, the Unix `grep` command) are algorithmic (functional) in nature. Similarly the semantic processes that make up a sequential interactive system and general information systems are single-stream interactive and multi-stream interactive respectively.

A semantic process that a system executes is called as a Problem Solving Process (PSP). A PSP either changes the state of a system from a given "problem" state to a desired "solution" state, or provides an output "solution" to an input "problem." A PSP is formally represented as follows:

Definition 3.4: A PSP is a set of strings of the form $\perp(i,o)^*\top$. The term $\perp$ denotes the start of the semantic process, $\top$ denotes the end of the semantic process, i refers to an input element, and o refers to the obtained output. i and o range over their respective domains I and O. An algorithmic PSP (APSP), has only one (i,o) element, while an interactive PSP has more than one (i,o) element. □

The *solution space* of a PSP is the abstract domain characterized by a PSP. The domain characterized by a PSP is represented by a lattice of *fixpoints* and an ordering relationship between fixpoints. A fixpoint of a PSP is defined as follows:

Definition 3.5: Consider any PSP of the form $\perp(i,o)^*\top$. For the domain characterized by this PSP, the following rules hold:

- A *fixpoint* of the PSP is defined as $\perp(i,o)^*$, where $(i,o)^*$ is 0 or more occurrences of (i,o) after the start of the PSP $\perp$.
- $\perp \sqsubset s$ where s is any fixpoint of the PSP.

- For any fixpoint s of the PSP, $s \sqsubseteq s(i, o)$.
- Any fixpoint $s \sqsubseteq \top$.

□

A dynamic system is a collection of PSPs. If the system is made up of solely algorithmic PSPs, then the system is said to be an algorithmic system. If the system supports single-stream interactive PSPs but not multi-stream interactive PSPs, then the system is said to be a single-stream interactive system. A system that supports multi-stream interactive PSPs is said to correspond to information systems in general.

The solution space of a system is the domain that is obtained by the combination of the solution spaces of all PSPs that it supports. This is obtained by taking the union of all fixpoints from all the different PSPs of the system. Fixpoints which correspond to the same semantic state are combined. Using such a mechanism the solution spaces of algorithmic, single-stream and multi-stream interactive processes are characterized in the following sections.

3.3 Solution Space of an Algorithm

As seen in Chapter 2 the generic model of an algorithm is characterized by a start state s_0 and a set of dynamics defined from that state. A TM that models an algorithm starts from the same state s_0 for all PSPs, and ends when a solution state is reached.

An algorithmic system is made up of PSPs of the form $\perp(i, o)\top$. Each such PSP is made up of two fixpoints corresponding to $\perp$ and $\perp(i, o)$ respectively. The solution space of an algorithmic system is hence a pointed domain where every process starts at a fixpoint corresponding to s_0 and ends in one of the fixpoints corresponding to one of the halt states.

Definition 3.6: The *solution space of an algorithm* is defined by the set $\mathcal{S}_{TM} = \{s_0 h \mid s_0, h \in S; h \in H\}$. This denotes sequences of state transitions for all problems solving processes executed by the TM for the algorithm. □

Figure 3.3 schematically depicts the solution space of an algorithm. All PSPs begin at a fixpoint corresponding to $\perp$ or the start state s_0; and end at a fixpoint corresponding to one of the halt states h. The characteristics of such a domain are as follows:

- The domain is pointed, and every string that makes up the solution space begins from s_0.
- The fixpoint corresponding to s_0 is the least element of the domain. This fixpoint can be termed as the bottom element $\perp$ for the entire solution space.

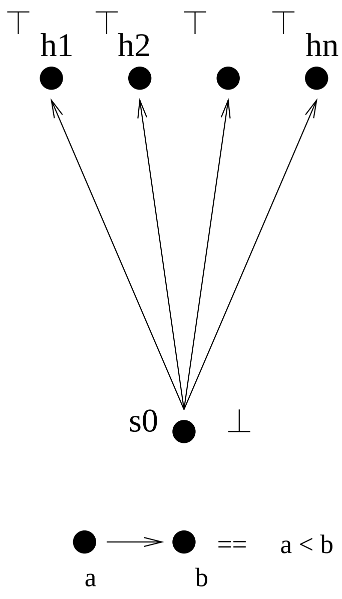


Figure 3.3: Solution space of an algorithm

- The solution space is *wellfounded.* In a canonical representation of $\mathcal{S}_{TM}$, the bottom element $\perp$ forms the minimal fixpoint. The solution space can be defined by a process of induction from the minimal fixpoint $\perp$.

The significance of a fixpoint based representation would be apparent when abstraction of a dynamic process is considered. The behavior of an algorithm (function) can be abstracted by a declaration showing the function name and the input and output parameter types. This is sufficient to abstract the behavior of a function because any function call begins computation from the same fixpoint. For example, consider an algorithmic system – the Unix `grep` command. The `grep` command computes an output solution in the form of a set of strings that match the input problem specification in the form of a regular expression. The `grep` command incorporates different PSPs like: regular expression match ignoring case, match strings with context, match line numbers, etc. The `grep` command is abstracted by its synopsis: `grep [-CVbchilnsvwx] [-f file] files`.

Such an abstraction is sufficient to represent the behavior of the command `grep` since it encapsulates every parameter that determines the behavior of the command whenever invoked. As we shall see in the subsequent sections, such a declaration is not sufficient to represent the behavior of interactive processes.

3.4 Solution Space of Sequential Interaction

Single-stream interaction is characterized by persistence of state across computations. A PTM which represents single-stream interaction begins computation from a state determined by the contents of its persistent worktape, rather than from the same start state for every computation.

For a series of PSPs executed on a sequential interactive machine, the system state transitions would be of the form $\perp(s_0s_1\dots s_n)\top, \perp(s_ns_{n+1}\dots s_{n+k})\top, \dots$. The persistent system state may determine where a new PSP begins and hence there may be no common starting point for all PSPs.

3.4.1 System state vs interaction state

Persistent state may be defined in two ways: *system state* and *interaction state.* It is important to distinguish between the two. The interaction state is specific to a PSP and is determined based on how many interactions have taken place since the start of the PSP. The system state is the state of the system itself. This is determined based on the values of state variables, and is persistent across interactions and across PSPs. Interaction state and the system state are formally contrasted as follows:

Definition 3.7: The k^{th} *interaction state* of a PSP $\perp(i_1, o_1)\dots(i_n, o_n)\top$ is defined as the stream $\perp(i_1, o_1)\dots(i_k, o_k)$. It is the semantic state of the interactive process reached after the first k interactions. An interactive state is represented by a *fixpoint* in the interactive solution space. □

Definition 3.8: The *system state* of a dynamic system is defined as a function over the values of a set of variables called the *state variables* of the system. □

Recall that a single-stream interactive process can be represented as a set of strings of the form $s_1..s_n$ where each s_i represents a fixpoint in the interactive behavior. An interaction state is the same as a fixpoint that is reached after a series of interactions; or alternatively, that defines a set of possible behaviors from that point. The system state in contrast, is a function of the values of data elements stored in the system's state variables. The number of system states is usually infinite, even for very small systems. On the other hand, the number of interaction states is finite. Interactive processes have to be finitely specifiable so as to be implementable. The set of interaction states hence divide (not necessarily partition) the set of system states into equivalence classes that associate each interaction state with all possible system states that the system can be in, under this fixpoint.

It can be noted that for algorithmic systems which do not have persistent state across computations, the system state and the computational state are the same.

3.4.2 Sketching the solution space of sequential interaction

Let SIM be the system whose behavior is single-stream interactive in nature. Let $PSP_0 \ldots PSP_n$ be the set of all interactive PSPs that can be run on the system. Each PSP can be defined as a set $\{s_0..s_m\}$ of fixpoints that depict interaction states. Let S be the set of all fixpoints from the set of all PSPs of SIM.

Each PSP has a number of interactions in mapping between $\perp$ and $\top$. At each interaction, the computational behavior of the system is dependent not only on the input provided, but also on the current fixpoint. The solution space is hence the set S and an ordering relationship $\sqsubseteq$ among elements of S. Figure 3.4 schematically depicts such a domain. The solution space is made up of fixpoints of all the PSPs adopted by the system. Fixpoints from different PSPs may be identical and may be combined. Some of the fixpoints in the domain depict the start of problem solving $\perp$, of specific PSPs; similarly few other fixpoints depict the end of problem solving $\top$ of specific PSPs. At any given moment when the system is ready to interact, it is in any one of the fixpoints. On interaction with the environment, the system computes its way to another fixpoint in the domain. The ordering relationship $\sqsubset$ is defined based on which fixpoints are reachable from which other fixpoints. However, there is no specific characteristic of the ordering relationship that is true for all single-stream interactive systems in general. The solution space of sequential interaction can have arbitrary ordering relationships among fixpoints. In fact, ordering relationship chains could also be circular such that some fixpoint $s \sqsubset_* s$.

In a formal sense, the solution space of sequential interaction is defined as follows:

Definition 3.9: The solution space of sequential interaction is defined by a set of strings of the form: $\mathcal{S}_{SIM} = \{q \mid q \in S^*\}$ where S is the set of fixpoints of the system. $\Box$

The notion of fixpoints is important for defining behavioral abstraction. As noted in the previous section, the behavior of an algorithmic process can be abstracted by a declaration of the function name and the input-output types. However, such an abstraction is insufficient for the behavior of an interactive process. The behavior of a (single-stream) interactive process is dependent not only on the method declaration, but also on the current fixpoint.

A single-stream interactive process can be abstracted by a pair denoting a fixpoint and the method declaration. This aspect is important in designing interactive behavior of an IS. In the dialog model proposed in Chapter 5, fixpoints are modeled as fundamental building blocks of an interactive process.

Lemma 3.1: The solution space of single stream interaction is richer than the solution space of an algorithm.

Proof: $\mathcal{S}_{SIM}$ contains strings of the form $q \in S^*$; while $\mathcal{S}_{TM}$ contains strings of the form

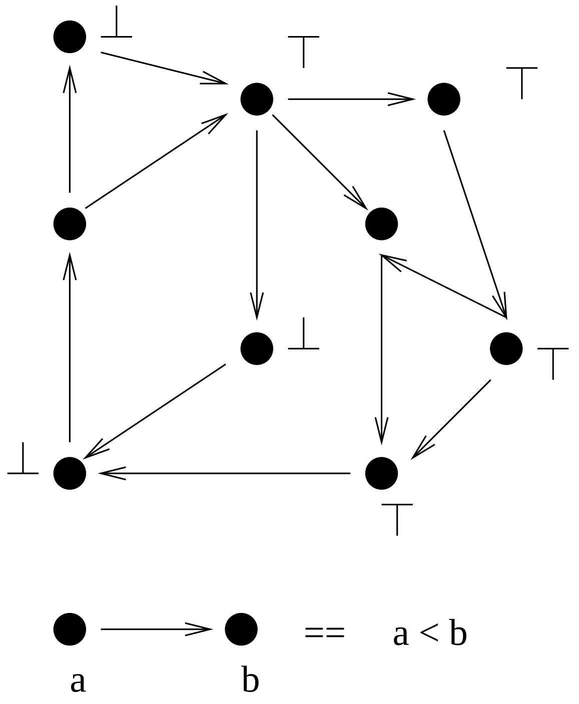


Figure 3.4: Solution space of single-stream interaction

$s_0h_i, s_0, h_i \in S$. $\mathcal{S}_{SIM}$ hence may contain strings that are not possible in $\mathcal{S}_{TM}$. □

The characteristics of a SIM solution space are as follows:

- There is no unique starting point that determines every string in the solution space.
- As a result, there might be no least elements in the domains.
- The partial ordering $\sqsubset$ across fixpoints may be circular such that for any fixpoint s, $s \sqsubset s$, making the domain non-wellfounded.

The non-wellfounded nature of interactive domains have been acknowledged by many authors, prominent among them are Milner [Mil89], Wegner [WG99] and Lenisa [Len98]. The above authors have concentrated on proof principles and characterization mechanisms for interactive domains based on their non-wellfounded nature. In this work, we address the question of how such a characteristic affects behavioral abstraction and the design process of an interactive information system.

A hyperset representation of a solution space is sufficient to describe only single stream interactions where it is not important which channels inputs are read from and to which channels outputs are given. In the following section we address how channel sensitivity affects the solution space of interactive processes.

3.5 Solution Space of Multi-stream Interaction

The characteristic feature of multi-stream interaction is channel sensitivity. For defining the solution space of multi-stream interaction, it is necessary to introduce the effect of channel sensitivity into a non-well-founded set representation.

3.5.1 Sketching the solution space of multi-stream interaction

One way to represent channel sensitivity is to use a channel identifier for each input and output element of a stream. Thus i^p would denote input from the p^{th} channel and o^q would denote output to the q^{th} channel. However such a representation has changed the input and output domains to $I \times \mathcal{E}$ and $O \times \mathcal{E}$ respectively, and has taken channels out of the model.

When channel sensitivity is introduced as part of the model, the input and output domains of the model remains unchanged. For any user interacting with a MIM, the behavior of the MIM as seen by the interaction channel of the user, is dependent not only on the interaction history, but also on interactions taking place on other interaction streams. For the user, the behavior seems to be affected not only by hidden variables, but also by hidden adversaries [WG99a].

In order to explicitly introduce the effect of channel sensitivity onto an interactive solution space, the notion of hidden adversaries is specified in the form of the following axioms:

1. If a MIM has n interaction channels and behavior S, then the behavior on each interaction channel at any point in time would be $S_i \subseteq S$, $i = 1..n$.

2. At any point in time on any interaction channel i, the behavior S_i on the channel is influenced by behaviors S_j $j = 1..n,\ j \neq i$ taking place on other channels.

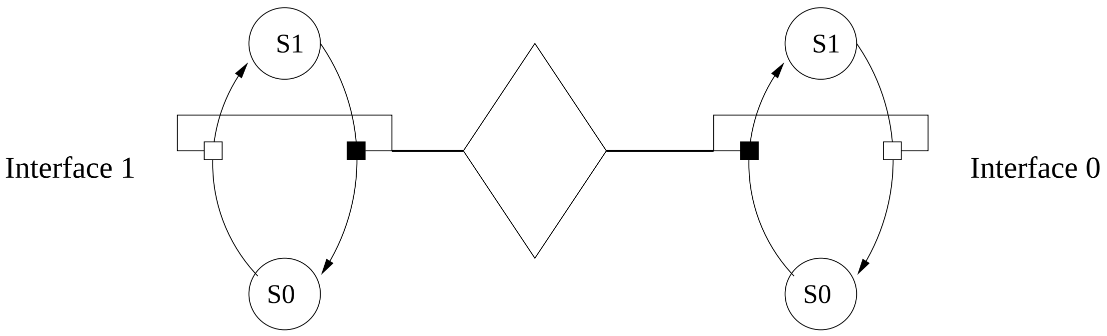


Figure 3.5: Channel sensitive behavior of a MIM

Figure 3.5 shows an example. Consider a MIM from a switching circuit whose channel insensitive behavior is described by the fixpoints $S = \{S_0, S_1\}$. The behavior from S_0 is defined as $S_0 \xrightarrow{a/a} S_1$. In other words, when the machine is at S_0 it takes an input and reflects the input as output, and moves to fixpoint S_1. The behavior from the fixpoint S_1 is defined as $S_1 \xrightarrow{a/\neg a} S_0$. At S_1, the machine accepts an input and outputs the negation of the input and moves to fixpoint S_0.

Let the machine have two interaction streams, I_0, I_1 which share S among them. At any point in time, let a behavior be available to at most one stream. Thus, if S_0 is available to I_0, it is not available to I_1 and vice versa. The same is true for S_1. Such a constraint introduces channel sensitive behavior. The fixpoint (and hence the behavior of the machine) available on any interaction stream is dependent not only on the interaction history on this stream, but also on interactions taking place on the other stream.

The figure represents channel sensitivity by replicating S on both interaction streams and associating them with a constraint relationship. The constraint relationship ensures that at any given point in time, only one of the fixpoints is accessible from any interaction stream. It also ensures that when behaviors change fixpoints on one stream, they also change correspondingly on the other stream. Such a constraint is said to have modeled the *dynamic integrity* of the MIM's behavior.

To generalize on the above example, the solution space of a multi-stream interaction (channel sensitive behavior) can be represented as an *algebra* over the solution space of a single stream interaction (channel insensitive behavior). It can be modeled as a *schema* of entities and relationships, where entities depict behavioral spaces on specific interaction channels; and relationships are constraints showing how streams affect one another. These constraints are also integrity constraints on MIM dynamics that specify correctness criteria for the MIM's channel sensitive behavior. Figure 3.6 depicts this domain schematically.

The pertinent question now would be: what is the nature of these constraints between interaction channels? This question is addressed in more detail in the next subsection. Before that, the solution space of a multi-stream interaction is formally defined as follows:

Definition 3.10: The solution space of a multi-stream interaction is defined as $\mathcal{S}_{MIM} = \langle S, \sqsubset, \psi \rangle$ where S is the set of fixpoints of interactive behavior on any interaction stream, $\sqsubset$ is an ordering relationship between fixpoints; and $\psi : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ is a constraint relationship that determines which fixpoints are available at which channel. $\square$

Lemma 3.2: The solution space of multi-stream interaction is richer than the solution space of single stream interaction.
Proof: $\mathcal{S}_{SIM}$ can be obtained from $\mathcal{S}_{MIM}$ if ψ is empty. $\square$

The fixpoint representation of a MIM solution space again elucidates on the nature of behavioral abstraction of a multi-stream interactive process. If the behavior of a MIM is to be represented as an abstract data type on any of its channels, it is required to have the

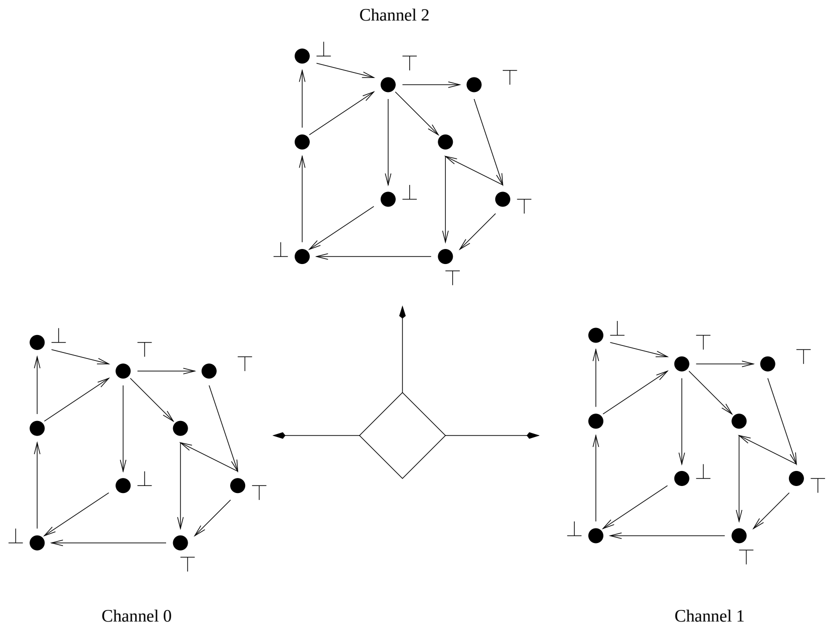


Figure 3.6: Solution space of multi-stream interaction

Algorithm	**Single-stream interaction**	**Multi-stream interaction**
function declaration	function declaration + fixpoint	function declaration + fixpoint + constraints
Example: `int strlen(char *)` The `strlen` function	`@prepaid.order(Order)` The `order` function at fixpoint `prepaid`	`[0]@prepaid.order(Order)` The `order` function at fixpoint `prepaid` when there is an obligation constraint.

Table 3.1: Behavioral abstractions of algorithms, single-stream and multi-stream interaction

following necessary information: method signature, current fixpoint, constraints acting on the fixpoint. It should be possible for program logic to invoke a multi-stream interactive process "if it is at fixpoint x and the constraint y holds." Table 3.1 compares behavioral abstractions of algorithms, single-stream and multi-stream interactions.

Another pertinent question at this time is to ask whether behavioral abstractions of interactive processes be reduced to that of an algorithm. After all, in the example given in Table 3.1 the terms `prepaid.order()` and `[0].prepaid.order()` may be considered as function declarations themselves. Such a question is analogous to the question of the reducibility of an interactive process to an algorithmic process. It also asks whether interactive solution spaces are reducible to algorithmic solution spaces.

In essence, it is true that interactive behaviors can be reduced to algorithmic behaviors (by changing the input and output domains). In an analogous fashion it can be argued that interactive solution spaces can be reduced to algorithmic solution spaces. However, the resultant algorithm that models an interactive behavior would be much more complex.

In order to model interactive behavior algorithmically, it is necessary to compute the state of the world and the interaction history. Wegner [Weg97] provides an example called "driving home from work" as a task that can be tractable interactively, but very complex algorithmically.

Referring back to the problem of fixpoints, it is insufficient to represent `prepaid.order()` and `[0].prepaid.order()` as separate functions because, an interactive program should be able to reason on an interactive functionality based on its fixpoint and constraints.

Consider a functor called `fixpoint(f)` that returns the current fixpoint of an interactive process `f()`. Thus `fixpoint(@prepaid.order) = prepaid`. An interactive software should be able to adopt decision logic of the form `if (fixpoint(order) == prepaid)`. The same could be extended analogously to the constraint acting on the fixpoint. Such a decision logic would not be possible if fixpoints and constraints were expanded to form separate functions.

3.5.2 Three-valued logic for interaction spaces

In order to determine the nature of constraints that make up ψ we need to consider different ways in which constraints on dynamic processes are specified.

One of the most common kinds of constraints on dynamics, is an exclusion constraint. This is implemented in various forms like semaphores and monitors in process scheduling, locks in databases, etc. An exclusion constraint *forbids* a process from performing a specified activity. The negation of an exclusion is *permission*, so that if a process has no exclusion constraint acting on it regarding any activity s, it is then said to be *permitted* to perform s.

On the other hand are process specifications, that describe activities that have to be performed by processes. A process that adheres to these specifications is expected to necessarily perform the specified activities. Such a specification of a process can also be considered as a "constraint" that *obligates* the process to perform certain activities, in order to maintain specific (liveness) properties. In conventional process specification, if a task s is not part of the specification, the process is said to be *forbidden* from executing s. However, if s is part of the specification (and no optional condition is explicitly specified regarding the execution of s), the process is said to be obligated to perform s.

As is evident here, both kinds of specification are constraints on an interaction space. We call the former as *negative* constraints and the latter as *affirmative* constraints. Figure 3.7 schematically depicts interaction spaces in the face of negative and affirmative constraints.

It is quite apparent that for a solution space of a system, either negative or affirmative constraints by themselves is inadequate. In any interactive process, there is not only a

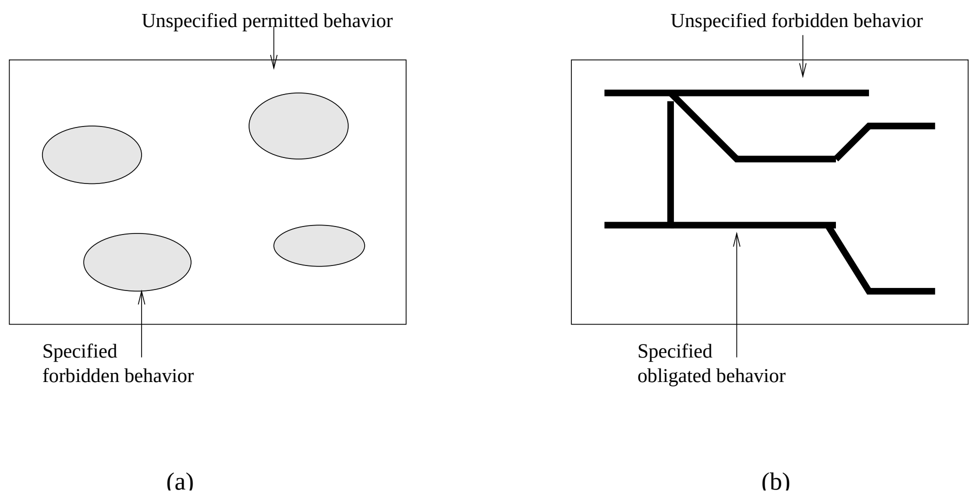

Unspecified permitted behavior
Specified
forbidden behavior
(a)
Unspecified forbidden behavior
Specified
obligated behavior
(b)

Figure 3.7: Negative and affirmative constraints

requirement for liveness, there are also many different means for achieving this liveness. For example, an environment that is interacting with any object may call any of the object's methods as part of an interaction. This specifies that any interaction may be represented by many *possible* behaviors. However, if the interaction by the environment is part of a PSP, there are usually certain sequences of methods that have to be *obligatorily* invoked to complete the PSP. Similarly, integrity issues may forbid the invocation of certain methods at certain interaction states.

The above elucidates the need for a three-valued logic to describe interactive spaces. There are at least three dimensions to an interactive behavior: its liveness, the set of possible behaviors and the set of forbidden or disabled behaviors. Usually, there are many more dimensions in any real world information system. Some example dimensions are behavioral priorities, rights, entitlements, etc. However, in this work we concentrate on a three-valued logic that forms the basic framework of an interaction space.

The essential point here is that, MIM spaces cannot be adequately characterized using two-valued logic, because of the large number of *possible* behaviors as against the required liveness behaviors. The space of possible behaviors is also responsible for the richness of open (interactive) system dynamics as against closed system dynamics. The next section formalizes this notion of a three-valued logic and develops an algebra for describing interaction spaces based on fixpoints.

3.6 A Fix-point Algebra for MIM Spaces

In this section we formalize on the ideas presented in this chapter. An algebra of fixpoints is developed to characterize the solution space of a MIM. The solution space of a MIM is considered to consist of a number of fixpoints. A fixpoint represents a point of interaction

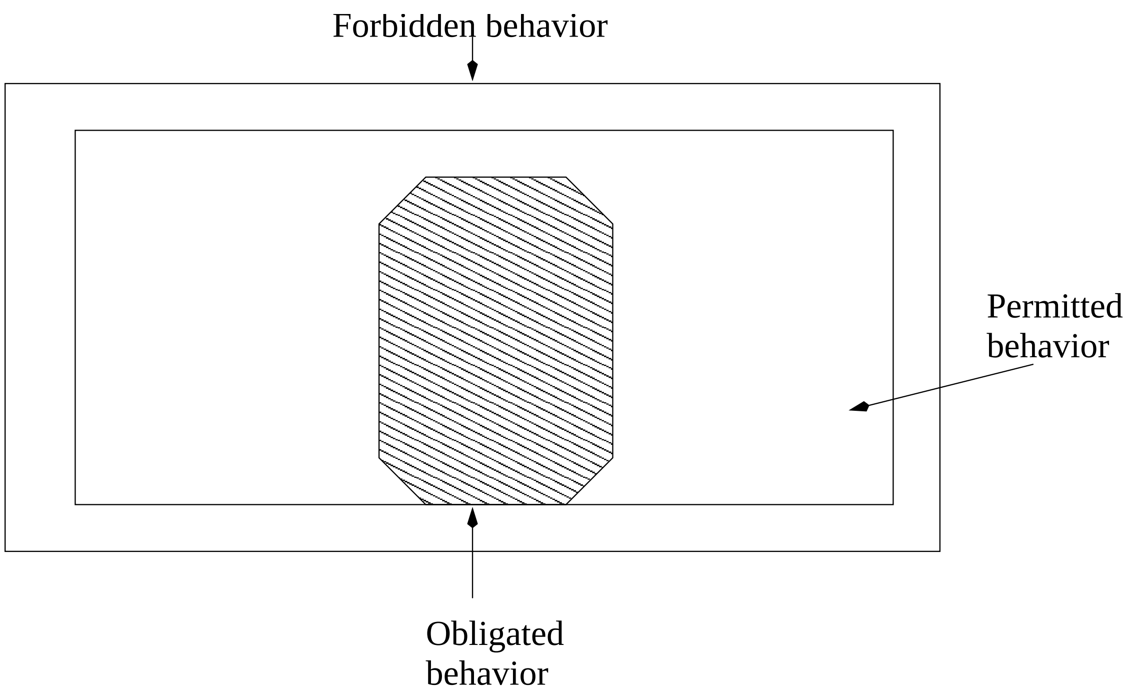


Figure 3.8: Three-valued logic for MIM spaces

and encapsulates computational behavior to possibly another fixpoint in the solution space. Each fixpoint can take on a truth-value in a three valued system of logic. The algebra presented here adopts such a three-valued logic showing liveness properties, possible behaviors and forbidden behaviors.

Deontic Axioms: Any formula in the MIM solution space can evaluate to a truth value corresponding to one of the following: O (obligatorily true), P (permissively true), or F (prohibitively true).

The relationship between O, P and F is schematically depicted in Figure 3.8. Obligation and permission are both evaluated to “true” while a contradiction evaluates to prohibition. The space of permitted behavior subsumes the space of obligated behavior and is mutually exclusive from forbidden behavior. In a formal sense, the norms O, P and F are related to one another by the following axioms:

1. $Oa \rightarrow Pa$ (what is obligated is also permitted)
2. $\neg Oa \rightarrow (\neg Pa \rightarrow Fa)$ (what is not obligated is forbidden if it is not permitted)
3. $\neg Oa \rightarrow (\neg Fa \rightarrow Pa)$ (what is not obligated is permitted if it is not forbidden)
4. $Pa \rightarrow \neg Fa$ (what is permitted is not forbidden)
5. $Fa \rightarrow \neg Pa$ (what is forbidden is not permitted)

6. $Oa \wedge Pa \rightarrow Oa$ (what is obligated and permitted if obligated)
7. $Oa \wedge Fa \rightarrow Fa$ (what is obligated and forbidden is forbidden)
8. $Pa \wedge Fa \rightarrow Fa$ (what is permitted and forbidden is forbidden)

Term elimination rules:
9. $Oa \wedge Fb \rightarrow Oa$
10. $Pa \wedge Fb \rightarrow Pa$
11. $Oa \vee Fb \rightarrow Oa$
12. $Pa \vee Fb \rightarrow Pa$

13. $Oa \vee Pa \rightarrow Oa$ (what is obligated or permitted is obligated)
14. $Oa \vee Fa \rightarrow Oa$ (what is obligated or forbidden is obligated)
15. $Pa \vee Fa \rightarrow Pa$ (what is permitted or forbidden is permitted)

16. $Oa \vee \neg Oa \rightarrow Oa$
17. $Pa \vee \neg Pa \rightarrow Pa$
18. $Fa \vee \neg Fa \rightarrow Pa$

19. $F(\neg a) \rightarrow Oa$ (it is forbidden for a to not hold – a is obligated to hold)
20. $P(\neg a) \rightarrow Pa$ (it is permitted for a to not hold – it is permitted for a to hold)
21. $O(\neg a) \rightarrow Fa$ (it is obligated for a to not hold – it is forbidden for a to hold)

22. $F(a \wedge b) \rightarrow Fa \vee Fb$
23. $F(a \vee b) \rightarrow Fa \wedge Fb$
24. $P(a \wedge b) \rightarrow Pa \wedge Pb$
25. $P(a \vee b) \rightarrow Pa \vee Pb$
26. $O(a \wedge b) \rightarrow Oa \wedge Ob$
27. $O(a \vee b) \rightarrow Oa \vee Ob$

Some derivable theorems:
$F(a \rightarrow b) \Rightarrow Oa \wedge Fb$ (by expanding $a \rightarrow b$ to $\neg a \vee b$)
$P(a \rightarrow b) \Rightarrow Pa \vee Pb$
$O(a \rightarrow b) \Rightarrow Fa \vee Ob$
$Oa \wedge \neg Oa \Rightarrow Oa$
$Pa \wedge \neg Pa \Rightarrow Fa$
$Fa \wedge \neg Fa \Rightarrow Fa$
$Oa \wedge \neg Pa \Rightarrow Fa$
$Pa \wedge \neg Oa \Rightarrow Pa$

In the system specified above, contradiction equates to a prohibition. This is not an exact interpretation in a semantic sense. A contradiction is something that is inconsistent with the system of axioms while a prohibition is something that is explicitly forbidden by the system. However, we equate contradiction to prohibition in order to retain simplicity. To be precise however, a fourth norm called "unknown" (represented by U) may be required. The norm U is defined as: $\neg O \wedge \neg P \wedge \neg F \rightarrow U$. A model is said to be *closed* if U is empty, or *open* $\exists s \in S, Us$. In this work we consider only closed worlds where contradictions are equated with prohibitions.

Signature: A MIM specification is a *signature* σ over a finite set of *constants* called "fixpoints" and an *ordering relationship* $\leq$ across fixpoints.

$$\sigma = (S, \leq)$$

In the above, S denotes a set of constants of the solution space called *fixpoints.* They represent points of interaction and encapsulate computational behavior from each interaction point. $\leq$ is an "ordering relationship" over the set S. The relationship $\leq$ is defined as a set of pairs of the form (s_1, s_2) $s_1, s_2 \in S$. The relationship is also represented as $s_1 < s_2$. If any fixpoint $s \in S$ is said to "hold" then it is an assertion that the MIM "behaves according to the behavior encapsulated by s on interaction with some environment." The signature of a MIM is also called its *vocabulary* that defines its behavior.

Model: A *model* M_σ of a MIM is a structure based on a signature σ, a universe of free variables X, a set of formulas R, and a traversal function $call()$ that represents interaction with the MIM. $call()$ is also said to "traverse S."

$$M_\sigma = (X, R, call, \sigma)$$

Atomic terms: An *atomic term* is any $s \in S \cup X$. The truth value of an atomic term s may have three values and are respectively represented as: Os, Ps and Fs.

Term: A term is any atomic term or an atomic term with the application of the traversal function $call()$. Let $Atoms_M$ be the set of all atomic terms over M. The behavior of $call()$ is defined as follows.

- For any $s \in Atoms_M$ and either Os or Ps holds, then $call(s) = p_1 \vee \cdots \vee p_n$ such that $p_1 \ldots p_n \in S$, and $s < p_1, \ldots, s < p_n$.
- For any $s \in Atoms_M$ and Fs holds, then $call(s) = \phi$ the empty set.

The traversal function $call()$ is defined over a fixpoint. For any fixpoint s, $call(s)$ returns the set of all fixpoints reachable from s by the ordering relationship $<$. If multiple fixpoints are reachable from s, which fixpoint the system actually moves to would be decided by the interaction and decision logic. We are not interested in the actual exchange of inputs and outputs that constitute an interaction; but only on transitions between fixpoints. So, the return value of *call*() is modeled as a disjunction of all possible fixpoints that are reachable.

call() is said to *traverse* the solution space moving from one fixpoint to another. Traversal through a fixpoint is possible only if the fixpoint is obligated or permitted. Traversal through a forbidden fixpoint is not possible.

For any model M_σ, the set $O(M_\sigma) \subseteq S \cup X$ denotes the set of all atomic terms that are obligated. The sets $P(M_\sigma)$ and $F(M_\sigma)$ denote the set of all atomic terms that are permitted and forbidden respectively.

Literal (Atomic formula): Let $Terms_M$ denote the set of terms on a model M. A *literal* is either a term or a relationship $r \in R$ of the form $\langle p_1, p_2, \ldots p_n \rangle$, where $p_1 \ldots p_n \in Terms_M$. Let the set of literals be denoted by Lit_M.

A relationship $r = \langle p_1, p_2, \ldots, p_n \rangle$ with arity n represents an condition of the form: $p_1, p_2, \ldots, p_{n-1} \rightarrow p_n$.

Formula: A formula in a model M_σ is defined as follows: (a). Any literal $a \in Lit_M$ is a formula. (b). if a and b are formulas, then so are $a \wedge b$, $a \vee b$, $\neg a$, $a \rightarrow b$, $\forall a$, $\exists a$, $call(a)$. The behavior of *call*() on formulas is defined as follows: $call(a \wedge b) = call(a) \wedge call(b)$, $call(a \vee b) = call(a) \vee call(b)$, $call(a \rightarrow b) = call(a) \rightarrow call(b)$, $call(\forall a) = \forall a.call(a)$, $call(\exists a) = \exists a.call(a)$

Channel: A *channel* of the interaction machine is any set $C \subseteq S \cup X$. For any channel C, the following subsets are defined: $O_C = \{s \mid s \in C, Os\}$; $P_C = \{s \mid s \in C, Ps\}$; and $F_C = \{s \mid s \in C, Fs\}$.

Theorem Proving: A given model M_σ is said to *model* a given formula φ given a formula η, if φ can be derived from η. This is represented as $M_\sigma(\eta) \models \varphi$.

Channel sensitivity of a formula and a process of derivation is a projection of the derivation process onto a channel. Hence for any $s \in S \cup X$, $O(C)s$ holds if $s \in C$ and Os holds. The norms for $P(C)s$ and $F(C)s$ are defined analogously.

At any point in a process of derivation, a model M_σ is said to be *deadlocked* if $O(M_\sigma) = P(M_\sigma) = \{\}$. A model M_σ is said to be *idle* if $O(M_\sigma) = \{\}$.

At any point in a process of derivation, a channel C is said to be *disabled* (not necessarily deadlocked) if $O(C) = P(C) = \{\}$. A channel C is said to be *idle* if $O(C) = \{\}$.

Validation: For validation of system dynamics the following modal operators are introduced on a formula: "next change" denoted by $\Box$, "previous change" denoted by $\Diamond$, and "precedes" denoted by $\triangle$. An assertion like $\Box Op$ means that the next change that applies to p is an obligation to p. Similarly, an assertion like $\Diamond Pp$ means that the previous change that applied to p was a permission to p; and $\triangle(Pp, Op)$ means that a permission to p precedes an obligation to p.

Based on these modal operators, the following properties of a MIM behavior can be validated:

Liveness: $Op \rightarrow \Box call(Op)$ (whatever is obligated is done)

Consistency: $call(p) \rightarrow \Diamond Pp$ (whatever is done is permitted)

Safety:

$Fp \rightarrow \neg\Box call(p)$ (whatever is forbidden is not done)

$\triangle(Pp, Op)$ (a permission to p precedes an obligation to p)

summarily permitted)

$Op \rightarrow \triangle(\neg Op, Fp)$ (whatever is obligated is not summarily forbidden)

The salient features of the fixpoint algebra are explained below in an intuitive fashion. The solution space of the interacting machine is made up of a set of "fixpoints". Each fixpoint represents a point of interaction and encapsulated computational behavior from that point. The interactive behavior of the machine is defined in terms of first order logic using free variables above fixpoints.

The domain representing channel insensitive behavior of the MIM is modeled by an ordering relationship $\leq$ on the set of fixpoints. The relationship $<$ between fixpoints denotes which interaction states are reachable from which other. The domain of fixpoints can be traversed using the *call*() function on fixpoints. A *call*() corresponds to an interaction with an environment that calls behavior from a particular fixpoint. *call*() is also defined on free variables, and which behavior is invoked as a result of *call*() is determined by which fixpoint is bound to the free variable. Process semantics and integrity constraints are encoded into the set R that shows relationships across fixpoints.

Any fixpoint or variable evaluates to a truth value in a three-valued logic. A fixpoint can be obligated, permitted or forbidden. Obligation and permission evaluates to true and prohibition evaluates to false. An interaction with a fixpoint from the *call*() primitive functions only if the fixpoint (or the variable representing the fixpoint) is obligated or permitted. Traversal through the solution space (which is the same as interacting with the MIM) fails at a fixpoint if it is forbidden.

Given a particular state of the system, any other assertion can be proved or refuted by traversal and deduction. If an assertion evaluates to a contradiction it is said to be forbidden. If it evaluates to true, it can either be obligated or permitted. Validation of system dynamics can be carried out for certain assertions in a straightforward manner. These are enumerated above and have been categorized into validation of system liveness, consistency and safety.

3.7 Comparison with Process Calculus

Process calculus, also called the calculus of communicating systems was proposed by Milner [Mil89] for modeling communication behavior. Here, complete system dynamics are represented in terms of interaction. This calculus has also been extended in various forms, like the π-calculus for mobile systems.

It is hence a necessary question to address the need for a different paradigm like the fixpoint algebra for modeling interactive systems. The concept of fixpoints is more abstract than communication in CCS. In fact, the fixpoint algebra is not concerned with actual data that is exchanged during an interaction. However, a concept of fixpoints addresses the semantic nature of interaction spaces that aids in the *design process* of an open system. Fixpoint algebra and process calculus are contrasted in a more detailed fashion below.

Simple interaction: In process calculus, interaction is modeled as handshake protocols where state transition occurs on interaction with an environment. State transitions may also occur due to internal interactions within the machine. For example:

$$C =_{def} in(x).C'(x)$$
$$C'(x) =_{def} out(x).C$$

Such a specification denotes a simple interacting machine which accepts an input at state C and goes to C'; and gives an output at state C' and reaches C.

In the fixpoint algebra, such a machine is represented as a set of fixpoints $\{C, C'\}$ where $call(C) = C'$ and $call(C') = C$. The $call()$ function traverses the solution space from each fixpoint. It may correspond to an interaction with an outside environment, internal interaction, or maybe simply a synchronization mechanism with other process segments with no exchange of data at all. The fixpoint algebra notation hence models a *class* of interacting systems with two alternating fixpoints. In an implementation of this model, input may be read from C' and output may be provided at C. Or multiple inputs (x, y) may be read at C and maybe a single output y given out at C'. Specific implementational models may be inherited from the same abstract model $\{C, C'\}$, $call(C) = C'$; $call(C') = C$ specified above. Inheritance of interactive processes in introduced in a detailed fashion in Chapter 5.

Composition: In process calculus, interacting agents are composed to form compound agents with the operator $|$. Thus $P|Q$ represents a compound agent obtained by the combination of agents P and Q. Combination semantics are modeled by assigning the output of one agent to the input of another.

In fixpoint algebra, composition is obtained by set union of fixpoints and definition of the ordering relationship $<$. Thus if $P = \langle\{P_1, P_2\}, <_P\rangle$ and $Q = \langle\{Q_1, Q_2\}, <_Q\rangle$, then the combined system $P \cup Q = \langle\{P_1, P_2, Q_1, Q_2\}, <_{PQ}\rangle$. The redefinition of the ordering operator $<$ defines the combination semantics. In the definition of an interaction machine that also contains a relationship set R, even this set is redefined for the combined system.

Expressiveness: The expressiveness of an interacting agent is determined from the *perspective of a user* interacting with the agent. Expressiveness is based upon assertions on *observable behavior.* Equality of interacting agents are established by the *bisimilar until* assertion.

In fixpoint algebra, expressiveness of an interacting machine is established from a *designer's perspective.* It is based upon assertions on *what should happen*, *what may happen* and *what may not happen* when the interacting machine is in operation. Equality of interacting machines is established by a congruence of their solution spaces in terms of fixpoints, ordering relationship and integrity constraints.

Modeling Paradigm: Process calculus takes an *operational* approach towards modeling interaction. Interactive activity is defined based on handshake protocols and reasoning is carried out on what is observable by interaction. It is best suitable when protocols are well known and complex.

Fixpoint algebra on the other hand, takes a *semantic* approach towards modeling. It is directed towards IS design which is made up of many *semantic processes* or PSPs. The goal here is to design IS interaction based on its functionality requirements. Reasoning about a model is carried out by assertions that verify that functionality constraints are not violated. For example, whatever is obligated is done, whatever is forbidden is not done, whatever is forbidden is not summarily obligated, etc.

Part II of the thesis looks at interaction modeling from a more concrete perspective. A paradigm called "dualism" is proposed to model information systems. This is based on the tenet that interactive behavior which models IS functionality enjoys a degree of autonomy over behavior that is oriented towards maintaining structural integrity of the IS. The fixpoint algebra is used to propose a model called "dialogs" and "interaction schema" to characterize interactive IS behavior.

Part II

Modeling Interactive IS Dynamics

Putting to use the insight that interaction is richer than algorithms

Chapter 4

The Dual Nature of an IS

Freedom from the desire for an answer is essential to the understanding of a problem. – *Jiddu Krishnamurti*

Part II of the thesis is directed towards modeling interactive behavior in information systems. Interaction is addressed in a more concrete fashion and issues like specification, design complexity, maintainability and verification are addressed.

The interactive dynamics that an IS carries out with its environments are specified based on functionality requirements. Characterization of interactive behavior implies definition and maintenance of integrity of IS functionality. In this work, we address interactive IS dynamics by characterizing an "interaction space" of the IS, in the form of an "interaction schema." The interaction space is the interactive solution space of IS dynamics.

An IS is said to have a dualism of properties that concerns structural integrity and dynamic integrity. These are called the *object space* and the *interaction space* respectively. Each space is semantically distinct from the other, and we argue that separating concerns of the interaction space from that of the object space is important to avoid conflicting problems that occur in IS design.

4.1 Contemporary Approaches to Interaction Modeling

Contemporary models for interactive IS dynamics can be divided into three categories. They are respectively called: *behavioral* models, *semantic* models and *hybrid* models.

Behavioral models show how a process performs its task. They depict conditionals, parallelism, synchronization and other related properties. Their generic mathematical structure is a directed graph, where graph nodes depict tasks or activities and graph edges depict control flow or dependencies. On the other hand, a semantic model depicts the "what" of a process. They show semantic states of the process and how states are

Behavioral models	**Semantic models**	**Hybrid models**
Models the "how" of a process	Models the "what" of a process	Combines "how" and "what"

Directed graph
nodes = tasks;
edges = dependencies or flow
	Directed graph
	nodes = state;
	edges = transitions
		Directed bipartite graph
		nodes = tasks/state;
		edges = flow/transitions

Examples: Task-dependency graph, workflow, flowcharts, object interaction diagrams, etc.	**Examples:** LTS, automata, Transducers, etc.	**Examples:** Petri nets, Predicate transition nets, etc.

Table 4.1: Contemporary approaches to interaction modeling

reachable from one another. Their generic model is also a directed graph, but here graph nodes represent semantic states and edges represent state transitions.

A third category of models like Petri Nets are hybrid in the sense that they combine both states and activities. Their generic model is a directed *bipartite* graph consisting of two kinds of nodes representing tasks and states respectively.

In conceptual design of IS dynamics, it is sometimes desirable to separate control flow semantics from functionality semantics [LLMT00]. This is because, different implementational situations require different control flow structures to achieve the same semantic functionality. Functionality specifications encode policy and other behavioral constraints on the dynamics. These constraints should be preserved in analogous forms for every behavioral model of a semantic functionality. Customization of a semantic process to different behavioral structures is ubiquitous in large scale information systems like supply chain management and organizational workflows.

Most contemporary IS design paradigms have separate models that show control flow and functionality semantics. The different dynamic models provide different views onto the space of IS dynamics. Some examples are considered below:

In the Object Modeling Technique (OMT) [RBP+90] IS dynamics are divided between two models: the *functional model* and the *dynamic model*. The functional model specifies "what happens." It describes flow of data and control and constraints acting on them. The dynamic model specifies "when something happens." It identifies different events that cause control flows in the system and relationships between events.

In UML [UML], there are many different facets for representing interactive dynamics. An interaction is defined in UML as the "dynamic behavior of the message sequences exchanged among objects to accomplish a specific purpose." From this definition, interaction is represented in two forms – as a collaboration diagram and a sequence diagram. A collaboration diagram shows the static collaboration structure among domain objects for a particular interactive process. A sequence diagram depicts the dynamic unfolding of the interaction sequence in executing a process. In addition, behavior is also represented in terms of state charts and activity diagrams which denote semantic states of the actual process itself. An activity diagram represents activities as action states, the completion of which would involve a transition.

Similarly, many other approaches to IS design have multiple models that depict different views on IS dynamics. However, such approaches have a few disadvantages:

- Without a common framework that defines the boundaries of IS dynamics it is hard to reconcile between the different models;
- Dynamic models are tightly bound to domain objects. But in reality, there exists a degree of independence between IS processes and the roles and activities carried out by domain objects. The same process may need to have different specialized implementations. Process semantics in such cases have a lot of autonomy with respect to domain objects or their roles.
- With most contemporary approaches it is difficult to encode policy constraints that determine interactive dynamics, and flexibly manipulate them;
- It is difficult to identify relationships across processes and constraints that may be implicit in the face of multiple processes.

We propose a different approach towards IS design based on a conception of IS *dualism*. In this model, an IS is considered to be a collection of semantic processes or PSPs. The IS may have different structural designs and implementations which ultimately have to adhere to the functionality specifications of the PSPs. Functionality issues have a degree of independence with respect to structural issues.

The abstract space of IS functionality is called the *interaction space* of the IS. Semantic processes are usually interactive in nature. This can be either single-stream or multi-stream interaction. The interaction space of an IS is the solution space of the set of PSPs that make up the IS. In the dualism model, the interaction space of an IS is characterized by an interaction schema.

The interaction space is supported by an IS implementation that has a specific structural design. Dynamics of the IS structure is oriented towards maintaining the system state. Such dynamics are called the "database" dynamics of an IS. And the space of IS structure is called the *object space*.

The object space maintains structural integrity and the interaction space maintains dynamic (functional) integrity of the IS. Such an approach reconciles different views of

IS dynamics under a common framework of the interaction space. It also addresses and solves the shortcomings presented above. The bottom line of a dualism model may be summed up by the following equation: `IS = Database systems + Interaction`. Interaction modeling is often addressed as part of database design; and issues that relate to interactive integrity are often clubbed together with issues that relate to data integrity. The dualism model advocates separation of these two concerns. The next section introduces the philosophical underpinnings for the dualism model and it's bottomline equation.

4.2 IS = Database Systems + Interaction

An *information system* (IS) is a generic term referring to software systems that provide information-based services. The notion of an IS has been addressed from various perspectives such as managerial, technical, organizational, etc. [Alt96, Sen97]. However, even with active research communities studying IS design, the term *information system* still lacks precise formal underpinnings. Unlike for say, databases, there is no agreement on what constitutes "principles of information systems."

Any significantly advanced IS contains some kind of a database system. As a consequence, IS design addresses a number of database issues like conceptual modeling, metadata management, etc. On the other hand, any large contemporary database system is actually an IS, providing additional services beyond simply storing data and running queries and updates. As a result, the distinction between a database and an IS tends to be blurred, and in common discourse it is not clear that the principles underlining the study of information systems should be different than those for databases.

As an example, Loucopoulos [LZ92] defines information systems as "systems which are data-intensive, transaction-oriented, with a substantial element of human-computer interaction." It could well be argued that contemporary database research defines a database system in a similar fashion.

However, from a historical sense, information systems and databases began from different concerns. The term information system was coined from a managerial and organizational perspective. It addressed issues like systems analysis and design, requirements engineering, data flow design, system development life cycles, etc. Over time, many of these ideas became more formalized. Many paradigms and tools were developed to support life cycle models. In addition, other aspects of IS design like organizational models, process models, coordination and collaboration came to be treated in mathematical terms [MC94, Law97, JL96].

In contrast, database systems began by addressing problems of data management. Early database systems were built over flat file systems of data. For many years, the main set of problems addressed by database research concerned efficient storage and retrieval of data. However database systems have now progressed greatly to include a number of

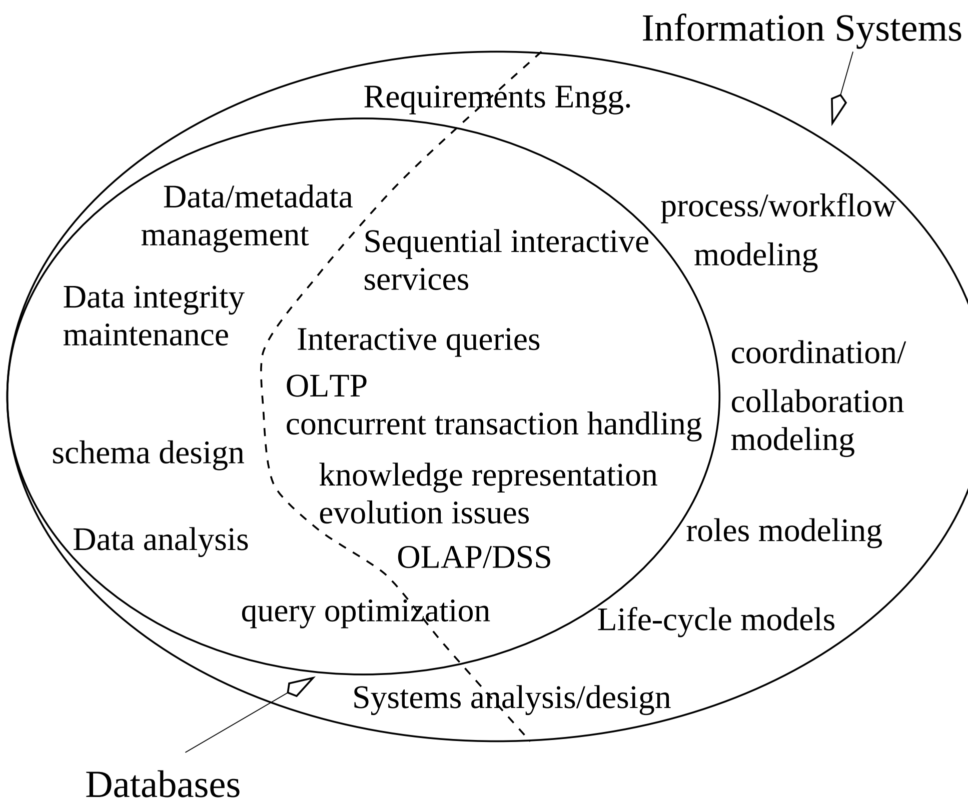


Figure 4.1: Databases vs Information Systems

other issues. Contemporary database systems are no longer just sophisticated file systems; they are actually information systems that provide different kinds of specialized services like transaction processing, data mining, analytical processing, etc.

The expansion of concerns in both information systems and databases is schematically depicted in Figure 4.1. The inner ellipse in the figure shows how database concerns have changed over the years; and the outer ellipse shows how IS concerns have changed over the years. The expansion of concerns have blurred boundaries between information systems and databases. In common discourse it is sometimes difficult to distinguish between approaches that address database design and those which address IS design.

However, at the level of foundations the outlook is very different. Research on *principles of database systems* is well-defined, at least since the middle '80s [Ull88]. It includes issues such as schema design, data modeling, query evaluation and expressiveness, and dependency preservation. On the other hand, as far as we know, there is no consensus on what are *principles of information systems*, and there are no accepted formalisms for studying them.

The distinction between a database and an IS is best appreciated when we consider their function, or "job." The "*job of a database*" is to store data and answer queries. This entails addressing issues like data models, schema design, handling distributed data, maintaining data consistency, query evaluation, etc.

On the other hand, the *job of an IS* is to provide a *service*, which entails considerations that span the life cycle of the larger system. Some examples are: a service for train reservation, a catering service, a service for calendar management, etc. Services over time represent semantic processes and determine IS functionality. They are usually interactive in nature, involving user sessions with one or more users, and are specified by models of interaction.

The concept of a *job* drives design decisions and models. The following subsections show how a change in emphasis from data management to service provision can affect design models and processes.

4.3 Statics and Dynamics

An IS is defined within an organizational context or some larger system framework. The static structure of an IS is a conceptual schema of the larger system. In order to implement the job of an IS, static structures meant to provide services have to address issues like the following: (a). the relevance of structural elements to the semantic service, (b). the comprehensiveness of the structural description, (c). the robustness of the conceptual structure in the face of changes.

On the other hand, the job of a database is to manage data and handle queries. The static structure of a database is oriented towards handling queries and updates efficiently. They address issues like the following: (a). indexing (b). normalization (c). anamoly detection and correction (d). computation of (materialized) views, etc. A database schema that is normalized for efficient queries and updates, may not always reflect the users perspective of the conceptual structure of the larger system.

A description of the "job" also illustrates the kind of dynamics that is characteristic of databases and of information systems. Database dynamics are algorithmic in nature. Answering queries are modeled by functions that return an answer to the input problem. Updates are modeled by partial functions which represent transitions from one valid system state to another in atomic steps.

The dynamics of an IS are usually interactive in nature. Services may involve interaction with one or more environments. They are driven by policy specifications governing IS functionality. Some examples of policy specifications are:

- A reservation process for a train may not be run before 60 days of the departure date;
- Payment by credit card requires an interaction session with the bank to check validity before delivery, etc.

Services have to model interaction protocols, integrity constraints on interactive dynamics and enforce policy decisions that govern interactive processes. Many rules of database dynamics like atomic transitions and isolation may be violated in IS dynamics in order to maintain integrity in providing services.

As noted in Chapter 3, interactive dynamics characterize richer domains than algorithms. Interactive behavior has its own set of concerns that are independent of database integrity concerns.

4.4 Individualization

Users of IS services play much more of a role than just supplying queries. For a database, everything that is required by the environment is assumed to be encapsulated as part of the database query. Ideally the database does nothing else other than answering queries. Hence in the absence of updates, a query evaluates to the same result every time it is called.

For information systems, user interaction is much more complicated. A user uses the IS in order to obtain a semantic functionality. There are many different aspects that need to be modeled – the most fundamental being the reconciliation of "semantic functionality" between the user and the IS. In addition, other characteristics of the user like *observations* (what the user should see), *intentions* (what the user wants) and *beliefs* (what are the implicit user knowledge that affects interaction) impact the design of an interactive service [Lew00].

Hence, tasks like user modeling and maintaining user profiles or *individualization* forms an important aspect in the design of IS dynamics. Individualization can be viewed as a projection of the IS onto the user's space, tailoring available services to user preferences and characteristics. Examples are: (a) *List the current valuation for all houses on my street* and, (b) *Supply purchase recommendations for me based on my purchase history.*

Individualization requires awareness of user characteristics and user preferences, as well as of the history of user interaction with the system. To individualize information services, an IS must be able to create and maintain user profiles with the relevant information. Some user characteristics are fixed (such as gender) while others may be reset by the user (such as their nickname). Some characteristics are never set by the user explicitly; these include simple facts (such as the user's software version number) as well as facts that are derived directly from the interaction history (such as the user's interests and patterns of behavior). The latter category of user preferences is the most interesting, from a research point of view.

Individualization of interactive services can be considered dual to the concept of views in database design. Database views are static individualized structures, while IS individualization also involves dynamics. However individualization of dynamics has some characteristic differences from computing database views. Database views contain data from the central database, but with possibly a different schematic structure. Individualized IS views may contain information about user preferences and interaction history that are not part of the larger IS. A database view is a subset of the larger database; but an individualized service may have a set of completely different or new services that are not part of the larger IS.

A simple example of individualization of services concerns how services need to be redefined for different target platforms [LS00]. The same service (for example, train reservation) may need to have very different interaction protocols on different target platforms (for example, telephone, over the counter, WWW browser, etc.) that the user is interacting on. All of them represent the same service, but may have different interaction sequences and may maintain vastly different sets of data items for carrying out the process. But a given policy decision on the semantic process affects all the different interaction protocols in the same way.

4.5 Some Models of IS Interaction

Some existing paradigms of IS modeling are reviewed in this section with emphasis on how they address the interactive nature of IS dynamics. The approaches presented is by no means comprehensive. The emphasis here is on the contrasting models in related literature that concern interactive dynamics of an IS.

4.5.1 Patterns

Patterns are a first step towards addressing the complex nature of interactive solution spaces. Since there is no characteristic structure of interactive solution spaces, patterns capture and document recurring structures of interaction. Patterns record both static structure (relationship between objects) and behavioral structure (interaction sequences and protocols). Patterns address different stages of a life cycle model like analysis patterns [Fow96], design patterns, process patterns [Cop98] and organizational patterns. Some well known patterns of behavior in information systems include the Model/View/Controller framework, the Publisher-Subscriber pattern, and the Observer pattern [GHJV95, Pre95]. The handbook of patterns by Gamma *et. al* [GHJV95] documents a number of other patterns which have been popularly used in real life design problems.

However, patterns as a paradigm for modeling interactive IS behavior lacks precise definition and notational semantics, which impedes the formalization and automatization of their usage.

4.5.2 Actors and roles

Actors [AMST92] are autonomous objects of a system, whose behavior at any given instant is determined by the role that they have adopted. This paradigm has also been used to model use cases [Fow97] and in Artificial Intelligence (AI) [AR96]. The evolution in AI from logic and search to agent-oriented models is not a tactical change, but a strategic paradigm shift to more expressive interactive models [Weg96]. IS modeling based on actors has extended to different related paradigms like mobile agents and collaborative agents [Wei99, HSG98]. Actors also have been formalized using underpinnings from process calculi and deontic logics [AMST92, ES99].

While the actor and agent paradigms provide an intuitive way of modeling IS dynamics, they are object centric in nature. That is, domain entities or actors form the building blocks of the model, and system dynamics are represented "on top" of the system of actors. An object centric approach towards IS design makes it difficult to encode integrity constraints that are characteristic of the interaction space. For example, in a system of actors that form an IS, it is difficult to difficult to flexibly manipulate policy decisions on IS services, and translate them automatically to the roles of actors.

4.5.3 Stocks and Flows

Stocks and flows is a formalized model of managerial and organizational perspectives on IS design [HR94]. In this model, a system consists of static elements called *stocks* and dynamic elements called *flows.* A system of stocks and flows interact with one or more *environments.* Stocks are called the "nouns" of the system and represent the system state, and flows are called the "verbs" of the system and represent system dynamics. Stocks are represented by variables and flows are represented by difference equations over stocks, flows and environments.

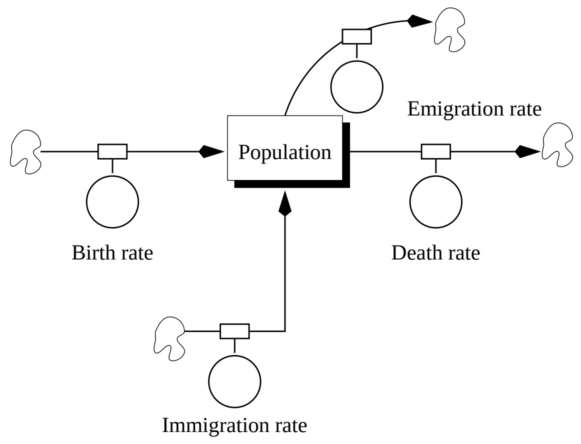


Figure 4.2: Dynamic modeling using stocks and flows

Figure 4.2 illustrates a typical stocks and flows model that shows population dynamics. Boxes represent stocks that maintain the system state, arrows represent flows or system dynamics and clouds represent environments. Control variables shown as circles with knobs affect the flows. Stocks and flows modeling provides an intuitive abstraction for IS design. However, it lacks mechanisms for abstraction, reasoning, and integrity checks on the system that are required for evaluation and verification.

4.5.4 Speech-Act Formalisms

IS modeling based on speech-act formalisms [Joh95] considers IS dynamics to be made up of interrelated events. Events are formalized using speech-act theory [Sea69]. Speech acts are classified into different types such as *assertive, commissive, directive, declarative* and *expressive*, that affect actions. The IS in a speech-act model is made up of *events* and *discourses.* Discourses are speech acts that connect events which comprise IS dynamics. Discourses also connect events with *objects* that comprise the IS statics. The logical formalisms for such a model are based on deontic logics.

Like stocks and flows, IS modeling based on speech-acts provides an intuitive mechanism to reconcile between semantic services and IS processes. However, there does not seem to be a clear separation between structural concerns and service concerns. This would hamper the manipulation of service concerns in a flexible manner.

4.5.5 Activity Schema

Liu and Meersman [Liu96] build an "activity schema," to model IS dynamics. In an activity schema, each object in the schema represents a semantic process. The approach here is based on object-oriented concepts. Activities are related by two kinds of relationships – specialization and aggregation. An activity is an encapsulation of a sequence of messages exchanged between objects to achieve a particular semantic process. They are governed by local and global constraints and pre and post conditions. Verification and reasoning about the activity schema is done using first order temporal logic.

The approach presented in our work is similar to the above; however with a number of new concepts that form the underpinnings of an activity schema. Firstly, constraints acting on an activity are made explicit and represented in terms of three deontics; secondly, an activity (an interactive process) is defined in terms of fixpoints that introduces specific semantics into activity specialization; and thirdly, translation of an activity (interaction) schema into an entity schema is done using translation rules that resolves contentions between actor autonomy and system integrity.

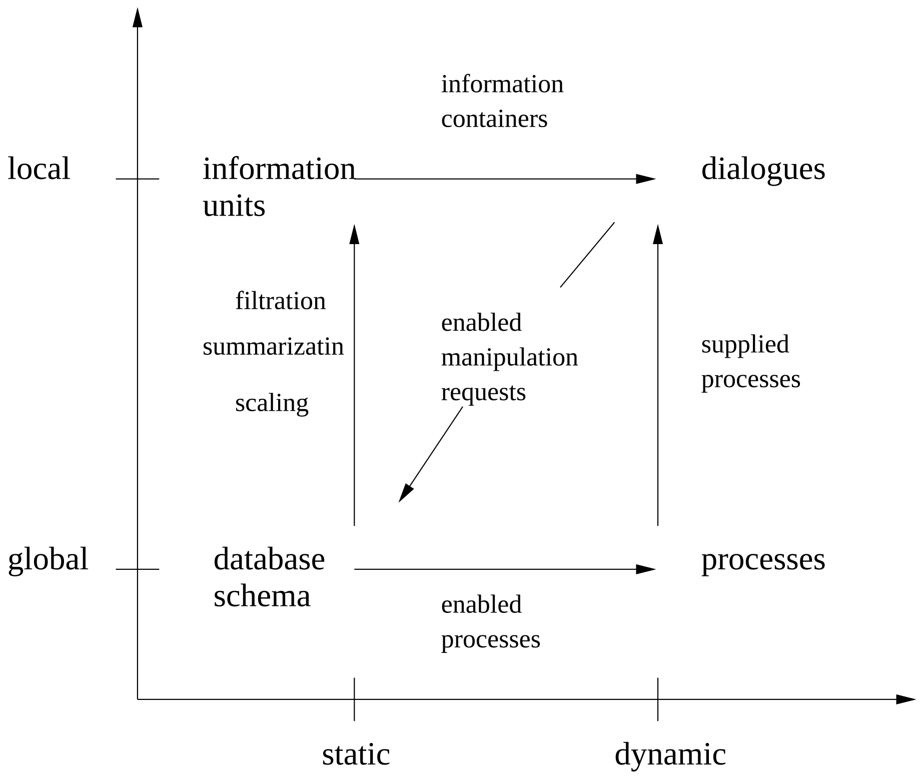


Figure 4.3: Codesign Framework

4.5.6 Codesign

Codesign [CT97] is a paradigm proposed for IS design that addresses the design of both statics and dynamics in a concurrent and autonomous fashion. Global and local (individualized) views are also defined for the dynamics, analogous to corresponding paradigms regarding the statics. Codesign addresses system design in a unified manner by handling the design of data, views, processes and human-computer interaction autonomously but inside an integrated framework [Lew00]. Figure 4.3 illustrates the overall framework of the codesign model.

Codesign divides IS design into four different aspects of concern. These are: individual-static, individual-dynamic, global-static and global-dynamic. The *individual-static* aspect concerns database views and their computation. They model different user perspectives of the IS static structure. The *global-static* aspect concerns design of the database schema and the IS conceptual model. They model IS static structure from the IS service provider's perspective. The *local-dynamic* aspect addresses human-computer interaction. It models semantic IS processes that make up the IS functionality. The *global-dynamic* aspect addresses IS dynamics from a designer's perspective. It constitutes designing the dynamics of application programs that are developed as part of the IS.

The four aspects of a codesign model are used throughout the IS life cycle and not

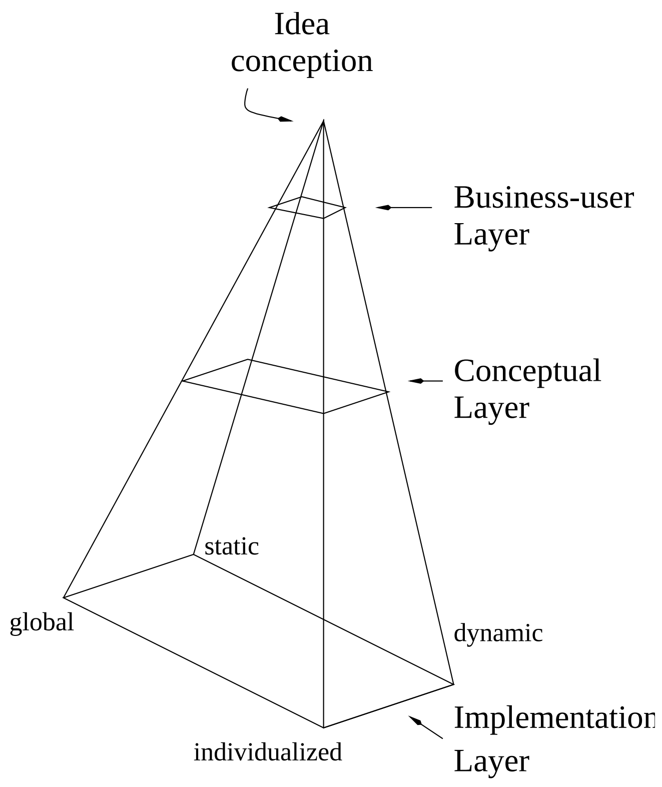


Figure 4.4: Codesign Process Framework

just during design or implementation. A process framework is defined for the codesign paradigm that incorporates these four dimensions into the design process.

Figure 4.4 illustrates the codesign process framework. The IS lifecycle is organized as a "pyramid" where the tip of the pyramid represents the idea conception that starts the entire project. The pyramid starts from the tip and finally reaches the implementation, progressively increasing in its complexity and the scope of issues that have to be addressed.

The pyramid is considered to be made up of different *layers* representing different phases of the process. The *business-user layer*, concerns requirements elicitation and design of the IS from the user perspective. The *conceptual layer* of the codesign model contains the design model of the IS as decided by the designers; and the *implementation layer* of the model addresses implementational issues of the IS. Any process model that adopts the codesign paradigm may define additional layers specific to its requirements.

Each layer in a codesign process adopts the four dimensions of the codesign model. For example, the global-static aspects of the business-user layer would specify the scope of the project, different kinds of users, the customer profile, etc. The local-dynamic aspect of the implementation layer would address the design of user interfaces in terms of HTML or Tcl-Tk, etc. The global-static aspect of the conceptual layer addresses the IS conceptual schema from the designer's perspective.

4.6 The Dualism Model

4.6.1 Dualism and codesign

The IS design model proposed in this work is part of the codesign approach. This model is called the *dualism model* and concentrates on modeling IS interactive dynamics. The dualism model is based on the following axioms:

- IS functionality is defined by semantic processes that are interactive in nature rather than algorithmic,
- Interaction is richer than algorithms and needs to be addressed separately,
- The interactive (functionality) concerns of an IS have a degree of autonomy over the structural (state maintenance) concerns, and
- Interaction space of an IS requires at least a three valued logic for characterization and reasoning.

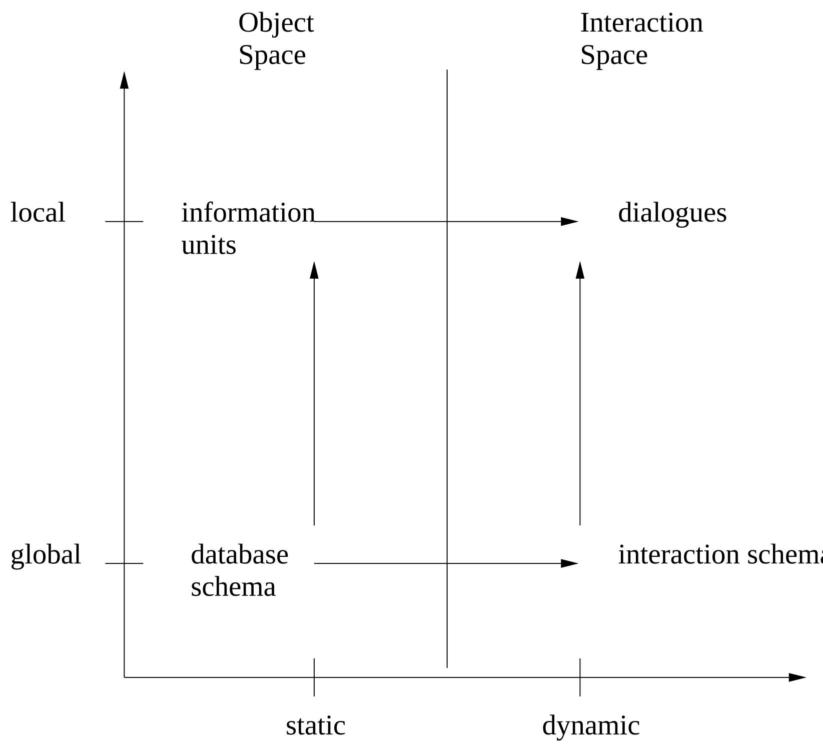


Figure 4.5: Dualism Framework

Figure 4.5 shows how the dualism model relates to the codesign framework. The codesign "square" is divided into two halves that are named the object space and the interaction space respectively. The object space concerns everything that deals with the structural aspects of the IS. The object space also has a set of associated dynamics. These are called "database" dynamics whose main function is to modify the system state.

The interaction space addresses IS functionality. Dynamics of the interaction space characterizes behavior that defines IS functionality. The local-dynamic parts of the interaction space are semantic processes of the IS based on its functionality specifications. These are called "dialogs." (A different spelling "dialogs" is used to differentiate between the dualism dialog that is based on fixpoints and a codesign dialogue). The global-dynamic aspect of the model is called an "interaction schema" that characterizes the functionality dynamics of the entire IS. An interaction schema models relationships between dialogs and integrity constraints on IS dynamics.

In the rest of the thesis, a running example of an IS meant to handle activities of a conference, is used as an example to illustrate different aspects of the dualism model. The requirements of this IS are intuitively introduced below. They will be introduced in a more formal manner in Chapter 6 when the dualism process framework is introduced.

A committee for conducting a technical conference requires an information system that can handle all activities of the conference as comprehensively as possible. This includes handling submissions of technical papers, posters and software which sets the ball rolling for the conference; to end conference activities like negotiating a committee for the next conference, announcing a preliminary call for papers, online sale of conference proceedings, etc.

There are different kinds of actors who make up the conference system like the PC chair, reviewers, authors, advisory committee, support team, etc. Each of them have different roles at different point in time. They interact with other actors and infrastructure that make up the conference system to perform semantic activities that makes up the conference functional.

The dualism designer who is asked to design an IS for such a system maintains a two-way perspective of the system throughout the lifecycle. The designer categorizes any given requirement or constraint into either a structural requirement/constraint or a functional requirement/constraint. As an example, consider the following constraints:

C0: An author may not submit a paper after the submission deadline is over.

C1: Every author entry in the list of authors should be associated with at least one paper in the list of papers.

C2: An author cannot be the reviewer of his own paper.

C3: An author is not intimated of the status of his/her paper before the announced date of intimation.

C4: A reviewer should have a minimum of one paper assigned for review and a maximum of four papers.

C5: The PC chair should send out a general call for reviewers on internet forums if all papers have not been assigned by three working days after the deadline for paper submissions.

C6: Any move towards extending the deadline for paper submissions should take place *before* the close of the current deadline.

C7: Paper submission deadline can be extended at most three times.

C8: A student delegate is entitled to attend the conference dinner for free

Constraint **C0** is a *functionality* constraint. It represents an constraint on dynamics resulting from functionality requirements. On the other hand, constraint *C1* is an integrity constraint on the system's state. It represents constraints on data items that make up the system state. The dualism designer places **C1** into the object space and **C0** into the interaction space.

In a similar sense the other constraints are assigned as follows. **C2** belongs to the object space, **C3** belongs to the interaction space, **C4** belongs to the object space, **C5, C6** and **C7** belong to the interaction space, and **C8** belongs to the object space.

The dualism designer strives to maintain a degree of autonomy between the object space and the interaction space. The effects of changes and enhancement requests are localized to the space in which they originated. To what extent concerns of the object space and the interaction space can be kept separated is a philosophical question which is addressed in more detail in the next chapter. The dualism model also incorporates the codesign process framework pyramid, where dualism of concerns is maintained in every layer of the pyramid. Different kinds of support in the form of languages and tools are proposed for the different layers of the dualism pyramid. The dualism process framework is addressed in detail in Chapter 6.

4.6.2 Dualism versus object-centrism and process-centrism

Figure 4.6 compares the dualism model with "object-centric" and "process-centric" models. Most current approaches to IS design may be categorized to be either object-centric or process-centric in nature.

In an object-centric model, the system is considered to be made up of domain entities (or objects or actors or agents). System dynamics are defined on top of the domain entities in the form of various interaction protocols or roles. Figure 4.6(a) shows an object-centric model. A UML model may be considered object-centric in nature. The building blocks

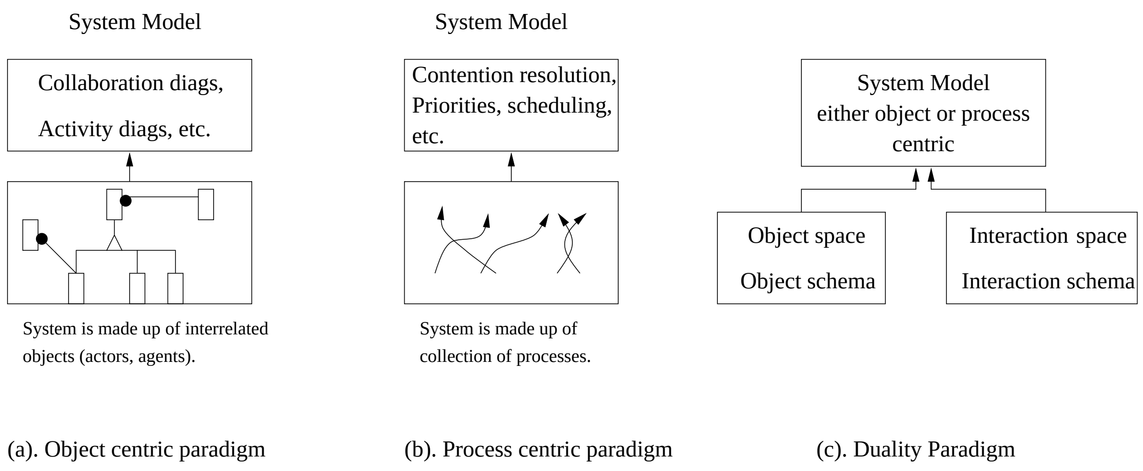

System Model
Collaboration diags,
Activity diags, etc.
System is made up of interrelated objects (actors, agents).
(a). Object centric paradigm
System Model
Contention resolution, Priorities, scheduling, etc.
System is made up of collection of processes.
(b). Process centric paradigm
System Model either object or process centric
Object space
Object schema
Interaction space
Interaction schema
(c). Duality Paradigm

Figure 4.6: Object-centric, process-centric and dualism Frameworks

Object space	**Interaction space**
Represents system structure	Represents system functionality
Depicts domain objects and relationships	Depicts semantic processes and relationships
Maintains integrity constraints on system state	Maintains integrity constraints on system dynamics
Has algorithmic dynamics that query and update system state	Has interactive dynamics that constitute semantic processes
Dynamics are ACID in nature	Atomicity and isolation no longer necessary conditions
Characterized by object schema	Characterized by interaction schema

Table 4.2: The dualism model

of the model are domain objects that make up the system's actors and infrastructure. System dynamics is defined by a variety of dynamic models like collaboration diagrams, activity diagrams, etc. on top of the domain objects.

In a process-centric paradigm, a system is considered to be made up of processes each of which achieve specific functionality. Domain entities are meant to facilitate execution of processes. They are dynamically allocated to processes based on different strategies. The characteristics of domain entities are defined based on the underlying process semantics. Figure 4.6(b) shows a process-centric model. Process centric models have been used for organizational workflows, business process engineering, job shop scheduling and other areas.

Both object-centric and process-centric approaches have their own set of advantages and limitations. These have already been addressed in Chapter 1. By dividing IS concerns into an object space and an interaction space, the dualism model also solves conflicting problems like actor autonomy vs system integrity. In addition, by formulating functional dynamics in the form of a schema maintainability of the dualism model is also high. These issues are addressed in more detail later in the thesis. Table 4.6.2 contrasts the object space and the interaction space of a dualism model.

In this work we concentrate only on the interaction space of a dualism model. The interaction space is characterized by an *interaction schema.* The next chapter introduces the building blocks of an interaction schema.

Chapter 5

Dialogs and Interaction Schema

Fundamental ideas play the most essential role in forming a physical theory. Books on physics are full of complicated mathematical formulae. But thought and ideas, not formulae, are the beginning of every physical theory. – Albert Einstein

This chapter explains the dualism model in detail. The dualism model divides an IS into an object space and an interaction space. The object space is concerned with structural properties of the IS and carries out state maintenance operations. The interaction space is the solution space of IS functionality dynamics. The two spaces are characterized using an object schema and an interaction schema respectively.

Modeling of conceptual structures and state maintenance dynamics have been extensively addressed in the field of databases. These include areas like ER modeling, database design and transaction processing. In this work we concentrate only on the interaction space. An interaction schema is used to model the interaction space. The interaction schema is based on an underlying formalism of fixpoints representing interaction and a three-valued logic for representing obligated, possible and forbidden behaviors.

5.1 Dialogs

An interaction schema is a semantic model that characterizes the interaction space. It models the "what" of the interaction space showing semantic states, state transitions and integrity constraints on states and transitions. The "how" of the semantic processes may be vastly different depending on the particular implementation. In an object-oriented model, the "how" would be represented by message sequences between domain objects. Similarly in an implementation based around an RDBMS, the "how" is represented by a series of queries and updates. The interaction schema models dynamic concerns that are independent of the behavioral dynamics. Such a separation of functional semantics from behavioral semantics is a conscious decision to maintain a degree of autonomy between the interaction space and the object space; and to separate description of IS functionality from their implementation.

For our purposes, we define the generic structure of an interaction schema in the following form:

Definition 5.1: An interaction schema is defined as $S = \langle E, R \rangle$, where S is the schema, E is a set of entity types or semantic process types in the schema, and R is a set of relationship types that relates entities of the schema. □

In some approaches, integrity constraints are introduced as a separate element that make up the conceptual schema. In our model, we consider an integrity constraint as a type of relationship.

The entity types or the types of semantic processes that make up an interaction schema called "dialogs." A dialog is a semantic process type or a PSP, that is part of the IS functionality. In contrast to a dialog, an entity type that belongs to the object space is called a "domain object." Both the terms *dialog* and *domain object* refer to classes (types) and not instances.

Definition 5.2: A dialog is an entity type that exists in the interaction space. It represents a semantic interactive process that a domain object (an object of the object space) or a group of domain objects (a subsystem) carries out with their environment. □

Figure 5.1 schematically depicts a dialog as opposed to domain objects. A domain object can be identified in any static snapshot of the problem domain even without a clear definition of the functionality of the system. Some examples of domain objects are: *Account*, *Transaction Log*, *User*, etc. in a banking setting; *Airplane*, *Control Tower*, *Runway*, etc. in an air traffic control setting; and *PC Chair*, *Author*, *Paper*, *PaperDB*, etc. in the conference IS example.

A dialog on the other hand, can be identified only from descriptions of the system's intended functionality. It may be attributed to a single domain object as in Figure 5.1(a), or to a group of domain objects (a subsystem) as in Figure 5.1(b). The "environment" of a dialog may be an other dialog within the IS; or it may be the larger environment outside of the IS.

Examples of dialogs are objects like *OpenNewCreditAccount*, *OpenNewDebitAccount* and *MoneyTransfer*, in the banking situation; or objects like *LandingProtocol*, *TakeoffProtocol* and *CruiseControlProtocol* in the air traffic control scenario; or objects like *CallforPapers*, *SubmitPaper*, *ReviewProcess*, etc. in the conference example. Each of the above dialogs involve one or more domain objects like *User*, *Teller*, *Account*; or *Airplane*, *ControlTower*; or *PC Chair*, *Author*, *PaperDB*, etc. and interact with a larger environment.

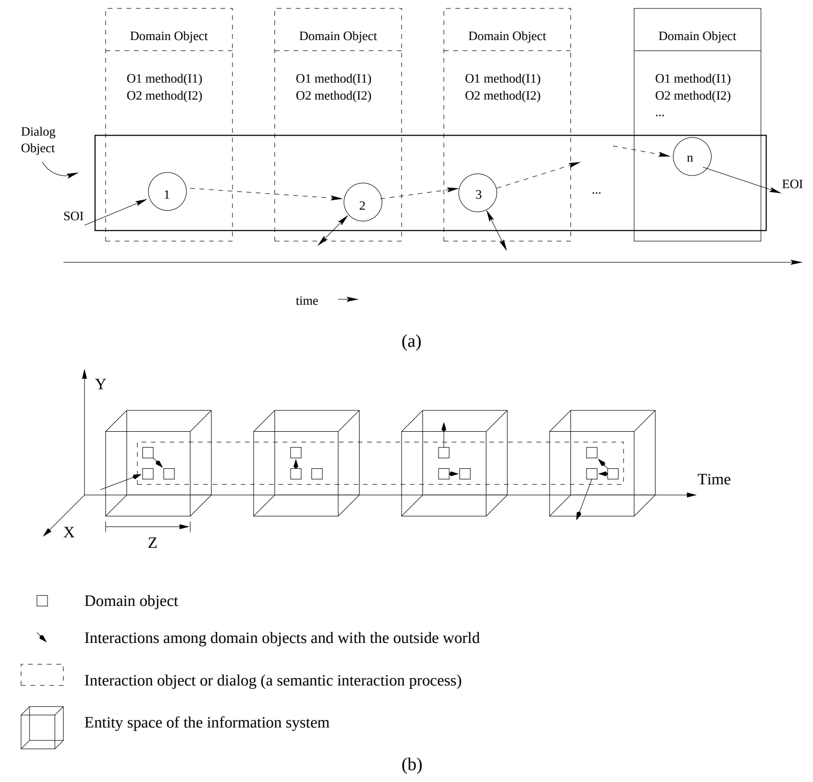

Domain Object
O1 method(I1)
O2 method(I2)
...
Dialog Object
SOI
1
2
3
n
EOI
time
(a)
Y
X
Z
Time
Domain object
Interactions among domain objects and with the outside world
Interaction object or dialog (a semantic interaction process)
Entity space of the information system
(b)

Figure 5.1: Domain objects and dialogs

A dialog is said to "represent" one or more domain objects; and the domain objects are said to have (jointly) "adopted" the dialog.

A dialog is a semantic characterization of functionality dynamics. It does not represent any specific sequence of tasks. It can be visualized as characterizing a subspace of the interaction space, representing *all* possible task sequences that can take place as part of a semantic activity. Also, a dialog does not distinguish between two or more agents (environments) which are interacting with it. It reads all its inputs from a single input port and provides all its outputs on a single output port. Hence by itself, it can model only single-stream interaction and not multi-stream interaction.

As seen in Chapter 3, single-stream interaction is modeled by a set of interaction states or *fixpoints* each of which represents a point of interaction and encapsulates computational behavior to another fixpoint. The formal model of a dialog follows a similar design and is introduced as follows:

Definition 5.3: A *dialog* is a 3-tuple $D = \langle A, M, S \rangle$, where:

- A is the set of domain objects that the dialog represents, and is called the *attribute* set of the dialog.
- M is a set of *method interfaces* for the dialog which can be invoked from the dialog's envi-

ronment.

- S is the *interaction state space*, or the set of fixpoints.

Each method $m_i \in M$ has a *behavior* in each fixpoint $s \in S$ that can take the dialog from s to any other fixpoint $s' \in S$. $m_i : S \rightarrow \mathcal{P}(S)$. □

Each dialog is modeled by a set of interaction states or fixpoints S and the set of methods are defined in each fixpoint that maps to possibly another fixpoint. Although, the definition of dialog behavior looks similar to the behavior of an object in conventional object-orientation; the definition of a dialog has some important differences. In a conventional object, the object state is defined as a function of the object's attributes; in a dialog, the interaction state is a fundamental construct of the dialog that represents computational behaviors unfolding from that point. In a conventional object, method behaviors determine the object's state; in a dialog, the interaction state determines method behaviors. In a conventional object, the behavior of a method is abstracted by method interfaces; in a dialog, the behavior of a method is abstracted by a pair (s, m), where $s \in S$ and $m \in M$, which is a combination of interaction state and method interface. Behavioral abstraction of single-stream interaction that was introduced in Chapter 3 may be recalled here. The fundamental nature of an interaction state in defining a dialog will be more apparent when dialog specialization is introduced.

Some more rules that hold for dialogs in the dualism model are as follows:

- The interaction schema is considered to represent the entire set of IS dynamics. Hence in a dualism model, any message passing among domain objects is always qualified within the context of some dialog.

- A dialog may send messages to other dialogs, but may not send messages to other domain objects that are not in it's attribute set. A dialog can communicate with another domain object only through some other dialog that represents the domain object.

Figure 5.2 shows an example declaration of a dialog from the running example of the conference IS. The dialog is called *AuthorSubmitsPaper*, and represents a semantic process in the conference IS. It describes a process by which an author may submit a paper to the conference. The dialog has four interaction states and six method interfaces. The methods are defined for each interaction state and takes the dialog to possibly another interaction state. The dialog represents four domain objects namely: WebSite, Paper, AuthorDatabase and PaperDatabase. The attribute set of the dialog indicates specific instances of the domain objects rather than the classes themselves. This is indicated by a declaration of the form `AuthorDB : AuthorDatabase` which says that the dialog applies to an instance of `AuthorDatabase` that is named `AuthorDB` inside the dialog. This means

```
Dialog AuthorSubmitsPaper

attributes // Domain objects adopting the dialog
ws : WebSite;
ps : Paper;
AuthorDB : AuthorDatabase;
PaperDB : PaperDatabase;
end attributes

states
NOLOGIN; // dialog is in this state if userid is not known
NOPAPERID; // userid is known and paper id is not known
PAPERID; // both userid and paperid are known and no paper matching paperid is in PaperDB
UPLOADED; // userid, paperid are known and there is a paper matching paperid in PaperDB
end states

methods
String login(String, String);
String setPaperID(String);
Boolean newSubmission();
Boolean upload(String, Paper);
Paper check(String);
String logout(String);
end methods

startstates
NOLOGIN, NOPAPERID;
end startstates

obligations
UPLOADED;
end obligations

prohibitions
end prohibitions

definitions

@NOLOGIN login() : {...} // checks userid and passwd and takes to either NOPAPERID or NOLOGIN
        setPaperID(), newSubmission(), upload(), check(), logout(): {} // disabled

@NOPAPERID login(), upload(), check() : {} // disabled
        setPaperID() : {...} // takes a paper id as parameter; sets paper id if it matches paper entry for userid in AuthorDB;
        destination state is either PAPERID, if no entry for paperid exists in PaperDB; or UPLOADED if paperid exists in
        PaperDB; or NOPAPERID if failure
        newSubmission() : {...} // Generates a new key for paper id and takes dialog to PAPERID
        logout() : {...} // unsets userid and changes state to NOLOGIN


@PAPERID login(), check() : {} // disabled
        setPaperID() : {...} // tries to set paper id to the new value provided as parameter. Changes state to NOPAPERID,
        PAPERID or UPLOADED.
        newSubission() : {...} // generates a new key for paper id; destination state is PAPERID.
        upload() : {...} // accepts uploaded paper and adds to PaperDB and makes entry in AuthorDB; destination state is
        UPLOADED.
        logout() : {...} // unsets userid and changes state to NOLOGIN


@UPLOADED login() : {} // disabled
        setPaperID() : {...} // tries to set paper id to the new value provided as parameter. Changes state to NOPAPERID, PA-
        PERID or UPLOADED.
        newSubission() : {...} // generates a new key for paper id; destination state is PAPERID.
        upload() : {...} // accepts uploaded paper and adds to PaperDB and makes entry in AuthorDB; destination state is UP-
        LOADED.
        check() : {...} // returns paper matching paperid for checking.
        logout() : {...} // unsets userid and changes state to NOLOGIN


end definitions

end Dialog
```

Figure 5.2: An example dialog: AuthorSubmitsPaper

that a different `AuthorSubmitsPaper` dialog would interact with a different instance of the same set of domain objects. This is analogous to class variables versus instance variables in conventional object orientation. A formal description language for dialogs called the "Dialog Description Language (DDL)" is introduced later in the thesis.

A dialog also has other declarations that determine it's correspondence to the semantic process. The dialog may begin in any interaction state that is specified in the list of "startstates" in the definition. The dialog is said to have successfully executed the semantic process if it's traversal contains at least one of the strings in the set of "obligations" of the dialog. In this example, the only obligation is the UPLOADED fixpoint. The dialog is said to have successfully executed the semantic process of paper submission if it's execution contains at least one traversal through this fixpoint. This means that there should be at least one upload as part of the process to call it a successful execution. The set of "prohibitions" denote sequences of fixpoints which indicate a failure in the execution of the semantic process. In this case there are no entries in the prohibitions declaration. An execution of `AuthorSubmitsPaper` process where there are no paper uploads cannot be termed to be successful executions; however, they also cannot be termed as "failed" executions.

The interaction state in which a dialog starts execution can be provided as an argument to its constructor function. Alternatively, the constructor function can have a decision tree which decides the starting fixpoint based on the states of the dialog's attributes.

5.2 Constrained Association

In Chapter 3, channel sensitivity in multi-stream interaction was modeled by constraint relationships that related interactive behaviors on each channel of a MIM. The constraint relationship was shown to need at least a three valued system of logic.

The same approach is used in the interaction schema. A dialog models single-stream interaction. In order to model multi-stream interaction, a relationship type called *constrained association* is introduced that specifies how the behavior of one dialog may affect the behavior of another. These are also called the integrity constraints of system dynamics.

In real world information systems semantic processes often affect the functioning of one another. However, IS models that characterize systems of multiple dynamic processes, often do not adequately address integrity constraints on system dynamics. Inter-process constraint relationships proposed in current paradigms can be categorized into either *minimalist* constraints that specify required behavior and forbids the rest; or *maximalist* constraints that specify forbidden behavior and permits the rest. A minimalist constraint tries to specify the exact sequence of activities that determine how processes affect one another. On the other hand, a maximalist constraint specifies the set of forbidden sequences of activities in a system of multiple processes in order to maintain system integrity.

In the dualism model a more general approach is adopted. Constrained association combines minimalist and maximalist constraints into a three-valued system of constraints.

Constrained association is in the form of one or more constraint equations that holds across one or more dialogs. These constraint equations are in the form of $M[head] \leftarrow body$, which is read as: if *body* holds, then *head* has a modality of M. The modality M can take on three values, namely, (a). *Obligation*, that requires *head* to hold, (b). *Permission*, that permits *head* to hold, and (c). *Prohibition* that forbids *head* to hold. The formal definition of a constrained association is as follows:

Definition 5.4: A *constrained association* between n dialogs is an *n-ary* association denoted by $R(D_1..D_n) = \langle \psi, D_1, D_2, \ldots, D_n \rangle$, where $D_1..D_n$ are n dialogs, and ψ is a set of one or more constraint relationships.

A constraint relationship $\psi_i \in \psi$ is in one of the following forms:

- $\psi_i : O(head) \leftarrow body$, which says *head* is *obligated* to hold if *body* holds;
- $\psi_i : P(head) \leftarrow body$, which says *head* is *permitted* to hold if *body* holds;
- $\psi_i : F(head) \leftarrow body$, which says *head* is *forbidden* to hold if body holds.

head and *body* are made up of predicates of the following form:

1. d : dialog (O: should exist, P: may exist, F: may not exist)
2. $d.s$: dialog state (O: set state to, P: may be in, F: may not be in)
3. $\wedge$, $\neg$, applied to the above in *body*
4. $\vee$, $\neg$, applied to the above in *head*

□

Figure 5.3(a) shows a simple example of constrained association. It depicts a constraint between two dialogs called "MovePiece" of opposing players in a board game. MovePiece specifies an interactive process by which a player moves his/her piece on the board. The dialog has two states labeled **C0** and **C1** that determine whether it is the user's turn to play or not. The constraint specifies that both users should never be in the same state simultaneously. Figure 5.3(b) depicts this relationship in a more intuitive fashion.

A constrained association between dialogs incorporates channel sensitivity into single-stream interaction. A set of dialogs and constrained associations between them can be collectively considered to be a multi-stream interactive process. In the example above, the pair of MovePiece dialogs along with the constrained association forms a single multi-stream interactive process – that of the board game. The board game can be considered to be a single process that not only interacts with the players, but is also channel-sensitive in its interaction.

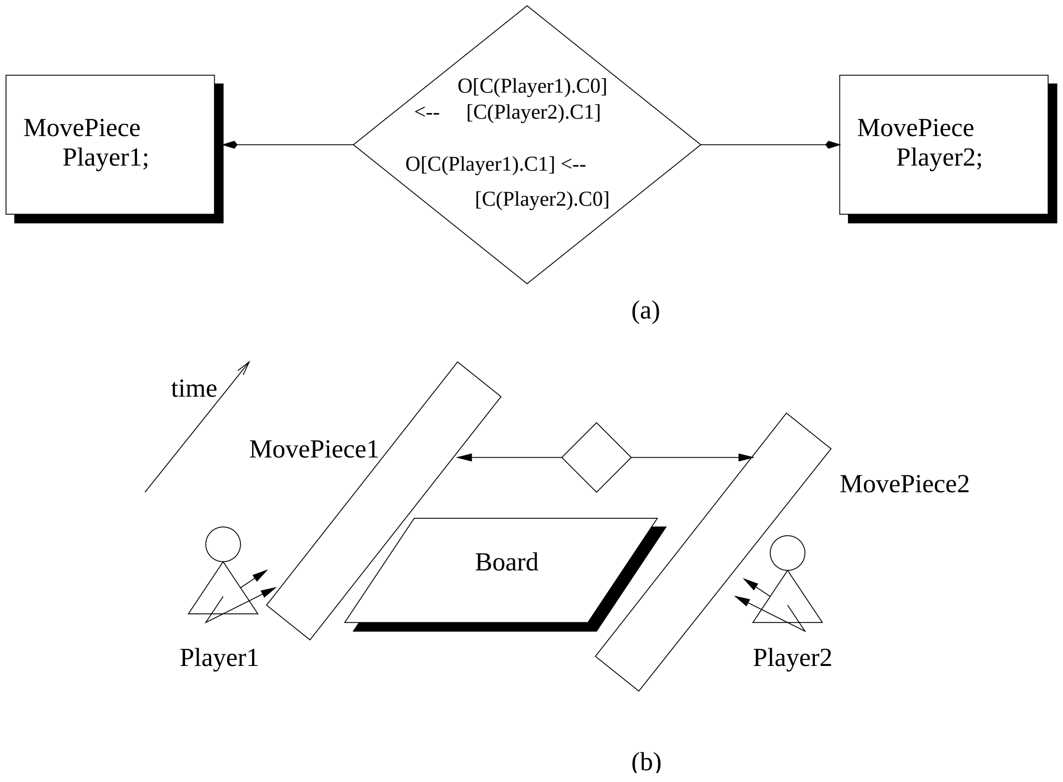


Figure 5.3: Constrained association between dialogs

Dialogs and constrained association are concerned with *types* and not *instances*. In order to represent specific instances of dialogs, constraint equations can contain free variables and inhabitation of variables into types.

Example: The MovePiece dialog in a two-player board game has to be always in pairs. It is meaningless to have a single MovePiece dialog with no complementary dialog representing the other player's move. A constrained association that relates MovePiece dialogs of opposing players can be represented as follows:

$R = \langle \psi, MovePiece \rangle$
$\psi = \{$
$O[(Player1 : MovePiece).C0] \leftarrow (Player2 : MovePiece).C1$
$O[(Player2 : MovePiece).C0] \leftarrow (Player1 : MovePiece).C1$
$\}$

In the above example, $Player1$ and $Player2$ are free variables that are inhabited into the MovePiece type. State **C0** specifies that it is the player's turn to play, and state **C1** specifies that it is the other player's turn to play. The constrained association ensures that states **C0** and **C1** are always pairwise symmetric between the two players.

In a constraint equation, a predicate that is specified over a type, applies to all

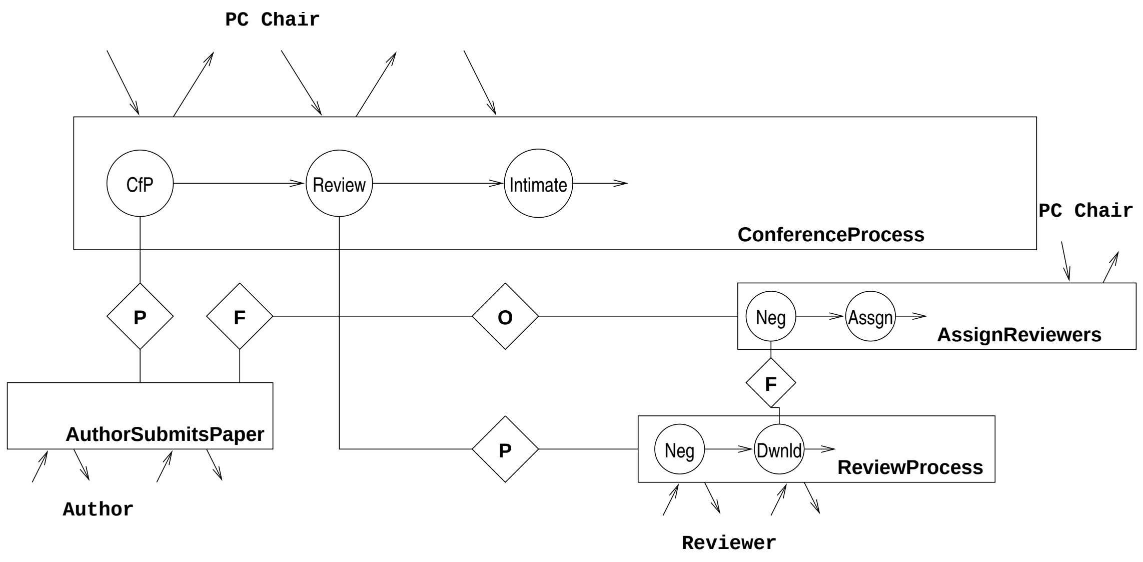


Figure 5.4: Constrained associations in the conference IS

instances of the type; while a predicate that is specified over a variable of a type, applies to a specific instance of a dialog that has been bound to the variable.

Example: Let state c of dialog D represent a critical section such that at most one dialog may be in that state at any given time. This can be represented as follows:
$CriticalSection = \langle \psi, D \rangle$
$\psi = P[(d:D).c] \leftarrow [\neg D.c]$

The constraint ψ is read as follows: when it is not the case that any dialog instance of type D is in state c, then it is permitted for some instance d to be in state c.

Constrained association can be used to coordinate between different activities of the IS at different levels of granularity. They represent coordination semantics that are characteristic of the IS rather than the objects or actors of the IS. Returning to the running example of the conference IS, some constrained associations that can coordinate activities of the IS are as follows:

$P[AuthorSubmitsPaper] \leftarrow ConferenceProcess.CfP$ (If ConferenceProcess is in state CfP, it is permitted to start an AuthorSubmitsPaper dialog)
$O[AssignReviewers] \leftarrow ConferenceProcess.Review$ (If ConferenceProcess is in state Review, it is obligated to start a dialog to assign reviewers)
$F[AuthorSubmitsPaper] \leftarrow ConferenceProcess.Review$ (If ConferenceProcess is in state Review, it is forbidden to start a AuthorSubmitsPaper dialog)

Figure 5.4 schematically depicts constrained associations in the conference IS. The entire process of the conference is modeled as a dialog called `ConferenceProcess`. The PC chair interacts with this dialog. This dialog has many fixpoints like *CfP*, *Review*, etc. Each fixpoint denotes certain behavioral properties of the conference process. Each fixpoint also has constrained associations which control other dialogs in the conference.

5.3 Dialog Specialization

Semantic processes often occur in the form of families, where many processes specialize on a single general process. For example, a semantic process like `AuthorSubmitsPaper` explained earlier, may be enacted in different ways. An author may submit a paper over the internet; or the paper may be submitted by post; or perhaps by hand. Each of the above process may have very different operational structure; but they form the same *semantic* process.

Such a scenario is not different from the generalization-specialization relationships encountered in classifying domain objects. Each of the above processes are specialized variations of the same semantic process for an author submitting a paper.

Specialization is modeled by inheritance. The specialized class inherits properties of the general class. By identifying properties that hold on a general class, the entire subdomain that the general class represents, is addressed. A good illustration of how inheritance helps bring down design complexity is shown by Haythorn [Hay94].

While inheritance is widely used to classify domain objects, inheritance in the dynamic domain has not been understood very well. There have been some approaches to introduce inheritance in workflows [Aal99]. However, it is still unclear what are the semantics of such an inheritance and how exactly can they mitigate design complexity.

The conventional notion of object inheritance is insufficient for dialog inheritance since it does not characterize interactive dynamics. The behavior of a dialog is abstracted both by state and method interface, than just method interface in conventional objects. Based on this, dialog specialization is introduced as follows:

Definition 5.5: A dialog $D' = \langle A', M', S' \rangle$ is said to be a specialization of another dialog $D = \langle A, M, S \rangle$ if:

1. $S \subseteq S'$
2. $M \subseteq M'$
3. $|A| \leq |A'|$
4. $\forall a \in A$, $\exists a' \in A'$, such that either $a = a'$ or a is a superclass of a'.

□

In a dialog inheritance, the specialized class inherits *interaction states* in addition to attributes and method interfaces from the general class. This is because, a state in an interactive process is a basic construct that encapsulates computational behavior from that point. Interaction state is necessary for behavioral abstraction of an interactive process. The specialized class may change definitions of computational behavior, and may also add more interaction states and method interfaces. It may also override behaviors from states. The specialized dialog represents the same domain objects as the general dialog. However it can represent subclasses of domain objects of the general dialog, and can add some more domain objects to its attribute set. Specialization of a dialog is said to perform two kinds of behavioral specialization:

Specialization of interactive behavior: A specialized interactive behavior is obtained by the addition of new fixpoints in the specialized dialog. A dialog $D' = \langle A', M', S' \rangle$ is said to specialize on the interactive behavior of another dialog $D = \langle A, M, S \rangle$, if $S \subset S'$.

Specialization of computational behavior: Computational behavior is specialized by overriding method definitions from fixpoints. A dialog $D' = \langle A', M', S' \rangle$ is said to specialize on the computational behavior of another dialog $D = \langle A, M, S \rangle$, if $M \subset M'$ or if $M \neq M'$.

In addition to the above, any constraints that apply to the general dialog or its fixpoints, apply correspondingly to the specialized dialog. A dialog may *not* override constraints acting on its fixpoints. Constraints are a characteristic of the IS and not completely in control of the dialog.

Example: Dialog `StudentSubmitsPaper` is a specialization of `AuthorSubmitsPaper`. It represents a semantic process for submission of a student paper. A student paper is authored completely by students and is eligible for a student award. A separate database called StudentDB keeps track of student submissions.
`StudentSubmitsPaper` is defined as $\langle A', M', S' \rangle$ where $A' = A \cup \{StudentDB\}$; $M' = M \cup \{newStudent()\}$; $S' = S \cup \{NOSTUDENTID\}$. The method definitions for `StudentSubmitsPaper` is the same as for `AuthorSubmitsPaper`, except for the following changes:

NOSTUDENTID dialog is in this state if userid is known and both paperid and student id are unknown
login(), upload(), check() : disabled
newStudent() : generates a new key for studentid and takes dialog to state NOPAPERID
setPaperID() : takes a paper id as parameter; sets paper id if it matches paper entry for userid in AuthorDB; destination state is either PAPERID, if no entry for paperid exists in PaperDB; or UPLOADED if paperid exists in PaperDB; or NOPAPERID if failure
newSubmission() : Generates a new key for paper id and takes dialog to PAPERID

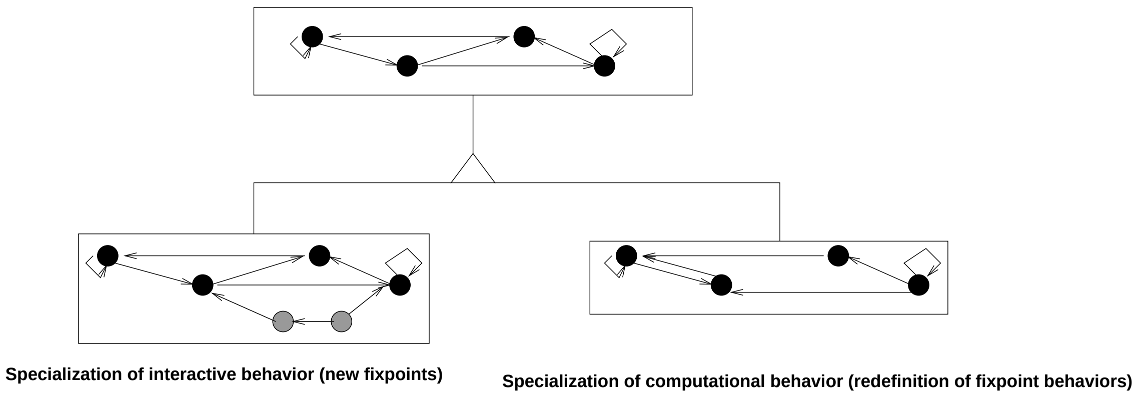

Specialization of interactive behavior (new fixpoints)
Specialization of computational behavior (redefinition of fixpoint behaviors)

Figure 5.5: Dialog Inheritance

logout() : unsets userid and changes state to NOLOGIN

NOPAPERID dialog is in this state if userid is known and paperid is unknown, and studentid is known (or is assumed non student)
login(), upload(), check(), newStudent() : disabled
setPaperID() : takes a paper id as parameter; sets paper id if it matches paper entry for userid in AuthorDB; destination state is either PAPERID, if no entry for paperid exists in PaperDB; or UPLOADED if paperid exists in PaperDB; or NOPAPERID if failure
newSubmission() : Generates a new key for paper id and takes dialog to PAPERID
logout() : unsets userid and changes state to NOLOGIN

In all other states method newStudent() is disabled.

Figure 5.5 depicts dialog inheritance in a schematic fashion. Inheritance may either add more fixpoints thus specializing interactive behavior, or may override behaviors from fixpoints thus specializing computational behavior.

Inheritance, along with constrained association can mitigate a lot of design complexity of IS dynamics. Constrained associations represent integrity constraints on system dynamics. These are constraints posed by the IS itself, and are independent of any implementation of the IS. With the use of inheritance, different levels of characterization may be obtained without having to rewrite semantic states and integrity constraints.

5.4 Interaction Schema and Maintainability

A modeling paradigm is considered beneficial based on the perspectives it provides into the system being modeled. Application domains in IS design problems are very complex which makes it difficult to precisely define what kinds of perspectives are better.

Nevertheless, different modeling paradigms are preferred for different domains because of the support they provide to mitigate design complexity specific to that domain. The dualism paradigm offers many advantages over object-centric or process-centric models. In particular:

- The dualism model provides sound logical underpinnings for the interactive behavior of an IS. The formalism based on fixpoints are used to specify integrity constraints on dynamics, behavioral specialization and also theorem proving and verification of properties which will be introduced in the next chapter.
- By separating the structural concerns from the dynamic concerns of an IS, constraints that affect one dimension need not hamper the other. For example, agent based modeling have used deontic constructs like obligation, permission, priority, etc. [SBD+00]. Behaviors of agents in such a system are determined by the deontic constructs associated with each of the agents. However, this makes it difficult to determine whether a constraint that hampers agent behavior is due to the agent or is a constraint of the IS itself. Identifying this becomes even more difficult when the number of agents in the system and interactions between agents increase.
- In process-centric modeling, specifics of the structural properties of the system are not explored in detail. This would increase design complexity when a given behavior needs to be specialized for implementation over multiple structures. Presently this is achieved by redefining behavioral processes and semantic conversion between processes. But such a conversion may not be possible between every pair of processes, and is likely to be inefficient in performance.

In addition to the above points, a large part of design complexity comes in the *maintainability* of the design. Software systems are generally employed for much longer periods than their initial estimated lifetime. The maintenance period of a software system is many times larger than the design period. During this time, maintainability of the running system is heavily dependent on how flexible and scalable is the design. Bad designs can result from any modeling paradigms. Good designs in terms of flexibility and scalability is often dependent on the designer's decisions. However, good designs also depend to a large part on the support provided by the underlying modeling paradigm.

In this regard, Haythorn [Hay94] provides an illustration how object orientation ameliorates design by providing better maintainability. Here, we consider Haythorn's example, and show that in addition to object orientation, Haythorn's model is also a dualism model. We show that dualism in the design has contributed as much towards maintainability as object orientation.

In order to create an argument for good designs, Haythorn provides an example problem of simulating the working of a bank. This consists of simulating customer arrival, and one or more tellers. The system is then designed using structured (modular) design, object-based and object-oriented designs. An object-based design was defined to be one which

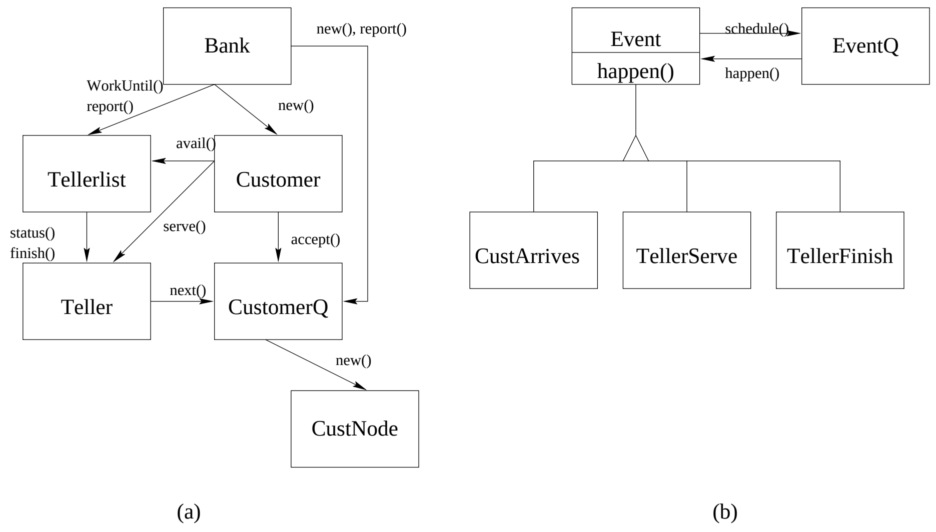


Figure 5.6: Object-based and object-oriented designs for the bank simulation problem

merely identified domain objects and interactions among them, and an object-oriented design was one which made use of inheritance and polymorphism.

The soundness of the three designs is then evaluated by enumerating a series of anticipated changes and measuring the impact of the changes on the designs. The calibration used is the percentage of code that was needed to be studied for incorporating the changes. While structured design and object-based design were shown to perform poorly, true object-oriented design was shown to reduce the impact of changes by almost half.

The essential differences between object-based and object-oriented design for the particular simulation problem are illustrated in Figure 5.6. The figure has been reproduced directly from [Hay94] which was published before standard notations for objects existed. In the figure, rectangles depict objects that have been identified, and directed lines among objects indicate associations between objects. Inheritance is depicted in Figure 5.6(b) by the triangle. Figure 5.6(a) depicts object-based design and Figure 5.6(b) depicts object-oriented design. Object-oriented design consists of polymorphic behavior by the *Event* class. There are also some domain objects in this design, which are not shown in the figure. Object-based design on the other hand, consists solely of domain objects and interactions among them.

The object-oriented design was shown to be robust in the face of the listed changes. The average amount of code that needed to be studied for incorporating a change was brought down by half in the object-oriented design relative to the object-based design. Haythorn attributed better maintainability of object-oriented design to polymorphism. But the paper also notices that the class *Event* is a "non-intuitive" class. Haythorn then proceeds to ask how to find these "great" classes in a given problem domain. A methodology is then proposed to find such "great" classes. It consisted of listing the set of all enhancement requirements that could be anticipated, and to identify commonalities in the changes, which *could* bring out the need for an abstraction like *Event*.

However, the above argument is somewhat unsatisfactory because there are no reliable means to *anticipate* in a general sense, the kinds of requirement changes that could occur. And even if a set of anticipated changes are listed, there are no general techniques by which an appropriate abstraction may be found which would encompass the changes.

Also, by looking at the class structures involving *Event* and *EventQ* in Figure 5.6(b), it is not apparent that those classes were identified based on a set of anticipated changes. In fact, the schema of classes consisting of *Event*, *EventQ* and associated subclasses actually portray an interaction schema. They represent interactions among entities of the bank, rather than any domain object per se. Haythorn's object-oriented model is actually a dualism model. An event schema is a rudimentary form of an interaction schema. An event like `TellerServesCustomer` encapsulates interactive behavior among domain objects. Sketching an event schema is now popularly used for IS design [SHF00].

We contend that, more than just polymorphism, it is also the interaction schema that has contributed towards maintainability. This is because, inheritance may also be introduced in Figure 5.6(a) for achieving polymorphic behavior in the domain objects, but without achieving the same kind of robustness arising from a class like *Event*.

In order to test our hypothesis, we separated the object-oriented design of Haythorn into an object schema and interaction schema. The object schema consisted of all domain objects like *Teller*, *Customer*, *CustomerQ*, etc., and the interaction schema consisted of classes like *Event*, *EventQ*, etc. The 8 design changes identified by Haythorn required a total of 341 lines to be modified. The paper also details how many lines in each class were affected. By categorizing these changes into object and interaction schema, we found that the interaction schema required 158 lines of change among the total of 341 lines. Also, 7 new subclasses were created of which 4 subclasses belong to the interaction schema.

As is evident here, with just polymorphism among domain objects and without an interaction schema, almost half of the changes could not have been incorporated in a straightforward manner. The changes would have been distributed across the domain objects and would have required workarounds to implement them.

This chapter introduced the main aspect of the dualism paradigm. The interactive behavior of the IS is modeled by an interaction schema. The interaction schema models IS functionality concerns independent from the implementational structure. Policy decisions on IS functionality may be embedded into the interaction schema in the form of constraints and may be flexibly manipulated. The next chapter introduces the dualism process framework. The dualism outlook is used throughout the process lifecycle. The chapter also proposes mechanisms for specification and verification of interaction spaces.

Chapter 6

The Dualism Process Framework

If the Tao is great, then the operating system is great. If the operating system is great, then the compiler is great. If the compiler is great, then the application is great. If the application is great, then the user is pleased and there is harmony in the world.
The Tao gave birth to machine language. Machine language gave birth to the assembler.
The assembler gave birth to the compiler. Now there are ten thousand languages.
Each language has its purpose, however humble. Each language expresses the Yin and Yang of software. Each language has its place within the Tao.
But do not program in COBOL if you can avoid it.
– Geoffrey James, "The Tao of Programming"

The dualism approach to IS design is based on a paradigm of reconciling between functional semantics (business logic) and operational semantics (structure and behavior), all the while maintaining a degree of autonomy between the two issues. The dualism perspective of an IS is not limited to conceptual modeling alone, but is important throughout the life cycle of an IS. In this work we also propose a process framework for the dualism model. This is based on the codesign "pyramid" framework. Dualism in IS concerns is addressed along three levels in the framework – specification, conceptual modeling and implementation. Three languages are proposed: Dialog Specification Language (DSL) for the specification layer, Dialog Description Language (DDL) for the conceptual layer, and the Dialog Presentation Framework (DPF) for the implementational layer.

6.1 The Dualism Pyramid

The dualism process framework is shown in Figure 6.1. The framework is the same pyramid as of codesign, introduced in Chapter 4. The figure depicts the pyramid plan – as seen from above.

An IS project begins with the idea conception at the top of the pyramid. The idea grows in scope until it is finally implemented at the base of the pyramid. The process framework

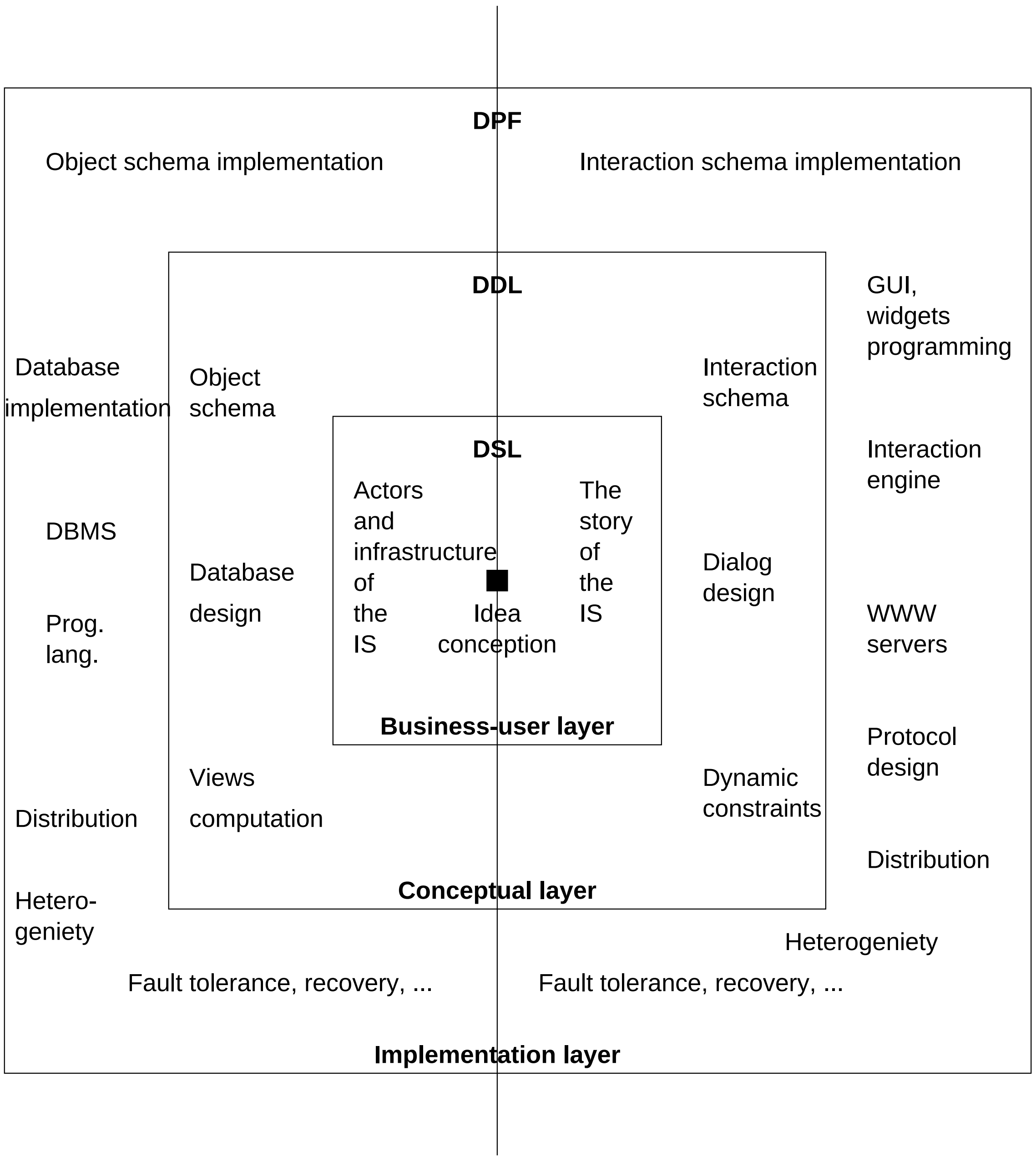


Figure 6.1: The dualism process framework

proposes three layers of concern between idea conception and implementation. These are respectively called: *Business-user layer*, *Conceptual layer* and the *Implementation layer*. Since this a process *framework*, there are no definite rules as to how these layers are interlinked. They might be cascaded like in a waterfall process model or may be executed in an iterative fashion as in a spiral model (c.f. [Som00] for process models). Any particular project may also define other intermediate layers.

Business-user layer: The business-user layer is the specification layer of the IS. The perspective of the IS in this layer is through the eyes of the business users. This layer is a direct expansion of idea conception. This layer is addressed by the system analyst. The role of the analyst in this layer is to capture and formalize user requirements in a comprehensive fashion.

IS dynamics in this layer is specified in the form of a “story” of the IS. The IS story specifies different “actors” and “infrastructure” that make up the IS. In addition, the IS story consists of many scenarios that describe different aspects of the story.

In the dualism paradigm, the IS story belongs to the interaction space and actors and infrastructure of the IS are documented in the object space. At this layer, the interaction space is called the “story space” of the IS.

Definition 6.1: The *story space* in a dualism model is the set of specifications that describe interactive behavior of the IS. □

Conceptual layer: Conceptual layer of the dualism pyramid is in the IS designer’s realm. It describes the conceptual model of the IS that is consistent with the IS story. This layer consists of an object schema and an interaction schema. The object schema describes the domain of actors and infrastructure. The interaction schema encodes the IS story into dialogs, and associations between dialogs. Dialogs model interaction states and constraints across interaction states. Processes that make up the IS story are translated into roles of domain objects and interaction sequences among them.

The interaction schema in the conceptual layer is described by a language called the dialog description language (DDL).

Implementation layer: The implementation layer of the dualism pyramid is in the realm of the developer. This layer consists of the implemented system that has to be consistent with the conceptual model. Implementation is concerned with a number of practical constraints arising from distribution, heterogeneity, crashes, etc.

In addition to the above, modalities like obligation, permission and prohibition have to be translated into corresponding implementational paradigms. Depending on the implementational paradigm, these modalities may mean different things. For example, in a networked environment, an obligation for a dialog may be implemented as an email message sent to the concerned actor to start a particular activity. In a GUI environment,

Business-user layer	**Conceptual layer**	**Implementation layer**
Specification	Conceptual modeling	Implementation
System analysts concern	System designers concern	Developers concern
IS story	Interaction schema	IS implementation
Actors, infrastructure, scenarios	Dialogs, deontic constraints	objects, messages, etc.

Table 6.1: Three layers of a dualism pyramid

an obligation may result in a popup window that prompts the user towards the specified interactive activity. Similarly, in a service environment like a hospital, an obligation may be displayed on a public terminal or a voice message.

Similarly, the presentation component for dialogs may vary considerably depending on the implementation. The IS is accessed by users over different target platforms. Depending on what kind of support is provided by these target platforms, the same dialog may have to be implemented using different sequences of activities.

Implementational layer is hence the largest layer of the pyramid. The scope of issues it has to address is much larger than either the conceptual layer or the business-user layer. The interaction space of the IS in the implementational layer is called the "system space." The three spaces are summarized in Table 6.1.

Definition 6.2: The *system space* of an IS in a dualism model is the set of all interactive behaviors that is performed by any given implementation of the IS. □

6.2 Hazards from Translation

The complexity of interactive dynamics comes from their three-valued system of description. An interactive process is not just described by liveness properties, but also by a set of *possible* behavioral properties. Any consistent instance of an interactive process displays it's set of obligated properties and zero or more of the permitted properties. Emergent behavior that is characteristic of open systems may be attributed to a large space of it's possible behaviors as against it's liveness behaviors. The space of possible behaviors may also result in inconsistent or unsafe behaviors if their correspondence to interaction specification is not exact.

To identify such modeling hazards, the dualism paradigm addresses interaction space from the three layers separately. The interaction schema at the conceptual layer must be consistent with the IS story at the business-user layer; and the IS implementation in the system space should be consistent with the interaction schema. Because of the three-valued nature of interactive spaces, it is possible to encounter different kinds of hazards while translating between the three spaces. These hazards are explored in this section.

6.2.1 The ideal translation

The following notation is used to represent the three modalities (obligation, permission and prohibition) from the three layers (business-user layer, conceptual layer, implementation layer): S for the business-user layer or story space, C for the conceptual layer or the interaction space, and I for the implementation layer or the system space. Thus O_S, P_S and F_S denote the three modalities for the business-user layer; O_C, P_C and F_C denote modalities of the conceptual layer; and, O_I, P_I and F_I denote modalities of the implementation layer.

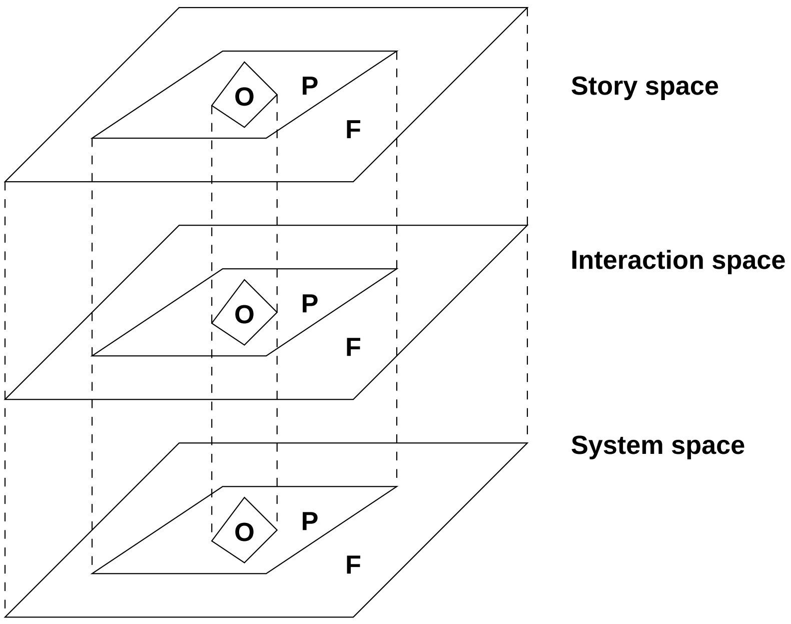


Figure 6.2: Ideal alignment of the three spaces

Figure 6.2 shows how the three spaces should be ideally aligned. In an ideal situation $O_S = O_C = O_I$; $P_S = P_C = P_I$ and $F_S = F_C = F_I$.

Specification paradigms at the business-user layer usually specifies only obligated behavior. The set of possible behaviors of the IS is implicit in the specifications. They have to be derived from generalizations over user specifications. Whatever is not specified and cannot be implied from generalizations, may be considered forbidden. In an ideal design and implementation, the set of all obligations in the interaction schema corresponds to the set of all specified behavior; the set of all permitted behavior in the interaction schema corresponds to the set of all generalizations made from the story space. The set of all prohibitions would correspond to the unspecified behaviors in the story space. Similarly, whatever is necessarily done by an implementation engine corresponds to the set of all obligations in the interaction schema; whatever can be done by an implementation corresponds to the set of all permissions; and whatever cannot be done corresponds to prohibited behavior.

Real life IS designs do not have such ideal mappings. In even fairly trivial implementations, discrepancies arise in translating specification to implementation. The nature of these discrepancies and what do they mean are explored in more detail below.

6.2.2 Discrepancies between O_S and O_C

If $O_C \subset O_S$, then the conceptual model of the IS is said to be *incomplete.* The IS does not perform all obligated behavior specified by the user. If $O_S \subset O_C$, then the IS model is said to be *malicious.* It (obligatorily) performs semantical activities that have not been specified by the user.

Example: Consider the running example of the conference IS. Let there be a specified obligation that the PC chair should start assigning papers to reviewers once the deadline for paper submission is over. This would be manifested in the interaction schema as a constraint of the form $O[AssignReviewers] \leftarrow [ConferenceProcess.Review]$. Refer to Figure 5.4 for a pictorial representation of a part of the interaction schema. `AssignReviewers` is the dialog that represents the assignment process and `ConferenceProcess` represents the overall conference process where *Review* is a fixpoint representing IS behavior after the paper submission is closed. If the design does not have this constraint it is said to be *incomplete.*

In the `AuthorSubmitsPaper` dialog the only fixpoint that is obligated is the $UPLOADED$ fixpoint. Every instance of the dialog should result in at least one upload. It is not obligated for the author to login at this dialog, if the author has already logged into the system. If the conceptual layer includes the $LOGIN$ fixpoint as the set of obligated strings, it is said to be *malicious.* It is obligating something that has not been specified as an obligation. □

If $(O_S - O_C) \subset P_C$, then the IS is said to be *conceptually inefficient.* There are some specified activities that are not incorporated by the IS model, but it is possible to perform them anyway. If $(O_C - O_S) \subset P_S$, then the IS is said to be *conceptually overloaded.* It obligatorily performs some activities that are only implied from generalizations over the user specifications not explicitly specified.

Example: In the conference IS, if the interaction schema contains a constraint $P[AssignReviewers] \leftarrow [ConferenceProcess.Review]$ instead of $O[AssignReviewers] \leftarrow [ConferenceProcess.Review]$, the interaction schema is *conceptually inefficient.* When the `ConferenceProcess` reaches the *Review* fixpoint, it is permitted, but not required for the PC chair to start the `AssignReviewers` dialog. However, the design is not incorrect since there is nothing to prevent starting the `AssignReviewers` dialog at the *Review* fixpoint.

Similarly, the above case where the fixpoint *LOGIN* was included in the success string represents a *conceptually overloaded* design aspect. User login is permitted by the design although it is not obligated for the dialog to succeed. □

If $(O_S - O_C) \subset F_C$, then the IS is said to be *incorrect and incomplete.* The IS design prevents the IS from performing certain obligated behavior. If $(O_C - O_S) \subset F_S$, then the IS is said to be *incorrect and malicious.* The IS necessarily performs some activities that are forbidden by the specification.

Example: In the conference IS, if no constraint of either of the following forms $O[AssignReviewers] \leftarrow [ConferenceProcess.Review]$ or $P[AssignReviewers] \leftarrow [ConferenceProcess.Review]$ exist in the interaction schema the dialog `AssignReviewers` evaluates to **F** (contradiction = prohibition). This forbids the chair from starting the assignment process after the paper submission process deadline. The design is said to be incorrect and incomplete.

Let the dialog `AssignReviewers` have a fixpoint called negotiate which is defined as follows: $AssignReviewers.negotiate(negiotiate\ with\ reviewers\ for\ assigning\ papers) \rightarrow AssignPapers$. The negotiate fixpoint represents a process where the PC chair negotiates with reviewers to assign papers to them. After the negotiation process is over, `AssignReviewers` moves to fixpoint *AssignPapers*, where reviewers can download papers assigned to them. Another dialog called `Review` represents the review process as seen from the reviewer's end. It consists of the following fixpoints: *Negotiate* where the reviewer negotiates with the PC chair for papers, *Download* where the reviewer downloads papers for review, and *UploadReview*, where the reviewer uploads his review.

The following constraints hold between `AssignReviewers` and `ReviewProcess`: $F[ReviewProcess.Download] \leftarrow [AssignReviewers.Negotiate]$, and $P[ReviewProcess.Download] \leftarrow [AssignReviewers.AssignPapers]$. This prevents the reviewer from downloading a paper prematurely, and allows downloads after the negotiation process is over. In such a case, if the designer mistakenly also includes a constraint of the form $O[ReviewProcess.Download] \leftarrow [ConferenceProcess.Review]$, the design is incorrect and malicious. It obligates the reviewer to start his paper download even before `AssignReviewers` has reached it's *AssignPapers* fixpoint. □

A summary of the hazards arising from discrepancies between O_S and O_C are shown in Table 6.2.2.

Hazard	**Discrepancy**
Incomplete	$O_C \subset O_S$
Malicious	$O_S \subset O_C$
Conceptually inefficient	$(O_S - O_C) \subset P_C$
Conceptually overloaded	$(O_C - O_S) \subset P_S$
Incorrect and incomplete	$(O_S - O_C) \subset F_C$
Incorrect and malicious	$(O_C - O_S) \subset F_S$

Table 6.2: Discrepancies between O_S and O_C

6.2.3 Discrepancies between P_S and P_C

There are only two kinds of hazards resulting from discrepancies between P_S and P_C. These are as follows: if $P_S \subset P_C$ or $(P_C - P_S) \subset F_S$, then the IS design is said to be *unsafe*. It is possible to perform certain interactive activities from the IS which are forbidden by the specification. Note that since $O \subseteq P$ and $P \cap F = \phi$, any discrepancies between P_S and P_C will always lie in the forbidden region.

If $P_C \subset P_S$ or $(P_S - P_C) \subset F_C$, then the IS design is said to be *conceptually inefficient*. The design has not taken into consideration implicit generalizations that could be made from user specifications.

Example: In the conference IS, once the call for papers is closed and the review process has begun, it is forbidden to receive any more submissions from authors. In the interaction schema this is modeled by having the following constraints: $P[AuthorSubmitsPaper] \leftarrow [ConferenceProcess.CfP]$ that permits an author to submit a paper when the call for papers is still open, and $F[AuthorSubmitsPaper] \leftarrow [ConferenceProcess.Review]$ that forbids the author to submit a paper once the call for papers is closed. However, if the latter constraint is missing from the interaction schema, it would permit an operation (submission of papers) that is forbidden, making the conceptual model unsafe.

In the `AuthorSubmitsPaper` dialog, the semantic process is said to have succeeded if there is at least one upload as part of the process. This is modeled by including the $UPLOADED$ fixpoint in the set of success strings. On the other hand, an author may login into the conference IS to submit a paper and decide to logout without submitting a paper. This is a perfectly permitted operation. Although it does not constitute a successful execution of the process, it also does not mean a failed execution of the process. However, if the interaction schema specifies strings not containing the $UPLOADED$ fixpoint as the set of failure strings of the dialog, it is conceptually inefficient. The interaction schema places unnecessary prohibitions on it's interactive processes. $\square$

Table 6.2.3 summarizes the hazards resulting from discrepancies between P_S and P_C. Figure 6.3 pictorially depicts translation hazards between story and interaction spaces.

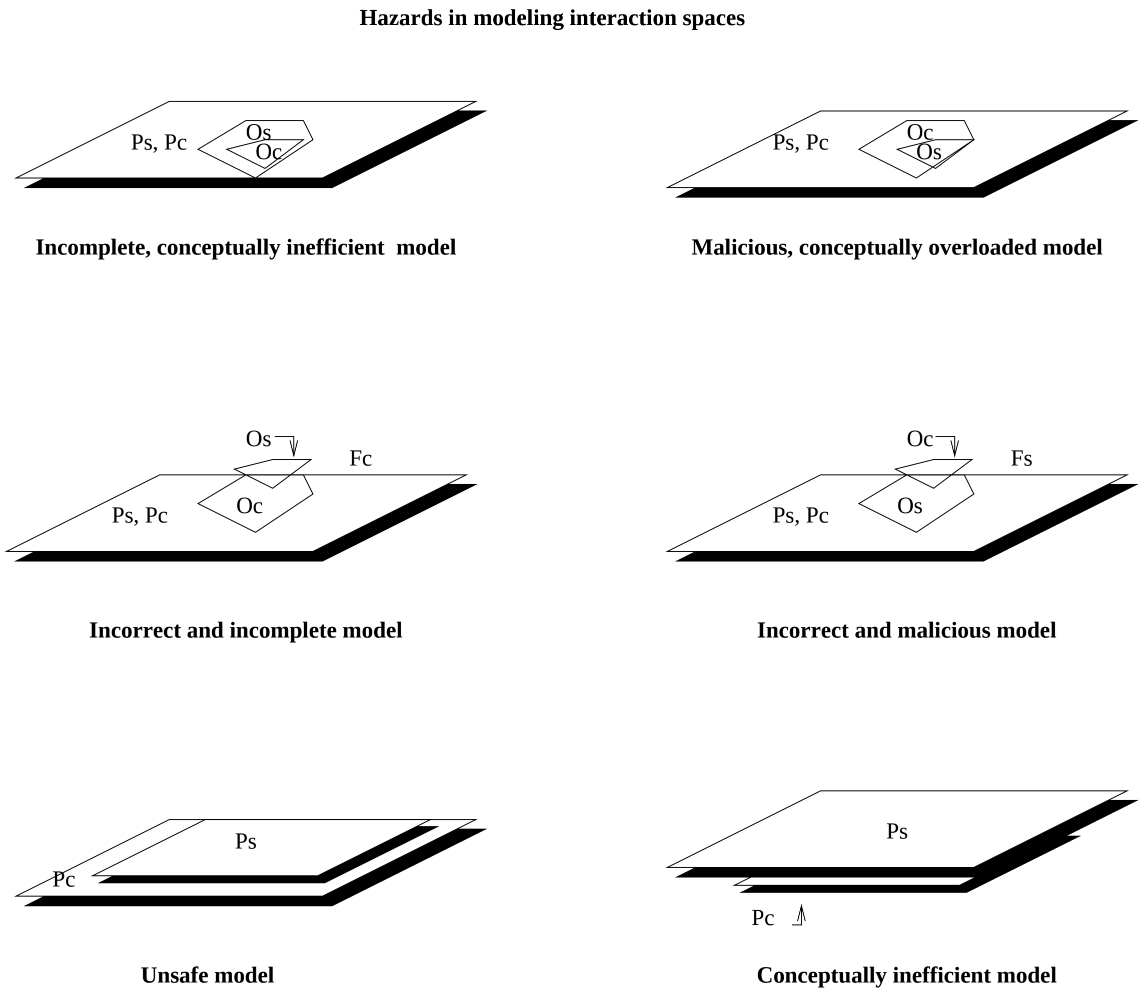

Hazards in modeling interaction spaces
Ps, Pc
Os
Oc
Incomplete, conceptually inefficient model
Ps, Pc
Oc
Os
Malicious, conceptually overloaded model
Os
Fc
Ps, Pc
Oc
Incorrect and incomplete model
Oc
Fs
Ps, Pc
Os
Incorrect and malicious model
Ps
Pc
Unsafe model
Ps
Pc
Conceptually inefficient model

Figure 6.3: Translation hazards between story and interaction spaces

Hazard	Discrepancy
Unsafe	$P_S \subset P_C$, $(P_C - P_S) \subset F_S$
Conceptually inefficient	$P_C \subset P_S$, $(P_S - P_C) \subset F_C$

Table 6.3: Discrepancies between P_S and P_C

6.2.4 Discrepancies between O_C and O_I

Hazards resulting from discrepancies between O_C and O_I are analogous to those resulting from discrepancies between O_S and O_C.

If $O_I \subset O_C$, the implementation is said to be *unreliable*. It does not perform some obligated activities in the IS design, and so may fail at a crucial time. If $O_C \subset O_I$, then the implementation is said to be *malicious*. It performs activities that are not part of the IS design.

If $(O_C - O_I) \subset P_I$, then the IS implementation is said to be *unreliable, but safe*. The IS implementation does not perform some obligated activities, but it is possible to do them with the current implementation. If $(O_I - O_C) \subset P_C$, then the implementation is said to be *overloaded and inefficient*. It necessarily performs activities that it need not perform.

If $(O_C - O_I) \subset F_I$, then the IS implementation is said to be *unreliable and unsafe*. The implemented IS may fail at a crucial stage by failing to perform obligated activities, and with no alternative means to perform the obligated activities. If $(O_I - O_C) \subset F_C$, then the IS implementation is said to be *malicious and unsafe*. The implementation performs activities that are forbidden by the design.

Example: Modalities like O, P and F have different kinds of implementation depending on the implementation context. For example, in the conference IS, an obligation for the PC chair to start the `AssignReviewers` dialog may be manifested by sending an email to the PC chair to begin the dialog. Suppose in an implementation the email is not sent but the `AssignReviewers` dialog is simply permitted to run when `ConferenceProcess` reaches the state *Review*, the implementation is said to be unreliable, but safe. The IS cannot be relied upon to let the PC chair know his obligations, but the PC chair can perform them anyway.

On the other hand, let this obligation to start the `AssignReviewers` dialog be implemented by sending an email to the PC chair that contains an access code. The access code is required to start the dialog without which it is forbidden to run the dialog. In such a case, if there is a bug in the sending of the email, the implementation would be unreliable and unsafe. It does not perform it's obligations, and nor does it allow the actors of the IS to perform the required obligatory behavior.

Let the conference IS be implemented as a website powered by CGI or PHP scripts.

Hazard	**Discrepancy**
Unreliable	$O_I \subset O_C$
Malicious	$O_C \subset O_I$
Unreliable, but safe	$(O_C - O_I) \subset P_I$
Overloaded and inefficient	$(O_I - O_C) \subset P_C$
Unreliable and unsafe	$(O_C - O_I) \subset F_I$
Malicious and unsafe	$(O_I - O_C) \subset F_C$

Table 6.4: Discrepancies between O_C and O_I

When an author visits the website, he is presented with a front page and a number of links like the conference venue, call for papers, registration, etc. The author may visit any of these links, and some of the links require the author to login with a username and password. In order to facilitate the author to do this, the front page also contains a form for username and password that allows the author to login. It is not obligated for the author to login whenever visiting the web site. But if the implementation does not do this and instead shows a login page as the front page of the website, it is in effect obligating the user to login every time when visiting the web pages. Such an implementation is said to be overloaded and inefficient; it is obligating a task that is only permitted. □

Hazards resulting from discrepancies between O_C and O_I are summarized in Table 6.2.4.

6.2.5 Discrepancies between P_C and P_I

Similar to the hazards between P_S and P_C, there are only two kinds of hazards resulting from discrepancies between P_C and P_I.

If $P_C \subset P_I$ or $(P_I - P_C) \subset F_C$, then the implementation is said to be *unsafe*. It is possible to do certain forbidden activities by interacting with the IS implementation. If $P_I \subset P_C$ or $(P_C - P_I) \subset F_I$, then the implementation is said to be a *no-frills (but safe) implementation*. The IS implementation does not support added features and activities that may be performed as per the IS design.

Example: In the conference IS, it is forbidden for authors to submit papers after the call for papers is closed. This is modeled by the constraint $F[AuthorSubmitsPaper] \leftarrow [ConferenceProcess.Review]$ in the interaction schema. In the implementation of the IS as a website, this can be implemented by disabling the hyperlink from the main page to the paper submission page after the call for papers is closed. If this is not done, then the implementation is said to be unsafe. It permits operations that are forbidden.

Similarly, the interaction schema allows a reviewer to start a `ReviewProcess` dialog right after the `ConferenceProcess` has reached the *Review* fixpoint. However, the reviewer is not allowed to download papers until the negotiation process is over in the

Hazard	Discrepancy
Unsafe	$P_C \subset P_I$, $(P_I - P_C) \subset F_C$
No-frills implementation (safe)	$P_I \subset P_C$, $(P_C - P_I) \subset F_I$

Table 6.5: Discrepancies between P_C and P_I

`AssignReviewers` dialog. This can be implemented by enabling the link that allows reviewers to check on the submitted papers right after the call for papers is closed. But if the implementation has this link disabled until the negotiation is over, the implementation is said to be no-frills but safe. It forbids operations which are permitted but not obligated. $\square$

Hazards resulting from discrepancies between P_C and P_I are summarized in Table 6.2.5. Figure 6.4 pictorially depicts translation hazards between interaction and system spaces.

6.3 The Story Space

6.3.1 Structure of the story space

The IS story specifies dynamic properties of the IS. There are many specification languages that have been proposed in IS literature. Hence we do not concentrate on developing a detailed specification language. The endeavor in the dualism framework is to show how to divide a story space into three sub spaces depicting obligations, permissions and prohibitions respectively. Obligated space represents behavior which have been specified explicitly. Permitted space consists of implications and generalizations made from the specifications. Prohibited space consists of everything that is not obligated or permitted.

For specifying IS dynamics, we propose a specification language called the Dialog Specification Language or DSL. Figure 6.5 shows the logical structure of a DSL story. In the figure, the following convention is used to represent the DSL grammar. Non terminal symbols are enclosed within angular brackets < and >. The definition of a non terminal is denoted by the symbol ::=. Terminal symbols are denoted without any surrounding structure. A domain whose terminal symbols may be too long to list is specified within parenthesis, for example (*set of integers*). When parenthesis is part of the syntax, it is preceded by a backslash. Control characters are indicated with a backslash and a following letter. For example, `\n` represents the newline character and `\s` represents a white space.

The IS story consists of a set of *actors* of the IS, who use a set of *infrastructure* that make up the IS. The story is organized as a set of *scenarios*. Each scenario is enacted by a set of actors, who use one or more of the IS infrastructure. Scenario dynamics are

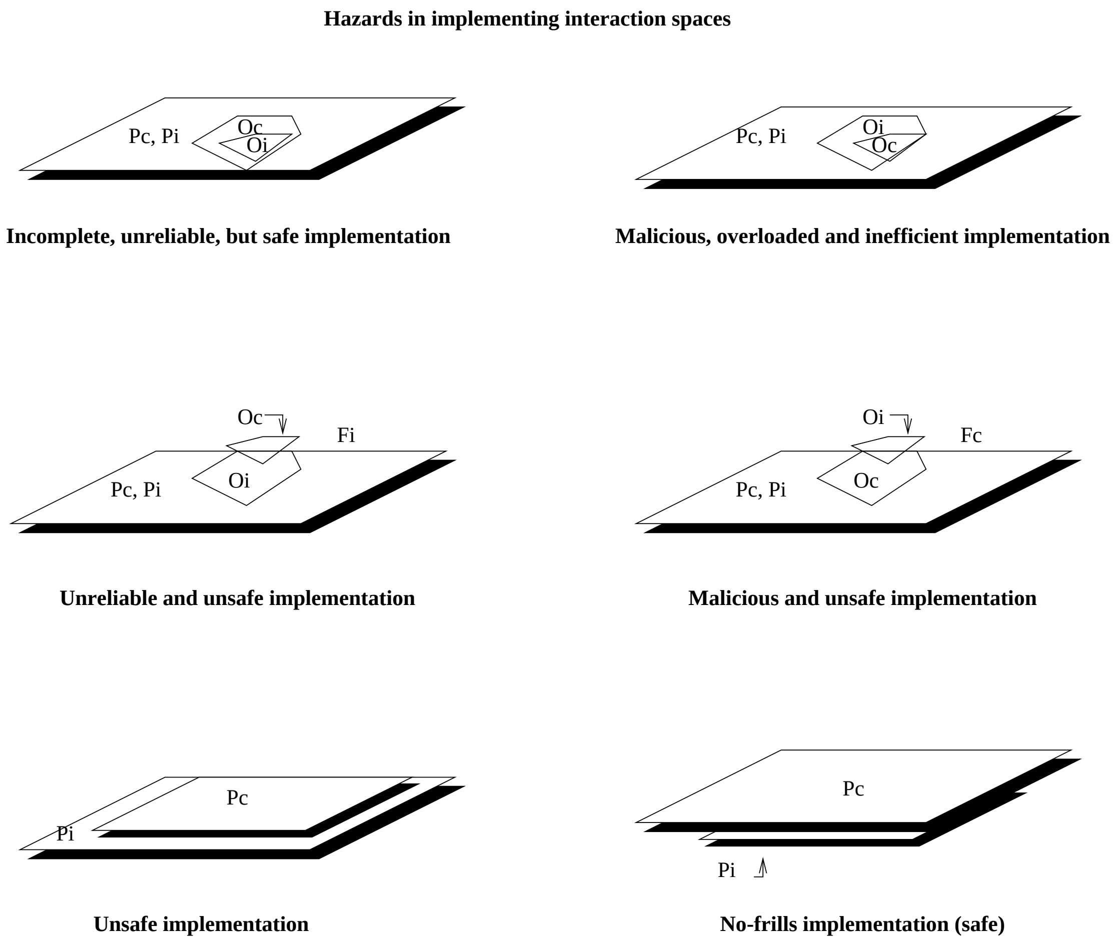

Hazards in implementing interaction spaces
Pc, Pi
Oc
Oi
Incomplete, unreliable, but safe implementation
Pc, Pi
Oi
Oc
Malicious, overloaded and inefficient implementation
Oc
Fi
Pc, Pi
Oi
Unreliable and unsafe implementation
Oi
Fc
Pc, Pi
Oc
Malicious and unsafe implementation
Pc
Pi
Unsafe implementation
Pc
Pi
No-frills implementation (safe)

Figure 6.4: Translation hazards between interaction and system spaces

```
<Story> ::= start <name> <delim> <actors> <infrastructure> <scenarios>
                                          endstory <delim>

<actors> ::= actors <delim> <actor_description> end actors <delim>
<actor_description> ::= <actorname> <delim> (textual description)
                         endactor <delim> <actor_description> | (null)
<actorname> ::= <name>

<infrastructure> ::= infrastructure <delim> <infrastructure_description>
                                               end infrastructure <delim>
<infrastructure_description> ::= <infrastructurename> <delim>
                   (textual description)  end <infrastructurename>
                            <delim> <infrastructure_description>  |
                                               (null)
<infrastructurename> ::= <name>

<scenarios> ::= <scenario> <scenarios> | (null)
<scenario> ::= scenario <name> <delim> <sdescription> endscenario <delim>
<sdescription> ::= <actordecl> <infdecl> <trails>
<actordecl> ::= <actorname> <delim> <actordecl> | (null)
<infdecl> ::= <infrastructurename> <delim> <infdecl> | (null)

<trails> ::= <trail> <trails> | (null)
<trail> ::= SOP <eventlist> EOP <delim>
<eventlist> ::= <levent> | <levent> + <eventlist> | (null)
<levent> ::= <label> | <event> | (.and. <levent> <levent>) |
(.or. <levent> <levent>)
<event> ::=  <label>: <action> | <action>
<action> ::= do(interaction_expression) | sync() | <action> & <action> | NOP
<label> ::= <name>

<name> ::= (string of alpha numeric characters)
<delim> ::= ;\s*
<num> ::= (natural number)
```

Figure 6.5: DSL grammar

represented as *trails* that make up the scenario.

Each trail consists of a beginning label called SOP and an end label called EOP. Between SOP and EOP, the trail consists of a sequence of labeled *events*. Event sequences may have branches, loops and parallel structures. Branching is represented by two kinds of constructs: `.and.` and `.or.` The `.and.` construct specifies concurrent execution of branches; and the `.or.` construct specifies decision making. A special event called `sync()` joins multiple paths in a synchronized fashion. At any place where an event is expected, a label of a previously declared event may be specified. This way loop structures are introduced.

Each event is described by a `do()` construct containing an interaction expression between domain objects (actors and infrastructure). An event may also be a *compound* event consisting of a conjunction of `do()` constructs. In DSL, there is no syntax for the interaction expression. It could be in the form of a textual description that describes the action.

Consider the running example of the conference IS. The semantic process that depicts paper submission by an author has many possible interaction sequences. The author may create a new account, before submitting a paper; the author may submit multiple papers in the same process, the author may replace a paper already uploaded with a newer version, etc.

The simplest way a user can specify such a semantic process is to list out as many interactive sequences as possible that make up the process. Here are an example set of DSL sequences that make up the process of paper submission:

$SOPdo(login) + l1 : do(newSubmission) + do(upload) + (.or.l1do(logout))EOP$
$SOPdo(login) + do(replace\ old\ submission) + do(upload) + do(logout)EOP$
$SOPdo(newSubmission) + do(upload) + do(checkSubmission) + do(newSubmission) + do(upload)EOP$

User specification typically starts with the most obvious interaction sequence. In this case, the interaction sequence that is most common (and is most likely to be specified first) is $do(login) + do(newSubmission) + do(upload) + do(logout)$. This represents one behavioral sequence that makes up the `AuthorSubmitsPaper` process. Depending on how comprehensively the requirements elicitation process is carried out, the user may explicitly think of the other sequences that make up the process, or of constructs like parallelism, branches and loops.

The task of the system analyst is to recognize that different sequences specified by the user make up the same semantic process. The analyst should generalize on these sequences and create a scenario that comprehensively describes the semantic process.

A scenario so specified can depict either an algorithmic, single-stream interactive or multi-stream interactive behavior. It is easy to identify whether a scenario represents an algorithmic activity. In this case, all trails in the scenario will have only one `do()` construct in their specification.

If a scenario represents sequential interaction, it can be modeled as a dialog. If it represents multi-stream interaction, it has to be modeled as a set of dialogs and constrained associations. In order to determine whether a scenario depicts multi-stream interaction, the following question needs to be asked: *In this scenario semantics, what forms the environment of the scenario and what forms the IS part of the scenario? And, when a message is sent by the IS part of this scenario, is it important to which environment it is sent?* In other words, is the input-output behavior of the scenario *channel-sensitive*? In the example for paper submission, the environment is the Author. The interactive behavior that is specified is not channel sensitive.

It is not a straightforward task to map between DSL scenarios and dialogs. How hard it is to design an interaction schema from a DSL specification would depend on how well the specification has classified semantic processes into scenarios and trails. In addition, no simple means exist to identify process families and build generalization-specialization relationships. For a large part, the soundness of a dualism conceptual model would still depend on the designer's skills.

Figure 6.6 shows part of the `AuthorSubmitsPaper` scenario as a combined DSL specification which has integrated all the three trails specified above. A pictorial representation of the scenario is also shown.

6.3.2 Generalization mechanisms

The task of the analyst during requirements elicitation is to identify that different sequences specified by the user belong to the same semantic process. The analyst should then combine the trails and perform generalizations on them to form a scenario. Generalization helps the analyst identify loops and branching structures which the user may not have specified explicitly. Some generalization mechanisms require domain knowledge, while a few other mechanisms may be applied without any explicit domain knowledge.

Generalization involves identifying a general process structure from specific substructures or execution exemplars. In a formal sense, this is a mapping from a carrier set Σ^* to a set of axioms Σ. Problems of this nature have been addressed in "grammatical inference" [MH96] that infers a grammar by looking at example sentences. Generalization of dynamic processes have also been addressed by Cook and Wolf [CW98], who build a state machine by looking at an example set of execution data. In [SS00], we compare these different approaches and present a mechanism of generalization by discerning patterns in transaction logs.

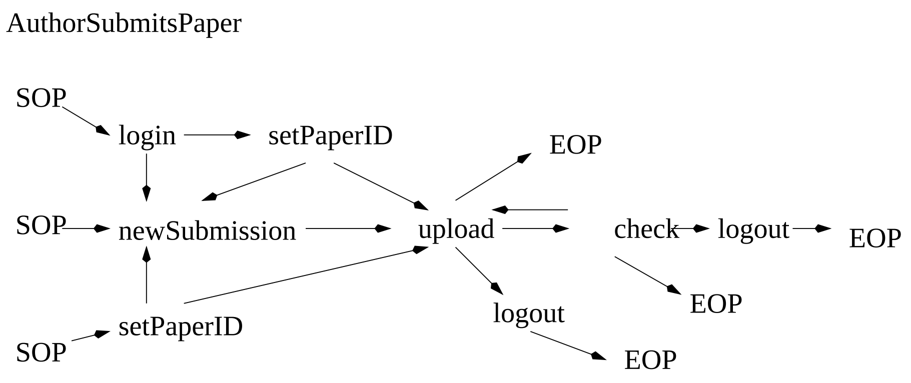
Combining trails: Trails of interaction sequences specified by the user have to be

Scenario AuthorSubmitsPaper
Actors: WebSite, Author;
Infrastructure: PaperDB, AuthorDB;

[T1,T2]SOP do(login) + (.or. [T1]l1: do(newSubmission) [T2] do(setPaperID)) + do(upload)
+ (.or. [T1] (.or. l1, do(logout)) [T2] do(logout)) [T1,T2] EOP

[T1,T2,T3]SOP [T1,T2](.or. do(login) [T3]l2: newSubmission) +
(.or. [T1]l1: do(newSubmission) [T2] do(setPaperID) [T3] do(upload)) +
(.or. [T1,T2] do(upload) [T3] do(check)) +
(.or. [T1] (.or. l1, do(logout)) [T2] do(logout) [T3] (.or. l2, NOP))[T1,T2,

(a)



(b)

Figure 6.6: Scenario construction

combined to form a scenario. Combination of trails is achieved in DSL as follows:

- A trail is first unfolded to form a sequence of the form $SOP\ (event)^{*}\ EOP$. This sequence is the unfolded set of all activities in the trail. Hence loops and branches are unfolded to form a long string of sequences.
- Consider a trail $SOP(event)_1 \ldots (event)_n EOP$ consisting of n events. Assign a logical timestamp to each event that shows it's temporal dependency with it's neighboring events. Hence if event i has a logical time stamp t (represented as $(event)_i^t$), then event $i+1$ would have a logical time stamp $t+1$ if there is a temporal dependency between events i and $i+1$. If event $i+1$ can be performed concurrently with event i, then it is given the same logical timestamp t.
- A trail is now considered as a sequence $SOP\ E^{t_0} \ldots E^{t_m}\ EOP$, of m compound events. Each compound event E^{t_k} is of the form $E^{t_k} = e_1 \wedge e_2 \wedge \ldots$ where $\wedge$ is logical conjunction of all events having the same logical timestamp.
- Prefix each event of a trail with a label showing the name of the trail. Hence if trail T1 contains events $E^{t_1} \ldots E^{t_m}$, then every $E^{t_i} = e_1 \wedge e_2 \cdots \in T1$ is represented as $[T1]e_1 \wedge [T1]e_2 \ldots$.
- Combine different trails by performing a logical disjunction among events having the same logical time. Hence if $[T1]E^{t_1} = e_1 \wedge e_2$ and $[T2]E^{t_1} = e_2 \wedge e_3$, then $[T1, T2]E^{t_1} = e_2 \wedge ([T1]e_1 \vee [T2]e_3)$. Note that since e_2 is necessarily performed in both T1 and T2, e_2 has no prefix label, while e_1 and e_3 have a prefix label which shows the trail context in which they are defined.

In the combined set of trails, the analyst can remove trail labels based on domain knowledge. Any event which has no attached trail label is applicable to all trails. For example, in Figure 6.6(a), the second trail depicts a combination of the three specified trails. Some labels that may be removed in this combined trail are as follows: `[T1,T2]upload` can be changed to `upload` since it is possible for a user to invoke upload twice. Similarly `[T2]logout` can be replaced with `logout` since a user may logout regardless of whichever is the current trail.

Loop structures in the combined trail may be identified by generalizing on regular expressions. A regular expression represents a string of characters with constructs that support repetitions. For any sequence of events $E^{t_1} \ldots E_{t_k}$, a generalized event sequence is obtained by building an automaton that recognizes this generalized sequence. Constructing automata from carrier sets is again a deeper issue which have been addressed in detail elsewhere [SS00]. A generated automaton is characterized between two boundaries – "most general" and "most specific". A "most general" automaton contains only one state and recognizes every string sequence. A "most specific" automaton recognizes only those set of strings which have been presented to it and nothing else. Generalization mechanisms fall somewhere between these extremes. The reader is refered to [MH96] and [SS99] for detailed approaches into generalization mechanisms.

6.3.3 Identifying OPF modalities in an IS story

Based on the DSL syntax and the combination of trails, the story space of an IS is defined formally as follows:

Definition 6.3: The story space of an IS is a tuple $Story = \langle E, C, SOP, EOP \rangle$, where E is a set of *events* of the form e^{t_k}, where t_k is the logical timestamp associated to them. Each event in the story has one of the following constructs: `do()`, `sync()`, `.and.`, `.or.`, or NOP. C is a set of relationships across events which are of the form $e_1^{t_k} \wedge e_2^{t_k}$, $e_1^{t_k} \vee e_2^{t_k}$, or $e_1^{t_k} + e_2^{t_{k+1}}$. Relationships $\wedge$ and $\vee$ are defined on events having the same logical timestamp; and $+$ is defined on events having consecutive timestamps. SOP denotes the head or start of a process and EOP denotes the tail or end of a process. $\square$

Walks in the story space: A *walk* in the story space, is any event sequence of the form $SOPe^*EOP$, where $e \in E$ and for any pair of consecutive events $e_1, e_2 \in E$, $e_1 + e_2 \in C$.

A walk is said to be an *obligated* walk if there is no event of the form `.or.` in the walk. It means that, if a semantic process that corresponds to the walk begins the walk at SOP, it will obligatorily result in the sequence denoted by e^* till EOP.

A sequence is said to be *permitted* if there exists at least one `.or.` event in e^*. This means that a semantic process that contains this walk may produce the sequence represented by the walk.

For verification of an interaction schema against the IS story walks in the story space are used as specifications of functionalities. Any walk in the story space is placed in one of three sets describing their modalities. These sets are named O_C, P_C and F_C respectively.

6.4 The Interaction Space

6.4.1 Structure of the interaction space

The interaction space contains the conceptual model of the IS. It is characterized by an interaction schema. The components of an interaction schema include dialogs modeled by a set of interaction states or fixpoints; constrained associations and dialog specialization relationships.

Interaction schema is modeled by the Dialog Description Language (DDL). The grammar describing DDL is shown below.

```
<InteractionSchema> ::= <Dialogs> <Relationships>

<Dialogs> ::= <Dialog> <Dialogs> | (null)
<Dialog> ::= Dialog <DialogName> <delim> <Attributes> <MethodInterfaces>
             <Fixpoints> <DialogDefinition> <Startstates> <Obligations>
             <Prohibitions> End Dialog

<DialogName> ::= <name>
<Attributes> ::= attributes <AttributeDescription> end attributes <delim>
<AttributeDescription> ::= <Object> : <Type> <delim>
                           <AttributeDescription> | (null)
<Object> ::= <name>
<Type> ::= <name>

<MethodInterfaces> ::= methods <MethodDeclaration> end methods <delim>
<MethodDeclaration> ::= <Type> <MethodName>\(<Params>\) <delim>
<MethodName> ::= <name>
<Params> ::= <Type> | <Type>, <Params> | (null)
<Type> ::= <name>

<Fixpoints> ::= states <FixpointDeclaration> end states <delim>
<FixpointDeclaration> ::= <fixpoint> <delim>
<fixpoint> ::= <name>

<DialogDefinition> ::= definition <FixpointDefinition> end definition
<delim>

<FixpointDefinition> ::= @<fixpoint> <MethodDefinitions>
                         <FixPointDefinition> | (null)
<MethodDefinitions> ::= <MethodDefinition> <MethodDefinitions> |
                        (null)
<MethodDefinition> ::= <MethodName> {} | <MethodName> { <intexp> }
<intexp> ::= <delim> | <varname> = <callexp> | return <callexp> |
             return <varname> | if (<intexp>) {<intexp>} else
             {<intexp>} | while (<intexp>) {<intexp>} | (null)
<callexp> ::= call <Object>.<name>(<name> (,<name>)*) <delim>
```

```
<Startstates> ::= startstates <Statelist> end startstates <delim>
<Statelist> ::= <fixpoint> , <Statelist> | <fixpoint> <delim> | (null)

<Obligations> ::= obligations <ODefinition> end obligations <delim>
<ODefinition> ::= <fixpoint> , <ODefinition> |
                  <fixpoint> <delim> <Odefinition> | <delim>

<Prohibitions> ::= prohibitions <PDefinition> end prohibitions <delim>
<PDefinition> ::= <fixpoint> , <PDefinition> |
                  <fixpoint> <delim> <PDefinition> | (null)

<Relationships> ::= <Constraint> <Relationships> | <Specialization>
                    <Relationships> | (null)

<Constraint> ::= constraints <vardecl> <constraintdecl> end
                 constraints <delim>
<vardecl> ::= <Varname> : <DialogName> <delim> <vardecl> | (null)
<constraintdecl> ::= <consequent> \<-- <antecedent> <delim>
                 <constraintdecl> | (null)
<consequent> ::= <deontic>[<Varname>] |
                 <deontic>[<Varname>.<fixpoint>] |
                 <deontic>[<DialogName>] |
                 <deontic>[DialogName>.<fixpoint>] | (null)
<deontic> ::= O | P | F
<antecedent> ::= <Varname> | <DialogName> | <antecedent> &&
                 <antecedent> | (null)

<Specialization> ::= specialization <SpecializationRel> end
                     specialization <delim>
<SpecializationRel> ::= <DialogName> \<-- <DialogName> <delim>
                     <SpecializationRel> | (null)

<name> ::= (alphanumeric string)
<delim> ::= ;\s*
```

As shown in the grammar, an interaction schema involves description of the dialogs in the schema and relationships across dialogs. Dialog relationships can be of two kinds: constrained association or dialog specialization. While the concept of a dialog relationship is more general, DDL allows only the above two kinds of relationships.

6.4.2 OPF modalities in the interaction space

In order to identify obligated, permitted and forbidden parts of the interaction space, we provide a redefinition of the interaction space as follows:

Definition 6.4: The interaction space is represented as $\langle D, S, \sqsubset, C \rangle$, where D is the set of all dialog names, S is the set of fixpoints from all dialogs and all instances of dialogs active in the interaction space; $\sqsubset$ is the traversal relationship across fixpoints; and C is the constraint relationship across fixpoints. $\square$

Procedures: A procedure in the interaction space is a sequence of fixpoints of the form $seq \in (S \cup D)^*$, such that for any $s \in seq$ either of the following holds:

- $s \in D$, or
- $\exists p$ before s such that $p \sqsubset s$, or
- $\exists p_1, p_2, \dots p_n$ before s such that $s \leftarrow p_1 \wedge p_2 \wedge \dots \wedge p_n \in C$

A procedure represents any interactive activity performed by the IS on one or more channels. A procedure traverses through the interaction space based on the traversal relationship $\sqsubset$ and on constraints between interaction states.

A procedure P is said to be *obligated* if: (a). At least one dialog name $s \in D$ present in the procedure P is obligated; and (b). For every dialog s in P, at least one of it's obligated string of fixpoints is also present in P.

A procedure P is said to be *permitted* if: (a). $\nexists s \in P$, such that $s \in D$ and Os. That is, there are no obligated dialogs which have been involved as part of the procedure; and (b). P does not end with any forbidden elements.

A procedure P is said to be *forbidden* if P ends on forbidden elements. That is, the procedure is given up because it encounters forbidden fixpoints and cannot proceed further.

Any procedure in the interaction space is placed in one of three sets describing their modalities. These sets are named O_C, P_C and F_C respectively. Verification of a conceptual model against a specification is made by establishing a mapping between the story space and the interaction space. An ideal mapping is bijective between O_S and O_C and between P_S and P_C. Mappings which are not bijective lead to translation hazards which have been discussed earlier in this chapter.

Example: Refer to the interaction schema of the conference IS from Figure 5.4. In this interaction, the following sequence forms a procedure: `ConferenceProcess . CfP . AuthorSubmitsPaper . login . newSubmission`

. `upload` . `logout`. Every element of this sequence is related to it's previous element either by the traversal relationship $\sqsubset$ or a constrained association. For example $P[AuthorSubmitsPaper] \leftarrow [ConferenceProcess.CfP]$ and $login \sqsubset newSubmission$. If the dialog `ConferenceProcess` is considered obligated (it is obligated to run the conference!) then the above procedure forms an obligated procedure, else the above procedure is a permitted procedure.

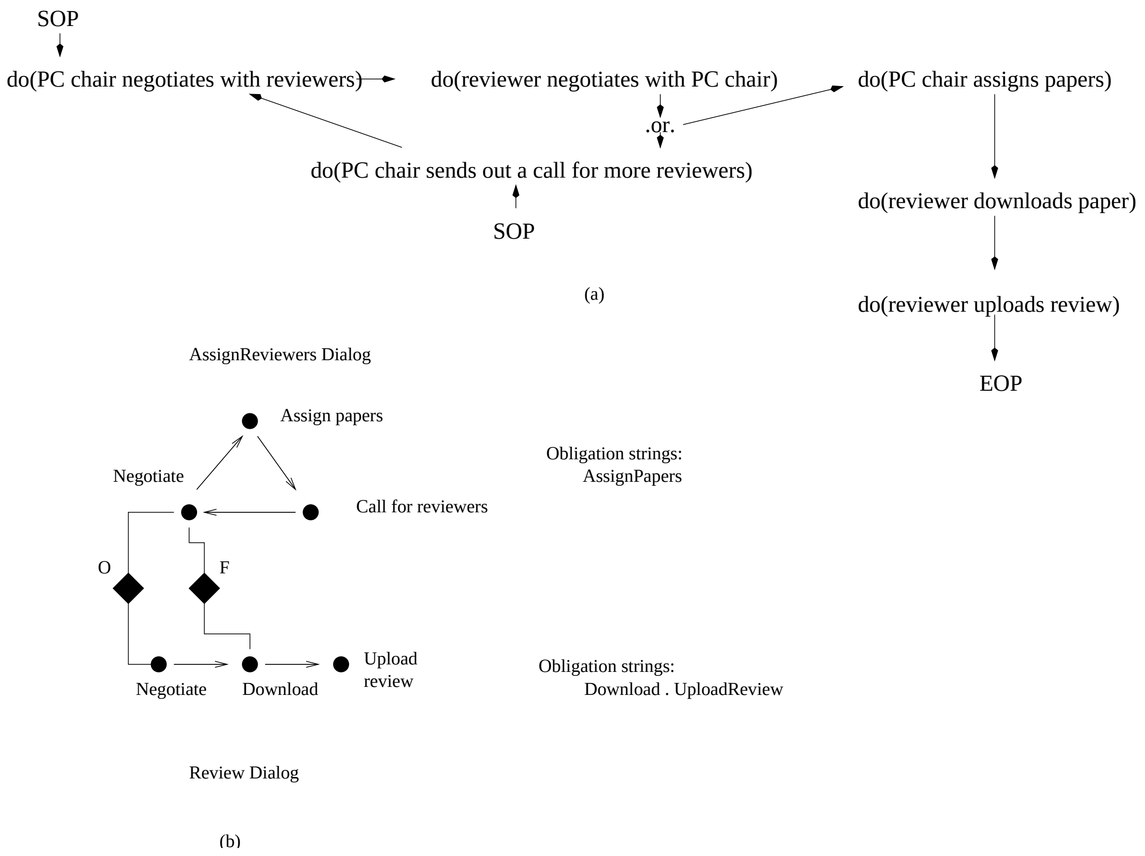

SOP
do(PC chair negotiates with reviewers)
do(reviewer negotiates with PC chair)
do(PC chair assigns papers)
.or.
do(PC chair sends out a call for more reviewers)
SOP
do(reviewer downloads paper)
do(reviewer uploads review)
EOP
(a)
AssignReviewers Dialog
Assign papers
Negotiate
Call for reviewers
Obligation strings:
AssignPapers
O
F
Negotiate
Download
Upload review
Obligation strings:
Download . UploadReview
Review Dialog
(b)

Figure 6.7: Example translation between story and interaction spaces

Translation Example: Consider the specification of a review scenario depicted pictorially in Figure 6.7(a). The scenario is a combination of many trails which specify how a PC chair performs the review process. This scenario is channel sensitive. It interacts with two environments – the PC chair and the reviewer and it is important which task is done by whom. Figure 6.7(b) shows part of the interaction schema that models this scenario. It consists of two dialogs and constraints between them. The dialogs are named `AssignReviewers` and `Review` respectively. `AssignReviewers` interacts with the PC chair and is used to negotiate with reviewers, call for new reviewers and assign reviewers. `Review` is used by the reviewer to negotiate with the PC chair and to review papers. The obligated string of interaction states for `AssignReviewers` is the lone state *AssignPapers* since the only thing that is expected of this process is that it assigns papers to reviewers. The obligated string for the `Review` dialog is $Download \cdot UploadReview$.

In order to validate the interaction schema against the DSL scenario, the following test is performed: any walk in the DSL scenario contains a corresponding procedure in the interaction schema and vice versa. The following is an obligated walk of the scenario: *SOP + do(PC chair negotiates with reviewers) + do(reviewer negotiates with PC chair) + do(PC chair assigns papers) + do(reviewer downloads paper) + do(reviewer uploads review) + EOP*. The corresponding procedure in the interaction schema is as follows: `AssignReviewers . Review . PCNegotiate . ReviewerNegotiate . AssignPapers . Download . UploadReview`. In order for this procedure to be an obligated procedure, at least one of the dialogs that begin this procedure should be obligated. This is verified by looking at the larger interaction schema from Figure 5.4 where the `AssignReviewers` dialog is obligated when the `ConferenceProcess` dialog reaches the *Review* state. □

6.5 The System Space

6.5.1 Structure of the system space

The system space is in the realm of the IS developer. It concerns implementational aspects and has to address many practical issues like distribution, heterogeneity, efficiency, etc.

The number of issues that have to be addressed in this space is too numerous to be able to address them comprehensively with one implementational language for dialogs. Hence we propose an implementational framework that guides the developer to implement a given interaction schema.

This framework is called the Dialog Presentation Framework (DPF) and is depicted schematically in Figure 6.8. The interaction schema is stored in a database which is accessed by one or more constraint manager processes. Constraint managers act as local servers on different locations which provide IS functionality. Dialogs may be started on any local server and the dialog is managed by the local constraint manager. The constraint manager keeps track of the states of each of it's dialogs and propagates constraints to other managers when necessary.

Since the interaction schema is shared by all constraint managers, locks are maintained in the central database where the interaction schema is stored. Locks ensure that any given constraint is managed by at most one constraint manager.

The life cycle of a dialog in this framework is as follows:

- When a dialog D_l is started on any local server l, it registers itself with the local constraint manager C_l.

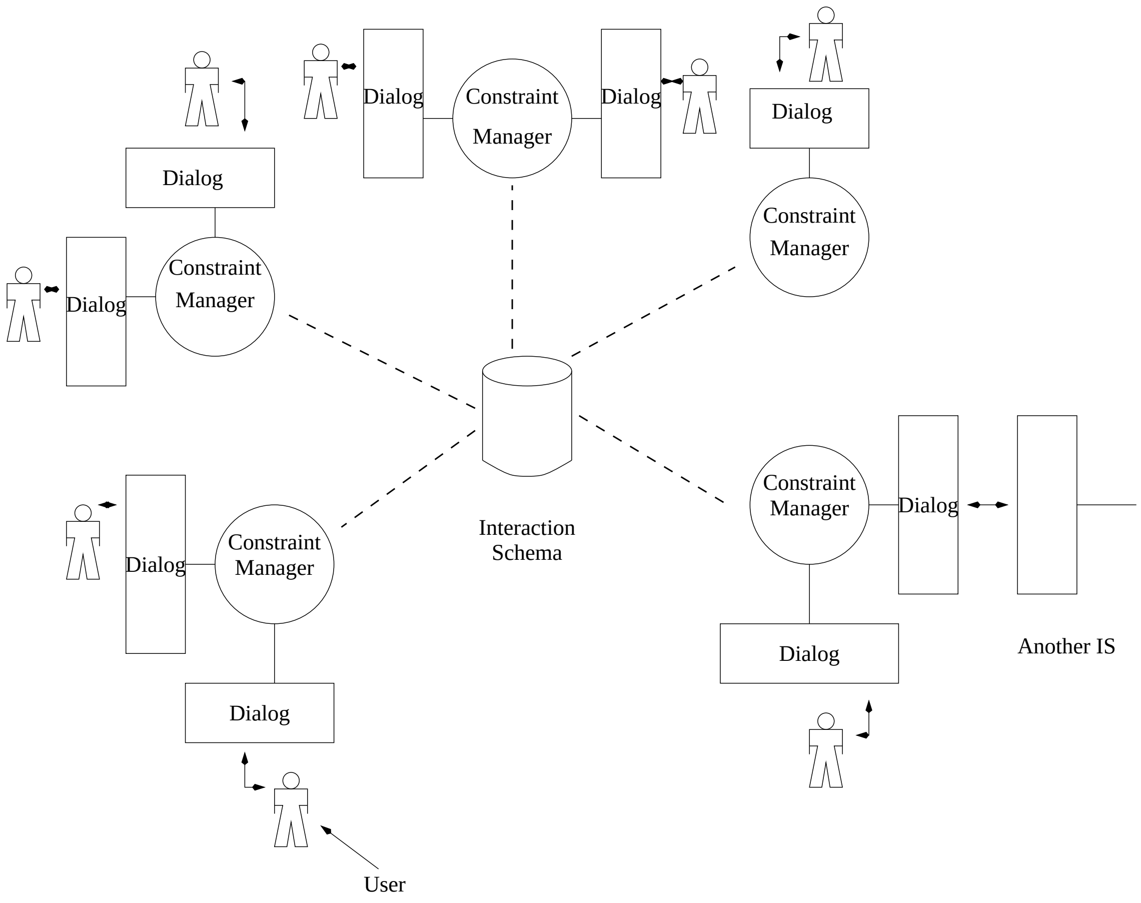


Figure 6.8: The dialog presentation framework

- The constraint manager C_l checks the central database whether D_l is free of any constraints acting on it.
- If constraint ψ contains the dialog in *head* part of the constraint, then C_l tries to lock the constraint.
- If no other constraint manager has already held ψ, the lock is granted to C_l.
- If some other constraint manager C_k has already held the lock for ψ, then C_l registers itself with C_k. Whenever the status of the *body* of ψ changes, C_k would now inform C_l, which in turn would propagate the constraint to D_l.
- The process by which a dialog finishes execution is also achieved by an analogous process of intimations and release of locks.

For a dialog itself, there are certain aspects which have to be implemented by the developer. These concern handling OPF modalities for the dialog and each of it's fixpoints.

The developer needs to decide what is performed when the dialog or any of it's fixpoints are respectively obligated, permitted or forbidden. For example, in some cases, obligation towards a dialog may mean sending an email to the concerned people to start those dialogs. In some other cases, an obligation to a dialog may simply mean displaying a pop-up window on a user's terminal. The DPF framework hence requires the developer to fill in a template for each dialog in the system that specifies how each of the dialog's modalities are presented. The syntax of the template is as follows:

```
<DPFDescription> ::= Dialog <DialogName> <delim> <DO> <DP> <DF>
                     <FM> End <DialogName> <delim>

<DialogName> ::= <name>

<DO> ::= O { <command> }
<DP> ::= P { <command> }
<DF> ::= F { <command> }

<FM> ::= @<fixpoint>.O { <command> } <FM> | @<fixpoint>.P { <command> }
         <FM> | @<fixpoint>.F { <command> } <FM> | (null)

<command> ::= (string of commands in implementation language)
<fixpoint> ::= <name>
<name> ::= (alphanumeric string)
<delim> ::= ;\s*
```

In addition to the above, each implementation of a dialog should implement the following methods:

set(fixpoint, modality) This function is invoked by the constraint manager to change the modalities of the dialog's fixpoints. Modalities that apply to the dialog itself (dialog is obligated, dialog is permitted, etc.) are maintained by the constraint manager.

schedule() This function determines the order in which a dialog performs it's obligations.

6.5.2 OPF modalities in the system space

The system space concerns implementational aspects. Behavioral semantics of the system space depends on the implementational paradigm. For example, the behavior of an RDBMS that implements the IS would be very different from the behavior of a set of objects. But from a generic sense, the system space dynamics are actually the database dynamics of the IS. While the implemented system functions to execute semantic processes, it's own dynamics are characterized by transitions across system states. The formal definition of system space dynamics is as follows:

Definition 6.5: The system space is represented as $\langle S, \rightarrow \rangle$ consisting of an infinite set of states S and a transition relationship $\rightarrow$ among states of the state space. $\square$

Identifying OPF modalities in the system space is achieved by mapping these modalities from the interaction space onto the set of system states. We propose to do this by executing queries on the system space that look for violations of dynamic integrity. A set of queries called *verification queries* are designed to look for violations of specific modalities. These queries are run on the database or set of databases that maintain the system state.

Dialog trace: In order to be able to run verification queries it is required to store execution *traces* of dialogs in the system space. The execution trace of a dialog is a sequence of the form $\langle id, s^* \rangle$ where id is a unique process id that identifies the specific instance of the dialog, and s is one of the following: the name of the dialog, or the name of one of it's fixpoints, or the name of one of it's methods, or constraint actions.

Example: The following sequence represents a trace of the dialog `Review`. ⟨ `10124, Review . Negotiate . O(Negotiate) . F(Download) . Download . UploadReview`⟩. The trace represents a particular execution with id 10124 of the `Review` dialog. The trace records only those operations that have happened after the dialog has started. Refer to Figure 6.7 which shows two constraints on this dialog. The constraints $O[Review.Negotiate]$ and $F[Review.Download]$ hold when $AssignReviewers.Negotiate$ holds. The execution trace records execution after this constraint has held. During the execution, $AssignReviewers.Negotiate$ no longer holds and so the above constraint too ceases to hold. $\square$

Analogous to a dialog trace is a trace of a constrained association. This records when constraints across dialogs were activated and which dialogs were affected as a result. The trace is of the form: $\langle id, head, body \rangle$, where id is the unique process id of the constraint process, *head* and *body* are constraints where the name of the dialog is replaced with their process ids. An example of the trace of a constrained association would be an entry of a tuple of the form: $\langle 23529, O[10124.Negotiate], 7893.Negotiate \rangle$. This entry captures the execution of a constrained execution between dialogs `AssignReviewers` and `Review` concerning their $Negotiate$ fixpoints.

Verification Example: Consider the example procedure visited before:

```
AssignReviewers . Review . PCNegotiate . ReviewerNegotiate .
AssignPapers . Download . UploadReview
```

In order to verify it's correspondence in the system space, some of the verification queries that are designed are as follows.

```
SELECT Review.id FROM Review, AssignReviewers, Constraints WHERE
         O[Review.id.Negotiate], [AssignReviewers.id.Negotiate]
         IN Constraints AND NOT (Review.id, Negotiate IN Review)
```

The above query searches for instances of the dialog `Review` which have violated their obligations to be in fixpoint *Negotiate* when the `AssignReviewers` dialog was in *Negotiate*. The above query assumes that dialog traces are placed in a table with the same name as the dialog; and constraint traces are placed in a common table called Constraints.

The following constructs are also proposed on top of standard SQL constructs to query dialog traces: a `BEFORE` b, a `AFTER` b, `NEXT`(a,b). They respectively match cases where a comes before b, a comes after b and where b comes right after a.

The following query returns all instances of the dialog `Review` which has violated the prohibition constraint from the `AssignReviewers` dialog.

```
SELECT Review.id FROM Review, AssignReviewers, Constraints WHERE
         F[Review.id.Download], [AssignReviewers.id.Negotiate]
         IN Constraints AND Download BEFORE ~F(Download) IN Review.id
```

6.6 Complexity and Decidability of Verifications

Verifications of translation correctness between the story, interaction and system spaces were presented in this chapter. Verification techniques proposed here are limited in scope and are based on a paradigm of sequence matching.

Verification technique between the story space and the interaction space is summarized as follows. Given a pair (w_i, p_i) consisting of a walk and a procedure, an assertion is made that they are equivalent. Verifying this assertion entails proving that p_i can produce w_i and both p_i and w_i have the same modality.

This technique can be used to identify hazards like incomplete models (where p_i cannot produce w_i) and malicious designs (where p_i produces more than w_i). Similarly if the modality of w_i and p_i is P, the above verification can detect hazards leading to unsafe models and conceptually inefficient models. The verification is based on stream matching, whose complexity is $O(mn)$ where m is the number of elements in w_i and n is the number of elements in p_i.

Verification of arbitrary subspaces between the story and interaction spaces involves determining equivalence between two possibly infinite streams. This is a known non-decidable problem.

Verification between the interaction space and the system space is carried out by querying the system space for violations of constraints. A summary of this technique is as follows: given a pair (p_i, Q) consisting of a procedure p_i and a set of queries Q that seek to refute p_i, prove that Q addresses all constraints present in p_i and Q produces an empty set when run on the system space.

Execution traces in the system space is assumed to be organized as tables corresponding to dialogs and one table corresponding to the set of all constraints. The complexity of evaluating a particular query is dependent on the number of joins required for answering the query. However, a procedure can be potentially infinite in length which makes the verification problem undecidable. But for any finite length of the procedure, the verification problem is decidable, albeit dependent on the number of dialogs and constraints in the procedure for its complexity.

Chapter 7

Conclusions

I've found my niche. If you're wondering why I'm not there, there was this little hole in the bottom ... – *John Croll*

7.1 Lessons from Interaction

This thesis addressed two aspects of interaction modeling: (a). Modeling interactive behavior as a domain independent issue of concern, and (b). Modeling the dynamics of open (interactive) information systems.

In modeling interactive behavior as a domain independent issue of concern, the following insights were obtained:

- Interaction is made of three components: computation, persistence of state and channel sensitivity.
- Interactive domains have non-wellfounded structures
- To describe properties of an interactive domain, it requires at least a three-valued system of logic.

By applying the above to information system design, the following insights were obtained:

- Information system dynamics are not only described be computational processes that change the system state, but also by sequences of interaction.
- Interactive sequences represent functionality dynamics.
- The interaction space of an information system represents dynamic integrity of the information system based on it's functionality semantics.
- Functionality dynamics of an information system are autonomous to some degree with respect to state maintenance or structural dynamics.

- Interaction spaces are described by a three-valued system of logic whose modalities are named obligations, permissions and prohibitions respectively.
- Many semantic errors in information system design may be attributed to mismatches between these modalities across specification, design and implementation.

7.2 Open Questions and Future Directions

A concept of solution space of a dynamic process only scratches the surface of what is likely to be a much deeper issue. There are many open questions concerning interaction modeling. The recognition that information systems can be modeled as a collection of semantic processes or PSPs, is significant. It shows in a formal fashion how interactive PSPs (and hence open systems) are richer than algorithmic PSPs (or closed systems).

A significant insight is also about the constituents of interaction: computation, persistence of state and channel sensitivity. Persistence of state introduces properties like evolution, learning and history dependent behavior into a computational system. Channel sensitivity introduces possibilities for coordination, collaboration and networking of computational systems. It is also significant to note that open systems can be modeled fundamentally by a MIM which consists of all the above three properties. However, the entire range of characteristics resulting from these properties are still largely unexplored.

In Chapter 2 a few questions were posed about MIMs. Some of these were:

1. What characterizes a MIM computation?
2. How can MIMs be compared?
3. Are there functions which are not computable by an algorithm but are computable by a MIM?
4. What are the boundaries of MIM computation?
5. What are the characteristics (like complexity, decidability, etc.) of problems that are solvable by a MIM?

This thesis has perhaps partially answered the first and the second questions. MIM computations are characterized by three kinds of behaviors: liveness or obligated behaviors, possible or permitted behaviors and forbidden behaviors. However, this is just most general characterization of MIM computations. There could be other properties like prioritized behavior or evolution of behaviors, which have not been addressed here.

MIMs can be compared by sketching their solution spaces of interactive fixpoints. This is again based on the model of a solution space presented in this thesis. Such a model is nowhere comprehensive since it reduces MIM computation into only three behavioral deontics.

The rest of the questions are still open and there are likely to be more such questions regarding MIM computation.

A concept of interaction modeling in a domain independent fashion also can open new paradigms of designing and managing open systems. The term “open system” gets a formal definition and legitimacy from interaction modeling. Interaction modeling also helps in separating the functionality concerns of an open system from the structural concerns. Such a separation of concerns helps in better portability and maintainability of information systems.

However, the formal definition of an open system also brings in a number of open questions. Some of these are as follows:

1. What properties about open systems can be formally proved?
2. Is it possible to formalize the design complexity of an open system in a domain independent fashion?
3. Does interaction modeling bring down design complexity?
4. Is interaction modeling the elusive “silver bullet” [Bro95] of software engineering?

An exploration into some of the above questions may perhaps become imminent as interaction modeling is better understood and applied to practical problems.

Bibliography

[Aal99] W.M.P. van der Aalst. Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information? *Proc. of Fourth IFCIS Int'l Conf on Cooperative Information Systems (CoopIS)*, Edinburgh, IEEE Computer Society Press, Sep 1999.

[AC96] Martin Abadi, Luca Cardelli. *A Theory of Objects.* Springer-Verlag, New York 1996.

[AGM92] S. Abramsky, Dov M. Gabbay, T.S.E. Maibaum (Eds.). *Handbook of Logic in Computer Science.* Oxford Science Publications. 1992.

[Acz88] Peter Aczel, Non Well-Founded Sets, CSLI Lecture Notes #14, Stanford, 1988.

[AMST92] Gul Agha, Ian A. Mason, Scott Smith, Carolyn Talcott. Towards a Theory of Actor Computation. Proc of CONCUR '92, R. Cleaveland (Ed.), LNCS 630, pp. 565-579, Springer-Verlag, 1992.

[AR96] Philip E. Agre, Stanley J. Rosenschein. Computational Theories of Interaction and Agency. *MIT Press*, 1996.

[AB71] C. E. Alchourrón, E. Bulygin. *Normative Systems.* Springer, 1971.

[Alt96] Alter, S. *Information systems: A Management Perspective.* Benjamin/Cummins, 1996.

[Awa93] Elias M. Awad. *Systems Analysis and Design.* Irwin Publishers, 1993.

[BM96] John Barwise, Lawrence Moss. Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena. CSLI Lecture Notes, No. 60. 1996.

[Beh99] Mohsen Beheshti. A High-Level Language for Object-Oriented Transactions. *Proc. of the Ninth International Conference on Heterogenous and Internet Databases*, Hong Kong, July 1999.

[Bro95] Friedrick Brooks. The Mythical Man-Month, The Twentieth Anniversary Edition. *Addison-Wesley Publishing Company*, Reading, Mass. 1995.

[BC96] Mark A. Brown, Jose Carmo (Eds.). *Deontic Logic, Agency and Normative Systems*, Springer, 1996.

[Bro97] Manfred Broy. Compositional Refinement of Interactive Systems. *Journal of the ACM*, Vol. 44, No. 6, Nov 1997, pp. 850-891.

[CG89] N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, Vol. 32, No. 4, 1989, pp 444–458.

[CP99] Fabio Casati, Giuseppe Pozzi. Modeling Exceptional Behaviors in Commercial Workflow Management Systems. *Proc. of Fourth IFCIS Int'l Conf on Cooperative Information Systems (CoopIS)*, Edinburgh, IEEE Computer Society Press, Sep 1999.

[CT97] Wolfram Clauß, Bernhard Thalheim. Abstraction Layered Structure Process Codesign. *Proceedings of the 8th International Conference on Management of Data (COMAD '97)*, Narosa Publishers, Chennai, India, 1997.

[CNM95] P. Coad, D. North, M. Mayfield. *Object Models: Strategies, Patterns and Applications*, Prentice Hall, Englewood Cliffs, 1995.

[CW98] J. E. Cook and A. L. Wolf. Discovering models from software processes from event-based data. *ACM TOSEM*, Vol. 7, No. 3, July 1998, pp 215–247.

[Cop98] James O. Coplien. A Generative Development Process Pattern Language. In Linda Rising, (ed.), *The Patterns Handbook: Techniques, Strategies, and Applications*, pp. 243-300. Cambridge University Press, New York, January 1998.

[DeV98] Richard W. DeVaul. Decidability – Truth or Turing? *Technical Report*, Aesthetics and Computation Group, MIT Media Lab, June 1998.

[DC95] Prasun Dewan, Rajiv Choudhary. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer Human Interaction*, Vol. 2, No. 1, Mar 1995, pp. 1-39.

[ES99] T. Eiter and V. S. Subrahmanian. Deontic Action Programs. In K.-D. Schewe, T. Ripke and T. Polle, (Eds.) *Fundamentals of Information Systems: Selected Papers presented at the 7th International Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO 98)*, October 5–8, 1998, Timmel, Germany, pages 37–54. Kluwer, 1999.

[Elm92] A.K. Elmagarmid (Ed). *Transaction Models for Advanced Database Applications.* Morgan-Kaufmann, 1992.

[GST00] Dina Goldin, Srinath Srinivasa, Bernhard Thalheim. IS = DBS + Interaction: Towards Principles of Information Systems. *Proc. of ER 2000*, Salt Lake City, USA, Oct 2000.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996

[Fow97] Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language*, Reading, MA: Addison-Wesley 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GW98] Dina Goldin, Peter Wegner. Persistence as a Form of Interaction. *Technical Report CS 98-07*, Brown University, March 1998.

[Gol00] Dina Goldin. Persistent Turing Machines as a Model of Interactive Computation. *Proceedings of the FoIKS 2000*, Burg, Germany, Feb 2000.

[GR93] Jim Gray, Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[HR94] B. Hannon, M. Ruth. *Dynamic Modeling.* Springer-Verlag 1994.

[HS98] H. Haugeneder, D. Steiner. Co-operating Agents: Concepts and Applications. *in [JW98]*, pp 174–202.

[Hay94] Wayne Haythorn. What is Object-oriented Design? *JOOP*, March 1994.

[HU79] John Hopcroft, Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[How93] Dennis Howe. Free Online Dictionary of Computing. *http://www.foldoc.org/*

[HSG98] M. N. Huhns. M. P. Singh. Les Gasser (Eds.). *Readings in Agents.* Morgan Kaufmann Publishers 1998.

[Hug89] John Hughes. Why Functional Programming Matters. *The Computer Journal*, Vol. 32, No. 2, 1989, pp. 98-107.

[Jac96] Bart Jacobs. Coalgebraic Specifications and Models of Deterministic Hybrid Systems. *In M. Wirsing and M. Nivat (Eds.) Algebraic Methodology and Software Technology (AMAST 1996)*, Springer LNCS 1101, Berlin, 1996, pp. 520–535.

[Jac96a] Bart Jacobs. Objects and Classes, Co-algebraically. *In B. Freitag, C .B. Jones, C. Lengauer, and H.-J. Schek (Eds.) Object-Orientation with Parallelism and Persistence* Kluwer Acad. Publ., 1996, pp. 83–103.

[JR97] B. Jacobs, J.J.M.M. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of EATCS*, Vol. 62, 1997, pp. 222–259.

[JKSS90] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. Proceedings of ICSE 1990, pages 63-71. IEEE Computer Society Press, 1990.

[JW98] Nicholas R. Jennings, Michael J. Wooldridge. (Eds.) *Agent Technology. Foundations, Applications and Markets,* Springer, 1998.

[JL96] Y. Jin, R. E. Levitt. The Virtual Design Team: A Computational Model of Project Organizations. *Computational and Mathematical Organization Theory* Vol. 2, No. 3 1996, pp. 171-196.

[Joh95] Paul Johannesson. Representation and Communication – A Speech Act Based Approach to Information Systems Design. *Information Systems* Vol. 20, No. 4, 1995, pp. 291–303.

[KR96] Mohan U. Kamath, Krithi Ramamritham. Correctness Issues in Workflow Management. *Distributed Systems Engineering (DSE) Journal : Special Issue on Workflow Management Systems*, Vol 3, No 4, Dec 1996, pp. 213-221.

[Kie97] Richard Kieburtz. Reactive Functional Programming. *Technical Report*, Oregon Graduate Institute of Science and Technology, CS-97-008, 1997.

[Kim95] Won Kim. *Modern Database Systems.* ACM Press, 1995.

[Kow92] James A. Kowal. *Behavior Models: Specifying User's Expectations.* Prentice Hall, 1992.

[Len98] Marina Lenisa. Themes in Final Semantics. *PhD Thesis*, Dipartimento di Informatica, Universita‘ di Pisa, March 1998.

[LS00] Jana Lewerenz, Srinath Srinivasa. Abstraction of the Interaction Properties of an Information System for Achieving Target Platform Independence. *Proc. of Modellierung 2000*, St. Goar, Germany, April 2000.

[Lew00] Jana Lewerenz. Human-Computer Interaction in Heterogenous and Dynamic Environments. *Phd Thesis*, BTU-Cottbus, Germany, 2000.

[Lip87] Richard P. Lippman. An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine*, April 1987.

[Liu96] Ling Liu, Robert Meersman. The Building Blocks for Specifying Communication Behavior of Complex Objects: An Activity-Driven Approach. *ACM Transactions on Database Systems*, Vol 21, No 2, June 1996, pages 157-207.

[LZ92] Loucopoulos, P., Zicari, R. (eds.). *Conceptual Modeling, Databases and CASE.* John Wiley & Sons, (1992).

[MC94] Thomas Malone, Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys* Vol. 26, No. 1 1994, pp. 87-119.

[MP92] Zohar Manna, Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1992.

[May97] Richard Mayr. Decidability and Complexity of Model-Checking Problems for Infinite State Systems. *PhD Thesis*, Technical University of Munich, Munich, Germany, 1997.

[MW93] John-Jules Ch. Meyer, Roel J. Wieringa (Eds). Deontic Logic in Computer Science: Normative System Specification. John Wiley and Sons, 1993.

[MH96] L. Miclet and C. de la Higuera (Eds.). *Grammatical Inference: Learning Syntax from Sentences.* Lecture Notes in Artificial Intelligence #1147, 1996.

[Mil96] Dale Miller. Logical Foundations for Open System Design. *ACM Computing Surveys*, Vol 28, Dec 1996.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mor99] Gareth Morgan. Images of Organization. *Sage Publications*, May 1999.

[LLMT00] Annette Laue, Matthias Liedtke, Daniel Moldt, Ivana Tričković. Modeling Intra- and Inter-Object Control Using Reference Nets. *Proceedings of Modellierung 2000*, St. Goar, Germany 2000.

[Law97] Peter Lawrence (ed.). *Workflow Handbook.* Workflow Management Coalition, 1997.

[Nut92] Gary J. Nutt. *Open Systems.* Prentice Hall, New Jersey, 1992.

[Pit93] Andrew M. Pitts. Bisimulation and Co-induction (tutorial). *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, Montreal, Canada, June 1993. IEEE Computer Society Press.

[PR98] Michael Prasse, Peter Rittgen. Why Church's Thesis still holds - Some Notes on Peter Wegner's Tracts on Interaction and Computability, *Computer Journal* Vol. 41 (1998) No. 6, pages 357-362

[Pre95] Wolfgang Pree. Design Patterns for Object-Oriented Software Development. *Addison-Wesley*, 1995.

[RL99] Fethi Rabhi, Guy Lapalme. *Algorithms: A functional programming approach.* Addison-Wesley, 1999.

[RG00] Raghu Ramakrishnan, Johannes Gehrke. *Database Management Systems* (2^{nd} Edition). McGraw Hill, 2000.

[RC96] Krithi Ramamritham and Panos K. Chrysanthis. Advances in Concurrency Control and Transaction Processing. *IEEE Computer Society Press*, September 1996.

[Ros82] J. Barkley Rosser. Highlights of the history of the lambda-calculus. *ACM Lisp and Functional Programming*, 1982.

[RBP+90] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, 1990.

[Rut96] J.J.M.M. Rutten. Universal Coalgebra: a theory of systems. *Technical Report*, CS-R9652, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, 1996.

[Sab98] Amr Sabry. What is a purely functional language. Journal of Functional Programming. Vol 8, No 1, pp 1-22, Jan 1998.

[SHF00] Heinz Schweppe, Annika Hinze, Daniel Faensen. Event-based Notification on the Web. *Tutorial at the 9th International World Wide Web Conference (WWW9)*, Amsterdam, 2000.

[Sea69] J. Searle. *Speech Acts – An Essay in the Philosophy of Language.* Cambridge University Press 1969.

[Sen97] James A. Senn. *Information Technology in Business: Principles, Practices, and Opportunities.* Prentice Hall 1997.

[SR93] A. Sheth, M. Rusinkiewicz. On Transactional Workflows. *Bulletin of the Technical Committee on Data Engineering*, Vol 16, No 2, June 1993, IEEE Computer Society.

[Som00] Ian Somerville. *Software Engineering* (6th Edition), Addison-Wesley, 2000.

[SS99] Srinath Srinivasa, Myra Spiliopoulou. Analyzing Transaction Logs for Building Coordination Models. *Technical Report 12/99*, BTU-Cottbus, Cottbus, Germany, Dec 1999.

[SS00] Srinath Srinivasa, Myra Spiliopoulou. Discerning Behavioral Properties by Analyzing Transaction Logs. *Proceedings of ACM Symposium on Applied Computing (SAC'00)*, Como, Italy, March 2000.

[SBD+00] V. S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, Robert Ross. Heterogenous Agent Systems. MIT Press, 2000.

[Tha00] Bernhard Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*, Springer Publications, Berlin, 2000.

[TS94] Roshan K. Thomas, Ravi S. Sandhu. Conceptual Foundations for a Model of Task-Based Authorizations. *IEEE Computer Security Foundations Workshop*, Franconia, New Hampshire, June 1994.

[Ull88] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems.* W. H. Freeman & Co. 1988.

[UML] UML Resource Center. *http://www.rational.com/uml/index.jtmpl*

[Vel94] D. Velleman. *How to Prove It.* Cambridge University Press, New York, 1994.

[Wad97] Philip Wadler. How to declare an imperative. ACM Computing Surveys, Vol 29, No 3, pp 240-263, Sep 1997.

[Wei99] Gerhard Weiss. Multiagent Systems : A Modern Approach to Distributed AI. *MIT Press*, 1999.

[Weg96] P. Wegner. Interactive Software Technology. *CRC Handbook of Computer Science and Engineering*, 1996.

[Weg97] Peter Wegner. Why Interaction is More Powerful than Algorithms? *Communications of the ACM*, May 1997.

[WG99] Peter Wegner, Dina Goldin. Interaction as a Framework for Modeling. In LNCS #1565, 1999.

[WG99a] Peter Wegner, Dina Goldin. Mathematical Models of Interactive Computing. *Technical Report*, Brown University, Jan 1999.

[WG99b] Peter Wegner, Dina Goldin. Coinductive Models of Finite Computing Agents. *Electronic Notes in Theoretical Computer Science*, Vol 19, Elsevier, 1999.