A FORMALISM FOR OBJECT-BASED INFORMATION SYSTEMS DEVELOPMENT

By

Ken Takagaki

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

COMMERCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

March, 1990

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMMERCE

The University of British Columbia
Vancouver, Canada

Date Apr 19/90

Abstract

Most current approaches to Information Systems Development (ISD) tend to derive from past experience and practice, rules of thumb and technology trends. The lack of theoretical foundations hinders the systematic development and evaluation of new ISD methodologies. The research undertaken in this thesis addresses this issue by proposing a formal, theory-based model, Ontology/Object-Based Conceptual Modelling (OBCM), for conceptually representing IS applications. The formalism is novel in that it is grounded in first principles derived from metaphysics, in particular the system of Ontology developed by Mario Bunge. Underlying this approach is the premise that an Information System is a model of reality and that model should be therefore rooted in a theory of reality, ie. a metaphysics. As a result, basic assumptions in reality such as thing, substance, property, attribute, time, state and change are explicitly and rigorously addressed. OBCM features an ontologically well-defined construct, "object", which is used to directly represent entities in reality, thus lending theoretical credence to the so-called object-oriented paradigm found in recent programming languages and databases.

In addition, the thesis presents a framework, Ontology/Object-Based Information System (OBIS), for systems implementation based on this model. This framework directly implements the object construct so that it can be immediately utilized by the information systems user in a "direct manipulation" style of end-user interaction. Further, OBIS strives for a single, homogeneous concept of system operation drawn from ontology rather than in terms of IS or computing technology. In principle, this one concept can be applied to any object in the IS, this simplifying the understanding and use of the Information System. In this way, the model attempts to unify the analysis,

implementation and user-interface aspects of Information Systems Development, thereby reducing the so-called "semantic gap" which has often been observed between the reality of the application and its final implementation in an IS.

A "proof of concept" prototype is described which illustrates the main principles and explores practical applications of the proposed model. This prototype is implemented as a single, stand-alone "shell" which can be used to support a wide variety of applications as well as providing the basis of a rapid prototyping or CASE tool. The prototype is used to implement sample problems including the well-known IFIP Working Conference problem, thus demonstrating the feasibility of the overall approach.

Table of Contents

List of Figures

Acknowledgement

It gives me great pleasure to acknowledge the many individuals who participated in the preparation of this dissertation. Without them, it would simply not have been possible.

I freely acknowledge my special debt to Yair Wand who was generous both with his time and his ideas. The basic idea of the ontological approach came from him and his influence is evident throughout the thesis. Professor Richard Mattessich was responsible for rekindling my interest in basic philosophy and metaphysics. I learned much from him. Son Vuong and Carson Woo were the other two members of my supervisory committee and they supported me constantly throughout this project. I also wish to thank Jim Varah, Al Dexter, Paul Gilmore and Fred Lochovsky for their careful reading of the thesis and their many observations and suggestions. They have contributed significantly to the final result. Further, I am grateful for the professional and intellectual environment provided by the faculty and students of the MIS Division at the University of British Columbia.

Finally, I thank Anita and Tetsu. I owe them everything.

Chapter 1

Introduction

The purpose of this thesis is to offer a contribution to a theory of *object-oriented software development*.

More specifically, this thesis proposes a new approach to Information Systems Development (ISD) involving the formalization and extension of the *object-oriented paradigm* to ISD. The result is an ISD model which emphasizes a single, well-defined, theory-based construct, *object*, which provides a central, unifying principle for the conceptual modelling, design, implementation and end-use of Information Systems.

The model is an attempt to address two important issues in the development of Information Systems: (1) the need for suitable theories of Information Systems (IS), in particular for conceptual modelling, and (2) the desire to narrow the "semantic gap" between user views of IS applications and final working systems. This chapter introduces the issues, articulates the research problem, and presents the research plan followed in the thesis.

1.1 Information Systems Development and Conceptual Modelling

The development of Information Systems encompasses a wide range of technical, organizational, and social issues including individual and organizational behaviours, project management, data base design, documentation and software engineering. A vast collection of tools, techniques and methodologies has accumulated over the years to assist IS

designers with this process. Nevertheless, the design and construction of complex information systems is still generally conceded to be difficult and error-prone [Sibley, 1986; Bubenko, 1986; Brooks, 1987]. The study and improvement of Information Systems Development, therefore, continues to be the subject of intensive research effort.

One aspect of ISD which has attracted considerable interest is an approach to systems description known as *conceptual modelling* [Brodie et al., 1984; Kung & Solvberg, 1986].

Traditional ways of understanding and describing information systems usually involve concepts such as programs, processes, files, data flows and other computer-related constructs. These are becoming recognized as being too remote from the problem domain, which typically consists of "real world" entities and phenomena such as people, products, services, and activities. The result is systems which are difficult to understand, validate and use [Brodie, Mylopoulos & Schmidt, 1984; Abbott, 1987].

Conceptual modelling, on the other hand, is based on the premise that an IS is, in part at least, an *abstract model* of some reality underlying the application. Its goal is to discover and refine modelling approaches which allow users and designers to describe information systems in ways which are as "natural" to the application as possible, rather than in terms of the computer; that is, to narrow the "semantic gap" between the application reality and its implementation in an Information System.

The value of conceptual modelling (also called reality modelling, semantic modelling and enterprise modelling) has been widely recognized [ANSI/X3/SPARC; Bubenko, 1980; Essink, 1986; Gibbs, 1985; Kung & Solvberg, 1986; Kung, 1983; Brodie et al., 1984]. For example, Kung & Solvberg [1986] describe the role of conceptual models as providing:

- a common reference framework for users and system designers, ie. a communications tool;
- improved modelling of the reality underlying the application;

- a blueprint for implementation;
- a basis for validation;
- documentation.

And Gibbs [1985] suggests that conceptual models

> simplify the design of complex systems by providing a modelling methodology; give a high degree of logical independence, as is required by evolving applications; document the structure of a system at varying levels of detail; and aid the user in interpretation of data [p. 195].

In general, it has been observed that as the scope of computer-based information systems widens, applications become more complex and comprehensive, and the number of users expands, the value of conceptual models will continue to increase correspondingly [Bubenko, 1986].

1.2 Object-Oriented Modelling

This thesis explores the link between conceptual modelling and an evolving approach or paradigm in software development referred to as *object-centered* or *object-oriented.*

The term 'object-oriented' has been used in a number of contexts including programming languages, database, and knowledge representation. Perhaps the best established use of the term to date is in software engineering and the tradition of the object-oriented programming languages such as Simula, Smalltalk and its derivatives [Robson & Goldberg, 1981; Stefik & Bobrow, 1986; Goldberg & Robson, 1984; Cox, 1986; Booch, 1986]. To oversimplify somewhat, an "object" in this context is a software entity which combines both data, ie. static information, and allowable operations on this data, ie. dynamics or behaviour, into a single construct.

There also exists, however, a more general view of object-orientation: that of a framework for direct and natural representations of the world. In this view, "objects" are seen as a convenient way of directly describing entities in reality. Greenspan expresses the concept as follows:

> In [an object-centered] framework, concepts/entities of the world are represented by units of description called *objects*. The creation, modification, and manipulation of objects are taken to represent the behaviour of their counterparts in the world [1984, p. 3].

In both the programming and the more general representation contexts, important techniques have been developed for dealing with objects such as their interrelation by *properties* and their organization into *classes* and *class hierarchies*. Other techniques allow objects to *inherit* properties or behaviours along these class hierarchies. These fundamental ideas also appear in Artificial Intelligence techniques such as semantic networks, frames and related knowledge representation schemes as well as in Object-Oriented Programming Languages, Semantic Data Models and other forms of information modelling [Quillian, 1968; Greenspan, 1984; Fikes & Kehler, 1985; Gibbs, 1985].

The idea of directly representing real-world constructs as software "objects" has led to a growing appreciation of the object paradigm as a powerful model for software and databases [Greenspan, 1984; Borgida, Mylopoulos & Wong, 1984; Nierstrasz, 1985; Gibbs, 1985; Tsichritzis, 1985; Tsichritzis et al., 1986]. For example, ideas such as object classification and inheritance of properties along class hierarchies appear to have epistemological[1] parallels in how we organize knowledge about the world. For these and other similar reasons, the object-oriented approach has been described as an important development in narrowing the semantic gap between the human understanding of a real

[1]That is, theories of how humans understand and organize knowledge.

world application and the information system as implemented in the computer.

Brooks [1987], for example, views object-oriented programming as allowing

> ...the designer to express the essence of the design without having to express large amounts of syntactic material that add no information content.

Danforth and Tomlinson [1988] have pointed out how object-oriented language concepts make possible

> ...a natural connection between the formal, extensional objects represented by a program and their concrete, intensional representations with memory or a file system [p. 33].

And Wand [1989] poses the question,

> Is the emergence of the object paradigm an *empirical indication* that humans find it easier to describe perceptions of the world through the notion of object?

This growing interest in exploiting the object paradigm for abstract, high-level systems and software description is evident in a wide range of research contexts including *programming languages* [Brookes, 1987; Shriver & Wegner, 1987; Danforth & Tomlinson, 1988], *semantic database modelling*, [Banerjee et al., 1987; Fishman et al., 1987; Hull & King, 1987], *office systems modelling* [Tsichritzis et al., 1987; Nierstrasz, 1985; Woo, 1988] and *software engineering* [Booch, 1986; Cox, 1986].

This thesis follows in the footsteps of this research by extending the object paradigm to the ISD process, first by providing an appropriate theoretical context and second, by formalizing an approach to object-oriented Information Systems Development. A concept of *object* is proposed which can be transported intact from the conceptual model through to the implementation and end-use of the resulting IS.

1.3 Research Objectives and Motivation

The overall goal of the research can be expressed in the following way:

> *To define and formalize a theory-based, object-oriented metamodel* [2] *for describing and developing Information Systems.*

To accomplish this goal, this thesis focuses on two questions.

- *Can the intuitions in the object paradigm be sufficiently formalized into a theory-based, Conceptual Modelling scheme for Information Systems Development?*
- *Can such a formalization be sufficiently operationalized to provide a single, unifying principle for the conceptual modelling, implementation and end-use of Information Systems?*

The motivation behind these questions and the research approach to their resolution are discussed below.

1.3.1 Formalization of Object-Oriented Conceptual Modelling

The first objective of this thesis is to propose a theoretical basis for object-oriented conceptual modelling of information systems.

Many well-known and useful conceptual modelling schemes have been proposed, including the semantic data models such as Entity-Relationship (ER) [Chen, 1976], Taxis [Borgida et al., 1984], RM/T [Codd, 1979], Semantic Database Model (SDM) [Hammer & McLeod, 1981], and the data abstractions of [Smith & Smith, 1977, 1979]. In addition, high-level modelling approaches have been proposed and incorporated into numerous

[2] That is, a model for describing other models. The term "metamodel" or "model" in this case is not meant as a formal mathematical or logical model, but rather a framework or approach for describing and developing information systems.

design methodologies which support the systems development life cycle. Examples include Jackson's Method (JSD) [Jackson, 1983; Cameron, 1986], NIAM [Verheijen & van Bekkum, 1982], ACM/PCM [Brodie & Ridjanovic, 1984], and PAMS [Flint, 1986].

Increasingly, however, the literature has observed that such models and methodologies have proliferated and led to a profusion of analytical constructs such as transactions, activities, events, processes and data flows, but for the most part with little or no theoretical justification [Bubenko, 1986; Floyd, 1986; Sibley, 1986; Wand & Weber, 1987; Wand, 1988; Brachman, 1983; Stamper, 1986]. This has resulted in a number of calls for a grounded theory of information systems development. For example, Floyd [1986] writes,

> Unfortunately, our understanding of the nature of systems development as a whole is haphazard at present and tends to be based on opinions and individual experiences rather than on systematic empirical research. In my view, the lack of a suitable theory about systems development as a whole explains many shortcomings in the existing methods [p. 31].

Bubenko [1986] has stated that "no generally accepted, workable theory of information systems and their development has evolved [p. 295]" and that

> ... there is a great need to 'clean up' this area [sic. IS methodologies] and to introduce and agree upon a coherent conceptual reference model and framework with formally well defined concepts, interactions, etc [p.307].

Wand [1988] observes

> It has been claimed that "hundreds, if not thousands" of methodologies for information systems development exist. All these methodologies have the same purpose - to support the successful development of information systems. Yet, there

> still exists no common set of criteria to compare methodologies. Indeed, when using different methodologies to describe the same system one may wonder how could there be such varying ways to describe it. Are the important constructs activities, processes, data flows, or objects [p. 1]?

And Hull and King [1987] ask

> An interesting question is why the central components of semantic models - objects, attributes, ISA relationships - are necessarily the best mechanisms to use to enrich a data model [p. 210].

Similarly, despite its powerful intuitive appeal, the object paradigm remains largely an *ad hoc* collection of techniques, methodologies and philosophies rooted mainly in common sense practice, metaphor, and simple cognitive views of the world.

The academic and practitioner literature reveals broad interpretations of the types of phenomena which can be described by the term "object" [King, 1989; Nierstrasz, 1987; Danforth & Tomlinson, 1988; Beech, 1987; Wegner, 1987]. Candidates have been proposed which range in diversity from *entity, activity*, and *assertions* [Greenspan, 1984] to *tangible things, roles, incidents, interactions*, and *specifications* [Schlaer & Mellor, 1988]. One practitioner-oriented reference lists as many as 18 separate object categories and dozens of associated relationships and rules [Ross, 1987]. Seemingly arbitrary distinctions are often made among object types such as *concrete* and *abstract* objects, *independent* objects and *characteristic* or *dependent* objects [Gibbs, 1985]. Organizational disciplines for objects such as *inheritance* have also met with controversy and many variations have been proposed including *multiple* and *incomplete* inheritance, *classless* inheritance, *delegation*, and *specialization* [Wegner, 1987; Sciore, 1989].

In addition, although the idea of representing phenomena in the world by the construct "object" is appealing, it is recognized that defining the correspondence between reality

and objects is largely intuition-based [Brodie & Mylopoulos, 1986; Kent, 1977; Stamper, 1986; Brachman, 1983]. The limited research to date in this area has led to complex and profound issues involving cognitive and epistemological questions of how we perceive and organize knowledge of the world into representations. Increasingly, therefore, researchers have called for a better understanding of the fundamental philosophical and metaphysical premises assumed by object-oriented modelling. These include such basic concepts as entities, relations and basic issues of existence itself [Borgida, Mylopoulos & Wong, 1984; Stamper, 1986; Beech, 1987; Wegner, 1987].

The research undertaken in this thesis addresses the call for a stronger theory for IS development by proposing a Conceptual Modelling system which is grounded in a formal *ontology* or theory of reality. This approach is motivated by the premise that any IS conceptual modelling system claiming to represent an aspect of reality should be rooted first and foremost in a theory of reality, that is, a systematic, coherent and consistent metaphysics.

Very briefly, the thesis develops a rigorous description of real world phenomena based on the system of ontology developed by Mario Bunge as part of his *Treatise on Basic Philosophy* [1977, 1979]. Notions such as *thing*, *property*, *state*, *event*, *process*, and *change* as well as a schema for representing reality are defined in strict ontological terms, without reference to the technology and techniques of Information Systems. It is shown how ontology can provide an underlying theoretical foundation for a formal definition of the *object* construct which is suitable for conceptual modelling. The thesis then demonstrates how this construct can be directly utilized in the implementation and end-use of Information Systems.

1.3.2 Operationalization of Object-Oriented Information Systems

Information Systems have a dual nature, that of both representation and artifact. That is, systems are (1) in some significant sense, models or representations of some "slice of reality", and (2) a human-created artifact which participates in reality as an entity in its own right [Essinck, 1986; Jansson, 1985].

Given an understandable conceptual model of some application reality, it is desirable to transform the model into a working IS artifact as directly as possible while preserving intact the essence of the original conceptual representation. In contrast, in the traditional "waterfall" systems life-cycle model [Royce, 1987; Boehm, 1976], systems development proceeds from some abstract specification of the system to increasingly more "physical" implementation in a series of steps. All too often, however, the original semantics of the application are hidden or lost with each succeeding step. The result is a final system consisting of software code, data files, hardware and procedures which are difficult for developers to validate and end-users to comprehend and use.

The proposed conceptual modelling system deals with this issue by supporting a direct *operationalization* of the object-oriented conceptual model into a working IS implementation. The design of the implementation is based on the same ontological principles as the underlying conceptual model. The modelling construct *object* is implemented directly in software and in addition, is directly manipulable by end-users of the system. In this way, the model strives to provide a *seamless, homogeneous* path from the conceptual model to implementation and through to final end-use of systems based on this approach.

To briefly anticipate the work to follow, the proposed model formally defines the systems description construct, *object*, which incorporates ontological concepts of identity, property, state, and change in a single, consistent, and unified entity. An implementation approach is developed whereby the essence of the conceptual model is preserved in the

implemented system. The *object* construct serves as both high level, conceptual modelling units and simultaneously as highly visible artifacts which can be directly manipulated by users of the information system much in the same manner as they would manipulate entities in reality.

In this environment, the users interact directly with the same objects used in conceptual modelling rather than traditional, implementation-level constructs such as files, programs, records, and data fields. As a result, this model provides fundamental support for modes of user-interface whereby users have a sense of *directly manipulating* real-world objects when dealing with the system [Schneiderman, 1980]. This is in contrast to most other approaches to ISD where user-interface is typically treated as a separate implementation issue in systems design with little or no intrinsic relation to the application.

1.3.3 Some Caveats

At this point, it should be recognized that this notion of "seamless" or "homogeneous operationalization" makes a basic assumption concerning the value of conceptual modelling.

Conceptual models are intended to facilitate IS users and developers in perceiving, understanding and describing information systems in terms appropriate to the application rather than in terms related to computer technology or implementation detail. Motivating conceptual models is a basic premise that as system descriptions, implementations and end-user interfaces become more and more separated from the concepts and terminology of the application domain, there is a greater danger of mistranslation and erroneous transformation, hence "semantic gaps", between the original problem and the final IS. The result is increased probability of system failure.

This thesis accepts the need for high-level conceptual models. Its primary purpose is not to support or dispute the basic premise motivating such models. Nor does it attempt

to demonstrate its corollary, ie. that the proposed framework will *necessarily* reduce the probability of IS failures. These issues are viewed as subjects of separate research efforts [Lyytinen, 1987].

What this research does attempt to show, however, is the *feasibility* of an ontology-based conceptual modelling scheme whose basic constructs can be preserved intact through problem analysis, conceptual design, implementation and end-use of an IS. It will do this via theory-building and "proof of concept" prototyping of systems which illustrate the homogeneous path from conceptualization to implementation and end-use. The thesis leaves the problem of evaluating the value and benefits of the proposed conceptual modelling approach to subsequent empirical validation and research.

Further, it should be emphasized that the model presented in this thesis is not intended to be a complete ISD methodology. Practical methodologies are the cumulative result of much experience and development effort by numerous contributors. Popular methodologies are supported by many tools, training programs, project management techniques and a vast literature by both academic researchers and practitioners. As such, the development of an object-oriented ISD methodology is beyond the scope of this dissertation.

On the other hand, this research does attempt to suggest foundation principles upon which future methodologies based on this model can be built. Further, it suggests specific tools, techniques and interfaces which appear suited to this approach.

Finally, it should be noted that this research is based specifically on Bunge's system of ontology. Therefore, the resulting model has a "look and feel" which reflects his general approach, terminology and notation. Moreover, since this thesis builds on Bunge's system as given, with only minor modifications, the model will reflect any controversies associated with his work. In particular, it does not presume to revise Bunge nor to build a new ontology. Since the focus of this thesis to operationalize fundamental concepts of an

existing ontology into an ISD model, such controversies do not affect the main thrust of this research.

It is recognized, however, that ontology has a long and rich history with contributions from many sources. Different ontological assumptions will very likely lead to quite different IS models. Further, existing ontologies, including Bunge's, are continually undergoing rethinking and reassessment. In this light, this thesis recognizes that the model proposed here is only one step toward other possible ontology-based models of Information Systems.

1.4 Research Methodology

The research methodology includes model building, comparative analysis, prototype implementation and case study.

Model building is directed at the first question:

> *Can the intuitions in the object paradigm be sufficiently formalized into a theory-based, Conceptual Modelling scheme for information systems development?*

In addition, some comparative analysis will be performed to evaluate the sufficiency of the model against some of the more important modelling approaches. The second question

> *Can such a formalization be sufficiently operationalized to provide a single, unifying principle for the conceptual modelling, implementation and end-use of Information Systems?*

is interpreted as a feasibility question, and addressed by the prototype implementation and case study. The prototype and case study are seen as providing a test bed for experimentation, development and validation of the theoretic model. In this thesis,

experience with the prototype played an important role in influencing and refining the model over several iterative cycles.

In this respect, this research and in particular, the prototype, can be viewed as an *empirical inquiry* into conceptual modelling in the spirit advocated by Newell and Simon [1975]. According to Newell and Simon, each new machine and program that is built is an experiment. It poses a question from which lessons can be drawn. Thus, each experiment provides the potential for the "development of new basic understanding by empirical inquiry".

> [As] basic scientists we build machines and programs as a way of discovering new phenomena and analyzing phenomena we already know about...[The] phenomena surrounding computers are deep and obscure, requiring much experimentation to assess their nature. ...[As] in any science, the gains that accrue from such experimentation and understanding pay off in the permanent acquisition of new techniques; and [it] is these techniques that will create the instruments to help society in achieving its goals [1975, p. 114].

1.5 Expected Contributions

The overall contribution of this research is to the growing theoretical effort in conceptual modelling and information systems development in general and object-oriented systems in particular. A brief summary of the specific contributions from this line of research includes:

- *A novel, theory-based approach to the conceptual modelling of information systems derived from basic principles in ontology.* This study responds to the cry for greater theoretical grounding in the study of information systems development [Bubenko,

1986; Floyd, 1986; Sibley, 1986; Wand, 1988]. The ontology-based approach is advanced as a means of providing a theoretical basis for the model and a framework for its evaluation. Further, since the model is based on a general system of metaphysics, it has the potential of being applicable to a wide range of systems rather than being limited to specific domains (such as office systems, business enterprises or transactions processing systems).

- *The formalization and extension of the object paradigm to information systems development.* The model contributes to recent interest in establishing better "philosophical" grounding for the object-oriented approach [Danforth and Tomlinson, 1988; Wegner, 1987].

- *An ISD model which reduces the semantic gap between the IS and its underlying application reality.* This study proposes a consistent and homogeneous construct which can be shared by analysts, programmers and end users of information systems. Instead of the typical "waterfall" approach, the proposed model facilitates smooth, equivalence-preserving[3] transformations from the initial conceptual model to final implementation, thereby narrowing the "semantic gap" between the user view of the application and its final implementation in a system.

- *A demonstration of a working prototype of IS based on the proposed model.* Program development methodologies, tools and software engineering techniques for use with object-oriented programming languages are still emerging in the literature and in the field. Although this thesis will not present a complete, working ISD methodology, the prototype implementation will illustrate applications of the proposed object-oriented model by designers, implementors and end-users. In this

[3]That is, the semantics of the original application are carried forward relatively intact in each stage of implementation.

way, it contributes to the growing collection of practical tools, ideas and techniques for working with the object paradigm. This is the major practical contribution of the thesis.

1.6 Thesis Overview and Organization

Chapter 2 of the thesis reviews the development of conceptual modelling and similar abstract, high-level systems description as found in the programming language, data base and software engineering literature. The review also examines the emergence and uses of the object paradigm as described in these various literatures. The common trend toward higher levels of abstract description through constructs such as *object* in all three research areas is analysed.

Chapter 3 presents the ontological background underlying the proposed conceptual modelling approach. This chapter describes previous work in IS which consider ontological principles. Finally, the chapter summarizes the relevant aspects of Bunge's system of ontology.

Chapter 4 formally presents the *Object/Ontology-Based Conceptual Model* (OBCM).[4]

Chapter 5 discusses the implementation framework for creating an *Object/Ontology-Based Information System* (OBIS) from the OBCM. A simple example is used to illustrate the principles.

Chapter 6 describes a working prototype in the programming language Smalltalk.

Chapter 7 presents an OBCM/IS solution to the IFIP Working Conference Problem [Olle, 1982].

Chapter 8 provides conclusions, summarizes the contributions of the thesis and suggests areas for future research.

[4]The model is both Ontology-based and Object-based. The acronyms OBCM and OBIS are intended to reflect both these aspects of the model.

Chapter 2

Conceptual Modelling and Abstraction

The scale and complexity of information systems have resulted in the drive toward higher, more *abstract* levels of system descriptions.

In the common dictionary meaning of the term, abstraction is concerned with the "essence" of things apart from their concrete realizations or instances.[1] The exact nature of this "essence" is elusive, as reflected in the various ways that abstraction is treated by different researchers in Computer Science and IS.

For example, in Computer Science, Goguen et al. [1978] have identified three distinct (although closely related) meanings for the term:

- A mathematical or conceptual model of something as in "abstract machine";
- The process of generalizing and thereby ignoring certain details, and;
- Consideration of a concept apart from its representation or implementation, eg. "abstract data type".

In the areas of Artificial Intelligence and Knowledge Representation, abstraction may refer to methods of representing the *idea* of something (as opposed to representing individual things or sets of things) [Nilsson, 1980]. Abstraction in AI also can refer to techniques for summarizing and simplifying complex representations or descriptive nets [Winston, 1984].

[1] Eg. ...the ideal or theoretical way of regarding things; consider apart from the concrete. *Concise Oxford Dictionary.*

In Semantic Data Modelling, abstraction is generally viewed as a specific set of tools which provide organizational guidelines and support database design methodologies. In this regard, abstraction mechanisms based on epistemological [2] intuitions such as generalization, aggregation, and classification have received considerable attention [Gibbs, 1986; Borgida et al., 1985].

This chapter reviews the background related to abstraction and conceptual modelling in the contexts of programming languages, data modelling, and information systems methodologies. It traces a common shift away from technology-based, implementation-specific levels of system description toward high-level, abstract descriptions in all three disciplines. In particular, it focuses on the emergence of *object-oriented* constructs as an important step in this process.

There are a large number of object-oriented programming languages, data models and IS methodologies in existence. Hence, this chapter is selective in highlighting the major issues relevant to this thesis, namely how the concept of *object* is used in forming high level, abstract descriptions of reality.

2.1 Programming Languages

2.1.1 Programming Abstraction

In the modern view of programming, the themes of high level modelling and abstraction emerge as the principal means of dealing with and controlling the complexity of software development [Shaw, 1984a; Zilles, 1984; Liskov & Guttag, 1986]. Technical innovations such as object-oriented programming are seen as techniques for enhancing the expressiveness of programming languages. They represent significant steps in the "continuing abstraction of programs away from the computer and toward the problem

[2] That is, theories of how humans understand and organize knowledge.

[Abbott, 1987]".

According to Abbott [1987], for example, in the early history of software development, programs were closely associated with the machines on which they ran. That is, application domain information tended to be invisible in the program which existed only as machine-level codes that reflected poorly, if at all, the nature of the application problem. Symbolic and high-level languages, macros and procedures, data structures, data types, domain-oriented languages, and so forth, gradually freed software from dependence on underlying hardware and permitted higher, more conceptual ways of expressing application domain information in programs. As a result, this encouraged the idea of regarding programs not as simply sequences of codes tied to some underlying hardware, but also as *representations* or *models* of a problem.

In modern programming languages, there is an emphasis on techniques for modelling the application through a many-to-one mapping whereby certain details are ignored and similarities highlighted. In this way, complex problems are converted into simpler ones. In addition, where possible, details of implementation are ignored to allow implementation-independent descriptions of programs and program objects. These ideas and approaches have been generally refered to as programming *abstractions.* According to this view, "good" abstractions emphasize significant information and suppress irrelevant or inappropriate details [Shaw, 1984a; Shaw, 1984b].

Some of the earliest software abstraction techniques centered around functions and procedures. For example, Liskov and Guttag [1986] distinguish between abstractions such as *parameterization* [3] and *specification.*[4]

[3]The abstraction of a set (potentially infinite) of computational instances by binding the *formal parameters* of a single program text an arbitrary number of times to *actual parameters* or *arguments* at run time.

[4]The use of a formal specification language or informal program comments or "meaningful" names and labels to hide the details of underlying implementation from the user during the design of the program.

Other programming abstractions are designed to reduce the difficulty of software construction by providing new ways of expressing the application in some *virtual machine* defined by the programming language. For example, *procedural abstraction* permits the extension of the virtual machine by adding new operations as required by the application problem. *Data abstraction* allows the virtual machine to be enhanced with user-specified data objects and operation sets characterizing their behaviour. Overall control of procedures over data objects is conveniently expressed by familiar *iteration* or *control abstractions* such as **if-then-else**, **for all-do**, **for each-do**, and **while-do** [Shaw, 1984a; Shaw, 1984b; Liskov & Guttag, 1986].

Together with the emergence of high-level and domain-oriented languages, these techniques can be viewed as a means of freeing computational models from underlying hardware architectures, providing the means of expressing programs in domain terms, and generally establishing programs as objects of study independent of the hardware on which they executed [Abbott, 1987].

2.1.2 Object-Oriented Programming

The development of abstract data types (ADT) and object-oriented programming represents a further important step toward the expression of programs in problem-domain terms and constructs. These concepts allow the creation of program units defined by a name, a set of possible values, and a set of operations rather than storage structures, program text, and other details of implementation [Brooks, 1987].

The virtual machine can thereby be populated with software units which, in theory, can be designed to reflect the static and behavioural characteristics of entities in the application domain. The goal is to improve the programmers' ability to deal with the essence of the problem and ignore those syntactic and machine-oriented issues which are not directly related to the problem at hand. Languages which stress these concepts are

generally known as *object-oriented.*

There exists a widely acknowledged confusion surrounding the term *object-oriented programming language* (OOPL) [Nierstrasz, 1986; Hailpern & Nguyen, 1987; Stefik & Bobrow, 1986; Pascoe, 1986]. The term *object-orientated* has not always been clearly distinguished from associated concepts such as abstract data types, information hiding, dynamic binding, inheritance, and class hierarchies. The literature on the topic has applied the term to a variety of languages including Smalltalk, Simula, Ada, and Modula [Stefik & Bobrow, 1986]. Object-related extensions and support have also been claimed for a large number of existing, more conventional languages including Pascal [Jacky & Kalet, 1987], Fortran [Isner, 1982], C [Cox, 1986; Stroustrup, 1986], Lisp [Stefik & Bobrow, 1986l and even COBOL [Bassett, 1987]. Stefik & Bobrow [1986] in their 1986 survey on the topic have estimated that over fifty languages exist which purport to be object-oriented.

Nierstrasz [1987] stresses that object-orientation is an *approach* rather than a set of specific language constructs. He identifies data abstraction, independence, message-passing paradigm, inheritance, homogeneity and possibly concurrency as hallmarks of this approach. Pascoe [1986] considers information hiding, data abstraction, dynamic binding, and inheritance as the critical of object-oriented languages.

Wegner [1987] lists seven "dimensions for objects-based language design": objects, classes or object types, inheritance, strong typing, data abstraction, concurrency, and persistence. He applies the term "object-oriented" only to those languages which possess the first three features, ie. objects, classes and inheritance, thus narrowing the use of the term. According to this definition, objects are considered to consist of a set of operations and a state that remembers the effects of the operations. Classes specify an interface for operations and serve as templates for object instances. Class inheritance is viewed as a mechanism for relating certain classes, ie. inheriting classes called subclasses and

inherited classes called superclasses.

Regardless of differences in terminology, the ideas behind object-orientation have been applied against many problem areas in software engineering and programming. The localization, or encapsulation, of data and operations into a single unit is said to promote more effective modularization of programs [Cox, 1984; Cox, 1986]. Object-oriented programming (OOP) has been claimed to smooth program design and implementation [Halbert & O'Brian, 1987; Wegner, 1987; Booch, 1986]. Object-oriented languages have also been proposed to address issues in reusability [Meyer, 1987; Woodfield et al., 1987], program readability [Goldberg, 1987], experimental prototyping [Diederich & Milton, 1987], and user interaction, particularly iconic interfaces [Cox & Hunt, 1986; Ledbetter & Cox, 1985].

Object-oriented programming languages have stimulated much research interest with respect to their conceptual modelling potential. For example, data abstraction, a cornerstone of object-orientation, provides two key modelling concepts: (1) *type*, a precise characterization of structural or behavioural properties shared by a collection of entities; and (2) *identity*, whereby a specific instance (of a type) can be distinguished from all other instances. Some researchers have viewed abstract data types (ADT) as an algebra, ie. a collection of types together with a family of operations closed upon those types, capable of capturing much of the behavioural semantics of an application [Deutsch, 1981; Zilles, 1984; Goguen et al., 1978].

Object-orientation research appears to have shifted attention away from many traditional concerns in programming (eg. data structures, algorithms and proofs of correctness) and toward other issues such as type theories, object classification, class hierarchies and class inheritance [Halbert & O'Brian, 1987; Wegner, 1987; Beech, 1987]. For instance, Wegner [1987] has suggested a possible new basis for the study and analysis of

programming, one which stresses the historical development of hierarchical class structures and incremental evolution of individuals and classes in the domain of discourse.

> The description of programs by their derivation history rather than by their execution properties has become a central goal of software engineering... Since historical evolution more accurately reflects the intellectual and social processes of program development than proofs of functional correctness, its formalization might identify primitives and inference rules more relevant to software technology than those of current program verification systems. Such formalization presents formidable problems and is still in its infancy. Object-oriented programming provides a starting point for a technology of evolutionary program development... [p. 482].

2.2 Semantic Data Modelling

In the database literature, the complexity of applications and the large numbers of data elements have resulted in researchers turning to high-level, abstract models for organizing and exploiting databases. Many of these constructs have been borrowed from developments in programming languages and artificial intelligence theory. As in the case of programming languages, a consistent theme is the gradual abstraction of data models from implementation and technology-based models to higher-level, conceptual models.

2.2.1 Semantic Data Models

Brodie [1984], Bubenko [1980], Kent [1978], Gibbs [1985], Hull and King [1987] and others have documented the evolution of data models toward ever "higher" levels of conceptualization. Successive models rely less and less on the computer-related mechanisms for representing and storing data (such as files, records, pointers, "navigation paths") and

strive to describe the data in terms of their "natural", real world structure and underlying meanings. The trend is from the "classic models" (hierarchical, network and relational) to data models which are increasingly semantically-based. Afsarmanesh and McLeod [1986] illustrate this progression as follows:

- Extensions to the relational model such as RM/T [Codd, 1979];
- Incorporation of abstraction mechanisms such as *aggregation* and *generalization* [Smith & Smith, 1977a, 1977b];
- Binary database models [Abrial, 1974];
- Functional database models;
- Entity-relationship models such as E-R [Chen, 1976] and Semantic Database Model [Hammer & McLeod, 1981];
- Behavioural models such as Taxis [Greenspan & Mylopoulos, 1983] and ACM/PCM [Brodie & Ridjanovic, 1984];
- Semantic-object-based models [Afsarmanesh & McLeod, 1986].

Brodie and Mylopoulos [1986] have argued that this trend represents a fundamental shift in information processing, from a primarily *computationally-based* to a *semantic* theory of information. The computational theory is concerned with supporting large amounts of data shared among many users and gives rise to issues such as data structures, concurrency, query languages, normalization, and integrity constraints. The semantic theory focuses on the representation of knowledge by applying such techniques as mathematical logic, semantic nets and frames. Throughout this evolutionary development, the goal has been to characterize the database and its operation in application-domain or

"real-world" terms, in short, to capture more semantics in the database. For example, the system should support the real world concept of "hiring" an employee rather than the technical operation of "inserting" an employee record into the database [Ledgard & Taylor, 1977].

Many of these alternative models are beginning to explicitly take on object-oriented constructs borrowed from Artificial Intelligence, ie. semantic network, frames and other knowledge representation schemes, as well as the object-oriented tradition in programming languages. The basic idea is to provide constructs which have direct correspondence to entities (or perceived entities) in the world of the application. These constructs (modelled as frames, semantic network nodes, objects, etc) are interrelated and manipulated to reflect corresponding situations in the real world. This basic idea is refined and extended in various proposals for semantic data models which are designed to provide better representations of real world phenomena and to achieve more "natural" manipulations on the data [Smith & Smith, 1977; Tsichritzis & Lochovsky, 1982; Brodie, 1984; Hull & King, 1987].

Generally speaking, most semantic models support direct representation of objects or entities, as distinct from indirect representation by attribute values [5]. They generally support constructs such as **IS-A** which determine object attributes and relationships among objects. Semantic data models make heavy use of abstract data types including *primitive* or *atomic* types as well as *constructed* and *derived* types, that is types constructed or derived from existing types or data in the system. Many models recognize that entities in the data base may not be uniquely identifiable with printable attribute values or keys, therefore they reference entities with internal identifiers not visible to the user.

[5] In a relational database, for example, a real world entity is typically represented as a collection of attribute values within a tuple.

Most semantic models support some concept of *attribute* or directed relationship among types. Different semantic models vary in the kinds of mechanisms used for interrelating types. Some emphasize the explicit construction of types while others may rely heavily on attributes. For instance, Figure 2.1, adapted from Hull and King [1987], shows four alternate ways of modelling the type ENROLLMENT. In Figure 2.1a, ENROLLMENT is an aggregation of types COURSE and STUDENT with the attribute GRADE.[6] Figure 2.1b views it as a ternary aggregation with no attributes. In Figure 2.1c, ENROLLMENT is an atomic type with three attributes. Finally, Figure 2.1d does not explicitly model the type ENROLLMENT but simply considers GRADE to be an attribute of the pair STUDENT and COURSE.

Data definition and manipulation languages have been proposed for specifying and using semantic models. In contrast to traditional record-oriented models, languages for semantic models must directly support the query and management of types and instances. Most semantic data models have concentrated on modelling the structural aspects of data, ie. the types and interrelationships among types. However, some semantic models have also explored facilities for handling dynamic and behavioural aspects of data [Brodie & Ridjanovic, 1984].

Comparative descriptions of the above features as well as specific proposals for semantic data models are described in detail in a number of sources including Hull and King [1987], Brodie [1984], Peckham and Maryanski [1988], and Tsichritzis and Lochovsky [1982].

2.2.2 Abstraction Mechanisms

In order to reduce the complexity of large data bases to manageable dimensions, abstraction mechanisms are typically used to organize objects by extracting essential and hiding

[6] In Figure 2, the aggregation of types is represented by dashed lines and attributes by solid lines.

irrelevant information. Many of these mechanisms have been inspired by theories about the way humans appear to represent and organize knowledge in their minds and the way they seem to express reality in language. Three specific organizing mechanisms, *classification, generalization* and *aggregation*, have received particular attention in database research.

Classification

Classification is usually defined as an abstraction in which a single construct, variously called *class, type* or sometimes *set*, is used to describe a set of related objects. The notion of a class implies both a property structure, ie. the relevant common properties which make the objects similar, and a set, ie. the objects in the class [Brodie, 1984; Gibbs, 1985].

Members of a class have an *instance-of* relation to the class. The reverse of classification is sometimes referred to as *instantiation*.

Generalization

Generalization is normally defined as an abstraction which organizes classes into useful hierarchies [Gibbs, 1985]. Generalization is often referred to as an IS-A relationship between a *specialized* object and some higher-level, generic one. The idea of hierarchy is emphasized as well as the *inheritance* of properties by the specialized object from the generic [Brodie, 1984].

The concept of generalization, however, is not straightforward and a number of controversies have been raised regarding its precise meaning as well as the notion of inheritance as a principle for organizing knowledge [Brachman, 1983].

Aggregation

Aggregation refers to an abstraction whereby a single object is viewed as a collection or aggregate of several other objects. Several distinct interpretations of aggregation have been proposed including Cartesian, cover and statistical aggregation.

Cartesian aggregation [Codd, 1979; Smith & Smith, 1977a], considers aggregations of properties, eg. the class PERSON is an aggregate of properties such as NAME, ADDRESS and other relevant properties.

Cover aggregation [Codd, 1979] abstracts an object as an aggregation of other objects. Both Cartesian and cover aggregation suggest a *part-of* relationship between the individual parts and the aggregate.

Statistical aggregation provides summary information related to a database such as the number of instances and averages of particular attribute values.

2.2.3 Object-Oriented Data Models

In many ways, the concepts in semantic data modelling are closely related to those found in object-oriented programming languages. Consider, for example, the following features of OOPL [Nierstrasz, 1986]:

- Encapsulation of statics and dynamics (eg. as instance variables and methods);
- Class hierarchy mechanisms;
- Homogeneity.

In fact, Maier and Stein [1987] have suggested that "an object-oriented language is complete enough to handle database design, database access, and applications". They offer a database model, Gemstone, and a language, OPAL, which are closely derived from

Smalltalk. Other models have been reported which allow data items to have attributes which can be database queries or even application programs [Hull & King, 1987]

Not surprisingly, some recent work in database research has been inspired directly by the object-oriented programming paradigm. Systems such as ORION [Banerjee et al., 1987], IRIS [Fishman et al., 1987], OMT [Blaha et al., 1988] and Gemstone [Maier & Stein, 1987] have explicitly acknowledged this relationship. One result of these trends is the gradual blurring of past distinctions between the more-or-less static aspects of data base design and the more dynamic aspects of programming and data query languages.

Afsarmenesh and McLeod [1986] have summarized these trends in semantic database modelling as follows:

- From a record-orientation to an object orientation
- From strictly static views of the database to accommodating dynamics
- From separate mechanisms for handling data and meta-data to a uniform treatment of all information.

Appendix F briefly reviews, without comment, three selected data models which are representative of these trends and illustrative of the wide variety of ways in which object-oriented concepts have been incorporated into data modelling. The first, by Kroenke and Dolan, focuses the object paradigm as a way of unifying disparate data base concepts. The second, ACM/PCM illustrates the incorporation of dynamics into a data model. The third, PAMS, is an attempt at integrating all aspects of data modelling, including both structural and dynamic modelling, in a consistent and homogeneous manner.

2.3 Systems Development Methodologies

2.3.1 Early Approaches to Information Systems Modelling

As with early approaches to programming and data base, traditional Information System Development methodologies usually reveal a strong technology orientation in their descriptions of information systems: files, processors, programs, data flows, inputs, outputs and so forth.

Important methodologies often emphasize one or another aspect of this focus. Data flow diagrams, for example, stress the transformation of data by different processes. Warnier-Orr [Warnier, 1981; Orr, 1982] and JSP [Jackson, 1975] on the other hand, relate data structure to program structure. And early formalizations of information systems modelling focused around abstract notions of inputs, outputs, files, procedures and other constructs closely related to systems implementation [Young & Kent, 1958; Langefors, 1966].

2.3.2 Modelling an Universe of Discourse

The explicit separation of logical versus physical design, the development of "business analysis" techniques (eg. BIAIT [Carlson, 1982] and BSP [IBM, 1982]) and the "top-down" methodologies reflect a shift to abstraction away from technology and implementation issues and toward the real-life situation motivating the IS. As noted by Bubenko [1986], most modern methodologies now employ some form of high-level, conceptual description of the information system, with varying degrees of formalism, the main purpose of which is to specify the system in machine-independent, user-oriented terms. An important theme of most modern ISD methodologies is that "information systems maintain a formal, symbolic model of some concrete and/or abstract universe of discourse (UoD) [Bubenko, 1986]".

This idea, which has only been explicitly articulated relatively recently [Wand & Weber, 1988; Bubenko, 1986; Essink, 1986; Greenspan, 1984], makes a strong and clear distinction between descriptions of the problem situation and descriptions of the final implemented system. The result has been to concentrate more and more upon the earlier stages of the ISD process in the conviction that functional requirements and implementation cannot be understood without first considering an appropriate model of the application or enterprise [Bubenko, 1986; Jackson, 1983].

Typical of these approaches are methodologies such as JSD [Jackson, 1983] which stresses explicit "reality modelling" of entities in an Universe of Discourse and NIAM [Verheijen & Van Bekkum, 1982] which exploits linguistic decomposition of sentences describing some real-world situation. Unlike most earlier methodologies, they also attempt to provide high-level descriptions of both structural and activity aspects of the enterprise.

Formal (mathematical) approaches to abstract information systems modelling have also been considered. Bubenko [1980] has proposed the Conceptual Information Modelling (CIM) which is based on first-order logic. Greenspan has devised a formal language, RML, using first-order logic extended to handle concepts dealing with time. These, and similar models are intended to generate high-level, formal descriptions of requirements modelling from a problem-domain perspective [Greenspan, 1984; Borgida, Greenspan & Mylopoulos, 1985].

Because of their explicit emphasis on "reality modelling", many of these approaches bear a strong flavour of *simulation.* That is, information systems are seen as mapping to real-world phenomena and states and tracking real-world events [Wand & Weber, 1988]. This is in contrast to the more conventional view of information systems as consisting of programs, computational processes, procedures, and data bases. It is evident that computer-based models or simulations of things in the real world can become extremely detailed. For example, Foley [1987] describes how advances in computer technology

have been exploited to generate sophisticated "artificial realities" which act as powerful, compelling models of some aspect of reality.

A good example of a systems development methodology which centers around "reality modelling" is Jackson's Method (JSD) [Jackson, 1983; Cameron, 1982; McNeile, 1986]. JSD insists upon defining an initial model of the part of the real world of interest even before functional requirements are fully analyzed. Functional requirements are then added to this model at a later stage.

> It is a fundamental principle of JSD that the developer must begin by modelling this reality, and only then to go on to consider in full detail the functions that the system is to perform. The system itself is regarded as a kind of simulation of the real world; as the real world goes about its business, the system goes about simulating that business, replicating within itself what is happening in the real world outside. The functions of the system are built upon this simulation; in JSD they are explicitly added in a latter development step [Jackson, 1983, p. 4].

JSD takes the position that although the model generated in this phase has no functionality (except to faithfully simulate real world behaviour), the model *implicitly defines* a complete range of possible system functionality. The methodology recommended for JSD consists of

- Identifying and defining real world entities and actions 'performed or suffered' by these entities.
- Creating an initial 'process model' in terms of the entities and their actions in terms of communicating sequential processes [Hoare, 1985].
- Adding functionality, mainly in the form of system outputs such as reports.

- Operationalizing the process model and outputs on appropriate hardware, software and data base technologies.

2.3.3 Object-Oriented Software Engineering

In the software engineering literature, developments in object-oriented programming languages have stimulated activity in object-oriented software development approaches which reflect OOPL principles. These approaches are generally viewed as partial lifecycle techniques which focus on the design and implementation phases of software development [Booch, 1983; Booch, 1986; Buzzard & Mudge, 1985; Meyer, 1988].

Booch [1986], for example, suggests that requirements analysis techniques such as JSD or SREM [Alford, 1985] can be integrated into object-oriented program design. The basic approach is to identify and represent real world entities with abstract data types or OOPL objects.

- Objects and their attributes in the real world are identified.
- Operations suffered by and required of each object are identified.
- The relationships among objects are established.
- Object interfaces are described by producing a module specification which can then be implemented in a suitable program module such as an ADA package [Booch, 1986].

2.3.4 Object-Oriented Methodologies

Recently, various database design and information system analysis methodologies which have clearly been influenced by the object-oriented programming paradigm have been

proposed by academics and practitioners. In many cases, these are basically "front-ends" to relational data models, data flow [de Marco, 1977] representations and other more conventional methodologies. In some cases, they are practical methodologies based on modelling approaches such as Entity-Relationship. For example, the Kroenke and Dolan proposal provides for a conceptual level of modelling in which the universe of discourse is first represented in terms of "objects". A translation process then converts the object-based model into a conventional relational data model.

To take another example, Blaha et al. [1988] have proposed a design methodology, Object Modelling Technique (OMT), for supporting the design of relational databases. OMT supports generalization and aggregation abstractions and is used to generate an abstract, high-level description of the problem domain in terms of objects. A high-level graphical notation further supports the methodology. This representation is then converted to a middle-level, DBMS-independent set of generic relational tables. This middle-level decouples the mapping of objects from the idiosyncrasies of a specific DBMS. Finally, the middle-level is converted into the tables of a specific DBMS. Blaha et al. claim that much of the conversion can be automated. OMT is seen as providing two important advantages over other relational database design methods:

- Objects provide a higher level of system description than expressing the application directly in tables (eg. via SQL) or even the more higher-level representations such as Entity-Relationship or Logical Relational Design Methodology (LRDM) [Teory et al., 1986];

- The use of objects in database design facilitates the integration of the database and procedural code expressed in object-oriented programming languages.

Shlaer and Mellor [1988] have presented a similar methodology which they call *Object-Oriented Systems Analysis.* Their approach appears to be closely derived from the Entity-Relationship model and employs three basic constructs: *object, attribute* and *relationship.* They provide some additional constructs for classifying and associating objects as well as concepts such as object *life cycles*, *states*, and *state transitions.* Traditional data flow diagrams are advocated for analyzing and describing state transformations.

Ross [1987] has described a methodology which he calls *Entity Modelling* which again appears to be directly inspired by the Entity-Relationship model. Entities are equated to things or objects in the real world and guidelines provided for identifying, naming and associating objects. Ross further introduces the concept of *Behaviour Modelling* which attempts to integrate dynamics into the data model. Included in behaviour modelling are various integrity rules, entity life-cycle transitions and triggers which attempt to describe and represent the "business rules" of the application.

2.4 Discussion

2.4.1 Conceptual Modelling and the Object Paradigm

The preceding sections have described some important trends in conceptual, or high-level modelling as found in programming, database design and systems development methodologies:

- *Abstraction Principle.* Separation from implementation issues (computer technology, program code, data storage detail, etc);
- *Reality Modelling.* Support for describing and organizing information on an epistemological or human conceptualization basis (generalization, aggregation, classification, etc);

- *Representation Principle.* Simulation or modelling of the Universe of Discourse rather than a strict computational or functional model;

- *Mapping Principle.* Direct correspondence between real world entities and representation or software units such as "objects".

Overall, there appears to be a general trend in the programming language, database, and systems development areas to include explicit, formal constructs to capture human perceptions and intuitions about the domain of discourse.

This trend is characterized by employing constructs which provide direct correspondences between aspects of reality and the model. These include constructs such as entities, attributes, relationships and their many refinements and derivatives which are independent of specific technologies or implementations. Further, modern technical innovations such as *abstract data types*, *objects* in programming languages, and *frames* in artificial intelligence languages appear to offer the potential of allowing the direct operationalization of conceptual modelling constructs in hardware or software.

Therein appears to lie much of the intuitive appeal of the object-oriented approach to conceptual modelling.

2.4.2 Issues in Object-Oriented Modelling

Although the object paradigm appears to offer powerful modelling and representational capabilities, it also gives rise to methodological and conceptual difficulties.

For one thing, the mappings between reality and their representations are not clearly understood. As Brodie and Mylopoulos observe, "...the semantics of most data models and database languages is intuitive and lacks formal definitions [1986]". The search for improved modelling precision is evidenced by the proliferation of different semantic model proposals and the profusion of modelling constructs: activities, events, processes, data

flows, entities, attributes, and so forth [Gibbs, 1985; Hull and King, 1987].

Further, the object-oriented modelling approaches reported in the literature appear to rely upon past practice, intuition and *ad hoc* rules of thumb in defining the basic constructs. In the requirements modelling language RML, for example, Greenspan [1984] proposes three object types: *entity, activity,* and *assertion.* Nelson [1982] presents an object-oriented prototyping approach which identifies four different object categories: *entity object, process object, event object* and *relationship object.* Often, there is an apparent failure to distinguish between objects as constructs for representing things in the real world and programming or software constructs [Wand, 1989].

In the practitioner-oriented literature, Ross [1987] shows six categories of objects: *people, places, things, organizations, concepts,* and *events.* At a later time, *measurement* is recognized as a seventh category. Intuitive distinctions are made between *things, concepts, events, relationships,* and so forth. Altogether, eighteen object types, nine relationship types and dozens of rules for their interaction are listed. Similarly, Shlaer and Mellor [1988] define five categories of objects: *tangible things, roles, incidents, interactions,* and *specifications* [p. 15]. Yet, in most cases, the choice of these categories appears to be basically *ad hoc* with little or no theoretical justification or rationalization.

On a more fundamental level, there has been discussion concerning the validity of different representations of reality as well as the nature of reality itself in the context of IS modelling. Kent [1978, 1977], for example, has provided especially compelling accounts of the difficulties in making consistent and coherent semantic correspondences between reality and data modelling constructs, including both record-oriented and semantic constructs such as Entity-Relationship. The ambiguities of abstraction mechanisms such as classification, aggregation and generalization and difficulties with semantic nets have been noted by several writers [Gibbs, 1986; Brachman, 1983; Kent, 1978; Israel & Brachman, 1984; Stamper [1986]; Brodie & Mylopoulos, 1986].

A basic question is how well these modelling constructs reflect human mental processes or reality in the world. Although such abstractions have been found to be powerful and useful, it would be naive to claim that they completely capture all aspects of the application, especially given that the mental and linguistic processes upon which they are based are so poorly understood. Indeed, much empirical evidence from the cognitive sciences has been presented which challenge the basic assumptions underlying these ways of organizing knowledge and information [Lakoff, 1987].

2.4.3 Theoretical Foundations for Object-Oriented Modelling

These concerns have led to several research directions in investigating theoretical foundations for the object-oriented paradigm. Among them can be included the *anthropromorphic*, *abstract data type* and *metaphysical* approaches.

The Anthropromorphic Approach

Danforth and Tomlinson [1988] observe that most programming languages incorporate "a metaphor in which computation is viewed in terms divorced from the details of actual computation". In some programming paradigms, such as logic programming or functional programming, the metaphor is usually mathematically precise. In the case of object-oriented programming, however, they see the metaphor "generally expressed in philosophical terms, resulting in a natural proliferation of opinions, concerning exactly what OOP really is". For Danforth and Tomlinson, the heart of the metaphor is the anthropromorphic view in which objects have identity, ie. a *self*, that persists over time and can respond intelligently to requests or *messages* addressed to them.

This anthropromorphic metaphor is frequently invoked in the context of object-oriented programming, and particularly in the practitioner-oriented literature. For example, the language manual for *Actor* [Whitewater, 1987], describes object-oriented programming languages in this way:

> "To many indigenous cultures, the world is populated by entities. Mountains, plants, animals, bodies of water are all governed by spirits that oversee the operation of that particular aspect of the world. This is directly opposed to the "civilized", western world view in which human beings are the only active entities around, and everything else is there for man to use for his own ends.
>
> Oddly enough, programming in Actor seems to have a lot in common with the first, animistic world view. In more traditional languages, we approach programming as writing a lot of code to do all the things that have to be done. The code is the rocks, plants, bricks and mortar that the programmer uses to build structures. The programmer is the only active entity, and the code just basically a lot of building materials.
>
> Object-oriented programming is more like creating a lot of helpers that take on an active role - a spirit - and form a community whose interactions become the application. When you design a class, you can think of the class as an expert or consultant that you can then use again wherever you need its specific expertise. Because of the loose coupling between classes, there is a high likelihood that you will be able to use them in more places than you had originally planned. After a while, classes become like old friends that you know are reliable, and always there when you need them [p. 380]."

To quote another example, the manual for the *Smalltalk V* programming language introduces OOPL in the following way.

> ...What could be more natural. We experience our world largely as a vast collection of discrete objects, acting and reacting in a shared environment.
>
> At the human social level we are a society of doctors, lawyers, beggars and thieves, etc. Although we are a population of unique individuals, we cluster in occupation groups based on the behavioural skills and knowledge we each develop and exhibit...
>
> Break a leg, call in a doctor and tell him or her about your condition. You trust the doctor's special knowledge and skills to help make you better...
>
> Want to become a lawyer? You learn the law and how to behave like a lawyer. Then as corporate counsel in response to the MagaCorp CEO's questions, "What's our exposure to this new project?", your answer is couched in legal considerations while the chief financial officer reflects on fiscal impacts.
>
> In Smalltalk's object-oriented terms, occupational abstractions like doctor, lawyer, programmer, etc., are *classes* of which we individuals are *instances.* To become a lawyer, we learn legal *methods.* Communications between individuals are comparable to Smalltalk *messages*, their context equivalent to Smalltalk *selectors*... Correspondence between our perception of the world and its representation in machine terms through Smalltalk gets at the heart of Smalltalk's power [pp. 11 - 13].

This anthropromorphic image has been pursued in detail by Tsichritzis [1985] who has proposed an environment in which objects called *KNOS* (for KNOwledge objects) behave according to analogies taken from the animal world: birth, movement, communication, and death. Tsichritzis, Fiume, Gibbs, and Nierstrasz have elaborated on some technical details for implementing KNOS in [Tsichritzis et al., 1987].

The Abstract Data Type Approach

Danforth and Tomlinson stress the need for well-grounded theories of OOP concepts such as object types and inheritance. In their opinion, a powerful type theory [7] in OOP, for instance, can provide potential benefits such as

- a uniform framework for the entities in a programming language;
- a means of relating and representing the *denotations* or meaning of program-level expressions;
- a framework for more "natural" connections between the application domain-level entities and their actual computer-based representations.

They point out that current type theories do not appear to provide a formal, consistent basis for supporting the objects and behaviours observed in OOP, and that most object-oriented programming languages attempt to provide such benefits, in *ad hoc* ways, for example, with special notations or language-specific mechanisms.

Another important OOP concept, inheritance, also poses theoretical issues as noted by several researchers [Danforth & Tomlinson, 1988; Brachman, 1985; Minsky, 1989]. Normally, inheritance reflects special relationships between entities. But not all the properties of a receiver are necessarily inherited. Nor are all the properties of a giver necessarily bequeathed. What formal theory, then, is appropriate for explaining inheritance including issues such as *multiple inheritance*? Depending upon the relationships supported by OOP (eg. **is-a**, **part-of**, etc), many different disciplines and restrictions on inheritance would appear to possess face validity. In most cases, object-oriented programming languages have provided their own disciplines and mechanisms related to inheritance, without considering any formal underlying models.

[7] Types in programming can be thought of as a way of describing the possible values in a computation even though they may not be known in advance [Danforth & Tomlinson, 1988].

Metaphysical Approaches

Recently, some researchers have looked to metaphysics to provide foundations for the object paradigm.

In the case of object-oriented database models, Beech [1987] argues that given the current state of our understanding, these models must still be developed informally. He looks to developments in metaphysics and philosophy such as recent axiomatizations and formalisms in the metaphysics of entities, relations and existence [eg. Parsons, 1980; Zalta, 1983] for providing a basis for a more rigorous validation and evaluation of object-oriented models.

In the search for better theories for object-oriented programming, Wegner [1987] observes that seemingly straightforward object-oriented concepts such as classification and types actually have rich and complex philosophical implications. He points out, for example, how classification can be viewed in terms of:

- evolutionary hierarchies and mathematical equivalencies;
- inheritance relationships;
- formalisms such as lambda calculus and algebras;
- metaphysical concepts such as "natural kinds" and the nature of reality.

He also observes that object-oriented concepts reach deeply into important philosophical controversies such as those concerning *typed* versus *untyped* universes. That is, while certain "realists" believe that types are man-made abstractions for organizing real world phenomena, there is also a strong philosophical tradition which stresses the primacy of abstractions such as types, ideals and categories. For example, Platonic *ideals* and Kantian *a priori* categories can be seen as possessing a greater reality than their often imperfect

specific instances. This raises questions about, for instance, whether *types* or *values*, ie. instances, should be the starting point for an object-oriented modelling scheme. In this context, therefore, Wegner sees simple, intuitive notions of classification and types as inadequate in developing truly powerful and comprehensive logical and computational formalisms or models of reality.

A recent step in developing more powerful models of Information Systems is the work of Wand and Weber who have drawn upon first principles taken from the philosophy of science [Wand & Weber, 1988; Wand, 1989]. Similarly, this thesis attempts to apply first principles in formal metaphysics (or ontology) to understanding and clarifying the object paradigm in the context of Information Systems. The next chapter provides the motivation and background for this approach.

Chapter 3

An Ontological Basis for Conceptual Modelling

3.1 Motivation for the Ontological Basis

This chapter lays the background for an object-oriented ISD model which is grounded in a formal ontology or theory of reality. More specifically, this model can be viewed as an *operationalization* of a substantial metaphysical system as formalized by Mario Bunge in his work, *Treatise on Basic Philosophy* [1977], published in seven volumes. The advantages of this approach for ISD include the following:

- the resulting model is more likely to be "in the world", rather than influenced by technology or implementation considerations;
- there is a theoretical basis for the selection of the constructs used in describing and implementing the information system;
- the completeness and consistency of the model can be established against the standard set by the underlying ontology.

Surveys of existing conceptual modelling approaches reveal that they are generally rooted in a combination of one or more of the following [Lyytinen, 1987]:

- process and functions (activities that process, transform, store, access and modify data);
- implementation terms (eg. files, programs, records);

- technological innovations (eg. abstract data types, objects, frames);
- mathematical formalisms (eg. first order logic); or
- intuitions, experience and common sense knowledge about the world.

This thesis, however, takes a different approach to conceptual modelling. It proposes that since the purpose of a conceptual model is to represent aspects of reality, the constructs in the model should, therefore, be based upon a systematic and preferably formal theory of reality itself. The goal is an IS conceptual modelling scheme that fits a given theory of the world, rather than trying to fit the world to some theory of representation [Gibbs, 1985].

In the philosophy of science, the study of theories or systems of reality is known as *metaphysics* or *ontology*. To function as a basis for an IS conceptual model, a system of ontology is required which is sufficiently formalized, complete and compelling. This thesis proposes that Bunge provides one such formalization which can be adapted to a formal model of IS conceptual modelling.[1]

The choice of Bunge will be discussed in more detail shortly. For now, it is suggested that his ontology results in the desired requirements as follows.

- Bunge's system represents a view of the world independent from any existing technological considerations related to computing or information systems. Hence, a model based strictly on his system is more likely to be free from technological or implementation influence.
- Bunge provides a rigorous, consistent formalism and axiomatization for the basic concepts in his system. Hence, there exists a well-defined foundation for the

[1]This is not to imply that other suitable ontological systems do not exist. In fact, it is recognized that other, possibly quite different metaphysical systems for conceptual modelling are possible, probably likely.

fundamental constructs in the conceptual modelling scheme to be proposed.

- The proposed model operationalizes Bunge's theory of reality. His ontology, therefore, provides a ready basis for evaluating the completeness of the proposed model as a way of describing reality.

- Bunge's ontology provides a foundation for the implementation and end-use of information systems based on the proposed conceptual modelling approach.

To summarize, the proposed model is motivated by the premise that an information system which purports to represent reality must make assumptions about the nature of reality itself. Therefore, as representations become more conceptual and technical innovations make possible increasingly more powerful modelling possibilities, the nature of this reality needs to be explicitly articulated and its implications clearly understood. Ontology provides one such articulation.

For comparison, it is useful at this point to briefly note the difference in emphasis between this approach and models based on formal or mathematical logic. It is accepted that mathematical logic supports a clean, straightforward, and well-accepted semantic theory once the connection is established between real-world phenomena and the individuals and predicates in the logical expressions [Mendelson, 1964; Tarski, 1941]. However, the task of making this connection is a matter of intuitive and imaginative *interpretation.* Logic says nothing about the nature of reality itself. Here, philosophy in general, and ontology in particular, is useful since they are concerned with the basic nature of the world.

Indeed, there is an important tradition in the philosophy of science which addresses issues in representation, reality and man's relation to reality [Hacking, 1983; van Frassen, 1980; Lakoff, 1987]. A theme of this literature is concerned with how basic assumptions about reality are inevitably reflected in our models and representations of the world. A

consequence of this tradition would indicate that information systems, as representations or models of reality, are similarly affected by the metaphysical assumptions underlying their construction. Most existing information system descriptions have typically been developed from "outside reality", for example, in terms of implementation, technology, hardware and software, formal and mathematical logic, and so forth. This research proposes to work in the opposite direction, from "within reality" by basing IS descriptions directly upon a theory of the world.

3.2 Previous Work with Ontological Formalisms in Information Systems

In some respects, it should be recognized that all IS models make *implicit* ontological assumptions about the applications they describe. Rarely, however, are these assumptions explicitly articulated or defined in a consistent or formal manner. Typically, fundamental concepts such as *entities*, *relationships*, and *activities* are undefined or defined informally. Recently, however, some researchers have begun to address these basic issues.

3.2.1 Stamper

In a series of works, Stamper [1986, 1979, 1978, 1977] has observed the need for better understanding of the links between data and reality. He criticizes conventional approaches such as first-order languages by observing that they model sets of individuals, hence make the assumption that these individuals exist, that they can be distinguished from each other, and that there is no equivocation about identity. He points out that any interpretation in a logic model, therefore, *necessarily* begs questions of individuation and identity. That is, real world entities must be sufficiently individuated in order to sustain an *interpretation* [Tarski, 1941] into the set-theoretic logic model.

Even if individuation is obtained, one must assume that individuals can be identified,

again in order to sustain an interpretation. Stamper argues that in real world situations, much business activity is concerned with disputing the boundaries of entities. In fact, the entire point of some information systems is to identify objects in the real world. Also, the reality of business is not just in easily identified physical objects but in social constructs (rights, contracts, orders, etc). He observes that social constructs are often modelled as *documents* in information systems because most logical formalisms have difficulties with abstract social constructs.

For example, a business concept such as an "order" is almost invariably modelled as a document, ie. a header and one or more order-lines and ignores the social reality of an order as a contract, a directive, or a particular kind of relationship between vendor and buyer, with all its social and legal richness which only marginally involves the document. As a result, semantic ambiguities are inevitable in these models. Stamper expresses this viewpoint as follows:

> Language, logic and mathematics all employ discrete tokens. These tokens can be studied in a formal way using themselves as instruments of enquiry without raising the problems of ontology I have identified. However, when we turn our attention to language, logic and mathematics as a way of expressing things about reality, we cannot ignore questions about the ways in which discrete, formal tokens are coordinated with a reality which has no such discrete structure [1986, p. 236].

He concludes by noting

> For the analysis of business information, a semantic theory should account for such notions as truth, individuation, identity, time, space and so on, instead of adopting them as primitive concepts [p. 251].

Stamper has proposed alternative formalisms which address these issues based on

natural language approaches such as the LEGOL Project [Stamper, 1978] and recent theories in psychobiology [Stamper, 1986].

3.2.2 Wand and Weber Ontological Formalisms

Recently, Wand and Weber have presented a formal model of information systems which draws upon Bunge's work in ontology [Wand & Weber, 1988; Wand & Weber, 1987; Weber, 1987; Wand, 1988a; Wand, 1988b]. Their objective is to develop a model of information systems which is independent of technological or implementation considerations and which can be used as a basis for a general theory of information systems design. They use their model to describe and explain important IS concepts such as batch and real time, transactions processing, and decision support systems.

Wand and Weber observe the wide differences among the many information system development methodologies used in practice and described in the literature. They note that these methodologies employ a wide range of underlying constructs and assumptions: activities, processes, data flows, objects, events, and so forth. They make the point that the lack of a suitable theory of information systems and IS development is hampering the comparison and evaluation of these competing methodologies and their underlying approaches.

They introduce their model by defining an information system as a "human created representation of a real world system as perceived by humans". In this light, information system development is seen as a "*transformation* from some perceptions of the real world into an implementation of a representation of these perceptions". According to their perspective, this transformation proceeds through three progressive stages:

- Analysis: From perceptions of reality into a formal model of this perception. It follows that a key element in IS design methodologies is the provision of a suitable

modelling tool for describing reality.

- Design: From the model of reality to a model of a representation.
- Implementation: From a model of the information system into a realization of the information system.

In their view, one important (though certainly not the only) measure of an information system depends upon the "goodness" of the representation of the real world. As a result, a formalization of "goodness of representation" is an important objective. They do this by positing characteristics, termed "invariants", which must be preserved during these transformations for a representation to qualify as a good design.

Their modelling formalism is an adaptation of some of the concepts and the formal notation used by Bunge in his system of ontology. Pertinent details of Bunge's ontology will be presented in the next section of this chapter. For now, a general overview of Wand and Weber's interpretation of Bunge's approach can be summarized here as follows:

- It is assumed that the real world is made of *things*, which have *properties.* [2] Things can be *composed* of other things, that is, things have a *composition.*
- It is assumed that things can be modelled by what Bunge calls a *functional schema*, ie. a set of functions which assign values to its properties.
- A combination of property values comprises the *state* of a thing and the set of possible states that a thing may assume is termed the *state space* of a thing. A specific information system will normally not attempt to model all the properties of a thing. The values for some particular set of properties of interest is the *state vector*, which

[2]Bunge's use of terms such as *thing*, *property*, *state* and so forth, have formal definitions which will be presented later in this chapter.

contains all necessary structural information about the thing as required by the system.

- The dynamics of a thing are modelled by state changes, called *events*. An event is fully defined by the pre- and post-states of a thing. The set of states over time is called the *history* of the thing.

- Two things are said to *interact* if their histories are not independent. A system is a set of interacting things. Interactions among things constrain the possible states of things in the system. Things that interact with the system but are not included in the composition are called the *environment* of the system.

- System dynamics are formalized by the concept of *stability*. Wand and Weber assume that

  1. a change of state will happen if and only if the system assumes a state that is not a stable state.
  2. a system in an unstable state will change to a stable state which is uniquely defined by the unstable state.

  Rules or laws define the possible unstable/stable pairs of states of the system and hence describe the behaviour of the system.

- An *external event* occurs when the environment forces the system to change from a current stable state to a new one. If the new state is unstable, *internal events* or *system responses* move the system to a stable state. From the systems point of view, then, the dynamics of the system are described in terms of the sequence

  stable state $\rightarrow$ *external event* $\rightarrow$ *unstable state* $\rightarrow$
  internal event (or system response) $\rightarrow$ *stable state.*

Based on these concepts, a full description of the statics and dynamics of a system is given by the pair $< S, L >$ where S is the possible state space of the system and L is the set of system laws. In practice, external events are limited to a subset of relevant events, hence, the formal scheme is a triple $< S, L, E >$.

In the model, the "goodness of representation" of the information system is its capacity as a *state tracking mechanism*, that is a mechanism whose states reflect the states in the real world. Four conditions are presented as necessary for an information system to qualify as a "good representation".

- *Mapping requirement.* All real-world states of interest must be mappable into the IS states.
- *Tracking requirement.* When an event occurs in the real-world, the sequence of real-world state changes initiated by that event must have a correspondence in the sequence of IS state changes.
- *Reporting requirement.* All real-world events of interest must be reported to the IS.
- *Sequencing requirement.* The sequence of reporting events in the IS must be the same as the corresponding sequence of events in the real world.

Wand and Weber claim that this model also explains fundamental concepts and intuitions in IS which have been used intuitively by practitioners. For example, definitions of transaction processing, management reporting and real-time systems are usually based on how systems are used. According to the model, however, transactions processing systems are seen as transformations of states according to system laws in response to transaction events. Management reporting and decision support systems are seen as *models* of the decision situation. Real time implies that any real event must be reported to the information system and processed before the next real event occurs.

Wand and Weber suggest the following advantages for this model.

- There is an ontological basis for modelling both reality and the IS.
- The model stresses laws rather than processes and activities. The advantage of laws are that they are testable for consistency and completeness.
- The model formalizes the notion of system decomposition.
- The model can compare methodologies independently of implementation.
- This approach provides a basis for automated analysis and design.

3.2.3 Wand's Ontological Model of Objects

A formal model of objects based on Bunge's ontology has been proposed in [Wand, 1989].

Wand suggests that object-oriented programming suffers from conceptual problems because the notion of objects "emerged as programming concepts and, therefore, were driven by implementation considerations". He advocates taking a modelling rather than a programming perspective to the object paradigm and looks to ontology to provide the basic principles for a theory of objects. In comparing Bunge's ontology to the object concept in object-oriented programming languages (OOPL), he finds some important relationships.

- The principles of *encapsulation, independence,* and *persistence* are fundamental to both OOPL and Ontology;
- *Message passing*, also a basic concept in OOPL, is not an intrinsic ontological concept. Messages can be viewed as one (perhaps out of many) mechanism for *interaction* in Ontology but not a basic principle;

- Classic OOPL's such as Smalltalk are characterized by *homogeneity* in that everything in the language is an object. Ontology does not insist on this principle;
- *Class inheritance* in OOPL has parallels in Bunge's theory of kinds which are classes of things in the world which share common properties and laws. In Bunge's system, the collection of kinds is a lattice under inclusion which, in Wand's view, parallels the concept of inheritance in OOPL.

Wand proposes a formal model of objects which includes the following characteristics.

- The world is made up of objects with properties and laws.
- Objects can be composite and possess *hereditary* (ie. inherited from its components) or *emergent* (ie. unique to the composite object) properties.
- Objects interact with each other and can affect each other's states.
- State transitions in objects are described as *events*. Objects switch states until they are in some lawful or stable state.

This model distinguishes clearly between the use of object as a modelling construct and an implementation or programming construct. And it has been acknowledged as an important initial step toward a formal and precise theory of objects based on ontological principles [Kilov, 1989]. This thesis shares the same approach and is motivated by many of the same considerations as Wand's model. There are, however, some significant differences.

- Wand's model is a general theory of objects which stresses first principles and overall approaches. This thesis proposes an *operationalization* of Bunge's Ontology in the context of Information Systems Development. There are, therefore, significant differences in level of detail and purpose.

- The model proposed in this thesis provides both a written and visual notation as well as an implementation framework which are not considerations in Wand's work.
- The proposed model develops principles for object interaction, object change, and end-user interfaces. Wand's model was not intended to consider these issues in any detail.

3.3 Bunge's Ontology

In the opinion of this thesis, Bunge's ontological system is compatible with IS conceptual modelling by virtue of its rigorous formalism, its commitment to realism, and its methodical distinction between reality and representation of reality.

The value of Bunge's work has been acknowledged by others for its (1) completeness, (2) rigorous formalization, (3) respect for the results of science as well as the needs of society, (4) recognition of emergent properties, and (5) wide acceptance among practicing scientists [Mattessich, 1986; Mattessich, 1982]. Moreover, as observed by Wand [1989], Bunge's objectives for a formal ontological theory are especially relevant to any attempt at developing a theoretical basis for information systems development:

> Metaphysics can dig up, clarify, and systematize some basic concepts and principles occurring in the course of scientific research and even in scientific theories...
>
> Metaphysics can render service by analyzing fashionable but obscure notions such as those of system, hierarchy, structure, event, information...[Bunge, 1977, pp. 23-24].

In the remainder of this chapter, features of Bunge's ontology salient to a theory of object-oriented conceptual modelling are summarized and interpreted. This material is drawn from [Bunge, 1977] and [Bunge, 1979].

3.3.1 Basic Assumptions

Bunge's basic ontological assumptions about the world are stated as follows [3]

1. *There is a world external to the cognitive subject.* If no such world existed, it would not be subject to scientific inquiry.

2. *The world is composed of things.* Consequently, the sciences of reality (natural or social) study things, their properties and changes.

3. *Forms are the properties of things.* We study and modify properties by examining things and forcing them to change. Properties are represented by predicates (eg. functions) defined on domains that are, at least in part, sets of concrete objects.

4. *Things are grouped into systems or aggregates of interacting components.* There is no thing that fails to be part of at least one system. There are no independent things: the borders we trace between entities are often imaginary. What there really is, are systems - physical, chemical, living, or social.

5. *Every system, except the universe, interacts with other systems in certain respects and is isolated from other systems in other respects.* Totally isolated things would be unknowable. And if there were no relative isolation we would be forced to know the whole before knowing any of its parts.

6. *Every thing changes.*

7. *Nothing comes out of nothing and no thing reduces to nothingness.*

[3] Assumptions M1 - M10 [pp. 16 - 17]. All references to Assumptions, Definitions, Postulates, and Theorems are to [Bunge, 1977] unless otherwise specified.

8. *Every thing abides by laws.* Whether natural or social, laws are invariant relations among properties, and they are just as objective as properties. Moreover, a law is a property.

9. *There are several kinds of law.* There are causal laws and stochastic laws. There are same-level (eg. biological) laws and cross-level (eg. psychosocial) laws.

10. *There are several levels of organization.* Eg. physical, chemical, biological, social, technological, etc. The so-called higher levels emerge from other levels in the course of processes: but once formed - with laws of their own - they enjoy a certain stability.

3.3.2 Substance, Association and Composition

Bunge begins his ontology with a theory of substance. He assumes the existence of *substantial individuals*[4] and a fundamental property of *association*. Formally, he proposes the following:

> POSTULATE.[5] There exists a non-empty set S and a binary operation $\circ$ such that:
>
> - S is the set of all substantial or concrete individuals;
> - $\circ$ represents the association of individuals which is associative and commutative;
> - members of S are idempotent under association (ie. $s \circ s = s$);
> - $s_1 \circ s_2 \circ ... \circ s_n$, where $s_i \in S$, represents a *composite* individual,
> - there exists a neutral or null element $\square \in S$ such that for any $x \in S$, $x \circ \square = x$.

[4] Ie. concrete or material things.
[5] Postulate 1.1.

Association can be either natural or man-made. For example, one view of $\circ$ is the "putting together of perceptible things by somebody" [P. 29]. The fundamental property of association is a step toward complexity[6] and is formalized in the concept of *composition.*

DEFINITION.[7] An individual x is *composite* iff it is composed of individuals other than itself and $\square$. Ie.

1. $x \neq \square$,
2. $x \in S$,
3. there exists $y, z \in S$,
4. $x \neq y, x \neq z$,
5. $x = y \circ z$.

Otherwise, the individual is *simple.*

A related term, *part-of* is defined as

DEFINITION.[8] For $x, y \in S$, x is *part-of* y iff $x \circ y = y$. In symbols, $x \sqsubset y$.

The relation $\sqsubset$ is also called the *whole-part* relation. The *composition* of an individual is the set of its parts.

DEFINITION.[9] The *composition* of an individual equals the set of its parts. Ie., let $\mathcal{C}{:}S \rightarrow 2^S$ be a function from individuals into sets of individuals, such that $\mathcal{C}(x) = \{y \in S | y \sqsubset x\}$ for any $x \in S$; then $\mathcal{C}(x)$ is called the *composition* of x.

[6] Ie. complex things.
[7] Definition 1.1.
[8] Definition 1.2.
[9] Definition 1.6.

A number of consequences flow from the definitions above.

COROLLARY.[10] The totality S of substantial individuals is partially ordered by the part-whole relation $\sqsubset$. Ie., $< S, \sqsubset >$ is a poset (partially ordered set).

and

THEOREM.[11] The association of any two individuals is the suprenum (least upper bound or l.u.b.) for them with respect to the part-whole ordering:

$$\text{If } x, y \in S \text{ then } sup\{x, y\} = x \circ y.$$

This concept of suprenum can be generalized by

POSTULATE.[12] Every set $T \subseteq S$ of substantial individuals has a suprenum, which is denoted by $[T]$. Ie., for any $T \subseteq S$, there exists an individual $[T] \in S$ such that

1. $x \sqsubset [T]$ for all $x \in T$;
2. if $y \in S$ is an upper bound of T, then $[T]$ precedes y, ie., if $x \sqsubset y$ for all $x \in T$, then $sup\ T = [T] \sqsubset y$.

DEFINITION.[13] Let $T \subseteq S$ be a set of substantial individuals. Then the *aggregation* or *association* of T, or $[T]$ for short, is the suprenum of T. Ie., $[T] = supT$.

THEOREM.[14] The world $\diamond$ is the aggregation of all individuals:

$$\diamond = [S] = sup\ S.$$

[10]Corollary 1.2.
[11]Theorem 1.1. Proofs for all theorems from Bunge are found in Appendix A.
[12]Postulate 1.3.
[13]Definition 1.7.
[14]Theorem 1.2.

THEOREM.[15] The ordered quadruple $< S, \circ, \square, \diamond >$ is a sup-semilattice with least element $\square$ and last element $\diamond$ with respect to the part-whole relation $\sqsubset$ (refer to Figure 3.1 adapted from [1977, p. 33]).

Bunge recognizes that the notion of *composition* is too general for most purposes. Therefore, he introduces the concept of *B-composition* and *A-composition.*

POSTULATE.[16] There is a proper subset B of S such that every substantial individual is the aggregation of members of B. Ie., for every $x \in S$ other than the null individual, there is a unique subset $B_x \subset B$ such that $x = [B_x]$.

This is, of course, the axiom of *atomism* expressed in terms of Bunge's association theory.

Another refinement of the notion of composition is:

DEFINITION.[17] Let $A \subset S$ be a set of entities. Then the *A-composition* (or *composition at the A level* of a substantial individual $x \in S$ is the set of parts of x belonging to A:

$$\mathcal{C}_A(x) = \mathcal{C}(x) \cap A = \{y \in A | y \sqsubset x\}.$$

Bunge uses the example of molecular composition of a body of water as the set of H_2O molecules. On the other hand, the atomic composition of the same body of water is a set of H and O atoms.

Bunge presents the notion of a *level* as follows:

...we may split the totality S of entities into a certain number n of disjoint sets A_i, each member of which is composed...by entities of the next lower level

[15]Theorem 1.3.
[16]Postulate 1.4.
[17]Definition 1.15.

A_{i-1}. The levels hypothesis may then be formulated as follows:

$$S = \bigcup_{i=1}^{n} A_i, \quad \bigcap_{i=1}^{n} A_i = \emptyset, \text{ with } A_i = \{a_i^n | n \in \mathbf{N}\}$$

and

$$a_i^n = [\alpha_{i-1}^n], \text{ where } \alpha_{i-1}^n \subseteq A_{i-1}.$$

3.3.3 Form

Bunge then proceeds to develop a theory of *form*. His position with respect to form is stated as a basic assumption:

> Forms are properties of things. There are no Platonic Forms in themselves flying above concrete things. This is why (a) we study and modify properties by examining things and forcing them to change, and (b) properties are represented by predicates (eg. functions) defined on domains that are, at least in part, sets of concrete objects [1977, p. 16].

For Bunge, all (concrete) entities have properties. A distinction is made between properties of substantial or concrete entities, called *substantial properties* or simply *properties*, and properties of constructs, call *formal properties, predicates,* or *attributes*. Substantial properties are possessed by substantial individuals even if we are ignorant of them. In contrast, an attribute is a feature we assign or attribute to an object. A correspondence can exist between attributes and properties in that an attribute can reflect or represent a property. However, the correspondence is not necessarily isomorphic. Some attributes may not represent any substantial properties, others may represent several, and there may exist properties to which we assign no attributes (either by ignorance or design).

We know properties through their attributes and we distinguish between them by a representation function

$$\rho : \mathbf{P} \rightarrow 2^{\mathbf{A}}$$

where **P** is the set of substantial properties and $2^{\mathbf{A}}$ is the power set of attributes **A**. That is, "the representation function is a correspondence between a proper subset of all conceivable attributes and the ill-defined set of all (known and unknown) substantial properties [1977, p. 60]". There can be, of course, attributes with no ontic correspondence such as set membership, disjunction and negation. More formally, Bunge presents the following:

> POSTULATE.[18] Let S be the set of substantial individuals or some subset thereof, and let T to Z be arbitrary nonempty sets, equal to or different from S. Then
>
> 1. any *substantial property in general* is representable as a predicate (or propositional function) of the form
>
> $A : S \times T \times ... \times Z \rightarrow$ *Propositions including* A,[19]
>
> 2. any *individual substantial property* or property of a particular substantial individual $s \in S$, is representable as the value of an attribute at s, ie. as $A(s, t, ..., z)$ where $t \in T ... z \in Z$.

Implicit in this postulate are the following:

1. Properties are attributable to individuals, ie. properties do not exist on their own.

2. Entities possess properties but are not "bundles" of properties. Specifically, an entity is not a set of properties. That is, a notion of identity exists which is independent of properties.

[18] Postulate 2.1.

[19] Bunge's example involves the property of being able to read represented by the function R, the set of humans H, and the set of propositions containing the predicate R such that $R : H \rightarrow P$ where $R(b)$ for $b \in H$ means "b can read" and $R(b) \in P$.

Bunge defines the *scope* of a substantial property as the collection of entities possessing it.

> DEFINITION.[20] The *scope* of a substantial property is the collection of entities possessing it. In other words, the scope $\mathcal{S}$ is the function $\mathcal{S}$: $\mathbf{P} \rightarrow 2^S$ from the set of all substantial properties to the set of all the subsets of substantial individuals, such that "$x \in \mathcal{S}(P)$", for $x \in S$, is interpreted as *"Individual x possesses property P"*.

Bunge postulates a special class of properties called *laws*. He assumes that "all entities satisfy some laws" and the following

> POSTULATE.[21] Every substantial property is lawfully related to some other substantial property.

Since laws interrelate substantial properties, laws themselves are properties of entities.

For composite individuals, global properties characterize the entity as a whole. There are two types of global properties: *hereditary* or *resultant*, possessed by at least one part of the whole; and *emergent*, possessed by the whole but no component. More formally,

> DEFINITION.[22] Given some entity $x \in S$ with properties $p(x)$ and composition $\mathcal{C}(x)$. Let $P \in p(x)$ be a property of x. Then, P is a *resultant* or *hereditary* property of x iff P is a property of some component $y \in \mathcal{C}(x)$ of x other than x; otherwise P is an *emergent* property of x. That is,
>
> 1. P is a *resultant* or *hereditary* property of x iff there exists $y \in \mathcal{C}(x)$, $y \neq x$, such that P is a property of y.

[20] Definition 2.5.
[21] Postulate 2.7.
[22] Definition 2.16.

2. P is an *emergent* property of x if there is no $y \in \mathcal{C}(x)$, $y \neq x$, such that P is a property of y.

3.3.4 Thing and Model Thing

A substantial individual endowed with all its properties is called a *thing*. Formally,

> Definition 3.1. Let $x \in S$ be a substantial individual and call $p(x)$ the collection of its properties. The individual together with its properties is called the *thing* X, ie.:
>
> $$X =_{df} < x, p(x) > .$$

For Bunge, "there are constructs, ie. creations of the human mind to be distinguished not only from things...but also from individual brain processes". This distinction between things and constructs is made in the following postulate:

> POSTULATE.[23] Every object is a thing or a construct; no object is neither and no object is both.

Theoretical science, ontology and information systems do not usually deal with things directly, but with perceptions or representations of things. Bunge formalizes a theory of representation by defining the notion of *model thing*:

> DEFINITION.[24] Let $X =< x, p(x) >$ be a thing of class T. A functional schema X_m of X is a certain nonempty set M together with a finite sequence $\mathbf{F}$ of nonpropositional functions on M, each of which represents a property of T's. That is,

[23] Postulate 3.4.
[24] Definition 3.6.

$X_m = M$ where

$\mathbf{F} =< F_i | F_i$ is a function on $M >$.

The following postulate relates the set of functional schemata to reality:

POSTULATE.[25] Any thing can be modelled as a functional schema.

For any representation X_m, Bunge defines the concept of *state.*

DEFINITION.[26] Let X be a thing with functional schema $X_m =< M, \mathbf{F} >$. For any $F_i \in \mathbf{F}, F_i : M \rightarrow V_i$, where V_i is the domain of values for the property represented by F_i. F_i are the *state variables* or *state functions* for X in the representation X_m. $\mathbf{F}$ is the *total state function* for X and its value

$$\mathbf{F}(m) =< F_1, F_2, ..., F_n > (m) =< F_1(m), F_2(m), ..., F_n(m) > \text{ for } m \in M$$

is said to represent the state of X at m in the representation X_m.

According to this conceptualization, there are no 'absolute' state functions, only state functions in a specified representation. There can be as many state functions as functional schemata, ie. different ways of conceiving a thing. Even within a specified functional schema, there are many choices of systems of units, each possibly leading to a different state function. As Bunge states,

> The sole test for the adequacy of a choice of state functions is the adequacy (factual truth) of the theory as a whole, in particular that of its key formulas, which are those interrelating the various components of the total state function - namely the law statements and constraint formulas of the theory...

[25] Postulate 3.4.
[26] Definition 3.9.

Bunge also points out that

> ...[The] choice of state functions is not uniquely determined by empirical data but depends partly on our available knowledge, as well as upon our abilities, goals, and even inclinations [1977, p. 127].

3.3.5 Laws and Lawful State Space

State functions represent properties. *Laws* are represented by *law statements.*

> DEFINITION.[27] Let $X_m =< M, \mathrm{F} >$ be a functional schema for a thing X. Any restriction on the possible values of the components of **F** and any relation among two or more such components is called a *law statement* iff (1) it belongs in a consistent theory about the X's and (2) it has been confirmed empirically to a satisfactory degree.

That is, a law statement $L(x)$ can be represented as a value of a certain function L, a *law function* which has as domain a class T of things and codomain the set of laws, ie.

> $L : T \rightarrow L(T)$ where T is the set of things possessing laws L.

Further, the notion of *lawful state space* is defined as

> DEFINITION.[28] Let $X_m =< M, F >$ be a functional schema for thing X where $F :< F_1, F_2, ..., F_n >: M \rightarrow V_1 \times V_2 \times ... \times V_n$ is the state function, and call $\mathrm{L}(X)$ the set of all law statements of X. Then the subset of the codomain $V =< V_1 \times V_2 \times ... \times V_n >$ of F restricted by the conditions (law statements) in $\mathrm{L}(X)$ is called the *lawful state space* of X in the representation

[27] Definition 3.10.
[28] Definition 3.11.

X_m or $S_{\mathbf{L}}(X)$ for short:

$$S_{\mathbf{L}}(X) = \{< x_1, x_2, ..., x_n > \in V_1 \times V_2 \times ... \times V_n |$$
$$\mathbf{F} \text{ satisfies jointly every member of } \mathbf{L}(X)\},$$

and every point of $S_{\mathbf{L}}(X)$ is called a *lawful* (or *really possible) state* of X in the representation X_m.

3.3.6 Class and Natural Kinds

Bunge develops an ontological theory of class as follows. As previously defined, the *scope* of a property, $\mathcal{S}$, is the set of things possessing it. That is, given a property P,

$$\mathcal{S}(P) = \{x | P \in p(x)\}.$$

A class then is defined as

DEFINITION.[29] A subset X of things is called a *class* iff there exists a property P such that X is the scope of P, ie.

$$X = \mathcal{S}(P).$$

The intersection of any two classes of things, if nonempty, is also a class.

DEFINITION.[30] Let Θ be the set of all things. Further, let $k : 2^{\mathbf{P}} \rightarrow 2^{\Theta}$, be the function assigning to each nonempty set $\mathbf{R} \in 2^{\mathbf{P}}$ of substantial properties the set

$$k(\mathbf{R}) = \bigcap_{P \in \mathbf{R}} \mathcal{S}(P)$$

of things sharing the properties in $\mathbf{R}$. This value $k(\mathbf{R})$ is called the $\mathbf{R}$-*kind* of things.

[29]Definition 3.14.
[30]Definition 3.17.

If **R** is finite, the corresponding **R**-kind is a class. That is, given $\mathbf{R} = \{P_1, P_2, ..., P_n\}$,

$$k(\mathbf{R}) = \bigcap_{i=1}^{n} \mathcal{S}(P_i) = \mathcal{S}(P_1 \wedge P_2 \wedge ... \wedge P_n),$$

The notion of **R**-kind allows a definition of equivalence.

> DEFINITION.[31] Let **R** be a set of substantial properties. Then two things $x, y \in \Theta$ are said to be **R**-*equivalent*, or *equal in every respect* $P \in \mathbf{R}$, iff they possess precisely the same properties in **R**:
>
> $$x \sim_{\mathbf{R}} y =_{df} (P)(P \in \mathbf{R} \Rightarrow (x \textit{ possesses } P \Leftrightarrow y \textit{ possesses } P))$$
>
> or equivalently
>
> $$x \sim_{\mathbf{R}} y =_{df} p(x) \cap \mathbf{R} = p(y) \cap \mathbf{R}.$$

R-kinds are defined with respect to a subset **R** of properties. A more restrictive notion of equivalence is the *natural kind*, defined with respect to subsets of laws (which are also properties). That is,

> DEFINITION.[32] Let **L** be the set of laws and let $k : 2^{\mathbf{L}} \rightarrow 2^{\Theta}$ be the function assigning to each $\mathbf{L}_i \subset \mathbf{L}$ of laws the set
>
> $$k(\mathbf{L}_i) = \bigcap_{L \in \mathbf{L}_i} \mathcal{S}(L)$$
>
> of entities sharing the laws in $\mathbf{L}_i$. This value $k(\mathbf{L}_i)$ is called the **L**-*species* or *natural kind*.

This leads to the concept of *nomological equivalence*:

---

[31] Definition 3.18.
[32] Definition 3.21.

DEFINITION.[33] Let x and y be two things and $\mathbf{L}_i \subset \mathbf{L}$ a set of laws. Then x and y are *nomologically equivalent relative to* $\mathbf{L}_i$ iff x and y share all their species-specific properties, ie. all the laws in $\mathbf{L}_i$:

$$x \sim_{\mathbf{L}_i} y =_{df} p(x) \cap \mathbf{L}_i = p(y) \cap \mathbf{L}_i.$$

Bunge asserts that the preceding provides a theory of reality

> We now have a theory of properties, distinct from the theory of predicates, and a theory of kinds, different from the algebra of sets. We can therefore use without qualms the concepts of a property and a kind. The differences between predicates and properties, and between sets and kinds, suffice to ruin the ontological interpretations of logic and of set theory. There is no reason to expect that pure mathematics is capable of disclosing, without further ado, the structure of reality [p. 150].

3.3.7 Change, Event and Interaction

Bunge begins his theory of change with a theory of *possibility*. Every "actual thing has some actual properties and others which it may - or may not - acquire [1977, p. 163]". That is, if a change occurs then it was possible to begin with. This *possibility* is one of the properties of the thing undergoing change [1977, p. 164]. Bunge links possibility to lawfulness, something is possible iff it is lawful. Therefore, "every law statement describes possibles [1977, p. 174]".

Change is seen as a construct which cannot be separated from things.

> A change is an event or process, whether quantitative or qualitative or both. Whatever its nature, a change is a modification in or of some thing or things: more

[33] Definition 2.23.

> precisely, it consists in a variation of the state of an entity. To put it negatively, there is no change separate from things - nor, indeed, are there changeless things even though some change slowly or only in certain limited respects [1977, p. 215].

Bunge bases his theory of change upon the concept of state space, that is, change is seen as a transition of a thing from one state to another. As such, his concept of change does not require detailed knowledge of the nature of the thing involved nor does it make explicit use of the time concept. Further, since states are representations of properties, our knowledge of states, and hence change, depends to some extent upon ourselves and the state of the art.

Recall the concept of the *lawful state space* of a thing in which every point in the space represents a lawful state of the thing. The actual state of the thing is represented by a specific point, the *representative point*, in the state space. A change in the thing can be represented by a *trajectory* of the representative points (see Figure 3.2 adapted from [1977, p. 218]). Bunge presents as a basic postulate, the following:

> POSTULATE.[34] Every thing has at least two distinct states.

That is, all (concrete) things are changeable.

Representing changes as state transitions (as opposed to turning into different things) allows Bunge to adopt that he calls the *principle of nominal invariance*, which may be stated as:

> PRINCIPLE.[35] A thing, if named, shall keep its name throughout its history as long as the latter does not include changes in natural kind - changes which call for changes of name.

[34]Postulate 5.1.
[35]Principle 5.1.

This implies, of course, that the state space of a thing must include all of the latter's possible states, from beginning to end. Since state spaces are representations, the principle of nominal invariance allows the designation of a permanent identity, even if the components of a thing change (eg. a given person is identified by the same name from age 5 to age 50, even if every atom in the physical body has been renewed over the years).

A change of state is called an *event*. Events are represented by the ordered pair $< s_1, s_2 >$ where s_1 and s_2 are the states before and after the change. Given a thing X with lawful state space $S_{\mathbf{L}}(X)$, the set of all possible events is

$$E(X) = S(X) \times S(X).$$

This includes the identity event that results in no change, ie. $< s_1, s_2 >$ where $s_1 = s_2$. Further, two or more events can combine to produce a complex event or *process*, ie. $< s_1, s_2 > \circ < s_2, s_3 > = < s_1, s_3 >$. These concepts can be summarized by

> DEFINITION.[36] Let $S(x) \neq \emptyset$ be a state space for a thing x and let $E(x) = S(x) \times S(x)$. The triple $\mathbf{E} = < S(x), E(x), * >$, where * is a partial (not every defined) binary operation in $E(x)$ such that, for all a, b, c, d in $S(x)$,
>
> $$< a, b > * < c, d > = \begin{cases} < a, d > & \text{if } b = c \\ \text{not defined} & \text{if } b \neq c, \end{cases}$$
>
> is the *event space* of x associated with $S(x)$ iff
>
> 1. every element of $E(x)$ represents a conceivable change of (or event in) thing x;
> 2. for any $e, f \in E(x), e * f$ represents the event consisting in that event e *composes* with event f in the indicated order;

[36] Definition 5.4.

3. for any $s \in S(x), < s, s > \in E(x)$ represents the *identity event* (or non-event) at s, ie. the staying of x in state s.

Two special cases are dealt with as follows:

COROLLARY.[37] Let $s, s' \in S(x)$ be states of x. Then

1. no event other than an identity event is immediately repeatable: $< s, s' > * < s, s' >$ is not defined in $E(x)$;
2. if an event is followed by its converse then no net change results: $< s, s' > * < s', s > = < s, s >$.

A concept of *precedence* is developed as follows.

DEFINITION.[38] Let e and e' be two events in a given event space, ie. $e, e' \in E(x)$ for some thing x, such that e and e' compose to form a third event $e'' = e * e'$. Then, e is said to *precede* e' relative to the reference frame involved in $E(x)$:

$$\text{If } e, e' \in E(x) \text{ then } e \prec e' =_{df} e * e' \in E(x).$$

And

Collorary.[39] For any given thing x and every state representation, $< E(x), \prec >$ is a strictly partially ordered set.

A mechanism for an event $< s_1, s_2 >$ can be described by functions which are laws or transformations of the state space $S_{\mathrm{L}}(x)$ compatible with the laws of x. That is,

[37] Corollary 5.4.
[38] Definition 5.7.
[39] Collorary 5.5.

DEFINITION.[40] Let $S_{\mathbf{L}}(x)$ be a lawful state space for a thing x. Then the family of *lawful transformations* of the state space into itself is the set of functions

$$G_{\mathbf{L}}(x) = \{g \text{ is a function } | \ g : S_{\mathbf{L}} \rightarrow S_{\mathbf{L}}(x)$$
$$\& \ g \text{ is compatible with the laws of } x\}.$$

Since change can thus be represented as a series of successive states, a concept of *history* is now defined as

DEFINITION.[41] Let $\mathbf{F}$ be a state function for a thing x which includes in its domain $\mathbf{T}$, the set of time instants. Then, the *history of* x during the interval $\tau \in \mathbf{T}$ is the set of ordered pairs

$$h(x) = \{< t, \mathbf{F}(t) > | t \in \tau\}.$$

The principle of change does not consider sources of change. Bunge recognizes spontaneous change in a thing or change in a thing induced by other things.[42] Induced change can be defined as

DEFINITION.[43] Let x and y be two different things with state functions $\mathbf{F}$ and $\mathbf{G}$ respectively relative to a common reference frame f, and let

$$h(x) = \{< t, \mathbf{F}(t) > | t \in S(f)\}, \textit{and } h(y) = \{< t, \mathbf{G}(t) > | t \in S(f)\}$$

be their respective histories. Further, let $\mathbf{H} = g(\mathbf{F}, \mathbf{G}) \neq \mathbf{G}$ be a third state function, depending upon both $\mathbf{F}$ and $\mathbf{G}$, and call

$$h(y|x) = \{< t, \mathbf{H}(t) > | t \in S(f)\}$$

[40]Definition 5.8.

[41]Definition 5.27 and [1979, p. 24]. Bunge's definition 5.7 is in terms of a time/space *frame of reference* in four dimensions.

[42]This thesis will be concerned only with *induced* change.

[43]Definition 5.29.

the corresponding history. Then, x *acts on* y, or $x \triangleright y$ for short, iff, for some state function **H** determining the trajectory $h(y|x), h(y|x) \neq h(y)$.

Further,

DEFINITION.[44] Two different things x and y *interact* iff each acts upon the other. In symbols:

$$x \bowtie y =_{df} x \triangleright y \ \& \ y \triangleright x.$$

DEFINITION.[45] Two different things are *bonded* (or *linked* or *coupled*) together iff at least one of them acts upon the other. In symbols: If x and y are things, then

$$Bxy =_{df} x \triangleright y \text{ or } y \triangleright x.$$

The above provides the basis for the following.

DEFINITION.[46] Let X be a thing composed of parts X_i for $1 \leq i \leq n$. Then X is an *aggregate* (or *conglomerate* or *heap*) iff, in every representation of X (ie. for every choice of state functions), its history $h(X)$ equals the union of the partial histories $h(X_i)$. Otherwise, X is a *system*.

3.3.8 Spacetime

Bunge then proceeds to demonstrate that the above principles are sufficient for a theory of spacetime.

Space and time are usually regarded as external to things and their changes. That is, they are seen as " a fixed scenario... absolute, autonomous, and self-existing [pp. 276-277]". Bunge, however, argues that space and time *result* from things.

[44]Definition 5.30.
[45]Definition 5.33.
[46]Definition 5.35.

> For, in the absence of things, there should be no spatial relations; and in the absence of change there should be no temporal relations.

Bunge begins his theory of space[47] with a formal definition of *interposition* or *betweenness* based on the fundamental principles of thing and change.[48] From this definition, he develops a formal notion of a *separation* function σ between any two arbitrary things.[49] Finally, he defines *space* as

> DEFINITION.[50] The set B of basic things, together with the separation function σ, is called the thing space, abbreviated $\vartheta =< B, \sigma >$.
>
> In other words, the thing space is nothing but *the collection of spaced things*, or the set of things related by their mutual separations.

The idea of a changing thing is also the basis for Bunge's theory of time.

> A duration is the duration of some event or process: a changeless universe would be timeless. Just as space is the spacing of things, so time is the pace of events. And just as spatial distance is the separation among things, so temporal interval is the separation between different states. This is the intuitive germ of the relational theory of time [pp. 296-297].

Bunge's theory of time begins with a definition of *temporal ordering* which is based on the ordering relation among the different states of a thing.[51] Definitions of *past*, *present*, and *future* of some thing x are developed relative to a distinguished *zero* or *origin* state

[47]Bunge's theory of spacetime is not directly relevant to this thesis. Therefore, definitions and postulates are not detailed here.

[48]Postulate 6.1.

[49]Definition 6.1.

[50]Definition 6.2.

[51]Postulates 6.7 & 6.8.

of x.[52] A potential *clock* is defined as a thing in which every one of its state spaces is strictly partially ordered.[53] An important conclusion resulting from these principles is

COROLLARY.[54] There is no time where there are no changing things.

3.4 Summary

This chapter has presented the motivation for the use of Ontology as the basis for an object-oriented model for ISD. Some previous work applying ontology to Information Systems was discussed. It has also briefly outlined the definitions and propositions of Bunge's formal system of ontology.

The next chapter develops an ontology-based conceptual modelling scheme which is based directly on these metaphysical principles.

[52]Definition 6.8.

[53]Definition 6.9. By "strict partial ordering", Bunge means asymmetric and transitive. Most things can, therefore, be considered to be clocks. Nor do clocks need to be regular.

[54]Corollary 6.3.

Chapter 4

Ontology-Based Conceptual Model - OBCM

This chapter proposes the Object-Based Conceptual Model (OBCM), a system for the conceptual modelling of IS applications. OBCM is an approach to conceptual modelling which is based directly on Bunge's ontological formalism presented in the previous chapter.

The formal description of Ontology-Based Conceptual Modelling is developed as follows. First, the main premises from Bunge upon which OBCM is based are highlighted. Next, details of OBCM are described, in particular the three major constructs *surrogate*, *model object* and *object*. Two different notations for OBCM are proposed: a formal written notation, and a somewhat more informal visual notation. Examples are presented to illustrate the conceptual material. The chapter concludes with a comparative analysis of OBCM to other, well-known modelling schemes.

OBCM is independent of implementation. In chapter 5, a framework is introduced for implementing an Ontology-Based Information System (OBIS) based on OBCM.

4.1 Background

The following premises from Bunge form the backbone of OBCM.

1. The world consists of *things* which possess both substance and form (ie. properties).[1]

[1]Definitions 3.1 & 3.2. All references to Definitions, Postulates. and Theorems in the footnotes are to [Bunge, 1977] unless otherwise stated.

2. Things are characterized by *properties*. Known properties are represented by one or more *attributes*, ie. concepts or values assigned to things by people.[2] Some properties are well-known, others are unknown and subject to further investigation. Further, the relation between properties and attributes is not necessarily isomorphic (ie. not one-to-one).

3. Things possess a fundamental property of *association* (symbol $\circ$),[3] which is the basis for forming complex things. Two or more things can associate to form another thing separate from the originals.[4]

4. Assume x and y associate to form z, ie.

$$x \circ y = z,$$

then x and y are said to be *part-of* z (written $x, y \sqsubset z$).[5] For the purposes of modelling, a *null element* $\square$ is assumed such that

$$\square \circ x = x.$$

5. A thing x is *composite* if it is composed (ie. is an association) of more than one thing other than itself and null. Otherwise, it is *simple*. The *composition* of a thing is defined as the set of its parts. More specifically, given a set of things T and a function, $\mathcal{C}$, such that

$$\mathcal{C} : T \rightarrow 2^T$$

where

$$\mathcal{C}(x) = \{y \in T | y \sqsubset x\}$$

[2]Postulate 2.1.
[3]Not related to the abstraction mechanism of *association* found in semantic data models.
[4]Definition 3.3 and Postulate 3.2.
[5]Definition 3.4.

for any $x \in T$, then $\mathcal{C}(x)$ is the *composition* of x.[6] In the case of a *simple* thing x, $x \sqsubset x$ and the composition of x is $\{x\}$ (ie. x has no parts other than itself).

6. The *A-composition* $\mathcal{C}_A$ of a thing x is defined as the set of parts of x belonging to some set A. That is, given x and its composition $\mathcal{C}(x)$,

$$\mathcal{C}_A(x) = \mathcal{C}(x) \cap A = \{y \in A | y \sqsubset x\}.$$

The A-composition is composition relative to a certain set or level of things. For example, the molecular composition of a body of water is some set of H_2O molecules. On the other hand, the atomic composition of the same body of water is a set of H and O atoms.[7]

7. For composite individuals, *global* properties characterize the entity as a whole. There are two types of global properties: *hereditary*, possessed by some part of the composite; and *emergent*, possessed by the whole but not any component of the whole. Given some entity $x \in S$ with properties $p(x)$ and composition $\mathcal{C}(x)$, let $P \in p(x)$ be a property of x. Then, P is a *resultant* or *hereditary* property of x iff P is a property of some component $y \in \mathcal{C}(x)$ of x other than x; otherwise P is an *emergent* property of x. That is,

   - P is a *resultant* or *hereditary* property of x iff there exists $y \in \mathcal{C}(x)$, $y \neq x$, such that P is a property of y.
   - P is an *emergent* property of x if there is no $y \in \mathcal{C}(x)$, $y \neq x$, such that P is a property of y.[8]

[6] Definition 1.6.
[7] Definition 1.15.
[8] Definition 2.16.

8. Properties are interrelated by relations called *laws* which are also properties of things.[9]

9. There exists a thing $\diamond$ such that every other thing is a part of $\diamond$. $\diamond$ is called the *world* or *universe*.[10] Assuming some null thing $\square$ such that

$$\square \circ x = x,$$

and the set of all things T, the totality of $< T, \square, \diamond, \circ >$ is a sup-semilattice[11] structure with last element $\diamond$ and least element $\square$ with respect to the part-whole relation $\sqsubset$. [12]

10. The *scope* of a property is the collection of things possessing that property.[13] A *class* of things are those things in the scope of a property or set of properties.[14] A *kind* is a class in the scope of a set of properties and laws.[15]

11. A thing of some class T can be represented by a *functional schema* which is a dual $< M, \mathbf{F} >$ where M is a base set and F a set of functions on M, each of which represents a property of T. The base set M can be interpreted as a means of parameterizing the functions in F according to the purpose of the representation. It can be, for example, a set of time instants or a mapping to some subset of spacetime. Each function in F is evaluated at a fixed $x \in T$ in addition to other variables in M.[16]

---

[9]Definition 3.10.

[10]Postulate 3.3.

[11]An ordered set (M,$\sqsubset$) is called *sup semi-lattice* if M contains an upper limit of any two elements [Gericke, 1966].

[12]Theorem 3.1. The nodes of the semi-lattice represent things and the edges represent the *part-whole* relations among the things. The lattice structure ensures an access path from any thing in the lattice to either $\diamond$ or $\square$. The significance of this will be explored later.

[13]Definition 3.13.

[14]Definition 3.14.

[15]Definition 3.21.

[16]Definition 3.6.

An example from Bunge is that of population, an important property of communities which changes constantly with time and can be represented as

$$Pop : \Sigma \times \mathbf{R} \rightarrow \mathbf{N}$$

in which Σ is the set of all communities, $M = \mathbf{R}$ is the real line representing time instants, and $\mathbf{N}$ the set of natural numbers.

12. *Law statements* express relationships among the functions in $\mathbf{F}$ or restrictions on their values.[17]

13. The *state* of a thing in some representation $< M, \mathbf{F} >$ is the value $\mathbf{F}(m)$ or $< F_1(m), F_2(m), ..., F_n(m) >$ where $m \in M$[18]

14. An *event* is a change in state which can be represented as an ordered pair $< s_1, s_2 >$ where s_1 and s_2 are the states before and after the change respectively. Events can also be represented as a triple $< s_1, s_2, g >$ where g is a function such that $s_2 = g(s_1)$. The function g may or may not be a law statement. For any thing x, there is a set $E(x)$ of *possible events*, which describes all possible changes in x. $E(x)$ is a *partially ordered* set.[19]

15. Given some class of things T and its representation $< M, \mathbf{F} >$ where M includes the set of time instants $\mathbf{T}$, the *history* of a thing $t \in T$ during the interval $\tau \subseteq \mathbf{T}$, is the set of ordered pairs[20]

$$h(x) = \{< t, \mathbf{F}(x, t) > | t \in \tau\}.$$

[17]Definition 3.10.
[18]Definition 3.9.
[19]Definitions 5.4, 5.5 & Corollary 5.5.
[20]Definition 5.27 and [p. 24, 1979].

16. A thing *acts* upon another if it modifies the latter's history.[21] A thing composed of interacting components is called a *system.*[22]

4.2 OBCM - Ontology-Based Conceptual Model

The Ontology-Based (or Object-Based) Conceptual Model (OBCM)[23] operationalizes Bunge's ontological system so that it can be used to describe some "slice of reality" or Universe of Discourse of an IS application. The implementation of an application modelled with OBCM is called an Ontology-Based (or Object-Based) Information System (OBIS).

Informally, OBCM can be viewed as an abstract space populated by constructs called *objects*, each of which is a representation of some thing in the world. An object is a pair <*surrogate, model object*>, where *surrogate* is an entity corresponding to some specific thing in the world and *model object* is an operationalization of Bunge's concept of functional schema or *model thing.*

The following sections develop the constructs *surrogate*, *model object* and *object* in detail.

4.3 Surrogate

4.3.1 Definition

DEFINITION 4.1 *A* surrogate *is a modelling element corresponding to some thing of interest in the real world.*[24] .

[21]Definition 5.29.

[22]More accurately, a thing is a system if its components interact in every representation (Definition 5.35).

[23]As mentioned earlier, the proposed model is *both* ontology-based and object-based. The acronym OBCM can be interpreted to refer to both of these aspects of the model.

[24]Implementation will be discussed later. For now, it may be useful to note that surrogates can be viewed roughly in the same way as abstract objects in semantic data bases and typically implemented

4.3.2 Composite Surrogates

Ontology allows for things to associate to form other things. Similarly, surrogates can associate to form other surrogates. That is, given the set of all surrogates S, there is an operation $\circ$ closed on S such that for any $x, y \in S$, $x \circ y$ is also an element in S. Surrogates are idempotent under the $\circ$ operation, ie. $x \circ x = x$.

DEFINITION 4.2 *A surrogate is* composite *if it is formed by the association of surrogates other than itself. Otherwise, it is* simple.

The *composition* of a surrogate is the set of its components. The composition of a simple surrogate is itself.

DEFINITION 4.3 *Assume a composite surrogate*

$$z = a \circ b,$$

then a is said to be part-of *z and b is said to be* part-of *z, or in symbols,* $a \sqsubset z$ *and* $b \sqsubset z$. *More formally, for surrogates x and y,*

$$x \sqsubset y \text{ iff } x \circ y = y.$$

In the world, a specific thing may be considered to be *part-of* several different associations. For instance, a person is *part-of* a family, a work group, a community, and so forth. Similarly, there is nothing to prevent a surrogate from being *part-of* several composite surrogates.

DEFINITION 4.4 *The* part-of context[25] *of a surrogate is defined as the set of surrogates to which the surrogate is in a* part-of *relationship.*

That is, if $s \sqsubset f, w, c$, then the *part-of context* of s is $\{f, w, c\}$.

---

as user transparent, system generated, internal identifiers [Hull & King, 1987].

[25]Bunge also refers to this concept as the *environment* of a thing [Bunge, 1979, p. 6].

4.3.3 Surrogate Semantics

Although the set of surrogates S is closed under $\circ$, not all associations of elements in S are of interest.[26] This leads to a discussion of the *semantics* of surrogates.

Whether a surrogate is simple or composite is a matter of interpretation or perception and the purpose of the application. That is, although the thing in reality may be composite, its surrogate can be simple. For example, a person is composed of various parts such as blood, bone, and tissue, but its surrogate may be simple because the application is only concerned with the person as a single unit and not its parts.

In addition, the nature of the part-of relation $\sqsubset$ can be interpreted broadly. It need not be viewed in simple physical or spatial terms. Bunge makes this point as follows:

> Indeed, an individual on our planet and another in a distant galaxy may be taken to associate to form a third individual, so that each component will be a part of the whole...[1977, p. 30].

To summarize, surrogates are substitutes for things in the real world. Ideally, there will be a one-to-one isomorphic correspondence between the surrogates in the OBCM and the things of relevance in the real world of the application. Surrogates correspond to Bunge's notion of "substantial individuals" and provide the bridging mechanism between reality and the OBCM. This reflects the fact that the OBCM (and its implementation in an OBIS) is a *representation* of some subset of reality, it is not the reality itself (although the implementation of the OBIS is also a real thing, or artifact, in its own right). Further, an information system normally does not deal directly with things in the world. It can, however, handle and manipulate their surrogates (or more accurately, implementations of surrogates).

[26] Bunge assumes that any two things can associate to form a third, hence the set of things is closed under association. However, when the things in a composition interact, the composite thing is called a *system* [1979, pp. 6-14].

On the other hand, surrogates have little significant descriptive power. Descriptive power is provided by model objects and objects.

4.4 Model Object

Information Systems normally do not handle concrete things but rather their representations.[27] A concept of representation is adapted from Bunge which in this thesis is called *model object.*

4.4.1 Definition

DEFINITION 4.5 . *Given a set of things T_K which share a set of properties P_K and laws L_K (ie. T_K is a* kind*), and a set S_K of surrogates in which each element corresponds to an individual in T_K, a* model object[28] *X_m of T_K is a named 4-tuple $< \mathbf{M}, \mathbf{F}, \mathbf{L}, \mathbf{C} >$ where*

- **M** is some base set appropriate to the purpose of the Information System to be implemented;
- **F** is a list of functions called *state functions* or *state variables* with domain $S_K \times \mathbf{M}$, each of which represents a property or properties of interest in P_K;
- **L** is a set of law statements, each of which represents a law or laws in L_K and expresses a relationship among the values of one or more of the state variables in **F** or restricts the values of state variables;
- **C** is a set of change functions, each of which specifies a possible change in the values of one or more state functions in **F**.

[27]Process control, embedded and similar systems will not be specifically addressed in this thesis.

[28]Alternate terms are simply *model*, or sometimes *view*. The notation X_m is Bunge's, presumably to suggest *model* of thing X.

Model object[29] operationalizes Bunge's notions of *model thing* or *functional schemata, state, law* and *event*.[30]

A discussion of each component of model object follows below.

The Base Set M

In the model, the domain of **F** is taken to be $S_K \times \mathbf{M}$, where S_K is the set of surrogates corresponding to the class of things T_K being represented by the model object.

Given a model object X_m, **M** is a cartesian product of certain sets appropriate to the purpose of the representation X_m.

Bunge does not clearly specify the components of **M** except in very general ways. Eg.

> The base set **M** will be denumerable or nondenumerable, as the case may be. It may or may not be thought of as mapped on a subset of physical spacetime. (In systems theory **M** is usually taken to be a set of time instants.) [1977, p. 120],

and

> The domain A of the state function **F** of systems of kind K is the cartesian product of certain sets, such as K, the family 2^E of sets of environmental items with which the members of K are coupled, the set F of reference frames, the set T of time instants, and so on [1979, p. 20].

In this thesis, **M** will be assumed to be at least the set of time instants **T**. The state functions in **F**, therefore, will be evaluated at some instant of time; that is, the domain of **F** is $S_K \times \mathbf{M}$ where $\mathbf{M} = \mathbf{T}$.

In most of the examples to follow, moreover, the functions in **F** will always be assumed to be evaluated at the *current point in time.* That is, each function is evaluated at some

[29]The term *model object* corresponds to Bunge's use of the term *model thing* for the functional schemata of a thing.

[30]Definitions 3.6, 3.9, 3.10, and Principle 5.8.

point $F_i(\mathsf{S}, \tau)$ where $\mathsf{S} \in S_K$ and τ is the current point in time. The parameter τ will normally be understood and not explicitly specified. This results in a *time dependent* model of the application. That is, the *values* of $\mathbf{F}$ may vary depending upon when the functions in $\mathbf{F}$ are evaluated.[31]

Other possibilities for $\mathbf{M}$ such as frames of reference[32] are outside the scope of this thesis and will not be pursued.

State Function and State

Each component $F_i \in \mathbf{F}$ represents a property or properties of the things in class T_K. Given $\mathsf{S} \in S_K$ corresponding to some specific thing $t \in T_K$, and a state variable F_i representing a property P_{K_i}, the *value* of F_i at S (ie. $F_i(\mathsf{S})$) represents an individual property of the t corresponding to S. That is, a property *in general* is represented by a function F_i and an *individual* property of a particular thing $t \in T_K$ is representable as the value of F_i at S, ie. $F_i(\mathsf{S})$.[33]

DEFINITION 4.6 *Given a model object X_m, the set of values of the components $F_i \in \mathbf{F}$,*

$$\mathbf{F}(s) =< F_1(s), F_2(s), ..., F_n(s) >$$

for any $s \in S_K$ is called the state *of the thing corresponding to s in the model X_m.*

This is directly adapted from Bunge's characterization of the state concept.[34] The cartesian product of the codomains of the components F_i in $\mathbf{F}$ is called the *conceivable state space* of the model object.[35]

[31]Bunge's ontology lays the basis for a more elaborate model involving an explicit time parameter. Enhancement of OBCM with time, however, is not pursued in this thesis.
[32]Definition 3.6 & 5.36.
[33]Postulate 2.1.
[34]Definition 3.9.
[35]Definition 3.9.

In Bunge's theory of form, the attribute-property correspondence is not necessarily isomorphic. Similarly, the state variables in **F** are not necessarily one-to-one with the set of properties P_K. Hence, any state variable F_i can represent one or more properties in P_K.

Law Statements

DEFINITION 4.7 Law statements *are restrictions on the* values *of the components* $F_i \in \mathbf{F}$ *or relationships among two or more such components.*[36]

Bunge uses the example of Ohm's law for DC circuits. Assume a class of circuits C, a specific circuit $c \in C$, and its corresponding surrogate c. Ohm's law can be expressed in OBCM as

$$\text{Ohm's Law} = [E(\mathsf{c}) = R(\mathsf{c}) \times I(\mathsf{c})]$$

where E is a state function representing voltage and R and I are state functions representing resistance and current respectively.

For the most part, Bunge is concerned with *natural* laws as found in the physical and biological sciences. In IS applications, it is more likely that law statements will refer to business and social laws or rules. A business example might be a law statement reflecting the permitted indebtedness for any customer of a firm, ie.

$$\mathsf{Debt\ Law} = [\mathsf{debt(c)} \leq \mathsf{creditlimit(c)}]$$

where c is a surrogate representing some customer, debt(c) is a value representing the indebtedness of the customer and creditlimit(c) is a value representing the firm's perception of the credit-worthiness of the customer.

[36]Definitions 2.7 & 3.10. Law statements can therefore express *constraints* in semantic databases.

Since law statements restrict the values of $F_i \in \mathbf{F}$, they define a subset of the conceivable state space of a model object. This subset is called the *lawful state space* or $S_{\mathbf{L}}$, of the model object.

As with state functions, law statements are not necessarily isomorphic to the set of laws L_K.

Change Functions

In the world, *change* is a modification in or of things.

In Bunge's ontology, change in a thing is represented by the transition of one state of the thing to another state. Such transitions are representable by the ordered triple $< s_1, s_2, g >$ where s_1 is the state of a thing before the change, s_2 the state after the change, and g is a function which provides the transformation mechanism between the two states.[37]

DEFINITION 4.8 *The* change functions *of a model object are mechanisms for changing the values of* $F_i \in \mathbf{F}$.

More precisely, they are mechanisms which define *possible* changes in the values of the F_i. That is, given a state s_1 of some thing, a change function $c \in \mathbf{C}$ potentially transforms s_1 to some other state s_2.[38]

All changes in a thing must be lawful. In OBCM, change functions may or may not be defined in terms of the law statements L. Ideally, the set of change functions C describes the transformation of lawful state spaces to themselves, ie.[39]

$$\mathbf{C} : S_{\mathbf{L}} \rightarrow S_{\mathbf{L}}$$

[37]Principle 5.3.
[38]The word *potential* is important since Bunge's ontology does not admit changeless things.
[39]Definition 5.8.

where $S_{\mathbf{L}}$ is the lawful state space of a model object.[40] The lawful state space $S_{\mathbf{L}}$ of a change function, however, pertains to a particular representation or model object. Therefore, although the result of a change might be lawful *locally*, ie. for some particular $s \in S_K$ and in some particular model X_m, a lawful local change may nevertheless result in unlawful changes elsewhere in the Information System. The ideas behind this issue are developed in more detail in section 4.5.4 to follow.

The set of change functions C is a partially ordered set. That is, there is a relation $\prec$ defined between some (but not necessarily all the) elements $g_i \in \mathbf{C}$ such that if $g_1 \prec g_2$ is permitted then $g_2 \prec g_1$ is not.[41] The partial ordering of change functions describes a *trajectory* or more accurately, set of possible trajectories. Such a potential trajectory can also be viewed as the (potential) *life cycle* for the thing represented by the model object.

The actual trajectory or ordered set of states (ie. the values of the state functions F actually taken) for a particular s over a period of time is called the *history* of s (more accurately, the *thing* represented by s).

Change functions are implemented as procedures and may involve input of values or data from sources external to the OBCM.

Recall that the set of change functions C is a mechanism for *local* changes in state. That is, C defines state changes for some specific surrogate as described by a particular model object. The value of a state variable for surrogate s_1 in representation X_m may affect the value of a state variable for some other surrogate s_2 in representation X_n. Such changes are effected through another mechanism, namely through law statements describing *object interaction*. This will be described in detail in the section describing *object interaction laws* (section 4.5.4).

---

[40] Definition 3.11.

[41] Definitions 5.4 - 5.7. Recall from Chapter 3 that Bunge's Corollary 5.4 (i) explicitly prohibits $< s, s' > * < s, s' >$ and (ii) treats an event followed by its converse as resulting in no change, ie. $< s, s' > * < s', s >=< s, s >$. Similarly, for any $g_i \in \mathbf{C}$, $g_i \prec g_1$ is not defined. Also, for any $g_1, g_2 \in \mathbf{C}$, if for any two states s_1 and s_2, $g_1(s_1) = s_2$ and $g_2(s_2) = s_1$, then neither $g_1 \prec g_2$ nor $g_2 \prec g_1$ is defined.

4.4.2 Multiple Views

Bunge assumes that the set of functional schemata is nonempty and can grow indefinitely.[42]

Similarly in OBCM, it is also assumed that for any class of things T_K, there can be more than one model object. In other words, there can be more than one representation for a class (or any particular thing in that class). Where there are several models for the same class T_K, each model is called a *view* of the class T_K.

For example, suppose the model object, PERSON, represents a class of persons of interest to the firm. The state functions in this representation refer to properties of the kind persons such as age, sex, height, blood type, and so forth. The model object, CUSTOMER, may be the representation for the class of customers of the firm. The state functions may refer to debt, credit limits and other properties of the customers of the firm.

Assuming that some members of the class of persons are also customers of the firm, there will be a set of surrogates Q_K which corresponds to individuals which are also customers. Q_K will, therefore, be represented by both model objects PERSON and CUSTOMER.

In this way, the information system can support more than one view or representation of a class of things in the real world.

4.4.3 Example

Customers in some restricted Universe of Discourse can be represented by the model object CUSTOMER = $< M, F, L, C >$ where

- T_K= {JOHN, MARY, JACK} where each element in the set is a customer of interest in the application.

[42]Postulate 3.4 and Definitions 3.7 & 3.8.

- S_K= {'JOHN', 'MARY', 'JACK'} where each element in the set is a surrogate corresponding to each customer in T_K.

- F is a set of functions consisting of

  1. fCreditLimit: $S_K \rightarrow$ *Integers*
  2. fDebt: $S_K \rightarrow$ *Integers*

  where fCreditLimit and fDebt represent credit worthiness and total indebtedness, two important properties of customers of a firm.

- L is a set of law statements, in this case consisting of a single statement

$$\begin{aligned} &\mathsf{lCredit} := \\ &\quad [\mathsf{fDebt(s)} \leq \mathsf{fCreditLimit(s)}] \end{aligned}$$

  for all s $\in S_K$.

- C is a set of change functions, in this case consisting of the single function

$$\begin{aligned} &\mathsf{cInit} = \\ &\quad [\mathsf{fDebt(s)} := 0] \ \& \\ &\quad [\mathsf{fCreditLimit(s)} := \textit{External Source}], \end{aligned}$$

  for all s $\in S_K$. That is, for any s $\in S$, the change function cInit defines a change from some initial state s_1 (say fDebt = *nil* and fCreditLimit = *nil*) to a new state with the values fDebt(s) and fCreditLimit(s) as defined above.

  The notation *External Source* indicates that the value for fCreditLimit is taken from a source external to the model (such as, for instance, keyboard entry or an external database).

4.5 Object

The main modelling construct in OBCM is the *object* which is a *representation* of some thing in reality. Objects are used as a conceptual modelling construct, are implemented as an Information Systems artifact, and are directly manipulated by the end-users of the final system.

4.5.1 Definition

DEFINITION 4.9 *Given a model object X_m representing a kind T_K, an* object *is a pair* $< \mathrm{S}, X_m >$ *in which*

- S is an element in S_K, where S_K is a set of surrogates each element of which corresponds to an individual in T_K, and
- X_m is a model object $< \mathbf{M}, \mathbf{F}, \mathbf{L}, \mathrm{C} >$, representing T_K.

Given an object $x = < \mathrm{S}, \mathrm{X}_m >$, the element S is called the *surrogate* for the object x, x is called the *object* for the surrogate S, and X_m is the *model object*, *model* or *view* for x. The object x is also called an object of *type* or *kind* X_m. Alternately, x can be said to be in the *scope* of X_m.

There can be more than one view of a thing. That is, the thing can be represented by more than one object. Therefore, the surrogate corresponding to the thing can participate in more than one object.

4.5.2 Example

In the model object CUSTOMER above, consider the surrogate 'JOHN'. The OBCM object *john* = < 'JOHN',CUSTOMER >[43] is an object of kind CUSTOMER representing

[43]If there are multiple views of JOHN, they can be distinguish as (for example) *john*.PERSON and *john*.CUSTOMER.

the real world thing, that is, person, corresponding to the surrogate 'JOHN'. The values of the state functions in **F** consist of

1. fDebt('JOHN')

2. fCreditLimit('JOHN').

These values represent the individual properties of JOHN. The *state* of the object *john* is the vector < fDebt('JOHN'),fCreditLimit('JOHN') >.

The notational convention used is UPPER CASE for model objects and their names, SMALL CAPS for things, quoted 'SMALL CAPS' or single SMALL CAP characters for surrogates and *italics* for OBCM objects and their names. Names of state functions, law statements and change functions are sans serif beginning with lowercase f,l or c respectively. A more convenient notation for the example above is *john*.fDebt and *john*.fCreditLimit for fDebt('JOHN') and fCreditLimit('JOHN') respectively.

Figure 4.1 summarizes the relationships among things in the world, surrogates, model objects and object. The vertical dotted line divides the Universe of Discourse, ie. the things to be represented, from the Information System. In reality, there can be several views or perceptions of the things in the world. This is indicated in Figure 4.1 by the *1:n* relationship between things and their views. Each thing is assigned a *1:1* relationship with a surrogate in the OBCM. In addition, each relevant view or perception of a thing has a *1:1* relationship with a model object.[44] A surrogate can be associated *1:n* with different models.

As the diagram indicates, then, for any thing in reality, there can be several objects in the OBCM, each corresponding to a different perception or view of that thing. However, there is only one surrogate.

[44]Which is the reason why *model object* is also referred to as a *view*.

4.5.3 Object Composition

Where the surrogate for an object is simple, the object is also called simple. When the surrogate for an object is composite, the object is also composite.

DEFINITION 4.10 *Given a composite surrogate* S *with composition* {A,B,...,N} *and objects* s *and* $a, b, ..., n$ *for these surrogates, the* S-composition C_S *of the object* s *is defined to be the set of objects* $\{a, b, ..., n\}$*, that is*

$$C_S(s) = \{a, b, ..., n\}.$$

All objects in OBCM have a composition. Simple objects consist of themselves, ie. $C(x) = \{x\}$ where x is simple. OBCM regards composition as a basic property of all things, a property which is representable as a state function.[45] All model objects, therefore, contain a state function fSComp representing this property defined as

$$\mathsf{fSComp} : S_K \rightarrow 2^{\Theta}$$

where Θ is the set of all objects in the OBCM.

The S-Composition of an object includes all possible views of the composition of its surrogate. This is because there may be several objects for any particular surrogate.

For instance, suppose in the example from section 4.5.2., the individual JOHN with corresponding surrogate 'JOHN' is represented by two models, CUSTOMER and PERSON. Then, there can be at least two objects representing JOHN, one of type CUSTOMER and one of type PERSON. Call them *john*.CUSTOMER and *john*.PERSON. Assume some object x with surrogate X which includes 'JOHN' in the composition of X. Then, the S-Composition of x will include *john*.CUSTOMER and *john*.PERSON.

This prompts some other, more specific notions of composition such as:

[45] Bunge also refers to *the property of being composed, ie. of having parts* [1977, p. 26] and [1979, p.20].

DEFINITION 4.11 *The* A-Composition, C_A, *of an object* x *is the set of objects in the S-Composition of* x *belonging to some set* A.

This is related to, but not exactly equivalent to Bunge's formulation of A-Composition.[46]

In OBCM, the set A is *implicitly* defined by K_A, a set of model objects. That is, given $K_A = \{K_1, K_2, ..., K_n\}$,

$$A = \bigcup_{i=1}^{n} \{x | x \text{ of type } K_i\}$$

That is, if for some object x, $K_A = \{K_m, K_q, K_s\}$, then the A-composition of x is the subset of objects in $C_S(x)$ of type K_m, K_q and K_s.

All model objects provide the state function fKinds such that

$$\mathsf{fKinds} : S_K \rightarrow 2^{\mathbf{K}}$$

where **K** is a set of model objects. This implicitly defines the set of objects permitted in the A-composition of an object. A-composition itself is represented in all model objects by a state function fAComp such that

$$\mathsf{fAComp} : S_K \rightarrow 2^{\Theta}.$$

In the previous example, if for object x,

$$\mathsf{fKinds} = \{\text{CUSTOMER}\},$$

then the A-composition of x will include *john*.CUSTOMER but not *john*.PERSON.[47]

[46]Definition 1.5. It should be noted that Bunge's theory of association and composition refers to things in the real world. The composition of *objects*, however, refers to representations, hence the need to modify definitions.

[47]Bunge's concept of A-composition is, in the opinion of this thesis, open to some interpretation. In certain respects, an A-composition of a thing depends upon its *representation*. For instance, in one representation, a body of water is a collection of H_2O molecules. In another representation, the same body of water is a collection of hydrogen and oxygen atoms. For this reason, in OBCM, every model object (ie. every representation) requires the state function fKinds which defines how the composition of a thing is to be represented.

Since objects are representations or views, for an object x, the A-composition of each object in $\mathcal{C}_A(x)$ yields a *consistent view* of the internal structure of x. For example, an anatomical representation of a person[48] may yield an A-composition of head, arms, legs, and trunk. An anatomical representation of the head, in turn, yields an A-composition which includes eyes, ears, nose and mouth. This yields an appropriate "explosion" of the person according to the anatomical representation. On the other hand, an *inconsistent view* might mix an anatomical view of a person with a molecular view of the head.

In the general case, a consistent representation of the structure of the A-composition of an object can be expressed by *V-Composition* or $\mathcal{C}_V$ which is defined recursively as follows.

DEFINITION 4.12 *The* V-Composition *of a simple object is itself. The* V-Composition *of a complex object x is the union of the $\mathcal{C}_A(x)$ and the V-Composition of each component of the A-Composition.*

That is, suppose $\mathcal{C}_A(x)$ consists of n objects $a_1, a_2, ..., a_n$ and denote the union of the V-compositions of the a_i as

$$V_c = \bigcup_{i=1}^{n} \mathcal{C}_V(a_i).$$

Then,

$$\mathcal{C}_V(x) = \mathcal{C}_A(x) \cup V_c$$

where $\mathcal{C}_V(x) = x$ when x is simple.

From this point on, unless explicitly specified, any reference to the *composition* of an object will always mean *V-Composition* since this provides a consistent internal structure of an object. V-composition is represented in all model objects by a state function fComp such that

$$\mathsf{fComp} : S_K \rightarrow 2^{\Theta}.$$

[48]That is, an object representing a person.

An object x in the composition of another object y is said to be *part-of* y, in symbols $x \sqsubset y$. An object may be part-of more than one object.

DEFINITION 4.13 *The* part-of context *of an object x is defined as the set of all objects of which x is part-of.*

All model objects provide a state function

$$\mathsf{fPartOf}: S \rightarrow 2^{\Theta}$$

where Θ is a set of objects. This operationalizes Bunge's definition of the *part-whole* relation.[49]

Figure 4.2, adapted from Bunge [1977, p.48] shows the relationships among S-composition, A-composition, and V-composition. Different surrogates are depicted by different geometric shapes. The figure at the top of the diagram shows a composite surrogate composed of four separate simple surrogates. Suppose that the object for the composite surrogate is z_0 and that three composing surrogates are represented by objects x_1, x_2, and y_1. The fourth surrogate is represented by two objects, y_2 and y_3 (ie. there are two different views of this surrogate). Suppose further that the four surrogates associate in different ways as shown with corresponding objects z_1, z_2, z_3 and z_4. Then,

$$C_S(z_0) = \{x_1, x_2, y_1, y_2, y_3, z_1, z_2, z_3, z_4\}.$$

Now, suppose that for z_0,

$$\mathsf{fKinds} = \{Z_1, Z_2\}$$

and z_1 and z_2 are in the scope of models Z_1 and Z_2 respectively. Then,

$$C_A(z_0) = \{z_1, z_2\}.$$

[49]Definition 1.2 [Bunge, 1977].

Further, suppose that for object z_1,

$$\mathsf{fKinds} = \{Y_1, Y_2\}$$

and y_1 and y_2 are in the scope of models Y_1 and Y_2. Finally, assume that for z_2,

$$\mathsf{fKinds} = \{X_1, X_2\}$$

and x_1 and x_2 are in the scope of models X_1 and X_2. Then,

$$\mathcal{C}_V(z_0) = \{x_1, x_2, y_1, y_2, z_1, z_2\}.$$

The properties of a thing can be affected by its composition or the things of which it is part-of.[50] This concept is operationalized in the model by allowing the law statements in the model for an object to reference the state functions of any object in its composition, A-composition or part-of context. This is clarified further in the next section.

4.5.4 Object Interaction

In the world, things interact with each other. That is, the properties of a thing can be affected by some other thing.

Bunge defines *interaction* or *bondage* between two things as occurring whenever the *history* or *trajectory* of one is affected by the other.[51] In his ontology, an aggregation of interacting things is a *system.*

> [In the case of a system], here the history of every component is determined at least partly by the states other components are in, so that the history of the whole does not equal the sum of the individual histories [p. 263].

[50][Bunge, 1979, Chapter 1].

[51]Definitions 5.27, 5.29, 5.30, 5.32 and 5.33. Refer also to Chapter 3 of this thesis.

Although Bunge defines the general concept of interaction, his ontology does not discuss specific ways by which interaction is achieved. In nature, the interaction of things would involve natural forces and mechanisms which could, at least partly, be described by the laws of the things concerned.

In OBCM, two mechanisms are introduced to model interaction: (1) local change: the invocation of change functions of one object by another, and (2) interobject change: inter-object referencing of state function values in the law statements of a model object.[52]

The first mechanism was briefly described in section 4.4.1 under the section on change functions. As an example of the second mechanism, consider some object p_t composed of objects p_1 and p_2. Assume further a state function fCost which represents the cost of the things corresponding to p_t, p_1 and p_2. If the cost of the thing corresponding to p_t is summative of its components, the cost of p_t can be represented by a law statement

$$\mathsf{lCost}: \quad p_t.\mathsf{fCost} = p_1.\mathsf{fCost} + p_2.\mathsf{fCost}$$

where p_t.fCost, p_1.fCost and p_2.fCost are the values of fCost for p_t, p_1, and p_2 respectively. That is, the values which can be taken on by p_t.fCost are determined by the law statement lCost. Such law statements are called *interobject law statements*.

The next section presents some notation to facilitate the expression of law statements.

4.6 An OBCM Notation

The basic constructs in OBCM are objects and model objects with their associated state functions, law statements and change functions.

This section develops a notation for dealing with these constructs. The format of the

[52]This is a relatively simple model of change. Other models are certainly possible. Unfortunately, the change models cited by Bunge [Appendix B, 1979] are too general and not directly applicable to this thesis. However, the development of a full model of change is beyond the scope of this thesis.

notation is functional, thus facilitating nested application.[53]

Much of the notation is drawn from conventional set theory. Despite the formal appearance of the notation, however, the ideas which it expresses invariably have simple, readily understood, natural language versions.

4.6.1 Names

Names of state variables, law statements and change functions need not be unique. Where potential ambiguities arise, names can be qualified with model object names or object names. For example, CUSTOMER.fDebt refers to the function fDebt $\in$ **F** of the model object CUSTOMER. Similarly, the law statement CUSTOMER.lComp refers to the law statement lComp $\in$ **L** of the model object CUSTOMER. And, the change function CUSTOMER.cInit refers to the event function cInit $\in$ **C** of the model object CUSTOMER.

Although state function values can be expressed in conventional functional notation such as $F_i.(s)$ where s is some surrogate, a more convenient notation is by reference to the object of interest. For example, for some object x, the value of state variable F_i is $x.F_i$.

An object can refer to itself with the keyword self. For example, a law statement constraining the composition of a simple object can be written

$$\begin{aligned} &\mathsf{lSimple}: \\ &\mathsf{fComp} = \{\mathsf{self}\}. \end{aligned}$$

In plain language, this *law of simple objects* says that the only object in the V-composition of a simple object is itself.

A further naming convention deals with references to *objects* as opposed to *model objects*. Often, references will be made not to specific objects but to all objects of a kind.

[53]The notation described here does not constitute a complete or formal language for OBCM. Such an undertaking is beyond the scope of the current work.

As a naming convention, where the context is clear, the name of the model object by itself may refer to all objects in the scope of the model object. For example, one may say that Q has a certain composition rather than "all objects of kind Q have a certain composition".

Finally, where the context is clear, given a model object X_m, its law statements and state variable names will be understood always to refer to all objects in the scope of X_m. For example, the law lSimple given above is equivalent to

$$X_m.\mathsf{lSimple}:$$
$$x.\mathsf{fComp} = \{\mathsf{self}\}$$

for all objects x in the scope of X_m.

4.6.2 Basic Object Functions

As modelling constructs, objects and model objects have a number of basic functions or operators.

The surrogate of an object x is returned by a function S_g on the domain of objects to the co-domain of surrogates, ie.

$$S_g : \Theta \rightarrow S,$$

where Θ is the set of objects in the system and S the set of surrogates. For example, any object can refer to its surrogate as $S_g(\mathsf{self})$.

The objects for a surrogate S are returned by a function O_b on the domain of surrogates to the powerset of objects, ie.

$$O_b : S \rightarrow 2^{\Theta},$$

where Θ is the set of objects in the system and S the set of surrogates. For instance, assume that some individual JOHN in the real world with corresponding surrogate 'JOHN' is

represented by object j_1 of type CUSTOMER and j_2 of type PERSON. Then, O_b('JOHN') returns the set $\{j_1, j_2\}$.

A useful notational convention is the function O_{Sg} which is simply $S_g \circ O_b$. This returns all other objects that have the same surrogate as some particular object. That is, for any object x,

$$O_{Sg}(x) = O_b(S_g(x)).$$

For example, given j_1 and j_2 above, $O_{Sg}(j_1)$ returns the set $\{j_1, j_2\}$. More simply, the function O_{Sg} can be said to return all *alternative views* of object x.

Figure 4.3 illustrates the relationships among thing, surrogate, model object and objects expressed by these functions.

4.6.3 · Object Sets

A number of useful functions on *sets* of objects are now defined.

Functions of Object Sets

Where all the objects in a set are of the same kind, the following notation is introduced to refer to the list (or vector) of values for a particular state function F_i for the set:

$$\{x_1, x_2, ..., x_n\}.F_i,$$

where all x_i are of the same kind. For example, assume the set {*john, mary*} of type PERSON and state function

PERSON.fAge: $S_K \rightarrow$ *Integers.*

The notation {*john, mary*}.fAge refers to the set {*john*.fAge, *mary*.fAge}. That is, it refers to a vector of the age of each person represented by the set of objects {*john,*

mary}.[54]

Object Set Partitioning

Often, there is an interest in partitions or subsets of some set of objects. Given a set of objects, a relation $\sim_n$ can be defined which returns a subset of the original defined in some nth respect. For example,

$$\sim_n: \Theta \rightarrow \Theta_n$$

where $\Theta_n \subseteq \Theta$. Such partitions are expressed in the form $\Theta_S / \sim_n$, where Θ_S is a set of objects.

For instance, the A-composition of an object is given by fAComp. The set of objects of type Q in the composition of the object is written

$$\mathsf{fAComp}/ \sim_{\mathsf{Q}} . \tag{4.1}$$

Similarly, the set of objects of type Q in the part-of context of the object is

$$\mathsf{fPartOf}/ \sim_{\mathsf{Q}} . \tag{4.2}$$

This partitioning also allows a specific view of an object to be returned as follows:

$$O_{Sg}/ \sim_{\mathsf{Q}} . \tag{4.3}$$

To explain further, O_{Sg} returns a set containing all alternate views of the object (ie. objects with the same surrogate as the given object). The partition $\sim_{\mathsf{Q}}$ then returns only those objects of type Q. The final result is an alternative *view* (ie. object) of type Q of the original object.

[54]This implies that x.fValue will return values either for a single object or a set depending upon whether x is an object or a set of objects.

To take a trivial example, given CUSTOMER object j_1 and PERSON object j_2 as described previously, the PERSON view[55] of j_1 is returned by

$$O_{Sg}(j_1)/\sim_{\text{PERSON}} .$$

Similarly, the CUSTOMER view of j_2 is given by

$$O_{Sg}(j_2)/\sim_{\text{CUSTOMER}} .$$

The relation $\sim_n$ can be more complex than the above. For example, the expression

$$\mathsf{fAComp}/\sim_{(\ Q\ \&\ \mathsf{fCost}\ =\ y)} \tag{4.4}$$

returns the subset of fAComp of type Q with values of Q.fCost equal to y.

Objects to Scalars

The following notation defines a class of functions which maps any set of objects to a scalar value, ie.

$$\mathcal{Q} : \{x_1, x_2, \ldots, x_n\} \rightarrow scalar$$

where x_i are objects. In particular, the function $\mathsf{qSize} \in \mathcal{Q}$ is defined such that given a set of objects Θ, $\mathsf{qSize}(\Theta)$ returns the *cardinality* (or number, size, population, etc) of objects in Θ.

For example,

$$\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathsf{Q}}) \tag{4.5}$$

returns the number of objects of kind Q in the A-composition of an object. This allows a convenient notation for expressing the cardinality, size or population of any particular group of objects.

[55] Ie. the object of type PERSON with the same surrogate as j_1.

Value Sets to Scalars

The notation

$$\Theta.F_i$$

refers to the vector of the values of state function F_i over the set of objects Θ. The following class of functions maps these value vectors to a scalar:

$$\mathcal{V} :< v_1, v_2, ..., v_n > \rightarrow scalar$$

where v_i are values. For instance, the function $\Sigma \in \mathcal{V}$ is defined such that given $V =< v_1, v_2, ..., v_n >$, $\Sigma(V)$ returns the sum of the individual values in V.

For example, the expression fAComp/ $\sim_Q$ denotes all objects of type Q in the A-composition of an object. The expression (fAComp/ $\sim_Q$).fValue denotes a vector of values of the state variable fValue for each object of type Q in the A-composition of the object. And the expression

$$\Sigma[(\text{fAComp/ } \sim_Q).\text{fValue}] \tag{4.6}$$

returns the sum of this vector. Suppose, for example, that an object represents a sales order, composed partially of several objects of type Q, each representing some specific item. This expression denotes the sum of the values of the items in the order.

4.7 A Visual Notation for OBCM

This section presents a simple visual notation to describe OBCM objects. The technique is a modification of a visual formalism known as *higraphs* [Harel, 1988]. Higraphs have been shown to be useful in diagramming E-R schemes, activity charts, state diagrams and generally, any situations which involve set-theoretic structures. Although the visual notation described here is based on higraphs, it is not defined as formally as in [Harel, 1988].

Figure 4.4 shows the basic format for the modified higraph used in OBCM. An object is depicted as a rounded rectangle. State functions are shown as circles along the top of the rectangle. Law statements are represented by smaller rectangles along the side. Change functions are also diagrammed as circles, but along the left side of the rectangle. The concept of *composition* of an object is diagrammed in a special way, by showing the composing objects directly within the original rounded rectangle.

The diagram in Figure 4.4 can actually refer to either a model object or a specific object. The name of the model or object can be placed in some convenient location within the rectangle. In this case, the diagram refers to a model object named X_m. The usual notational conventions are used to distinguish between object and model names, ie. lower case italics for objects and upper case for model object names.

Figure 4.4 shows some additional information which can be conveyed by this diagramming technique. In Harel's visual formalism, a dashed line allows representation of *cartesian product* of sets. Similarly, a dashed line is used to indicate the composition of an object of type X_m and consisting of objects of types A_m and B_m. Along the bottom is a rectangle representing a law statement which regulates composition. The *1:n* notation represents the essence of the law, that the composition of X_m must consist of one object of type A_m and one or more objects of type B_m.

The arrows (edges) in the diagram represent the various relationships among state functions, law statements, change functions, and objects in the composition or part-of context. The edges in themselves, however, have no formal semantics. Their meanings are obtained from the state variables, law statements and change functions of the model object being depicted.

The edges in Figure 4.4 do suggest, however, that the value of state variable $X_m.\mathsf{f}_3$ is affected by the values of $B_m.\mathsf{f}_5$ and that law $X_m.\mathsf{L}_2$ specifies the nature of this relationship.

Figure 4.5 shows a more complex situation. Here, object x_0 is composed of several

objects, $x_1, x_2, q_1, q_2, p_1, p_2, p_3$ and p_4. The diagram also shows the composition of x_1 as $\{q_2, p_3, p_4\}$ and x_2 as $\{p_1, p_2, q_2\}$. It can be seen that q_2 is *part-of* both of x_1 and x_2. Since this diagram shows specific objects, the value for the state function $x_0.\mathrm{f}_1$ is also shown (in this case, the integer 25).

Figure 4.6 shows the invocation of change functions. The edges from c1 indicates that this change function changes the value of f1 to some value from the set of integers N obtained from a source external to the object being diagrammed. The change invoked by c2 is slightly different. Here, the edges represent a change function which requires the original value of f2 as well as a value from an external source. The change invoked by c3 shows a change to the composition of the object. Here, an object of type Y_m external to the object is brought into its composition.

In the next section, a brief example shows how both the written and visual notations are used in an OBCM description of a simple application. Note throughout the example that despite the symbolic notation, an easily understood, natural language expression is always readily available.

4.7.1 Example

Suppose a manufacturer produces vehicles constructed of 4 wheels and a motor. Let S_W be the set of surrogates corresponding to wheels, S_M the surrogates corresponding to motors, and S_V the set of surrogates corresponding to complete vehicles. Further, let WHEEL, MOTOR and VEHICLE be model objects for wheels, motors and vehicles respectively.

Worth or value is a property of all the things (ie. wheels, motors and vehicles) and is represented by the state functions WHEEL.fCost, MOTOR.fCost, and VEHICLE.fCost, which map the surrogate for each thing to a number representing dollars, ie.

WHEEL.fCost:$S_W \rightarrow$ N

MOTOR.fCost: $S_M \rightarrow$ N

VEHICLE.fCost:$S_V \rightarrow$ N

where N is the subset of real numbers for expressing dollars.

Now, let v be an object representing a specific vehicle. Let w_1, w_2, w_3, w_4 be objects representing four different wheels and let m be an object representing a motor. The state variable v.fAComp represents the A-composition of v. A law statement can be formulated to constrain the values of v.fAComp as follows:

$$\begin{aligned} &\mathsf{lAComp}: \\ &\quad \mathsf{qSize[fAComp/} \sim_{\text{WHEEL}}] \leq 4 \ \ \& \\ &\quad \mathsf{qSize[fAComp/} \sim_{\text{MOTOR}}] \leq 1. \end{aligned}$$

The law statement lAComp states that the A-composition of objects of type VEHICLE contains a maximum of four wheels and one motor.[56]

The law statements of an object can reference state variables of objects in its composition or part-of context. Suppose the cost of a vehicle is determined by the manufacturer to be the sum of the costs of its wheels and motor. Then, a law statement can be defined to express this relation by the following steps.

- The object representing the motor in v is

$$\mathsf{fAComp/} \sim_{\text{MOTOR}} .$$

- The cost of the motor is represented by

$$(\mathsf{fAComp/} \sim_{\text{MOTOR}}).\mathsf{fCost}.$$

[56]That is, a maximum of four *objects* of type WHEEL and one *object* of type MOTOR. This particular formulation of lAComp does not limit the A-composition of VEHICLE only to objects of type WHEEL and MOTOR. Nor does it require objects of type VEHICLE to always have a certain A-composition (it could be, for example, a frame travelling down a production line).

- The objects representing the wheels in the composition of v are

$$\mathsf{fAComp}/\sim_{\mathsf{WHEEL}}.$$

- The cost of all the wheels is represented by

$$\Sigma[(\mathsf{fAComp}/\sim_{\mathsf{WHEEL}}).\mathsf{fCost}].$$

- The cost of the vehicle represented by v then is expressed by the law statement:

$$\begin{aligned}&\mathsf{lCost}:\\ &\mathsf{fCost} = (\mathsf{fAComp}/\sim_{\mathsf{MOTOR}}).\mathsf{fCost} \ +\\ &\Sigma[(\mathsf{fAComp}/\sim_{\mathsf{WHEEL}}).\mathsf{fCost}].\end{aligned}$$

  More informally, the law statement lCost defines the value of fCost for vehicles to be the sum of the cost of motor and the wheels of which it is composed.

The result of the above is equivalent to

$$v.\mathsf{fCost} = m.\mathsf{fCost} + w_1.\mathsf{fCost} + ... + w_4.\mathsf{fCost}$$

for any particular v with A-composition $\{m, w_1, w_2, w_3, w_4\}$.

Figure 4.7 depicts an object of type VEHICLE using the visual notation. In addition to diagramming the state variables and law statements discussed above, Figure 4.7 also shows two change functions, cM and cW. These functions can be invoked to add objects of type MOTOR and WHEEL to the composition of VEHICLE, subject of course to the constraints of lAComp.

4.8 OBCM and Semantic Modelling - Comparative Analysis

This section describes how OBCM supports various abstraction mechanisms commonly used for semantic data modelling.

A-Composition, for example, can be viewed as the direct equivalent of the *aggregation* abstraction mechanism when applied to aggregation of physical entities. The state functions and law statements of OBCM allow equivalents to other common modelling abstractions found in the conceptual modelling literature.

4.8.1 Pre-Defined vs Law-Based Models

In general, it can be observed that many conceptual modelling approaches described in the literature attempt to support one or more *pre-defined* structures which are considered useful for modelling the world. These structures must be generalized enough to meet a wide range of situations and intuitive enough to be understandable to users. IS designers then attempt to fit real world entities and phenomena into these pre-defined structures.

In contrast, OBCM is conceived as a collection of objects with no pre-defined structuring except for the basic concept of association which results in the composition and part-of relationships. It relies on the use of law statements to define and constrain the relationships among the state function values of objects. Each situation must be analysed individually and distilled into a set of law statements describing the situation.

On the other hand, it can be shown that many of the common modelling abstractions can be developed by selecting an appropriate set of laws. Further, they can be viewed as specializations of Bunge's concept of association. This provides the basis for a more generalized form of data modelling which can be called *law-based.*[57] The next few sections discusses how OBCM can provide the equivalent forms of some common, well-known semantic modelling schemes.

[57]As emphasized in the introduction, this thesis does not try to prove that OBCM can model *all* applications nor, for that matter, that OBCM is a *better* modelling scheme than other models. Such claims must be based, at least partly, on usage over a period of time and empirical study. On the other hand, it might be noted that there are no intrinsic grounds for supposing that other modelling schemes such as data flows diagrams, E-R, or abstraction mechanisms can themselves model all applications. In contrast, since OBCM is based on a metaphysics which purports to explain the nature of reality, there would appear to be a stronger basis for such claims.

4.8.2 OBCM and Entity-Relation Models

The Entity-Relationship or E-R model [Chen, 1976] is a well-known, high-level modelling scheme used in database design. A typical situation in the E-R model is a *CUSTOMER* set in an *ORDERS* relationship to a *PRODUCT* set. *Cardinality rules* are used to describe aspects of the *ORDERS* relationship. Customers and products may have certain *attributes* which describe them. Alternately, *ORDERS* is converted into a so-called *weak entity* to which *CUSTOMER* is in a *PLACE* relationship and *PRODUCT* is in a *FOR* relationship (such as in Figure 4.8 adapted from [Hall & Mosevich, 1988]).

As the above example shows, in E-R modelling, the distinctions between entities, relationships and even attributes are open to interpretation. OBCM, however, requires a strict ontological interpretation upon the application, ie. the world consists of things which are grouped into systems or aggregates; things have properties which can be represented by attributes or values; everything abides by laws.[58] The equivalent to E-R modelling is achieved in OBCM by viewing a relationship as a composite thing.[59] For example, an order can be viewed as a *thing* composed of a customer thing and one or more product things. Since the order is a thing, it has certain properties, particularly emergent ones such as its value and composition, as well as certain laws or rules by which these properties are governed.

All things of interest, *customers, products* and *orders*, are represented in the OBCM as objects. The state functions of each object represent the relevant properties of these things. For instance, in an ORDER object,[60] an emergent property, *order value*, can be represented by a state function fValue whose value is determined by a law statement relating ORDER.fValue to the sum of PRODUCT.fValue of all objects of type PRODUCT

[58]Assumptions M1 - M10 [Bunge, 1977].

[59]In a different context, others have also noted that the relationships in the E-R model can be generalized as a composite object [Hull & King, 1987].

[60]That is, an object of type ORDER.

in the composition of the ORDER object, ie.

$$\text{ORDER.fValue} : S_O \rightarrow \mathbf{N},$$

(S_O is a set of surrogates corresponding to the orders of a firm) and

$$\begin{aligned}&\mathsf{IValue}:\\ &\mathsf{fValue} = \Sigma[(\mathsf{fAComp}/\sim_{\text{PRODUCT}}).\mathsf{fValue}].\end{aligned}$$

4.8.3 Cardinality Constraints in OBCM

The *cardinalities* in the E-R model can be expressed by law statements constraining the composition or A-composition of objects. For example, a law statement may restrict the number of CUSTOMER and PRODUCT objects allowed in the composition of ORDER as follows:

$$\begin{aligned}&\mathsf{IAComp}:\\ &\mathsf{qSize}(\mathsf{fAComp}/\sim_{\text{CUSTOMER}}) = 1\ \&\\ &\mathit{minsize} \leq \mathsf{qSize}(\mathsf{fAComp}/\sim_{\text{PRODUCT}}) \leq \mathit{maxsize}.\end{aligned}$$

That is, the law statement IAComp restricts the A-composition of ORDER objects to exactly one CUSTOMER object and a range $< \mathit{minsize}, \mathit{maxsize} >$ of PRODUCT objects.

4.8.4 Classification and Generalization

Variations on classification or IS-A constructs are also supported since OBCM provides multiple views of any thing of interest. For instance, an individual in the world can be represented by several models such as STUDENT, GRAD-STUDENT, PERSON, etc., each with their own state functions, law statements and change functions.

One form of IS-A is *generalization with strict inheritance* where an object type *must* be a specialization of another. For example, suppose that all STUDENT objects must

also have a PERSON view (ie. all students are also persons). In this case, the law of strict specialization can be formulated as:

$$\begin{aligned}&\text{STUDENT.IIsA}:\\&\quad \mathsf{qSize}[O_{Sg}(\mathsf{self})/\sim_{\text{PERSON}}] = 1.\end{aligned}$$

That is, there must be exactly one object of type PERSON among all the alternate views of a STUDENT object.

Similarly, the equivalent of property inheritance can be achieved by law statements of one object referring to state variables of other objects. For instance, assuming the strict specialization above, the state variable STUDENT.fName can be defined to take its value from the state variable PERSON.fName as follows:

$$\begin{aligned}&\text{STUDENT.IName}:\\&\quad \text{STUDENT.fName} := O_{Sg}(\mathsf{self})/\sim_{\text{PERSON}}\text{.FNAME.}\end{aligned}$$

That is, STUDENT name must be the same as the name of the PERSON view of the STUDENT object. Or more informally, a student's name must be the same as his or her personal name.

OBCM can also be used to support a more permissive form of generalization where a strict inheritance is not necessarily required. For example, the customers of a firm can be other firms, individuals (ie. persons), or organizations such as government agencies. Suppose that individuals are represented by the model object PERSON and customers are represented by the model CUSTOMER. The following law statement relates objects of kind CUSTOMER to any existing PERSON view:

$$\begin{aligned}&\text{CUSTOMER.IName}:\\&\text{CUSTOMER.fName} := \begin{cases}\text{PERSON.fName if } \exists\, O_{Sg}(\mathsf{self})/\sim_{\text{PERSON}}\\ \textit{External source} \text{ otherwise.}\end{cases}\end{aligned}$$

This expresses the idea that if the customer is also a person, then the customer's name is the same as that person's name. Otherwise, the customer's name comes from an external source.

4.8.5 Aggregation

Aggregation is interpreted in a number of different ways in the literature. As discussed in Chapter 2, distinctions can include Cartesian, cover and statistical aggregation.

Cartesian aggregation refers to an aggregation of properties, eg. a PERSON is an aggregate of properties such as NAME, ADDRESS, and so forth. The set of state functions **F** in a model object directly captures this idea in OBCM. Cover aggregation refers to an aggregation of entities, eg. a VEHICLE is an aggregate of WHEELS and a MOTOR. This abstraction is directly captured by the concept of object *composition.* Statistical aggregation provides summary information. This is reflected in the notion of *emergent properties.* For example, the number of items in the composition of an object in OBCM is an attribute representing the emergent property of population, cardinality, or numerosity (ie. number of components) of the composite object.

In some cases, not only are the properties of the aggregate dependent upon the properties of its components but certain properties of a component may depend upon one or more properties of the aggregate. For example, an ORDER object may consist of a CUSTOMER object and one or more PRODUCT objects. The ORDER object will have a state function fValue representing the value of the order being represented. On the other hand, one of the state functions of the CUSTOMER object is fDebt which represents the total indebtedness of the customer being represented. The value of fDebt, however, depends upon fValue of all orders of which the customer is in a *part-of* relationship. This

idea is easily expressed by a law statement for CUSTOMER such as:

$$\mathsf{lDebt}:$$
$$\mathsf{fDebt} = \Sigma[(\mathsf{fPartOf}/\sim_{\mathrm{ORDER}}).\mathsf{fValue}].$$

In plain English, the law states that the debt of a customer is the sum of all orders of which the customer is part-of.

4.8.6 Association

Association is defined as an abstraction in which a collection of entities is considered a higher-level set [Brodie, 1984], although, this abstraction is not always clearly distinguished from classification [Peckham & Maryanski, 1988]. Generally, it is conceded that associations consist of similar, and not different, entities [Brodie, 1984].

In OBCM, abstractions equivalent to association can be modelled by explicitly formulating appropriate law statements. For example, a *club* might be viewed as an association of individuals. It may be represented by an object of type CLUB composed of individuals of type PERSON with an explicit law statement constraining all the objects in the composition of a CLUB object to be of type PERSON. One way of expressing this might be with a law of strict composition similar to

$$\mathsf{lComp}:$$
$$\mathsf{qSize(fComp)} = \mathsf{qSize(fComp}/\sim_{\mathrm{Q}})$$

which constrains composition to objects of a certain type only. For example, in the example of CLUB,

$$\mathrm{CLUB.lComp}:$$
$$\mathsf{qSize(fComp)} = \mathsf{qSize(fComp}/\sim_{\mathrm{PERSON}}).$$

This simply states that the composition of CLUB must consist only of objects of type PERSON.

4.9 Summary

This chapter has presented an Ontology-Based Conceptual Modelling (OBCM) scheme for describing information systems. This scheme is based on ontology or a system of metaphysics and not on concepts based on computing or implementation technologies. As such, it can be seen as one answer to the call for a more theory-based approach to information systems development [Wand, 1988; Floyd, 1986; Bubenko, 1986].

In summary, the model

- attempts to ground IS conceptual modelling in a consistent and formal metaphysics with a theoretical foundation for its basic constructs (eg. thing, property, change, state, etc.) and
- provides a representation for both structural and dynamic aspects of reality through state variables, law statements, and change functions.

Through examples and comparison to other conceptual modelling approaches, this chapter shows how a small number of ontologically-based concepts might be powerful enough to model a range of information systems and applications. The model appears general enough to encompass many of the features of other modelling techniques such as Entity-Relation and abstraction mechanisms such as generalization, aggregation and association. It further supports a concept of multiple views.

OBCM is a formal model and uses a formal notation. However, its ontological basis gives its formalism a straightforward, intuitively understood interpretation. For every situation described by its formal notation, a relatively simple, ontology-based description is possible, using natural language terms. This chapter has also proposed a simple and useful visual notation for OBCM.

The next chapter describes how OBCM leads to direct implementation into a working

system and details how support is provided to model the dynamic or behavioural aspects of an application.

Chapter 5

Object-Based Information Systems - An Implementation Framework

This chapter describes the relation between OBCM and its implementation as an Ontology-Based Information System or OBIS.

Like OBCM, the structure and operation of an OBIS derives directly from Bunge's ontology. Its design principles attempt to preserve a direct relationship between the conceptual (OBCM) description of the application and its implementation in a working system. And its underlying motivation is the seamless, homogeneous path from a conceptual model of the application to an implementation, as was first discussed in the introduction to this thesis.

5.1 Conceptual Framework

5.1.1 OBIS - Definition

The overall framework for OBIS results from Bunge's conceptualization of the *world* or the *universe*:

> POSTULATE: There exists a thing such that every other thing is part-of the former. That thing is unique and we call it *the world.*[1]

Similarly, an Ontology-Based Information System is conceptualized as *the total aggregation of all the objects of concern to the application.* This aggregation is itself treated as an object representing the Universe of Discourse of the application. In other words,

[1]Postulate 1.2 & 3.3 and Definition 1.3 [Bunge, 1977].

the OBIS is simply another object described by its own state functions, law statements and change functions. Conceptually, then, system implementation is homogeneous and consistent with its underlying OBCM description in the sense that no other additional constructs are required other than those related to object and model object.

As with any other object, the OBIS has a *composition.* Objects in this composition have, in turn, compositions of their own. One possible (although not the only) way of looking at the OBIS is as an object with an *A-composition* containing one or more objects representing specific applications areas. Each of these application objects has, in turn, its own A-composition containing other appropriate objects. This recursion can be repeated to any arbitrary level of granularity as required for the application.

For example, in a data processing application, the OBIS may have an A-composition of several *application objects* such as *Order*, *Payroll*, and *Inventory* objects. In turn, each application object is composed of other relevant objects. The *Order* object might, for instance, be composed of objects representing customers and products for sale. The composition of the *Payroll* object may include objects representing the employees of the firm. Figure 5.1 shows a possible OBIS composed in this way.

The preceding can be more formally expressed by the following.

POSTULATE 5.1 *Given a set of surrogates S_K each corresponding to some thing of interest in an application domain $\mathcal{A}$, there exists an individual $\diamond \in S_K$ such that every other element in S_K is part-of it.*[2] *Ie.* $(\exists x)[x \in S_K \;\&\; (\forall y)(y \in S_K \Rightarrow y \sqsubset x)]$.

DEFINITION 5.1 *An OBIS for the application domain $\mathcal{A}$ is an object for $\diamond$.*

That is, an OBIS is an object or representation for the aggregation $\diamond$ of the surrogates corresponding to the things of interest in the application domain. As with any surrogate, $\diamond$ may be represented by more than one model object.

[2]In Bunge's notation as described in Theorem 1.2, $\diamond = [S_K]$.

Since the composition of $\diamond$ consists of all surrogates in S_K, an OBIS provides information about any thing of interest in the application domain $\mathcal{A}$. Access to the constituent components of an OBIS can be described by the following.

POSTULATE 5.2 *There exists an element* $\square \in S_K$ *such that for any* $x \in S_K$, $x \circ \square = x$. $\square$ *is part-of every element in* S_K.

THEOREM 5.1 *The quadruple* $< S_K, \diamond, \circ, \square >$ *is a semi-lattice with least element* $\square$ *and last element* $\diamond$.

Proof. The proof is similar to Bunge's.[3] There exists a supremum for any two elements of S_K, namely their association. $\square$ is part of every individual. $\diamond$ contains every individual. These conditions define a sup-semilattice.

The nodes of the lattice are the surrogates and the edges are the *part-whole* relationships $\sqsubset$ formed by the $\circ$ operation.

COROLLARY 5.1 *Given an OBIS, an access path is available from an object to any other in the OBIS by virtue of its composition and part-of state functions.*

Proof. By Theorem 5.1, $< S_K, \diamond, \circ, \square >$ has the semi-lattice structure. Since an OBIS is an object for $\diamond$ and each object in the OBIS has a surrogate in S_K, an access path exists between any two objects.

This implies that conceptually[4], at least, the OBIS does not require a separate navigation mechanism to access its components other than *composition* or *part-of*. In other words, access to the components of the OBIS is completely defined on strict ontological grounds.[5]

[3]Theorems 1.3 & 3.1.

[4]Ignoring, for example, implementation or efficiency issues.

[5]Compared to, for example, Information Systems which require user knowledge of indexes, keys, relational joins, etc.

5.1.2 OBIS Design Approach

The implementation process does not include the explicit design phase found in some methodologies [Freeman & Wasserman, 1984]. Instead, the IS development process for OBCM/IS proceeds directly from analysis to implementation. Analysis involves not only understanding the application but also *interpreting* it in ontological terms so that it can be modelled by OBCM. Then, the OBCM is directly converted into a working OBIS without an intervening design phase in the usual sense.

The effect is to reduce the number of steps in the IS development process. Further, since both the conceptual model and implementation share the same representational constructs, ie. objects, there are fewer opportunities for "semantic gaps" to occur between the application world and the final IS. The goal is an IS implementation which is close to the original conceptual model of the application, is easily understood, and appeals to the intuitions of developers and end-users. The process for OBCM/IS development will be illustrated later in this chapter through an extended example.[6]

Since all the components of the OBIS including the OBIS itself are objects, in principle, a single, common concept of examining, manipulating and otherwise interfacing with objects is sufficient to provide a complete user-interface to the Information System. The two basic forms of end-user interaction with objects involve *inspection* of objects and *change* to objects.

5.2 The OBIS User Interface

5.2.1 Direct Manipulation Style

In OBIS, users interact directly with objects in accordance with ontological principles.

[6]Development of a complete ISD methodology for OBCM/IS is beyond the scope of this thesis. However, the principles presented here should form the foundations for any future practical methodology.

The aim is to achieve a close parallel between object manipulation and the way things are manipulated in the real world. Users work with constructs, operations, and manipulations in the OBIS which are semantically closely related to the real-life entities and phenomena they encounter in the application world. The overall result is a formalism to support the *direct manipulation* style of user interface [Schneidermann, 1983], in which the user enjoys a sense of manipulating real world entities when working with the system.

In this way, OBCM/IS is a representation which is not only *descriptive* of some aspect of reality but also can be *examined* and *manipulated* like the things in that reality. Thus, the representation can be richer and potentially more powerful than conventional IS representations such as files, records and programs.[7]

5.2.2 End-User Interface

The accessing of state variables by end-users is called *inspect*[8] or *view*. The invoking of change functions to modify the values of state variables is called *change*.

- *View.* This involves the examination of objects, more precisely, of the state of objects (ie. the values of the state functions). The underlying implementation technology must be capable of displaying these values in some appropriate manner

[7]Such representations may have significant intuitive power. For instance, Mattessich [1987] presents some interesting archaeological evidence of the use of tangible, visually identifiable *clay tokens* for sophisticated accounting and record-keeping purposes, even before the invention of writing and abstract counting systems. These fire-hardened artifacts from the Fertile Crescent in the Middle East date as far back as 8000 B.C. They often possessed *morphological* similarities (ie. in shape or by inscriptions) to the things being accounted for (eg. sheep, jars of oil, measures of grain). More importantly, however, tokens were often used in such a way that the "*morphological tokens and pictographs not only describe structures, they themselves are similar structures*". These prehistoric "accounting systems reveal the logical, indeed set-theoretical, structure inherent in certain economic aspects of reality [pp. 88 - 89]".

Similarly, objects in OBCM/IS are not only descriptive by virtue of their state variables but also reflect the composition and structure of the things they represent. For instance, by these standards OBCM/IS objects are inherently more powerful representations than *icons* which normally do not reflect such detailed structure.

[8]A number of OBIS terms, including *inspect*, are borrowed from the terminology used in the Smalltalk programming environment.

(eg. icons, images or text on a video display). Examining *sets* of objects (such as the *composition* of an object) is referred to as *scanning* or *browsing.*[9]

- *Change.* Change is defined as modification of the values of state functions. Change functions can be specified to modify the value of any state variable of interest, including the composition of objects. As will become apparent shortly, changing the composition of an object is an especially important manipulation in the OBIS. All changes must be lawful, that is, subject to validation with respect to the law statements of all objects affected by the change.

To give an intuitive example, *Order Entry*, a typical data processing application, can be modelled as a collection of three kinds of objects with the models CUSTOMER, PRODUCT and ORDER.

The real world activity of generating an *order* can be represented by aggregating together a CUSTOMER object and one or more PRODUCT objects into a composite ORDER object.[10] The law statements, L, declared in the models CUSTOMER, ORDER and PRODUCT ensure that the composition of the ORDER object is lawful. For instance, ORDER objects must have only one customer in their composition but can have several products. Or, customers may not be *part-of* orders which exceed a certain value. Since the interface involves objects only, the user is not concerned with technology-dependent operations typically found in traditional information systems such as file updates, record inserts, or manipulation of relations.

View and *change* can be related to fundamental and explicitly stated ontological principles. *View* is based on Bunge's assumption of an external world subject to inspection

[9]Again borrowed from Smalltalk terminology.

[10]A suitable graphical interface such as user manipulation of visual representations of customer and products, etc., can be employed to enhance and emphasize the user's perceptions and intuitions regarding this activity.

and inquiry.[11] *Change* assumes the principle of inherent flux in the world.[12] The next two sections discuss view and change in more detail.

5.2.3 View

Display Technology

Values of an object's state functions can include conventional numeric values, character strings, and sets of objects such as those returned by state functions representing *composition* or *part-of context*. *View* or *inspect* refers to displaying these values to the end-user. The term can also refer to the state function fHistory which returns an ordered set of *time/value* pairs giving the history of an object.

Normally, the actual physical display of information in an IS is technology-dependent. To minimize this dependency, in OBIS, the *values* of state variables can be thought of as being received and interpreted by the implementation technology which then presents or displays them to the user in an appropriate manner (Figure 5.2).

In many cases, values such as numbers and text strings are presented to the user conventionally. However, this approach also makes it possible to introduce state functions such as

$$\mathsf{fForm} : S_K \rightarrow \mathrm{G}$$

where G is a set of implementation-specific values or codes which is interpreted by the implementation technology as color, shape, placement and other parameters for displaying the object on the physical hardware. This state function can be viewed as representing certain properties of identity, shape, or form of the thing represented by the object and returning a value which can be appropriately interpreted by the implementation platform (such as, for instance, control sequences to drive a physical display device).

[11] *M1 There is a world external to the cognitive subject* [Bunge, 1977, p. 16].
[12] *M2 Every thing changes* [Bunge, 1977, p. 17].

Scanning or Browsing

Sets of objects such as the *composition* or *part-of context* of an object in OBIS are examined by *scanning* or *browsing.* Because of the lattice structure of the objects in the OBIS, users can reach any object in the system by scanning the composition or part-of context of any other object in the system.

The actual scan mechanism is implementation dependent. In particular, implementation specific mechanisms will be required to manage browsing large sets of objects.

In a realistic application, the composition of some objects (such as the OBIS itself) will be too large to display each component individually and some method of limiting the search space is required. For example, instead of immediately attempting to browse through the entire composition of an object, it may be more useful to initially *scope* the composition in terms of its models. That is, instead of returning a set of the individual objects in the composition, the scan mechanism returns a list of model object names related to the objects. Then, the user need only scan the subset of objects of a selected type of interest. The subset may, of course, still be too large. In that case, further mechanisms for selecting objects may be required such as by the values of certain state variables.

5.2.4 Change

In the OBCM/IS model, changes in state occur explicitly as a result of change functions or implicitly through interobject law statements.[13] Users of the OBIS can directly change state variables in objects by invoking *change functions* (subject to satisfying the appropriate law statements for the objects being changed). Further, a direct change in one object can result in *change propagation* to other objects in the OBIS as a consequence of

[13]This is consistent with Bunge's definition of *event* or change functions as either laws or transformations compatible with these laws (Definition 5.80.)

the interobject laws statements of affected objects.

Bunge distinguishes between *induced* and *spontaneous* change.[14] In OBIS, change is always *induced*.[15] Further, all objects are subject to change. Once an object comes into existence, it can be changed by user invocation of a change function or it may change as a result of changes to objects in its composition or part-of context as determined by its interobject laws. Conceptually, many objects may be changing simultaneously and in parallel (although in practice, they are subject to the limitations of the implementation technology).

Change functions are normally implemented as procedures and can include data from sources external to the OBIS such as keyboard input by human operators, external databases, and so forth. As such, they provide the primary mechanism by which the OBIS interfaces with the outside world. But invocation of change functions is not arbitrary. Since the set of change functions is *partially ordered*, not all change functions may be available to the user at any time (or in a given state). Further, every change in an object must be valid or lawful in that the resulting object must not contradict any of the law statements of the object itself or any other affected object in its composition or part-of context.

The "user" invoking the change may be a human operator. It can also be an automated agent programmed appropriately. Since the objects in an OBIS are governed by their own local, internal law statements, there is no requirement for global integrity controls in the OBIS. Further, there is no concept of the user needing to "know" the permitted legal changes for an object. This is in contrast to traditional DBMS or programming languages where integrity rules and constraints are often coded as part of the

[14]Definitions 5.28, 5.29 & 5.30.

[15]Bunge defines spontaneity as a change in a thing where no other thing has acted upon it. The case of true spontaneous change raises a number of philosophical issues and is not incorporated in the OBCM/IS as presented in this thesis.

application software.

The next sections describe some basic principles behind a single, common concept of object change which can be applied to any object within the OBIS (including the OBIS itself). The discussion includes ideas such as change propagation, change validation, change in composition, partial ordering of change functions, history, and the creation of new objects.

Change Propagation

Users directly modify objects by invoking change functions. The impact of such changes can be propagated to other objects through interobject law statements in **L** which refer to state functions of other objects in the composition and part-of context of the original object undergoing direct change.

For example, consider the example of an object p_t composed of p_1 and p_2, and the law statement

$$\mathsf{law}_1 : \; p_t.\mathsf{fValue} = p_1.\mathsf{fValue} + p_2.\mathsf{fValue}.$$

Suppose that p_1.fValue is changed by a change function. This change will result in a corresponding change to the value p_t.fValue by virtue of the interobject law statement law_1.

Conceptually, changes are not necessarily propagated serially but may be propagated to all affected object in parallel. Thus, many objects in the OBIS can be changing simultaneously.[16]

[16]Of course this parallelism must be simulated where the implementation technology is not able to support this concept directly.

Change Validation

Any change to an object must be lawful not only for the target object itself, but also for all other affected objects in its composition and part-of environment to which change effects are propagated. In practice, however, since change functions are local to each object, it is difficult to specify these functions in sufficient detail to ensure the validity of all affected objects after their invocation. In fact, it may be difficult even to *identify* which objects would be affected by the invocation of a particular change function.

In OBCM/IS, therefore, change functions are specified as transformations of existing states into *conceivable* new states, not necessarily *lawful* new states.[17] However, the result of invoking change functions must be *validated* to ensure that the new state is indeed lawful for all affected objects.

In order to achieve this, each object in the OBIS incorporates a *validation function* $\mathcal{V}_L$ such that

$$\mathcal{V}_L : \mathbf{L} \rightarrow B$$

where $\mathbf{L}$ is the set of law statements for an object and B the Boolean results {*true,false*}. The validation function $\mathcal{V}_L$ is invoked for each affected object following invocation of a change function. In the worst case, the validation function for *all* objects in the OBIS must be invoked.

If the validation function fails, that is, returns *false*, then the change is considered unlawful and is disallowed.

Change in Composition

The *association* of things in the real world is represented in OBIS by an important type of change, change in *composition*. The converse of association is *disassociation*, where

[17] Recall Bunge's distinction between the *conceivable state space* and *lawful state space*.

objects are *removed* from the composition of an object. As some of the examples later will demonstrate, in OBCM/IS, many significant phenomena in the world can be modelled by changing the composition of objects.

As the composition of an object is changed, either by association or disassociation, at some point, the object of interest may become non-existent. That is, the object has changed so much that it can no longer be described by its model (ie. state functions, law statements and change functions). This is Bunge's principle of *nominal invariance.*[18] In other words, there is a point at which the object of concern has changed so drastically that it can no longer represent the thing it is intended to represent.

Object Creation

An object enters the OBIS by being added to its composition (or the composition of another object within the OBIS). Whenever an object is added to the composition of another, it must first be located. If it does not already exist, it must be *created.*

Object creation must be distinguished from surrogate creation, because if Bunge's ontological principles are followed strictly, there is no true concept of *creating* things and in particular, Bunge does not allow the creation of things out of nothing.[19] In OBIS, therefore, surrogates are system generated and are assumed to always exist for any thing of interest. That is, OBIS assumes

POSTULATE 5.3 *A surrogate always exists in the OBIS for any thing of interest in the application.*

Generation of a new object is, therefore, a matter of assigning a model to the *assumed surrogate* and is not strictly a creation activity. Put differently, in the OBIS, new *objects*

[18]Principle 5.1 [Bunge, 1977].

[19]*M7 Nothing comes out of nothing and no thing reduces to nothingness.* [Bunge, 1977, p. 17] and Theorems 1.5, 1.6 & 1.7 and Corollaries 1.5, 1.6 & 1.7.

(ie. *representations* or *views*) can be created but not new surrogates. Surrogates without models, of course, are of little interest to users of the OBIS since they have no descriptive or representational power.

Partial Ordering of Change Functions

In Bunge's ontology, an *event space* is a set of events, each of which represents a change in state. Events compose with each other and are partially ordered. This partial ordering is not necessarily fully connected. That is, there may be events which neither precede nor succeed each other.[20] In the OBIS, the set of change functions C is a partially ordered set. The partial ordering reflects the sequence of changes that an object may go through during its existence from creation to extinction.

In the OBIS, partial ordering is best interpreted in terms of an *entity life cycle* which is found in certain ISD methodologies and semantic data models [Jackson, 1983; Ross, 1987; Rosenquist, 1982]. More accurately, it represents a *range of possible states* that an object can take.

OBIS supports the ordering of change functions through a *partial ordering function*

$$\mathcal{P} : \mathbf{T} \rightarrow 2^{\mathbf{C}}$$

where T is the set of time instances. $\mathcal{P}$ returns the subset of C which is available to be invoked by the user at any point in time.

The partial ordering of the change functions for an object can also be defined by reference to the *current state* of the object. That is, as a function

$$\mathcal{P} : S_{\mathbf{L}} \rightarrow 2^{\mathbf{C}}$$

where $S_{\mathbf{L}}$ is the lawful state space for the object concerned. This is similar to the

[20] Definitions 5.4, 5.5, 5.6 & 5.7 and Corollaries 5.3, 5.4, & 5.5.

use of *preconditions* for defining functions found in certain programming languages and modelling schemes [Meyer, 1988].

In this thesis, a simple and informal notation is used to indicate the partial ordering as follows.

- The $\prec$ operator indicates that ordering is in effect. For example, $\{\alpha_1 \prec \alpha_2\}$ (where $\alpha_1, \alpha_2 \in \mathrm{C}$) indicates that α_1 precedes α_2.
- Otherwise, no ordering is in effect. Eg. $\{\alpha_1, \alpha_2\}$.
- Ordered and unordered functions can be mixed. Parentheses are used to group ordered and unordered functions. Eg. $\{(\alpha_1 \prec \alpha_2) \prec (\alpha_3, \alpha_4)\}$ indicates that α_1 precedes α_2 and α_1 and α_2 *both* precede either α_3 or α_4. However, neither α_3 nor α_4 are ordered with respect to each other.

Just as Bunge allows the *composition* of events,[21] OBIS allows a change function α_0 to be *defined* as the composition of two or more other change functions α_i. Eg.

$$\alpha_0 = \alpha_1 * \alpha_2 * \ldots * \alpha_n.$$

History

In Bunge's ontology, the *history* of a thing is the succession of its states over time.[22]

The history of an object is its *actual* state trajectory or life cycle during its existence in the OBIS. The state function fHistory, defined for all objects in the OBIS, returns pairs of values $< t, \mathrm{F}(t) >$ where t is a point in time and $\mathrm{F}(t)$ the vector of values for the state functions of the object.

[21] Definitions 5.4, 5.5 & 5.6.

[22] Bunge defines the term more generally, ie. as a succession of states over a spacetime frame of reference. For the type of applications relevant to OBIS, only the time co-ordinate seems appropriate.

Change Invocation

Invocation of a change function results in a series of activities. The behaviour of objects in the OBIS depends on how their change functions are specified. The typical sequence for changing the composition of an object (call it the *target* object) illustrates many of the concepts related to change discussed above.

1. For the composition of the target object to be modifiable by the user, a *change function* must be defined in the object's model. Give this function the name cAddComp.

2. The target's partial ordering function $\mathcal{P}$ provides the user access to cAddComp at certain points in the target's existence. Suppose that at one of these points, the user invokes cAddComp to add an object x_a of type X_a to the composition of the target.

3. The change function is defined as a procedure which first permits the user to scan the composition of the OBIS (or other relevant object) for x_a. If the object exists, cAddComp adds it to the composition of the target. Note in this case that the target's underlying surrogate is now changed as well.[23]

4. If the object to be added does not exist, cAddComp attempts to create it by assigning the model X_a to a surrogate. cAddComp assists the user to locate an appropriate surrogate. It may be possible that an appropriate surrogate exists as an alternate view, let's say object y_a. If so, cAddComp assigns X_a to the surrogate for y_a, generating a new object x_a which is added to the target.[24]

[23]Recall that the surrogate of a composition object is also composite.

[24]If y_a *must* exist, the change function should return an error message is y_a is not found. This results in a form of *referential integrity constraint.*

5. If an appropriate surrogate does not exist, even as another object, then the *assumed*[25] surrogate is assigned to model X_a. The resulting object is added to the target. This sequence assumes that the object to be added, x_a, is simple. If x_a is complex, then a change function of x_a may be invoked to "build" x_a by adding to its composition. This procedure can be followed recursively to build new objects of any complexity.

6. The law statements **L** of the target as well as all affected objects in the target's composition and part-of context are validated. If any violations are found, the change is disallowed and the user informed.

7. The values of fHistory of the target as well as the histories of any other affected objects now reflect the change.

It should be noted that a change function such as cAddComp can be implemented as a procedure which invites considerable decision-making assistance from the user. However, it is also conceivable that change functions can be defined to minimize human intervention.

The above example showed how the composition of an object is changed in terms of its underlying surrogates. However, the user will normally perform this operation in terms which are more intuitive and natural to the situation. For example, suppose that objects of type ORDER are defined as the aggregate of one CUSTOMER object and one or more PRODUCT objects. Assume that the requirement is to add a new CUSTOMER object to an object *Order*. Assume further that the real world customer of interest is the individual JOHN.

1. *Order* supports a change function cAddComp for changing its composition. The user invokes this function.

[25]Recall the assumption that a surrogate always exists in the OBIS for all things of interest in the application.

2. cAddComp looks for a CUSTOMER view[26] of JOHN. cAddComp may, for example, initiate a scan of all objects of type CUSTOMER in the composition of the OBIS. Once the correct object is found, cAddComp adds it to the composition of *Order*.[27]

3. If a CUSTOMER view of JOHN does not exist, cAddComp checks for any other object with the surrogate for JOHN. For example, a PERSON view may exist.[28] A new CUSTOMER object, *john*, is created using the surrogate of the PERSON object. The new object is then added to the composition of system. There are now two (at least) different objects describing the real world individual JOHN in the OBIS.

4. If there are no views (or objects) for JOHN, a new CUSTOMER object is created using the assumed surrogate for JOHN and is added to *Order*.

5. The validation function for *Order* is invoked. The user is notified if the change violates any law statements.

5.3 Extended Example

This section presents a simple example illustrating the main principles of OBIS development. The example chosen involves a typical data processing application, Order Entry. The OBIS for this application is a single object of kind ORDER ENTRY composed of objects of kinds CUSTOMER, ORDER, and PRODUCT.

[26]Ie. an object of type CUSTOMER with surrogate corresponding to JOHN.

[27]In a small system, it is reasonable to assume that a human user can scan all CUSTOMER objects and identify the particular object of interest. With large OBIS, however, the cognitive load may be such that a human operator cannot be expected handle this task unaided. In such cases, the change function can provide more assistance or an automated process can be substituted.

[28]Again, in a small system, the user can be expected to know whether such a view already exists (or at least search for such a view.)

5.3.1 The OBIS Design Process

The overall implementation process for developing an OBCM/IS application can be outlined as follows.

1. The analyst/designer (or end-user) examines the application and identifies the things of interest in the application according to ontological principles.

2. These things are grouped into *kinds* by identifying common sets of properties and laws.

3. Within each kind, for each property of interest, one or more state functions are defined which map individual things to appropriate attribute values.

4. For each law of interest, one or more law statements are defined which constrain state function values or express relationships among state functions.

5. A set of change functions is defined which specifies possible transformations of object states. This set is partially-ordered and defines the potential *life-cycle* of the objects in a kind.

6. The state functions, law statements and change functions together form a *model object* for the kind. More than one model may be formed for any kind.

7. Model objects are implemented in a computer-based form. For each thing of interest in the application, surrogates are generated and objects created for each surrogate. The collection of objects constitutes the OBIS.

8. In practice, this process can be supported by special computer-based tools and "shells".

5.3.2 Order Example - Overview

An examination of this application reveals at least three major kinds of things related to Order Entry. *Customers* are individuals, other firms or perhaps government agencies. *Products* are items for sale by the firm. *Orders* are interpreted as collections of products to be sold to some customer. That is, they are aggregations of n products and 1 customer. Therefore, the *composition* of an order is a set consisting of products and a customer. And those products and the customer are *part-of* the order.

The next three sections develop the models for these three kinds.

5.3.3 Customers

The customers of the firm are represented by the model object CUSTOMER. Assume a collection of surrogates, S_C, each of which corresponds to a real customer of the firm.

State Functions

As with all model objects, CUSTOMER includes the following standard state functions:

$$\mathsf{fComp} : S_C \rightarrow 2^{\Theta},$$
$$\mathsf{fAKinds} : S_C \rightarrow 2^{\Theta_m},$$
$$\mathsf{fAComp} : S_C \rightarrow 2^{\Theta},$$
$$\mathsf{fPartOf} : S_C \rightarrow 2^{\Theta},$$
$$\mathsf{fForm} : S_C \rightarrow \{\mathbf{G}\},$$
$$\mathsf{fHistory} : S_C \rightarrow \{< t, \mathbf{F}(t) >\}.$$

The specific properties of customers include their indebtedness to the firm and their ability to pay, or credit worthiness. These are represented by the following functions in

F:

$$\mathsf{fDebt} : S_C \rightarrow \mathbf{N},$$

$$\mathsf{fCreditLimit} : S_C \rightarrow \mathbf{N},$$

where **N** is the set of numbers representing money.

Another useful property is the customer's identity. While this can be represented by any number of attributes, a conventional one is the use of a *name*:

$$\mathsf{fName} : S_C \rightarrow \mathbf{N}_s,$$

where $\mathbf{N}_s$ is the set of names.

Law Statements

For the purposes of this application, objects of type CUSTOMER are *simple*. The law statement lComp restricts the value of fComp as follows:

$$\mathsf{lComp}:$$

$$\mathsf{fComp} = \{\mathsf{self}\}.$$

That is, the composition of CUSTOMER objects will only be composed of one object of kind CUSTOMER (ie. CUSTOMER objects are *simple*).

The law lDebt relates fDebt to the value of orders to which the customer is in a *part-of* relationship. That is, fDebt is the sum of all orders of which the customer is part-of. Assuming the model object ORDER (to be defined later) and the state variable ORDER.fValue,

$$\mathsf{lDebt}:$$

$$\mathsf{fDebt} = \Sigma[(\mathsf{fPartOf}/\sim_{\mathrm{ORDER}}).\mathsf{fValue}].$$

That is, the total debt of a customer is the sum of the values of the orders to which the customer is in a *part-of* relationship.[29]

[29]More precisely, the value of fDebt of an *object* of type CUSTOMER is the sum of the values of fValue of all *objects* of type ORDER in fPartOf of the object of type CUSTOMER. Generally, the more intuitive wording will be used in this and other examples.

The law lCreditLimit restricts the value of fDebt according to

$$\text{lCreditLimit}:$$

$$\text{fDebt} \leq \text{fCreditLimit}.$$

That is, the total debt of a customer must not exceed its credit worthiness (or credit limit).

Change Functions

An analysis of possible changes for a customer indicates the following life cycle:

1. Customers are first recognized by the firm as individuals or other firms who express an interest in dealing with the firm.

2. Customers are provided with an initial credit limit and various identifying characteristics.

3. The credit limit can change over the duration of the customer's relation with the firm

4. Certain identifying characteristics may change during the customer's relation with the firm.

This suggests the following change functions.

A change function to change the value of fCreditLimit is defined as:

$$\text{cCreditLimit}:$$

$$\text{fCreditLimit} := \text{external source},$$

where the value of fCreditLimit is set from an external source. This change can occur at any time during the customer's relation with the firm.

A function to change the value of fName can be expressed as

cName :

fName := external source.

This change can occur at any time during the customer's relation with the firm.

A function to change the value of fForm to some implementation-specific value appropriate to the implementation technology is defined as

cForm :

fForm := external source.

This change can occur at any time during the customer's relation with the firm (whenever, for example, the implementation technology is changed).

A change function is required to initialize the value of fKinds as follows

cKinds :

fKinds := {CUSTOMER}.

A function is required to provide the initial values for the customer when it is first recognized by the firm and entered into the system. This function is defined as the composition of changes defined earlier as follows

cInit :

cKinds * cCreditLimit * cName * cForm.

This change can only occur once and precedes any of the other change functions.[30]

The partial ordering of these change functions can be depicted as

cInit $\prec$ (cCreditLimit, cForm, cName).

[30]Note that cInit $\prec$ {cCreditLimit, cName,cForm}, even though cInit is composed of the latter.

That is, cInit precedes cCreditLimit, cForm and cName but the latter three are not ordered.[31]

To summarize, the model CUSTOMER is a representational schema for surrogates S_C consisting of

$\mathbf{F} = \{\mathsf{fDebt}, \mathsf{fCreditLimit}, \mathsf{fName}\}$[32]

$\mathbf{L} = \{\mathsf{lCreditLimit}, \mathsf{lDebt}\}$

$\mathbf{C} = \{\mathsf{cInit} \prec (\mathsf{cCreditLimit}, \mathsf{cForm}, \mathsf{cName})\}$.

5.3.4 Products

The products for sale in the firm are modelled by the model object PRODUCT. Assume a collection of surrogates, S_P, each of which corresponds to a real product for sale by the firm.

State Functions

The following standard state functions apply.

$$\mathsf{fComp} : S_P \rightarrow 2^{\Theta},$$
$$\mathsf{fAKinds} : S_P \rightarrow 2^{\Theta_m},$$
$$\mathsf{fAComp} : S_P \rightarrow 2^{\Theta},$$
$$\mathsf{fPartOf} : S_P \rightarrow 2^{\Theta},$$
$$\mathsf{fForm} : S_P \rightarrow \{\mathbf{G}\},$$
$$\mathsf{fHistory} : S_P \rightarrow \{< t, \mathbf{F}(t) >\}.$$

[31]There is considerable flexibility in how change function definitions and ordering can be expressed. For example, instead of defining cInit, the ordering of change functions above can be expressed as

$$(\mathsf{cKinds} \prec \mathsf{cCreditLimit} \prec \mathsf{cName} \prec \mathsf{cForm}) \prec (\mathsf{cCreditLimit}, \mathsf{cForm}, \mathsf{cName}).$$

[32]In addition to the standard state functions for all objects.

Among the specific properties of products are their value, which is represented by the state function

$$\mathsf{fPrice} : P \rightarrow \mathbf{N}.$$

The identity of a product can be represented by the conventional state function fName

$$\mathsf{fName} : P \rightarrow \mathbf{N}_s.$$

Law Statements

The law lComp defines objects of kind PRODUCT as simple.

$$\mathsf{lComp}:$$
$$\mathsf{fComp} = \{\mathsf{self}\}.$$

Another law restricts a specific product to be *part-of* only one order

$$\mathsf{lOrder}:$$
$$[\mathsf{qSize}(\mathsf{fPartOf}/\sim_{\text{ORDER}}) \leq 1].$$

Change Functions

An analysis of a possible life cycle for a product indicates the following:

1. Products are recognized when they are first acquired by the firm.
2. Products are initialized with a name and price. It is assumed that neither the name nor the price will change during the life of the product with the firm.

The result of this analysis suggests the following change functions.

The standard change function to to modify the value of fForm is defined as

$$\mathsf{cForm}:$$
$$\mathsf{fForm} := \text{external source}.$$

This change can occur at any time during the product's life with the firm.

A change function to change the value of fPrice is defined as

cPrice :

fPrice := external source,

where the value of fPrice is set from an external source. This change occurs only once.

A function to change the value of fName is expressed as

cName :

fName := external source.

Again, this change occurs only once.

A change function is required to initialize the value of fKinds.

cKinds :

fKinds := {PRODUCT}.

A function is defined to provide the initial values for the product when it is first acquired by the firm and entered into the system. This is a composite function.

cInit :

cKinds * cPrice * cName * cForm.

This change can only occur once and precedes all of the other change functions.

The partial ordering for these change functions can be expressed as

$$\text{cInit} \prec \text{cName} \prec \text{cPrice} \prec (\text{cForm}).$$

In summary, the surrogates for the products of the firm is the set S_P. These products are described by the model object PRODUCT which consists of

$\mathbf{F}$ = {fPrice, fName}

$\mathbf{L}$ = {IComp,IOrder}

$\mathbf{C}$ = {cInit $\prec$ cName $\prec$ cPrice $\prec$ (cForm)}.

5.3.5 Orders

An *order* is viewed as a composition of a customer and one or more products. It is represented by the model object ORDER.

The collection of possible surrogates for the orders of the firm is therefore defined as

$$S_O = S_C \times 2^{S_P}.$$

That is, S_O is the cartesian product of one customer surrogate and one or more product surrogates.

State Functions

The standard state functions for ORDER are

$$\mathsf{fComp} : S_O \rightarrow 2^{\Theta},$$
$$\mathsf{fAKinds} : S_O \rightarrow 2^{\Theta_m},$$
$$\mathsf{fAComp} : S_O \rightarrow 2^{\Theta},$$
$$\mathsf{fPartOf} : S_O \rightarrow 2^{\Theta},$$
$$\mathsf{fForm} : S_O \rightarrow \{\mathbf{G}\},$$
$$\mathsf{fHistory} : S_O \rightarrow \{< t, \mathbf{F}(t) >\}.$$

Other relevant properties of orders include value and size (number of products ordered) represented by the functions

fValue: $S_O \rightarrow \mathbf{N}$,

fSize: $S_O \rightarrow \mathbf{I}$

where $\mathbf{I}$ is the set of integers.

The identity property of an order can be represented by a conventional order number as follows

fNumber: $S_O \rightarrow \mathbf{I}$.

Law Statements

Assume the following relationships among the properties of orders:

1. an order must have one and only one customer in its composition;
2. the size of the order is the number of products in its composition (order size can be zero); and
3. the value of an order is the sum of the prices of the products in its composition.

These relationships can be captured by the following law statements.

The law statement lComp restricts the value of fComp as follows:

$$\begin{aligned} &\mathsf{lComp}: \\ &\quad [\mathsf{qSize}(\mathsf{fAComp}/\sim_{\text{CUSTOMER}}) = 1 \ \ \& \\ &\quad \mathsf{qSize}(\mathsf{fAComp}/\sim_{\text{PRODUCT}}) \geq 0]. \end{aligned}$$

This is the cardinality law which specifies that every order must be composed of one customer and zero or more products.

The law statement lSize relates fSize to the number of objects of kind PRODUCT in the composition of an ORDER object.

$$\begin{aligned} &\mathsf{lSize}: \\ &\quad \mathsf{fSize} = \mathsf{qSize}[\mathsf{fAComp}/\sim_{\text{PRODUCT}}]. \end{aligned}$$

The law statement lValue relates fValue to the sum of the prices of all objects of kind PRODUCT in the composition of an order.

$$\begin{aligned} &\mathsf{lValue}: \\ &\quad [\mathsf{fValue} = \Sigma(\mathsf{fAComp}/\sim_{\text{PRODUCT}} .\mathsf{fPrice})]. \end{aligned}$$

Change Functions

The life cycle for an order can be analysed as follows.

1. An order is first created by bringing together a customer and zero or more products.
2. The composition of an order can be changed by adding products but not by adding customers. Assume that products can be added at any time during the life of the order.

A change function is required to add a CUSTOMER object to the composition of an ORDER object.

$$\mathsf{cAddCustomer}:$$
$$\mathsf{fAComp} := \mathsf{fA\widehat{Comp}} \circ c,$$

where c is an object of kind CUSTOMER. (This notation indicates that the value of fAComp is changed by adding c to its previous value $\mathsf{fA\widehat{Comp}}$.) This event can occur only once.

A change function is required to add a PRODUCT object to the composition of an ORDER object.

$$\mathsf{cAddProduct}:$$
$$\mathsf{fAComp} := \mathsf{fA\widehat{Comp}} \circ p,$$

where p is an object of kind PRODUCT. This event can occur more than once.

The standard change function cForm modifies the value of fForm.

$$\mathsf{cForm}:$$
$$\mathsf{fForm} := \textit{external source}.$$

This change can occur at any time during the order's life with the firm.

A change function changes the value of fNumber.

$$\mathsf{cName}:$$
$$\mathsf{fNumber} := \textit{external source}.$$

This change will occur only once during the life of the order. The external source for this function will typically be a serial numbering function which assigns each new order with a unique sequential order number.[33]

The change function cKinds is defined as

cKinds :

fKinds := {CUSTOMER,PRODUCT}.

A function provides the initial values for the order when it is first created.

cInit :

cKinds * cNumber * cAddCustomer * cForm.

This change can only occur once and precedes any of the other change functions.

The partial ordering of these change functions can be expressed as

cInit $\prec$ (cAddProduct, cForm).

In summary, ORDER is a model object for describing surrogates $S_O = S_C \times 2^{S_P}$. ORDER consists of the following

F = {fNumber, fValue, fSize}

L = {lComp, lValue, lSize}

C = {cInit $\prec$ (cAddProduct, cForm)}.

5.3.6 Order Entry System

The collection of all customers, items and orders in an Order Entry application can be viewed as a single aggregation, represented in the OBIS by an object of kind ORDER-ENTRY. That is, in the ontological view, the Order Entry *application* itself is seen as a single, composite thing.

[33]A law statement can be used to confirm unique serial numbering.

The surrogate for the Order Entry application can be defined as

$$S_{OE} = 2^{S_C} \times 2^{S_P} \times 2^{S_O}.$$

That is, the surrogate corresponding to the application is the aggregate of surrogates corresponding to customers, products and orders in the application.

State Functions

The standard state functions for ORDERENTRY are

$$\mathsf{fComp} : S_{OE} \rightarrow 2^{\Theta},$$
$$\mathsf{fAKinds} : S_{OE} \rightarrow 2^{\Theta_m},$$
$$\mathsf{fAComp} : S_{OE} \rightarrow 2^{\Theta},$$
$$\mathsf{fPartOf} : S_{OE} \rightarrow 2^{\Theta},$$
$$\mathsf{fForm} : S_{OE} \rightarrow \{\mathbf{G}\},$$
$$\mathsf{fHistory} : S_{OE} \rightarrow \{< t, \mathbf{F}(t) >\}.$$

A number of additional properties of the Order Entry application are of interest.

Population or *numerosity* is an important emergent property of any composite thing. The number of customers in the application is represented by the state function

$$\mathsf{fCustomerCount} : S_{OE} \rightarrow \mathbf{I},$$

where **I** is the set of integers.

The number of orders in the application is represented by

$$\mathsf{fOrderCount} : S_{OE} \rightarrow \mathbf{I}.$$

The number of products available for sale is represented by

$$\mathsf{fProductCount} : S_{OE} \rightarrow \mathbf{I}.$$

The value of this state variable is equivalent to the conventional notion of *available inventory.*

Other emergent properties of the application include the *value* of its various components. In this example, the total value of the orders in the application is represented by

$$\mathsf{fValue} : S_{OE} \rightarrow \mathrm{N}.$$

Laws Statements

The law lCustomerCount relates fCustomerCount to the number of objects of kind CUSTOMER in the composition of the order entry application.[34]

$$\begin{aligned}&\mathsf{lCustomerCount}:\\ &\quad \mathsf{fCustomerCount} = \mathsf{qSize}(\mathsf{fComp}/\sim_{\text{CUSTOMER}}).\end{aligned}$$

Similarly, the law lOrderCount relates fOrderCount to the number of objects of kind ORDER in the composition of the application.

$$\begin{aligned}&\mathsf{lOrderCount}:\\ &\quad \mathsf{fOrderCount} = \mathsf{qSize}(\mathsf{fComp}/\sim_{\text{ORDER}}).\end{aligned}$$

The law lProductCount relates fProductCount to the number of objects of kind PRODUCT in the application which are still available for sale, that is, which are *not* part-of any orders.

$$\begin{aligned}&\mathsf{lProductCount}:\\ &\quad \mathsf{fProductCount} = \mathsf{qSize}(\mathsf{fComp}/\sim_{\text{PRODUCT}})\ \&\\ &\quad \text{NOT}\ [(\mathsf{fComp}/\sim_{\text{ORDER}}.\mathsf{fComp})/\sim_{\text{PRODUCT}})].\end{aligned}$$

That is, all objects of type PRODUCT in the composition of the application which are not in the composition of objects of type ORDER.

[34]That is, an object of kind ORDERENTRY.

The value of fValue is given by the law

$$\begin{aligned} &\mathsf{lValue}: \\ &\mathsf{fValue} = \Sigma[(\mathsf{fComp}/\sim_{\mathrm{ORDER}}).\mathsf{fValue}]. \end{aligned}$$

That is, fValue is the sum of the values of fValue for all objects of kind ORDER in the application.

Change Functions

The life cycle of an order entry system can be summarized as

- The system is created (possibly with a null composition).
- Customers are added.
- Products are added.
- Customers and products are combined to create orders.

Change functions are defined to reflect this life cycle as indicated below.

The change function cAddCustomer adds objects of kind CUSTOMER to the composition of the application.

$$\begin{aligned} &\mathsf{cAddCustomer}: \\ &\mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ c, \end{aligned}$$

where c is an object of kind CUSTOMER. This event can occur many times.

The change function cAddProduct adds objects of kind PRODUCT to the composition of an object of type ORDERENTRY.

$$\begin{aligned} &\mathsf{cAddProduct}: \\ &\mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ p, \end{aligned}$$

where p is an object of kind PRODUCT. This change can occur many times.

The change function cAddOrder adds objects of kind ORDER to the composition of the application.

$$\mathsf{cAddOrder}:$$
$$\mathsf{fAComp} := \mathsf{fA}\widehat{\mathsf{Comp}} \circ o,$$

where o is an object of kind ORDER. This event can occur many times. Note that object o itself is a composition of objects of type CUSTOMER and PRODUCT. Therefore, this function may invoke change functions associated with o if o needs to be created.

The following change functions change the state variables fKinds and fForm.

$$\mathsf{cKinds}:$$
$$\mathsf{fKinds} := \{\text{PRODUCT,CUSTOMER,ORDER}\}$$

and

$$\mathsf{cForm}:$$
$$\mathsf{fForm} := \textit{external source}.$$

A composite change function cInit provides the initial values for the Order Entry object when it is first created.

$$\mathsf{cInit}:$$
$$\mathsf{cKinds} * \mathsf{cForm}.$$

This change can only occur once and precedes any of the other change functions.

The partial ordering for ORDERENTRY can be expressed as

$$\mathsf{cInit} \prec (\mathsf{cAddCustomer}, \mathsf{cAddProduct}, \mathsf{cAddOrder}, \mathsf{cForm}).$$

In summary, ORDERENTRY is a representation for the aggregation of customers, products and orders related to the application. The surrogate for this aggregation can be defined as $S_{OE} = 2^{S_C} \times 2^{S_P} \times 2^{S_O}$. The model consists of the following.

$\mathbf{F} = \{\mathsf{fValue}, \mathsf{fCustomerCount}, \mathsf{fOrderCount}, \mathsf{fProductCount}\}$

$\mathbf{L} = \{\mathsf{lComp}, \mathsf{lValue}, \mathsf{lCustomerCount}, \mathsf{lOrderCount}, \mathsf{lProductCount}\}$

$\mathbf{C} = \{\mathsf{cInit} \prec (\mathsf{cAddCustomer}, \mathsf{cAddProduct}, \mathsf{aAddOrder}, \mathsf{cForm})\}.$

5.3.7 Order Example - Operation

The OBIS for this example consists of objects representing all the customers, products and orders of interest to the application. The collection of all the objects in the OBIS is itself an object, in this case, of type ORDERENTRY. Call this object *OrderSystem*.

Populating the OBIS

The operation of the OBIS is described in OBCM/IS terms (and not in computer or implementation related terms). The user begins with the *OrderSystem* and populates it with CUSTOMER and PRODUCT objects by invoking the change functions cAddProduct and cAddCustomer. Each time a CUSTOMER or PRODUCT object is created, their respective cInit change functions are invoked, setting their initial values.[35]

Creating New Orders

In response to the real world situation in which an order is received, the user represents this activity by creating a new ORDER object and changing its composition with CUSTOMER and PRODUCT objects to reflect the real world situation. The procedure is as follows.

1. An object of type ORDER is created. At the moment of creation, this object will have a *null* composition (and is therefore an unlawful object).[36]

2. The composition of *OrderSystem* is scanned for the relevant customer which is then selected and added to the composition of the ORDER object.

[35]This is analogous to updating Customer and Inventory *master files* in a conventional IS.

[36]An unlawful object cannot be added to the OBIS but is permitted to exist temporarily outside the OBIS.

3. The composition of *OrderSystem* is then scanned for relevant PRODUCT objects which are also selected and added to the composition of the order.

4. The ORDER object (which is now lawful) is now added to the composition of *OrderSystem*.

5. The validation functions of the affected objects, in this case the customers and products in the composition of the order as well as the order, are invoked. Any violations are reported.[37]

Figure 5.3 shows selected aspects of *OrderSystem* using the informal visual notation of Chapter 4. In this example, there are two orders, *order#1* and *order#2* both with the same customer c_1. It is seen that c_1 is *part-of order#1*, *order#2*, and *OrderSystem*. The customer c_2, however, is *part-of OrderSystem* only.

OBIS Queries

The equivalent of conventional database queries are accomplished by scanning the *composition* of *OrderSystem* for the desired object then viewing the value of the state function of interest. Since all the objects in *OrderSystem* are connected in a lattice through their *composition* and *part-of* relationships, an object of interest generally can be accessed from any other object with a known relationship to the target object.

For example, suppose the user has current access to *OrderSystem*. If the user wishes to query a particular customer, the composition of *OrderSystem* is *scoped* for all objects of type CUSTOMER. This set is then scanned to locate the customer of interest. The customer's credit limit and outstanding debt can now be determined by viewing the values for state functions fCreditLimit and fDebt.

[37]The mechanisms for invoking validation functions are implementation-specific. It is also feasible (and reasonable) to validate the new ORDER object *before* it is added to *OrderSystem*.

The user can then scope the *part-of context* of the customer for all objects of type ORDER. This reveals all the orders for the customer. A particular order can then be selected and its value viewed. If desired, the composition of the order can then be scanned to determine the different products ordered and their individual values.[38]

For any object, its *history* (ie. fHistory) can be viewed to determine the *trajectory* generated by changes as they occurred to the object.

5.4 Summary

The OBIS framework presented in this chapter provides an integrated approach to implementation of an IS from the OBCM conceptual model proposed in Chapter 4. The result is a homogeneous path from conceptual model to implementation of an Information System. The goal is to reduce the opportunities for semantic gaps which develop as the concepts from the application domain are implemented into a working IS.

The proposed framework also provides a single, ontologically consistent concept of user-interface and system operation in terms drawn from a theory of reality rather than in terms which are computer or technology-dependent. In principle, this one model of user-interface and system operation can be applied to any object in the OBIS including the OBIS itself, thus simplifying the understanding and use of the IS.

The operation of the OBIS is defined around ontologically-oriented concepts of *view* and *change*. These concepts are developed into ontology-based interactions with the user rather than the technical operations typically associated with conventional information systems such as file updates, relational joins, and so forth.

In the next chapter, a prototype implementation in a microcomputer environment using Smalltalk is described. This prototype is intended to be a "proof of concept"

[38] An *ad hoc* query can be accommodated by defining a state function specifically for the query and adding it to the appropriate model.

implementation of OBCM/OBIS, that is, an experiment in software to determine the feasibility of the theoretical model [Newell & Simon, 1975].

Chapter 6

An OBIS Prototype Implementation

6.1 Role of the Prototype

This chapter describes a Smalltalk prototype of an Ontology-Based Information System as implemented on an IBM AT-class microcomputer. The prototype is implemented as a "shell" which provides basic support for OBCM/IS objects. The OBIS is then generated by populating the shell with objects specific to the application.

In this thesis, the prototype plays an important role for the following reasons.

- *Proof of concept.* The prototype implementation serves to demonstrate the feasibility of the theoretical model presented in Chapters 4 and 5. As such, it forms the most basic form of validation for the model. Although implementation is not an exhaustive validation, it can reveal inconsistencies or unexpected behaviour for further study.

- *Model refinement.* The prototype can be viewed as a series of experiments [Newell & Simon, 1975] designed to test and refine aspects of the theoretical model. In this respect, implementation is an important research tool in refining and testing the OBCM/IS framework.

- *Comparison to other models.* A working prototype facilitates the comparison of OBCM/IS to other systems development approaches.

- *Tools and techniques.* Through implementation, a number of practical tools and

techniques are demonstrated. Although a complete systems development methodology is beyond the scope of this project, such tools may prove useful in future work in OBCM/IS.

The following sections discuss the general approach to the prototype implementation and provide a brief overview of selected implementation issues. Details of the prototype are included in the Appendices to this thesis. The chapter concludes with a description of a working OBIS based on the extended example from Chapter 5.

6.1.1 Choice of Prototype Language

Smalltalk was selected for the implementation for the following reasons.

- *Object-oriented approach.* An implementation language consistent with the object-oriented approach was considered desirable for maintaining homogeneity and ease of implementation.

- *Availability.* Smalltalk is well-known as a pioneering object-oriented language which is well-documented and readily available [Goldberg & Robson, 1983]. Several versions of Smalltalk have been developed, some of which are implemented on microcomputers [Digitalk, 1986; ParcPlace, 1989]. The version selected for this implementation is Smalltalk/V 286 which executes on IBM AT-class microcomputers [Digitalk, 1986].

- *Software environment.* Smalltalk supports a programming environment which includes object browsers, editors and other object management and manipulation tools which can be adapted for implementing the OBIS. The Smalltalk library of objects provides basic constructs and operators such as collections and sets useful in implementing the OBCM concept. In addition, the Smalltalk environment

supports windowing and other graphical interfaces useful for displaying objects.

- *Embedded Prolog.* The particular version of Smalltalk used, Smalltalk V/286, supports an embedded Prolog which facilitates the expression of laws statements.

6.1.2 The Smalltalk Programming Language

Smalltalk is an integrated programming language and programming environment. As one of the first acknowledged Object-Oriented Programming Languages (OOPL), it has had considerable influence in OOPL design and implementation.

In Smalltalk, an *object*[1] is some private or local data and a collection of procedures, called *methods* that can access that data. The local data of an object cannot be accessed except through the methods of the object. Methods, on the other hand, are public to all other objects in a Smalltalk environment. To access a method of another object, Smalltalk objects send *messages* corresponding to the method which then executes. A message can also include parameters which are used by the method during its execution.

Objects are *instances* of special objects called *classes*. Classes describe the local data structures and the methods of objects. Sending the appropriate message to a class object results in the creation of a new instance as specified by the class object. Classes are organized into a *class hierarchy* in which classes lower in the hierarchy can *inherit* the data structures and methods from those higher in the hierarchy (called *superclasses*).

Smalltalk supports a comprehensive programming environment which fully integrates the programming language, support tools (such as editors, linkers, and debuggers) and the operating system. It exploits a highly visual environment with *browsers* and *windows* which can display objects and their behaviours. The visual environment supports display

[1]The Smalltalk *object* should not be confused with the OBCM/IS object. Unless specifically stated, the term *object* will always refer to the OBCM object as defined in Chapters 4 and 5.

formats such as *GraphPanes* which display graphics, *TextPanes* which display conventional text-based material, and *ListPanes* which supports lists of text-based information.

Further details on Smalltalk can be found in [Goldberg & Robson, 1983; Goldberg, 1984; Robson & Goldberg, 1981].

6.2 The Implementation Framework

The prototype is implemented as a *shell* which provides generic support for OBCM/IS objects. OBIS objects and their models are implemented within the shell by specifying a programming *protocol* or *discipline* in the Smalltalk language.

6.2.1 A Smalltalk Programming Protocol for OBIS

The programming protocol for the prototype implementation can be summarized as follows.

- A Smalltalk class is defined to represent surrogates. Surrogates are implemented as instances of this class.
- Smalltalk classes are defined for each model object. These are called *Model Object Classes* or MOC's. An object is implemented as an instance of an MOC paired with an instance of class Surrogate. There is no hierarchy defined among the MOC's.[2]
- State functions and change functions are implemented as methods of MOC's.[3] These methods are strictly partitioned into one of the following categories:

[2]Neither Bunge's ontology nor OBCM/IS supports a concept of inherent hierarchy or inheritance among the things in the world.

[3]In the prototype, a slightly modified version of the standard Smalltalk *Class Hierarchy Browser* along with its associated editor is used to created MOC's and implement state and change function methods. A more refined implementation would use specialized tools and editors to generate MOC's.

1. *State function methods.* These methods return values which are stored in instance variables or computed within the methods.
2. *Change methods.* These methods change the values returned by state function methods.

- Law statements for a model object are implemented as Prolog statements.
- The following state function methods are common to all MOC's.

1. fComp - answers the V-composition of the object.[4]
2. fKinds - answers the set of MOC names allowed in the A-composition of the object.
3. fAComp - answers the A-composition of the object.
4. fPartOf - answers the part-of context of the object.
5. fHistory - answers the history of the object.
6. fForm - answers a set of implementation-dependent parameters which control the appearance on an object on the display device.
7. fSurrogate - answers the set of objects which have the same surrogate as the object.

6.2.2 Prototype Operation - Overview

Users interface with OBIS implementations by viewing and changing objects in the system. These two activities are meaningful only in the context of a specific object with

[4]Recall from Chapter 4 that *V-composition* is recursively defined as the A-composition of other A-compositions.

which the user is interacting at any time. This object is called the *current object* and is said to have the *focus*[5].

The object of focus can be changed as a result of scanning. Scanning a set of objects reveals each of the objects in the set. By choosing one of these objects (eg. with a mouse click), the focus is transferred to the selected object.

Change involves the invocation of *change functions*. In the prototype, the user can display the allowable change functions of the object with the focus. By selecting a specific change function, the user invokes the Smalltalk change method associated with that change function.

In this prototype, the user is assumed to be a human operator working with a display screen, keyboard and pointing device (ie. mouse). On the other hand, the user may also be substituted by an automated process or procedure consisting of a set of rules (or possibly heuristics) which scan and manipulation the OBIS.

6.2.3 Prototype User Interface

Object Window

The user interface provides mechanisms for viewing, scanning and changing objects in the system. The interface employs a visual format which is similar, but not identical, to the visual notation introduced in Chapter 4.

The state function, fForm, provides some basic implementation-specific parameters which controls the appearance of an OBCM object in the Smalltalk environment as well as its interface to the keyboard and mouse. The default value for fForm of an object defines a Smalltalk GraphPane window for displaying graphical and text information

[5]This use of the term *focus* is borrowed from Smalltalk terminology. Technically, it is the *window* of the object (to be described shortly) which has the focus and not the object itself.

related to the object.[6] This window supports a "pop-up" service menu which offers the user a number of services including access to the state function values, law statements, and change functions of the object. This menu also provides mechanisms for manipulating sets of objects returned by state functions such as fComp or fPartOf. This window and the service menu implements the basic elements of the user interface described in Chapter 5. Figure 6.1 shows a sample window (A) opening on an object, in this case, of type ORDERENTRY.[7]

Typically, the user will be interacting with an object entirely through the services offered by this window. In addition to the basic elements of user interface, additional services actually offered to the user through the object window are implementation-specific. In the current prototype, the following are included in the choices offered by the service menu.

Viewing States, Law Statements and Change Functions

1. States - allows the scanning and selection of the state functions of the current object. Selecting this option displays all the state functions of the object in a separate "pop-up" window. The user can then select a specific state function to display its value.

2. Changes - allows the scanning and selection of the change functions of the current object. Selecting this option displays all the change functions defined for the object in a "pop-up" window. The user can invoke any display state function by selecting it. Since the set of change functions is partially ordered, certain functions may not be allowed at all times and are either not displayed or marked as unavailable.

[6]The Smalltalk environment and standard windows are described in detail in [Digitalk, 1988].

[7]Actual screen displays produced by the prototype will differ somewhat from these figures which do not completely capture the highly interactive dynamics of the Smalltalk windowing environment.

3. Laws - allows the scanning and viewing of the law statements of the current object. In the present implementation, a special Smalltalk "browser" is opened in which the Prolog statements corresponding to the laws are displayed.

In Figure 6.1, label (B) points to a "popup" menu providing these three functions.

Scanning and Selection

1. fComp - allows scanning of the composition of the current object and selection of any specific object in the composition. The focus changes to the selected object.

2. fAComp - allows scanning of the A-composition of the current object and selection of any specific object in the A-composition. The focus changes to the selected object.

3. fPartOf - allows scanning of the part-of context of the current object and selection of any specific object in the part-of context. The focus changes to the selected object.

4. fSurrogates - allows scanning of all objects with the same surrogate as the current object and selection of any specific object therein. The focus changes to the selected object.

The label (C) in Figure 6.1 points to a menu providing these services.

Manipulating Sets of Objects

The implementation provides services to manipulate *sets* of objects such as the A-composition or the part-of context of the current object. Some of these services include the following.

1. ofKind:k - selects the subset of objects of specified type k and allows scanning and selection over this subset. The focus changes to the selected object.

2. compOf - returns the union of the composition of each object in the original set. Allows scanning and selection over this new set. The focus changes to the selected object.

3. partOf - returns the union of the part-of context of each object in the original set. Allows scanning and selection over this new set. The focus changes to the selected object.

4. stateValueOf:f withValue:x - selects the subset of objects in which state function f has value x. Permits scanning and selection over this subset. All the objects in the original set must be of the same type.

Label (D) in Figure 6.1 shows a *ListPane* displaying a list of objects. A "popup" menu on the ListPane (E) provides the above services to the objects listed in (D).

Every collection of objects always includes one object of type NIL. This object is *part-of* all other objects in the OBIS. Therefore, invocation of partOf over the set {NIL} returns all the objects in the OBIS.

6.2.4 OBIS Prototype Shell

All these basic services as well as the protocol for defining MOC's is provided by the prototype *shell system.* This shell provides the implementation-level support to manage objects and surrogates, handle the windowing and menuing systems, and provide the scanning, viewing and selection facilities described above. Generating a working OBIS is a matter of using the shell to generate and maintain MOC's, then populating the shell with the specific objects related to the application.

Shell Structure

The basic structure of the shell consists of five components or subsystems.

1. *OBCM/IS Primitives.* Provides the basic support to manage objects and surrogates at the program code level.

2. *Windowing and Menuing Subsystem.* Supports the display of objects on the physical hardware. Much of this subsystem is by necessity implementation-specific to the Smalltalk environment.

3. *User Interface.* Provides the user with access to the system on the object level. That is, allows for scanning and selecting of OBCM objects.

4. *MOC Support Subsystem.* Supports the definition and maintenance of Model Objects for the system.

5. *Law Statement Subsystem.* Supports the definition and maintenance of law statements.

Figure 6.2 shows a standard Smalltalk *Class Hierarchy Broswer* which displays Smalltalk classes and their relationship to each other. As indicated, the shell components (with the exception of the Law Statement Subsystem) are implemented as Smalltalk classes arranged in a class hierarchy beginning with the primitives. In this particular implementation, since the MOC's are at the bottom of the class hierarchy, any instance of a MOC[8] inherits the functionality of the User Interface, Windowing Subsystem, and basic primitives.

Although the components of the shell are laid out as a hierarchy, it is important to note that this hierarchy is simply a technique used in the implementation of the

[8]That is, any *object* (or more accurately, any implementation of an object). Recall that an OBCM/IS object is implemented as an instance of a MOC coupled with a surrogate.

prototype itself. The Model Object Classes themselves do not form a hierarchy. Indeed, the OBCM/IS model as presented in this thesis does not support a concept of hierarchy nor is such a concept supported in the prototype.

Figure 6.2 also shows the Law Statement Subsystem in its own browser. In the prototype, the Law Statement Subsystem is implemented separately from the other four components although it is still considered part of the prototype shell system. This is strictly an implementation convenience related to the Smalltalk V/286 embedded Prolog interpreter. There is no inherent reason why shells implemented beyond the prototype stage (or, for that matter, in a different programming environment) cannot combine all five components into a single unit.

The shell components are described in more detail in the following paragraphs.[9]

OBCM/IS Primitives

This component, implemented as class **OBIS**, provides support for the basic ontological concepts in the OBCM/IS model. This support includes

1. Internal systems management for objects and surrogates on the implementation level.
2. Internal management of change function invocation.
3. Automatic update of the history as changes occur.
4. Support for validation functions.
5. Partial ordering of change functions. (Some further details of the mechanisms for validation functions and partial ordering will be described when discussing the MOC subsystem.)

[9]Appendix B contains the relevant Smalltalk code for a current version of the prototype shell.

In addition, a major function of this component provides methods for implementing the basic OBCM notation as described in Chapter 4 in Smalltalk. This includes returning surrogates from objects and vice-versa (S_g and O_b), object set partitioning ($\sim_n$), and scalar functions ($\mathcal{Q}$ and $\mathcal{V}$). For example, the sum of all values of the state function fValue of all objects of type Q in the *part-of context* of an object is expressed by the the notation[10]

$$\Sigma[(\text{fPartOf}/\sim_Q).\text{fValue}].$$

The implementation of this notation in Smalltalk takes the form of the following method of class **OBIS**

```
sumValuesOf:#fValue
    fromObjectsOfKind:Q
    in:#fPartOf.
```

Windowing and Menuing Subsystem

This component, implemented as class **OBISForm**, provides the windowing and menuing support required to display and view objects on the physical display. This component is technology-dependent and draws heavily upon the facilities provided by the Smalltalk programming environment. Any changes to the display technology are isolated in this component.

The methods in this component intercepts the values of fForm of an object and translates them into a display image for the object.

Some of the functionality provided by this component includes

1. Basic windowing support to convert the values of fForm into a displayable Smalltalk window.

[10]Refer, for example, to Section 4.6.3 of Chapter 4.

2. Menuing support for "popup menus" and their interface with a mouse device.

3. Additional window support for listing objects, displaying state functions values, law statements, history and other displayable items.

User Interface

This component implemented as class **OBISInterface** supports scanning and selection on an object level.

The functionality of this component includes

1. Display, scanning and selection of the *composition* and *part-of context* of an object.

2. Display, scanning and selection of objects with the same surrogate as the current object.

3. Manipulation of sets of objects such as selection of objects of a specified type, unions of the compositions and part-of contexts of sets of objects, and subsets of objects which meet specified conditions. These manipulations have been detailed in Section 6.2.2.

Law Statement Subsystem

This component defines and maintains the law statements of model objects. In Smalltalk V/286, the embedded Prolog interpreter is implemented in such a way that it is convenient to define laws statements in a separate component rather than in the MOC subsystem along with their associated state variables and change functions.

Law statements are expressed as conventional Prolog assertions. For example, in Chapter 5, the law statement ICreditLimit for the model CUSTOMER was expressed as

```
ICreditLimit :
```

$$\mathsf{fDebt} \leq \mathsf{fCreditLimit}.$$

Its equivalent Prolog assertion is

```
"Debt is <= Credit Limit"
lCreditLimit(fDebt, fCreditLimit) :-
   le(fDebt,fCreditLimit).
```

To take another example, the law statement CUSTOMER.lDebt in Chapter 5[11] was defined as

$$\mathsf{lDebt}:$$
$$\mathsf{fDebt} = \Sigma[(\mathsf{fPartOf}/\sim_{\text{ORDER}}).\mathsf{fValue}].$$

That is, the value of CUSTOMER.fDebt is equal to the sum of the values of the orders of which a customer is *part-of*. This is implemented as the Prolog assertion

```
"Customer debt = sum of order values"
lDebt(debt, aCustomer) :-
  is(orderValues,
     aCustomer sumValuesOf:#fValue
               fromObjectsOfKind:Order
               in:#fPartOf).
  eq(debt,orderValues).
```

(Here, the Prolog predicate *is* binds the variable *orderValues* to the sum of the values of objects of kind ORDER in the part-of context of the customer. Variable *orderValues* is then bound to the variable *debt*.)

Model Object Subsystem

In this subsystem, model objects are defined and maintained as Model Object Classes. A MOC is created for every model object and the state functions and change functions are

[11]Section 5.5.

defined in the form of Smalltalk methods as outlined earlier in Section 6.2.1. In addition, the state functions common to all objects in the OBIS are predefined in this component.

State Functions. State functions are implemented as state function methods of a MOC. These methods return values corresponding to the state of the object. These values are computed within the method or are carried in instance variables of the MOC. For example, the following method returns the value of the state function fCreditLimit.

```
fCreditLimit
"Answer credit limit"
↑ fCreditLimit.
```

In this case, the value is carried in the instance variable *fCreditLimit.*

The value for a state function can also be computed. For example, in Chapter 5, the state variable fDebt represented the total indebtedness of a customer. The value of this state variable can be defined according to the law statement CUSTOMER.lDebt (ie. the sum of the values of all orders of which the customer is part-of). The Prolog version of lDebt was described in the previous section. The value of fDebt can therefore be defined as

```
fDebt
  "Answers fDebt defined as law lDebt"
  |debt |
(CustomerLaws new :? lDebt(x,self)).
↑ x.
```

The expression[12]

CustomerLaws new :? lDebt(x,self)

refers to the Prolog clause **lDebt** and binds the free variable x to the value as defined within the clause. The value of x is then returned by the method **fDebt**. In this way, the values of state functions such as **fDebt** can be defined directly with respect to relevant law statements.

Change Functions. Change functions are implemented as *change methods* of the MOC. In the prototype, they always start with lowercase c. In Chapter 5, the change function to change the credit limit of a customer was defined as

cCreditLimit :

fCreditLimit := external source.

The Smalltalk implementation is

```
cCreditLimit
|oldCreditLimit|
    "Change fCreditLimit"
fCreditLimit isNil ifTrue:[fCreditLimit := 0].
oldCreditLimit := fCreditLimit.
fCreditLimit := (Prompter prompt: 'Credit Limit?'
                    default: (fCreditLimit printPaddedTo:1)) asInteger.
self validateLaws
    ifFalse:[fCreditLimit := oldCreditLimit.
        ↑ nil].
```

[12]Syntactical detail irrelevant to the example has been removed from this expression. The embedded Prolog in Smalltalk V/286 uses a non-standard syntax. The assertion is equivalent to the standard Prolog query

```
?- lDebt(X)
```

in which X is instantiated to a value according to the rules in its database.

↑ fCreditLimit.

This procedure obtains an input value from the external world (using the Smalltalk Prompter) and assigns it the instance variable *fCreditLimit*. The procedure also invokes a validation function. If the change validates *false*, the method restores the original value.

The *validation function* presents law statements coded as Prolog clauses with specific values. The function returns boolean values (*true* or *false*) depending upon whether Prolog succeeded or failed with these values. For example, the validation function in this example presents the Prolog clause **fDebt**[13] described previously with specific values for fDebt and fCreditLimit. The Prolog clause succeeds or fails according to whether these values satisfy its rules. Validation functions can be invoked at any time, either explicitly as above or automatically for an object whenever it is changed.[14]

The next example implements the change function CUSTOMER.cKinds from Chapter 5. Recall that this change was defined as

cKinds :

fKinds := {CUSTOMER}.

The Smalltalk implementation is

```
cKinds
        "Initialize fKinds."
fKinds := Customer asSet.
pOrder add:#cInit.
↑ self.
```

The reference to *pOrder* concerns the partial ordering of the change functions of CUSTOMER objects. In the prototype, a relatively simple form of partial ordering was

[13]The syntactical detail is omitted in the example.

[14]The mechanism for invoking validation functions is implementation-specific. Efficiency issues must be considered in designing such a mechanism. In the worst case, every object in the system can be validated whenever a change is made to *any* object.

implemented whereby each change function explicitly affects the partial ordering of the change functions of an object. Upon successful completion of a change, the change function can remove or add itself to the list of change functions disallowed for the object. In this example, after setting the values of fKinds, change function cKinds adds itself to the set of disallowed changes.

In the next section, highlights of a Smalltalk OBIS implementation for the Order example from Chapter 5 is described. The description here is necessarily incomplete and meant to be illustrative only of the basic approach. Details of the full working example including much of the Smalltalk code can be found in Appendix C.

6.3 Implementation Example

6.3.1 MOC

The basic structure of the sample implementation is as follows.

OBIS
OBISForm
OBISInterface
ModelObj
Customer
Order
Product
OrderEntry

The first four parts constitute the basic shell of the prototype and remains constant for all applications. The last four entries are the Model Object Classes which define the model objects for this application. The state functions, law statements and change functions

of a model object are implemented directly as a MOC. In the next few pages, a sample MOC implementation of the model CUSTOMER is presented. The MOC's for the other models (ie. ORDER, PRODUCT and ORDERENTRY) are detailed in the appendix.

State Functions

Within a MOC, state functions of the model object are implemented as methods of the MOC. For the MOC **Customer**, the following state function methods are defined.[15]

fCreditLimit

Answers the credit limit of the customer. This is a stored value.

```
fCreditLimit
"Answer credit limit"
↑ fCreditLimit
```

fDebt

Answers the total indebtedness of the customer. This is a value computed according to the law statement lDebt.

```
fDebt
  "Answers fDebt defined as law lDebt"
  |debt |
(CustomerLaws new :? lDebt(x,self)) do: [:each |
    debt := each at:1].
↑ debt
```

fName

Answers the name of the customer. This is a stored value.

[15]In some cases, irrelevant syntactic details have been removed to simplify the presentation.

fName

"Answer **customer name"**

↑ fName

fGForm

Answers a Form[16] controlling the size and position of the Customer display window. This is a stored value.

fGForm

↑ fGForm

Change Functions

Change functions are also implemented as methods of MOC's. The following are defined for the MOC **Customer.**

cCreditLimit

Changes the stored value of fCreditLimit. Obtains new value from a Prompter.[17]

fCreditLimit

"Answer credit limit"

↑ fCreditLimit

cKinds.

Changes the stored value of cKinds to {**Customer**}. Upon successful completion, removes itself from the list of allowable changes.

cInit

"Initialize fKinds."

[16]Equivalent to fForm. A *Form* is a special Smalltalk object which controls the display screen.
[17]A Prompter is a Smalltalk object which obtains input data from the user.

```
fKinds := Customer asSet.
pOrder add:#cInit.
↑ self.
```

cName

Changes the stored value of fName. Obtains a new value from either the Prompter or a PERSON object with the same surrogate as the customer.[18]

```
cName
        "Change the customer name."
|oldName name|
oldName := fName.
(name := CustomerLaws new :? lName(x,self)) notNil
    ifTrue:[name do:[:each | fName := each at:1]].
name isNil ifTrue:[fName := (Prompter prompt: 'Name?'
                        default: oldName)].
self validateLaws
   ifFalse:[fName := oldName.
          ↑ nil].
↑ fName
```

cGForm

Changes the stored value of fGForm. Obtains a new value from the Prompter.

```
cGForm
self changeVar:#fGForm to:('Customer' magnifyBy:2@6).
pOrder add:#cGForm
```

[18]This definition is enhanced somewhat from the original as presented in Chapter 5.

cInit

Sets initial values. Includes methods **cKinds**, **cCreditLimit**, **cName**, and **cForm**. Upon successful completion, removes itself from the list of allowable changes.

```
cInit
        "Initialize new object name & composition.

self cName;
    cGForm;
    cCreditLimit.
pOrder add:#cInit.
```

Law Statements

Law statements are defined as Prolog clauses in a separate section of the prototype. The following describes Prolog clauses which implement the law statements of the model object CUSTOMER.

lComp The composition of CUSTOMER must be simple.

```
"Composition Law - simple object"
lComp(aCustomer) :-
   is(cardinality,
      aCustomer ctObjectsIn:#composition),
   eq(cardinality,1).
```

lDebt

Indebtedness is the sum of the values of all orders of which the CUSTOMER is part-of.

```
"Customer debt = sum of order values"
```

```
lDebt(debt, aCustomer) :-
  is(orderValues,
    aCustomer sumValuesOf:#fValue
          fromObjectsOfKind:Order
          in:#fPartOf),
  eq(debt,orderValues).
```

lCreditLimit

Indebtedness is equal to or less than fCreditlimit.

```
"Debt is <= Credit Limit"
lCreditLimit(fDebt, fCreditLimit) :-
  le(fDebt,fCreditLimit).
```

The state functions, law statements and change functions of the other MOC's in this example are implemented similarly.

6.3.2 Prototype Operation

Recall that, in this example, the entire application is represented by the object *OrderSystem* which is the aggregation of all the Customers, Products and Orders of interest. Figure 6.3 shows the prototype implementation of *OrderSystem* as viewed through its Smalltalk window. The GraphPane of this window (A) provides a graphical view of the object, in this case, a logo or icon (although a more elaborate graphical image can be substituted). The pop-up service menu (B) displays the various services available to the user. For example, the *state functions* of *OrderSystem* can be display as in (C) by selecting the appropriate entry from the service menu. In this case, selecting *States* results in (C) as indicated by the arrow (a). The values of any state function can then be displayed by selecting from (C).

Similarly, the *change functions* (D) are displayed by selecting *Changes* as indicated by the arrow (b). A change function can then be invoked by choosing from (D). The partial ordering function in the prototype ensures that only the allowed change functions are displayed in (D). For instance, the change function cInit which is one of the original changes for *OrderSystem* is not displayed. This reflects the fact that this change was invoked when *OrderSystem* was created and the object cannot be initialized a second time.

In Figure 6.1, the ListPane (D) showed all the objects in the A-composition of *OrderSystem*. Figure 6.4 shows an ORDER object *No. 1* which is in the composition of *OrderSystem*. The service menu provides access to the objects in the *A-composition* and *part-of context* of *No. 1* as shown in (a) and (c) respectively. In this case, it can be seen that *No.1* is composed of the CUSTOMER object *john* and a PRODUCT object *a*. In addition, *No. 1* is part-of *OrderSystem*. Selecting any object in either of the ListPanes changes the focus to the selected object. In this way, the user can navigate from one object to any other object in the application.

Figure 6.5 shows how sets of objects can be manipulated in the prototype. The ListPane (A) displays all the objects in *OrderSystem*. A service menu (B) is brought up on the ListPane to provide a list of manipulations. In this example, the subset of objects *ofKind* PRODUCT are selected as shown by the sequence of arrows (a) and (b). This subset is then displayed in (C).

Figure 6.6 demonstrates some important principles when invoking change. Here, the ORDER object *No.1* is being created and added to *OrderSystem*. An order is created by making *OrderSystem* the object of focus then invoking the change function cACompOrder from its service menu (refer back to Figure 6.3). This function creates a new ORDER object and adds it to the composition of *OrderSystem*.

Since ORDER objects consist of a CUSTOMER and one or more PRODUCT objects, however, *No.1* cannot be lawfully added to *OrderSystem* until its own composition is complete. Therefore, as shown in Figure 6.6, the focus switches from *OrderSystem* to the newly created *No.1* (A). Change functions for *No.1* are then invoked to add CUSTOMER and PRODUCT objects to its composition. For example, the change function cACompCustomer for *No. 1* leads the user through a sequence of steps (as suggested by the arrow (a) but not shown in detail here) to create a CUSTOMER object, assign it a credit limit and then add it to the composition of *No.1.* Similarly, one or more products are added to the composition of *No. 1* as indicated by the arrow (b).

When the composition of *No.1* is complete (and validated) it can be added to *OrderSystem*, which will then regain the focus. In general, this recursiveness may be applied as many times as necessary.[19] The user can verify that *No.1* was added to *OrderSystem* by checking its state function fAComp. Displaying the value of this state function shows that the composition of *OrderSystem* includes an object of type ORDER (as in Figure 6.5).

Figure 6.7 shows the *history* of object *No.1.* The history window is selected from the service menu of *No.1* and displays a set of *time/state* value pairs corresponding to changes in *No.1* as they occurred. This history reveals, for example, that the state functions fName and fKinds were initialized when the object was created. Then, a customer and a product were added to its composition (fAComp). As a result of adding a product, the value of *No.1* represented by the state function fValue changed to 50 in accordance with its interobject law statements.

[19]In this example, sequential operation is assumed. That is, no other changes are allowed for *OrderSystem* until the *No.1* is complete. Relaxing this assumption results in concurrent manipulation of more than one object. Implementation of concurrent operation of OBIS is beyond the scope of this thesis.

6.3.3 The OBIS Implementation Process

The implementation process for developing an OBCM/IS application used in this prototype can be summarized as follows.

1. The analyst/designer (or end-user) examines the application and identifies the things of interest according to ontological principles. State functions, law statements and change functions are developed into an OBCM/IS specification, with the assistance of an appropriate notation.

2. Each state function, law statement and change function in the OBCM/IS is directly converted into Smalltalk code. Implementation consists of specifying classes (MOC's) for each model object and converting the OBCM/IS specifications into methods and Prolog assertions. A coding protocol or discipline guides this conversion. Although the conversion was manual in the prototype, support tools to automate the process are feasible.

3. The OBIS implementation is typically supported by an implementation "shell" consisting of object window handlers, service menus, and the various viewing, scanning, and manipulation operations. The shell will also support the OBCM/IS notation on the programming level.

4. Finally, since the prototype was designed to demonstrate proof of concept only, it utilized the standard Smalltalk windowing, menuing and user interface environment. However, the inherently visual form of OBIS encourages far more elaborate interfaces. For example, objects can be displayed in a variety of iconic images rather than simple windows. The user interface can also support more direct manipulation such as "dragging" object icons and placing them within each other icons to

represent composition and so forth. A direct implementation of the visual notation from Chapter 4 is a further possibility.

The traditional IS Systems Life Cycle [Freeman & Wasserman, 1984] usually specifies some form of explicit *design* phase, sometimes referred to as *logical* and *physical* design. The OBIS implementation process does not include such a phase. Instead, the conceptual model is *directly* converted into an implementation. There is no transformation of the constructs in the OBCM specification into a separate physical or logical design. In this way, the end-user interacts directly with an implementation which is semantically closely related to the original conceptual model of the application.[20]

6.4 Summary

This chapter has described a prototype implementing the OBCM/IS model proposed in Chapters 4 and 5. The prototype demonstrates that OBCM/IS has a relatively straightforward implementation in an object-oriented programming language such as Smalltalk. A basic "shell" and a simple protocol or set of rules for defining model objects are sufficient for the prototype. There is no inherent reason why implementation in other programming languages would not be possible.

Although the prototype used only the standard Smalltalk programming environment, it is able to support the basic principles behind the theoretical OBCM/IS model as well as the direct manipulation style of user interface. Given the many recent commercial developments in graphical interfaces and tools, it should be straightforward to implement more elaborate versions of the prototype, including an implementation of the visual notation discussed in Chapter 4.

[20]Of course, in practice, implementation variations to the basic OBIS framework may be necessary for performance or other reasons.

The following additional observations can be made regarding the operation of the prototype.

- The prototype avoids the technology-dependent operations typical of many conventional IS implementations. Users interface with the prototype on an *ontological level*, that is, a level that remains relatively close to the original conceptualization of the application. As such, the user deals with directly visible and manipulable representations of orders, customers, products and so on. There are no computer-dependent concepts such as updating data files, joining or selecting relations, or entering records.

- The prototype presents a highly homogeneous and consistent interface to the user. Every object essentially behaves in the same manner as any other object in the system, including the object representing the IS itself. In this way, the focus is on the properties and behaviour of the things represented by the object and not on the mechanics of the IS.

- Since the basic principles of *composition* and *part-of context* apply no matter which object the user is dealing with, navigation through the system and access to any object of interest is conceptually straightforward. That is, they involve only ontological concepts and not technology-dependent ones such as keys, data base navigation, and so forth.

It is important to emphasize that although the prototype is promising as a *proof of concept* to test out the theoretical model, it does not constitute proof that OBCM/IS will work for all real situations and in all cases. On the other hand, it is doubtful whether such a proof is possible for *any* IS methodology, data model or software development technique.

In the final analysis, moreover, the success of a software development methodology is probably best measured by practical application in the field over many years. Such an extended field test is beyond the scope and resources of this thesis. However, case study is often offered as an acceptable demonstration of the usefulness of an IS development approach. Indeed, certain cases have become established as "benchmark standards" in the literature for this purpose. The IFIP Working Conference Problem [Olle, Sol & Verron-Stuart, 1986] is one of these case studies.

The next chapter presents the OBCM/IS solution to the IFIP Working Conference problem.

Chapter 7

IFIP Working Conference Case Study

The IFIP Working Conference problem [Olle, 1982] is a popular reference case study for the evaluation and comparison of Information System Development methodologies and semantic data models [Olle et al., 1986]. Therefore, it provides a convenient "yardstick" for comparing and analysing different ISD approaches.

Recall from Chapter 1 that the intent of this thesis is *not* to develop a complete ISD methodology. It is limited to demonstrating the *feasibility* of an ontology-based approach to ISD. Hence, this thesis does not attempt to compare OBCM/IS to the many different methodologies and models applied against the IFIP case study. Nor does it attempt to implement a complete production-quality IFIP information system in all its detail. Rather it attempts to demonstrate in a general way how OBCM/IS can be applied to a significant real world problem.

The actual OBCM/IS solution to the IFIP case study is presented in Appendix D. This chapter focuses on two critical aspects of the OBCM/IS approach, namely (1) the interpretation of the IFIP problem in ontological terms so that it can be modelled by OBCM/IS, and (2) the ability of the prototype shell described in Chapter 6 to support an implementation of the IFIP case study. The chapter concludes with some brief comments relating OBCM/IS to other ISD approaches.

7.1 Requirements Analysis

The IFIP Working Conference problem is presented in its entirety in Appendix D. In this section, an OBCM/IS approach to requirement analysis is described.

Many approaches can be taken to perform the preliminary requirements analysis of this problem. One possible first step used here is an analysis of the nouns and noun phrases in the problem description to identify the *things* of interest.[1]

The extraction of the real world *things* of interest from the written problem description can be informal, intuitive and somewhat arbitrary. The following list summarizes some important components of the IFIP problem as deduced from the written description.

- *Working Conference.* A conference bringing together experts to discuss some topic of interest. Involves a Programme Committee and an Organizing Committee. Organized as a series of sessions. A Working Conference must achieve a break-even position.

- *Person.* A human being or individual.

- *Session.* A group of individuals meeting together at some specified time and place to discuss certain issues. The discussions will center around several papers accepted for the session.

- *Programme Committee* A group of individuals responsible for soliciting, collecting, evaluating, and approving the papers for a Working Conference. The Programme Committee organizes these papers into various sessions.

- *Organizing Committee* A group of individuals responsible for finding and organizing the facilities, issuing invitations and otherwise providing all the amenities required

[1] Grammatical and content analysis of problem descriptions in the context of ISD have been pioneered by methodologies such as NIAM [Verheijen & van Bekkum, 1982].

for a conference.

- *Invitation Letter.* A piece of text addressed to an individual inviting the individual to submit a paper to a conference.
- *Letter of Intent.* A piece of text addressed to the Programme Committee indicating an intent to submit a paper in response to a call for papers.
- *Author.* An individual who produces a paper.
- *Attendee.* An individual who has expressed an intent to attend a conference.
- *Paper.* A piece of writing or text, produced by an author or authors, which has been submitted to the Working Conference. A *selected* paper is one which has been duly refereed and recommended for inclusion in a particular session.
- *Text.* An organized collection of symbols expressing ideas, suggestions, and other conceptual information. Can be physically expressed in a variety of physical or electronic formats.
- *Facility.* A building, location, or other physical location at which some IFIP conference activity can occur.
- *Referee.* An individual who is responsible for evaluating a paper.
- *Referee Report.* A text giving a judgement upon a paper. Written by a Referee (or referees).
- *Chairman.* An individual who is responsible for a specific session.

7.2 OBCM Interpretation of the IFIP Case

It should be noted that much of the preliminary analysis above is open to interpretation and alternate formulation. Indeed, Bunge's Ontology stresses that our understanding of the world is a matter of state of art and depends on our "available knowledge, as well as upon our abilities, goals and even inclinations" [1977, p. 127].

Although other versions are possible, in this chapter the IFIP case study is described as an OBCM model in the following terms.

- Things which can be represented in OBCM by *simple* objects. These are described by the following model objects.

  PERSON
  AUTHOR
  ATTENDEE
  REFEREE
  TEXT
  FACILITY

- Things which can be represented as compositions of the simple objects implied above. These can be described by the following models.

  INVITATIONLETTER
  INTENTLETTER
  PAPER
  REFEREEREPORT.

- Further components of the IFIP case are viewed as aggregations of the objects

above. A Programme Committee, for example, is an aggregation of persons, invitation and intent letters, papers, referee reports and so forth. A Working Conference Session is organized around a number of selected papers and a facility (or facilities) in which to meet. A Working Conference is a collection of sessions. The IFIP system itself is an aggregation of all the objects of interest in the system. These components can be described by the models

PROGRAMME COMMITTEE
ORGANIZING COMMITTEE
SESSION
WORKING CONFERENCE
IFIP.

The next few sections briefly sketch the OBCM models for this interpretation. Details of the formal definitions of state functions, law statements and change functions are fully described in Appendix D.

7.2.1 Simple Objects

The model object PERSON describes all human beings (or individuals) of interest to the IFIP problem. A collection of surrogates S_P is assumed, each of which corresponds to some person of interest.

An author is an individual who is involved in producing a paper. The model object AUTHOR is a view of persons. That is, the surrogates for AUTHOR is a subset $S_{Pa} \subseteq S_P$.

An attendee is an individual who has declared an intent to attend a Working Conference. The model object ATTENDEE is a view of persons. That is, the surrogates for ATTENDEE is a subset $S_{Pt} \subseteq S_P$.

An referee is an individual who has agreed to referee papers for a Working Conference. The model object REFEREE is a view of persons. That is, the surrogates for REFEREE is a subset $S_{Pr} \subseteq S_P$.

A text is an organized collection of symbols expressing ideas, suggestions, and other information.[2] Assume a collection of surrogates S_{Tx} each corresponding to some unique text.

7.2.2 Composite Objects

An invitation letter or call for papers is a piece of text addressed to some individual inviting that individual to submit a paper on some topic. It is represented by the model object INVITATION. The possible surrogates for INVITATION is defined as

$$S_C = S_P \times S_{Tx},$$

that is, an invitation letter is composed of a PERSON object and a TEXT object.[3]

An intent letter is a text from some individual indicating an intent to submit a paper in response to an invitation. It is represented by the model object INTENT whose possible surrogates are defined as

$$S_I = S_P \times S_{Tx},$$

A paper is a text produced by an author or authors, which has been submitted to the Working Conference in response to a call for paper. It is modelled by PAPER with

[2]It can be argued that in Bunge's ontology, text is *not* a substantial things and cannot be represented by a model thing (see also [Bunge, 1977, pp. 116-117]). This thesis assumes, however, that a text has some physical form which can be described by a model object.

[3]In the interest of simplicity, the possibility of addressing an invitation letter jointly to more than one individual is not considered.

possible surrogates[4]

$$S_{Pp} = 2^{S_{Pa}} \times S_{Tx}.$$

That is, for OBCM modelling purposes, a paper is an aggregate of one or more authors and a text.

A Referee Report is a text written by a referee, commenting upon a paper. It is described by the model REFEREEREPORT. Assuming for simplicity that a paper is assessed by a single referee, the surrogate for a referee report is defined as

$$S_{RR} = S_R \times S_{Pp} \times S_{Tx}.$$

That is, for OBCM modelling purposes, a referee report is an aggregation of a referee, a paper and a text commenting upon the paper.

7.2.3 Committees and Conferences

For OBCM modelling purposes, the Programme Committee is viewed as an aggregation of the invitation letters, intent letters, papers, and referee reports which are required to support the activities of the Committee. The Programme Committee is represented by the model object PROGCOM and its surrogate is defined as

$$S_{PC} = 2^{S_C} \times 2^{S_I} \times 2^{S_{Pp}} \times 2^{S_{RR}}.$$

Conceptually, in the OBCM/IS model, as invitations are generated, they are added to the Programme Committee. As intent letters and papers are received, they are also added to the Programme Committee. And when a paper is distributed for refereeing, a referee report is generated and added to the Programme Committee.

[4]This is a fairly liberal interpretation. A more restrictive version might assume, for example, that the only papers of interest are those (1) in response to a call (ie. no unsolicited papers are acceptable) and (2) for which an intent to submit was declared (ie. there is an intent letter on file). In this case, the surrogates for PAPER would be modified as $S_{Pp} = 2^{S_{Pa}} \times S_{Tx} \times S_C \times S_I$.

For OBCM modelling purposes, the Organizing Committee is viewed as a set of facilities and attendees.

$$S_{OC} = S_F \times S_{Pt}.$$

Conceptually, in the model, as facilities are located, they are added to the composition of the Organizing Committee. Similarly, as individuals confirm their intent to attend, they are added to the Organizing Committee.

A Working Conference consists of a Programme Committee, an Organizing Committee and a number of sessions. It is described by the model WORKINGCONFERENCE. Its surrogate is

$$S_{WC} = 2^{S_S} \times S_{PCOM} \times S_{OCOM}$$

The IFIP is the collection of all Working Conferences and all individuals of interest to the IFIP system.[5] It is described by the model IFIP which has the surrogates

$$S_{IFIP} = 2^{S_{WC}} \times 2^{S_P}.$$

7.3 IFIP Case Study - Operation

The OBIS for the IFIP case study consists of the aggregation of all the objects of interest to the application. The OBIS is itself an object, in this case, of type IFIP. Call this object *IFIP*.

Figure 7.1 shows selected aspects of *IFIP* in OBCM visual notation. *IFIP* is viewed as an aggregation of Working Conferences and people. Change functions allow users to add new Working Conferences and people to the composition of *IFIP* in response to conditions in the real world. A Working Conference, in turn, is composed of one Programme Committee, one Organizing Committee and any number of Sessions.

[5]This is an appropriate place to populate with objects of type PERSON since individuals can participate in several Working Conferences.

7.3.1 Populating the IFIP

Starting from *IFIP*, new Working Conferences can be created and added by invoking *IFIP*.cAddConference. The act of creating a new Working Conference results in the creation and inclusion of a Program Committee and an Organizing Committee for the new conference.[6]

7.3.2 Programme Committee Activities

The activities of the Programme Committee, as specified in the problem statement, are supported as follows.

1. *Preparing a list for the call for papers.* The value of PROGCOM.fCallList returns a set of persons in the composition of *IFIP* from which invitations can be generated.

   For each object in fCallList, an object of type INVITATION is created. In the prototype, these are created manually and added to the composition of the Program Committee. However, in a working system, invitations would be generated automatically by the system.

2. *Registering letters of intent.* For each letter of intent received by the Programme Committee, an object of type INTENT is created and added to the Programme Committee.

3. *Registering papers.* For each paper received by the Programme Committee, an object of type PAPER is created and added to the Programme Committee.

4. *Distributing papers among referees.* The distribution of papers among referees is represented by creating objects of type REFEREEREPORT. This involves bringing together a referee with a paper.

[6]This is automatically performed as part of WORKINGCONFERENCE.cInit.

5. *Collecting Referee Reports.* The collection of referee reports is represented by adding evaluations to referee reports as they are received by the committee.

6. *Grouping papers into sessions and selecting chairmen.* This is represented by invoking WORKINGCONFERENCE.cAddSession.[7]

Figure 7.2 illustrates some of the basic structure of a Programme Committee in the OBCM visual notation. Invitations and letters of intent are seen as an aggregation of people and text. A paper is an aggregation of one or more authors and a text. A referee report is an aggregation of a paper, a referee and a text (commenting upon the paper).[8]

The visual notation as developed in Chapter 4 does not provide a specific way of showing that authors and referees are also persons but an additional comment such as that shown in the figure is useful.

7.3.3 Organizing Committee Activities

The activities of the Organizing Committee as specified in the problem description are supported as follows.

1. *Preparing a list of invitees.* The list of invitees is simply again a list of all persons in the composition of *IFIP*. Note that both the Organizing Committee and the Programme Committee are capable of generating the same list.

2. *Priority Invitations.* The IFIP problem description does not define the meaning of priority invitations or their significance. In this version of the case, it is assumed that invitations above include priority invitations.

3. *Invitations to authors of accepted papers.* Since all persons receive invitations and authors are persons, it is assumed that this is a redundant requirement for the

[7]The selection of chairmen is not detailed in this presentation of the case study.
[8]For clarity, Figure 7.2 repeats the model object TEXT several times.

OBIS. A list of authors only is possible, however, by scanning for all objects of type AUTHOR in the composition of the Working Conference.

4. *Invitations to authors of rejected papers.* This is also assumed to be a redundant requirement.

5. *Avoiding duplicate invitations.* Duplicate invitations are avoided because the collection of objects of type PERSON in the composition of *IFIP* is a set.

6. *Registering acceptance of invitations.* The registration of invitation acceptances is represented by the creation of objects of type ATTENDEE as each acceptance is received.

7. *Generating final list of attendees.* Lists of attendees can be generated by Session. The final list of attendees is union of the sets of attendees in the composition of all sessions of the Working Conference.

7.3.4 Implementation of the IFIP Case Study

Figures 7.3 through 7.5 depict the IFIP Case Study as implemented in the OBCM/IS Prototype Shell described in Chapter 6.[9]

In Figure 7.3, the arrow (a) represents the addition of a Working Conference, in this case, *WC#1*, to *IFIP*. When *WC#1* is created, Programme and Organizing Committees were also created and added to *WC#1*. Also at some time, a Session was created and added to *WC#1*. These activities are not shown but the resulting composition of *WC#1* is shown by scanning its A-composition as indicated by arrow (b).

Figure 7.4 shows how the prototype deals with the Programme Committee. Scanning its A-composition (arrow (a)) reveals that the committee has issued three invitations

[9]These figures attempt to summarize the dynamics of user interaction. Therefore, they vary somewhat from the actual screen images produced by the prototype.

letters and that two individuals (ie. objects *Mary* and *John*) have filed an intent to submit papers. Two papers (*Objects* and *Bunge*) have been received by the committee and a referee report has been commissioned for one of these (the *Objects* paper).

Arrow (b) shows a shift in focus to the paper *Bunge*. Scanning the composition of *Bunge* (arrow (c)) reveals how it is composed of the author *John* and a text. Shifting the focus to the text of the paper (arrow (d)) reveals an object with the actual written text.

Figure 7.5 shifts the focus to the paper's author, *John*. Arrow (a) shows objects with the same surrogate as the author *John*. As revealed, an object of type PERSON exists which shares the same surrogate. Arrow (b) shows the *part-of context* of the author *John*. It shows that *John* is part-of the paper *Bunge*, Programme Committee, a Working Conference, and finally, *IFIP* itself.

7.4 OBCM/IS and Other ISD Approaches

Although many solutions to the IFIP Working Conference problem have been published [Olle et al., 1986], the assessment and comparison of methodologies still poses enormous difficulties [Flint, 1986; Brodie et al., 1983; Bubenko et al., 1983; Rzevski, 1983]. For example, Sol [1983] suggests at least five different approaches for such assessments:

1. Describe an ideal methology and evaluate against it. The problem is, of course, to develop the ideal.

2. Distill a set of important features from various methodologies. Determination of which features are important, however, depends heavily on subjective evaluation.

3. Formulate *a priori* hypotheses on the partial ordering of features and their important. Again, such hypotheses are difficult to derive.

4. Define a meta-language for describing and comparing a wide range of methodologies. However, the basis of such a meta-language is not well-understood.

5. Follow a contingency approach whereby a methodology is evaluated for specific situations. A true comparison is difficult with such an approach.

Research is ongoing in understanding and developing an appropriate basis for evaluation and comparison, as well as for creating taxonomies of ISD methodologies [Olle et al., 1983].

Anything beyond a superficial comparison of OBCM/IS to other methodologies is, therefore, beyond the scope of the current work. Further, it is important to stress that the intent of this thesis is not simply to generate another Semantic Data Model or ISD Methodology but rather to explore the feasibility of an ontology-based approach to ISD. However, some brief comments placing OBCM/IS in relation to other ISD approaches are appropriate. To this end, these comments are organized according to Brandt's eight part taxonomy for comparing ISD methodologies [Brandt, 1983]. This taxonomy can be summarized as follows:

1. *Origin and Experience.* This reports on the environment in which the methodology has been developed and is practiced. Differences in academic and commercial methodologies are highlighted.

2. *Development Process.* Coverage of the development process is discussed here. The transformation of the final specification to programs in of special interest.

3. *Model.* The basic constructs, degree of formalism, mathematical or other foundations, etc. are of interest.

4. *Iteration and Tests.* This addresses the validation and verification aspects of a methodology.

5. *Representation Means.* Here, use of graphical elements, formal languages and other presentation features are covered.

6. *Documentation.* The integration of documentation and its importance are discussed.

7. *User Orientation* The knowledge expected of the end user, and ways of user participation are of interest here.

The following sections briefly summarize OBCM/IS according to these parameters.

7.4.1 Origin and Experience

OBCM/IS was developed over a two year period in an academic environment. Some experience has been gained utilizing the Smalltalk prototype shell, including an implementation of the IFIP problem.

A major feature is that its origins are based on a significant theory of the structure of the world, ie. ontology, rather than implementation considerations. Another important feature is its use of modern, object-oriented concepts (which are supported by its underlying theoretical foundations).

7.4.2 Development Process

OBCM requires an analysis of the application according to ontological principles. The analysis then proceeds to a formal OBCM specification, a direct implementation into an OBIS, and end-use. An explicit logical or physical design phase is missing. The original constructs in the OBCM are carried forward to the end use of the resulting system. Written and visual notations as well as an implementation shell support the process.

In particular, the ability to directly implement the OBCM specification and to directly interact with the results of the implementation should be noted. This is an attempt to

achieve the "homogeneous path" from conceptual specification to end-use discussed in Chapter 1 of this thesis.

7.4.3 Model

The model is ontology-based as derived from Bunge. The basic constructs are not related to any prominant database or ISD models such as entity-relationship or relational. Instead, they are grounded in ontology. Basic concepts such as thing, property, change, state, etc. are explicitly defined.

OBCM/IS shares with semantic models an emphasis on the "meaning" of data as well as its structure. Unlike most semantic models, however, OBCM/IS supports a very general structure and depends on the use of law statements to represent relationships among objects.

7.4.4 Iteration and Test

OBCM/IS can be applied iteratively starting with simple models which are continually being refined and enhanced with additional state functions, law statements and change functions. An OBIS is "self-testing" in the sense that the entire system and all objects within the system are always subject to validation against law statements.[10]

7.4.5 Representation Means

A written and visual notation have been described in Chapter 4 of this thesis. Improvements on these as well as a complete formal language are a desirable and possible enhancements.

[10]This does not imply that the OBIS is error-free since the original analysis and formulation of law statements might be in error.

7.4.6 Documentation

OBCM/IS is self-documenting in that, in theory, access is always available to state functions, law statements and change functions of model objects. These fully define all aspects of an implementation. State functions, law statements and change functions will be, in principle, be available to the system user in their original form although in practice, they may be more readily available in their implemented form, ie. as program code.

7.4.7 User Orientation

OBCM/IS is designed for use by both end-users and system developers. In principle, end-users can specify the conceptual model and operate an implementation which directly corresponds to the original conceptual model.

7.4.8 Tools and Prospects

A prototype shell is available but will require considerable refinement for practical use. Many of the ideas behind object-orientation can also be applied, often directly to OBCM/IS. In particular, concepts such as software reusability, graphical user interfaces (GUI), and OOPL software development environments are compatible with OBCM/IS.

Therefore, while OBCM/IS is still an academic product with little practical application, it can both take advantage of and contribute to modern software practice.

7.5 Summary

This chapter has described an OBCM/IS solution to the IFIP Working Conference problem. A detailed comparison of OBCM/IS to other Semantic Models and IS methodologies is beyond the scope of the current work. However, this chapter concludes with a brief set of comments attempting to relate OBCM/IS to current practice in ISD.

It was not the intent of this thesis to produce a complete and full OBCM/IS specification in all its fine detail.[11] Nor was there any attempt to generate a customized, production-quality implementation. Such an effort would primarily be a matter of additional detail but no significant new principles.

What was shown in this chapter and in Appendix D, however, is how an ontology-based interpretation of the IFIP problem can be sustained and how an OBCM representation of the problem can be generated. The adequacy of the prototype shell of Chapter 6 was illustrated, at least for the IFIP problem, by an actual OBIS implemention of the case study. In summary, the material in this chapter together with Appendix D demonstrates the *feasibility* of an ontology-based approach to IS development which unifies analysis, implementation and end-use.

[11]Moreover, the written problem description does not provide the necessary particulars.

Chapter 8

Contributions and Extensions

8.1 Thesis Summary

This thesis has developed an object-oriented model for the conceptual modelling of information systems with a theoretical grounding in ontology. Further, a framework for systems implementation based on this model was presented. A "proof of concept" prototype was implemented to advance the model-building, illustrate the major principles, and explore practical applications of the model.

Consider the overall goal of this research as defined in Chapter 1: *To define and formalize a theory-based, object-oriented metamodel for describing and developing Information Systems.*

In working toward this objective, two sub-goals were articulated in terms of the following research questions:

1. *Can the intuitions in the object paradigm be sufficiently formalized into a theory-based, Conceptual Modelling scheme for Information Systems Development?*

2. *Can such a formalization be sufficiently operationalized to provide a single, unifying principle for the conceptual modelling, implementation and end-use of Information Systems?*

The first of these was addressed by appealing to metaphysics or ontology. The basic premise behind this approach is that if a conceptual model is an abstraction of reality, then the constructs in the model should be based on a theory of reality, ie., an ontology.

The well-known system of Mario Bunge was chosen for its rigor, formalism and acceptance among the scientific community. This system was *operationalized* into the Object-Based Conceptual Model or OBCM. Thus, unlike many other ISD approaches, all of the underlying assumptions are explicitly stated and the basic principles and concepts are formally defined. In the sense that ontology helps us better understand reality, it is hoped that OBCM provides the potential for better models of IS applications.

The second sub-goal was addressed by the Object-Based Information System or OBIS, a framework for implementing IS applications expressed in terms of OBCM. This framework deals with issues such as implementing abstract OBCM constructs, maintaining semantic continuity between the application and the implementation, and producing user interfaces consistent with the ontological approach.

The OBIS is a *direct implementation* of the constructs in the OBCM model of the application. This has the following important implications.

1. The analyst and user *interpret* the application in accordance with ontological principles. This interpretation is the OBCM-based conceptual model or specification.

2. There is no concept of logical or physical design, in which the designer attempts to convert a specification to some technically feasible IS design. Instead, the implementor works directly from the OBCM using a set of tools and/or programming protocols to implement the analyst's specification immediately into a systems implementation.

3. The end-user interacts directly with this implementation.

Thus, OBIS strives for a *seamless, homogeneous* path from conceptual model to the end-user. As such, it is an attempt to unify three major aspects of information systems development: analysis, implementation and user interface.

The prototype which was developed as part of this thesis attempts to provide a practical demonstration of the potential of OBCM/IS as an approach to ISD. Its main features can be summarized as follows:

1. is theory-based and object-oriented;
2. employs a small set of simple, unifying principles;
3. has an inherently homogeneous structure composed of very high-level, abstract entities which can be directly related to the application domain; and
4. supports a direct manipulation style of end-user interface.

8.2 Contributions

In chapter 1, a number of potential contributions offered by the thesis were listed. These can be summarized as follows.

8.2.1 Theoretical Contributions

Theory-Based Model for ISD

OBCM/IS is an attempt to ground ISD in a basic metaphysics. Previous approaches to ISD are usually not theory-based and have tended to derive from individual experience, past practice, rules of thumb, and technological trends. The proposed model, therefore, provides one answer to the call for a theory-based approach to Information Systems Development. This is the main theoretical contribution of the thesis.

Extension of the Object Paradigm

The proposed model is *object-oriented* and provides a conceptual context within which the intuitive appeal and modelling power of the object paradigm can be better understood.

Thus, it extends the use of the object paradigm to the general case of Information Systems Development.

Reducing the Semantic Gap

This thesis proposes a well-defined, theory-based modelling construct *object* which can be shared by the analysts, implementors and end-users of information systems. Since all parties share the same basic construct, the likelihood of "semantic gaps" between users and systems developers is reduced.

8.2.2 Practical Contributions

Implementation

Although there have been a number of calls for theoretical models, it is not obvious how theory can actually be applied in practice. The prototype developed in this thesis is crude and designed to be illustrative only. However, it provides a demonstration of the practical impact of a basic theory such as ontology on the design, implementation and use of information systems. This demonstration is perhaps the primary practical contribution of this thesis.

OBIS Prototype Tools

The prototype utilizes a simple protocol or programming discipline to implement the OBCM *object* construct. Further, it incorporates a number of working tools and implementation ideas such as the "shell" concept, object scanners, service menus, and visual interfaces. Thus, it demonstrates how basic concepts from object-oriented programming environments can be adapted to the ISD context.

8.3 Future Extensions to the Research

The ideas arising from this research suggest several possible extensions.

8.3.1 Extensions and Improvements to OBCM/IS

Enhancements to the Model

As discussed in Chapter 4, Bunge allows for an explicit time parameter in his theory of representation. Operationalizing this aspect of this ontology can potentially result in an extension of OBCM with time.

Another possible enhancement is the development of a formal language for OBCM using Bunge's ontology as a base.

Improving the Prototype Shell

Since the contribution of this project focussed on the theoretical model and "proof of concept", neither the efficiency of the shell nor its user interface was given high priority. The prototype is, therefore, not suited for production use and would require performance enhancements such as (1) improved mechanisms for law validation and the partial ordering of change functions, (2) improved user interface (such as incorporation of the OBCM visual notation), and (3) general improvements in robustness, speed and performance.

Automated Tools

The current implementation makes extensive use of the standard Smalltalk programming environment to create Model Object Classes (MOC's), state function methods, change methods, and Prolog clauses for expressing law statements. A possible extension is to customize or create a programming environment to automate the task of generating MOC's directly from the OBCM specification. This would include parsers to interpret

OBCM notation and automatically generate the necessary software (eg. classes and methods if Smalltalk is used as the host language).

Rapid Prototyping and CASE

Many of the ideas suggested by the OBIS implementation framework potentially simplify and streamline the systems development process. As such, OBCM/IS can potentially serve as the basis of a rapid prototyping or Computer-aided software engineering (CASE) tool [Hughes & Clark, 1990].

Interfacing with Conventional Systems

In concept, an OBCM/IS can obtain the values for state variables of its objects from conventional databases. The values of state functions can also be returned to the conventional database. In this way, an OBCM/IS can act as a common access point to heterogeneous data bases [Chung, 1990].

8.3.2 Enhancements to the Ontological Basis

In a different direction, enhancements can be made to the Ontological basis itself. For instance, in the opinion of this thesis, Bunge's ontology provides only a very general model of change. A more specific model of change may improve substantially the dynamic aspects of OBCM.

More generally, a research programme to build an ontology (or ontologies) suitable for IS modelling may prove an important step in developing future theories of information systems.

Bibliography

[1] Abbott, R. "Knowledge Abstraction" *CACM.* August, 1987.

[2] Afsarmanesh, H. and McLeod, D. "A Framework for Semantic Database Models". *New Directions for Database Systems.* Ariav, G. and Clifford, J. (Eds). Ablex, 1986.

[3] Banerjee, J., Chou, H., Garza, J., Kim, W., Woelk, D., and Ballou, N. "Data Model Issues for Object-Oriented Applications". *ACM Transactions on Office Information Systems.* January, 1987.

[4] Baroody, A. and DeWitt, D. "An Object-Oriented Approach to Database System Implementation". *ACM Transactions on Database Systems.* December, 1981.

[5] Bassett, P. "Frame-Based Software Engineering". *IEEE Computer.* July, 1987.

[6] Beech, D. "Groundwork for an Object-Oriented Database Model". *Research Directions in Object-Oriented Programming.* Shriver, B. and Wegner, P. (Editors). MIT Press, 1987.

[7] Berzins, V., Gray, M. and Nauman, D. "Abstraction-Based Software Development". *CACM.* May, 1986.

[8] Blaha, M., Premerlani, W. and Rumbaugh, J. "Relational Database Design Using an Object-Oriented Methodology". *CACM.* April, 1988.

[9] Booch, G. "Object-Oriented Design". *Software Engineering with ADA.* Benjamin/Cummings, 1983.

[10] Booch, G. "Object-Oriented Design". *IEEE Transactions on Software Engineering.* February, 1986.

[11] Borgida, A., Greenspan, S. & Mylopoulos, J. "Knowledge Representation as the Basis for Requirements Specifications". *IEEE Computer.* April, 1985.

[12] Borgida, A., Mylopoulos, J., & Wong, H. "Generalization/Specialization as a Basis for Software Specification". *On Conceptual Modelling.* Brodie, Mylopoulos & Schmidt (Ed), Springer-Verlag, 1984.

[13] Brachman, R. "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks". *Computer.* October, 1983.

[14] Brandt, I. "A Comparative Study of Information System Design Methodologies". *Information Systems Design Methodologies: A Feature Analysis.* Olle, T., Sol, H., and Tully, C., Editors. North-Holland. 1983.

[15] Brodie, M. "On the Development of Data Models". *On Conceptual Modelling.* Brodie, Mylopoulos & Schmidt (Ed), Springer-Verlag, 1984.

[16] Brodie, M., Mylopoulos, J. & Schmidt, J. (Eds). *On Conceptual Modelling.* Springer-Verlag, 1984.

[17] Brodie, M. and Mylopoulos, J. "Knowledge Bases and Databases: Semantic vs. Computational Theories of Information". *New Directions for Database Systems.* Ariav, G. and Clifford, J. (Eds). Ablex, 1986.

[18] Brodie, M. and Ridjanovic, D. "On the Design and Specification of Database Transactions". *On Conceptual Modelling.* Brodie, Mylopoulos & Schmidt (Ed), Springer-Verlag, 1984.

[19] Brodie, M., Ridjanovic, D., and Silva, E. "On A Framework for Information System Design Methodologies". *Information Systems Design Methodologies: A Feature Analysis.* Olle, T., Sol, H., and Tully, C., Editors. North-Holland. 1983.

[20] Brooks, F. "No Silver Bullet: Essence and Accidents of Software Engineering". *Computer.* April, 1987.

[21] Bubenko, J. "Information Modelling in the Context of Systems Development". *Information Processing 80. Lavington, S. (Ed).* North-Holland, 1980.

[22] Bubenko, J. *Information System Methodologies - A Research View.* Syslab Report No 40, February, 1986.

[23] Bubenko, J., Gustafsson, M., and Karlsson, T. "Comments on Some Comparisons of Information System Design Methodologies". *Information Systems Design Methodologies: A Feature Analysis.* Olle, T., Sol, H., and Tully, C., Editors. North-Holland. 1983.

[24] Bunge, M. *Treatise on Basic Philosophy: Volume 3, The Furniture of the World.* D. Reidel Publishing. 1977.

[25] Bunge, M. *Treatise on Basic Philosophy: Volume 4, A World of Systems.* D. Reidel Publishing. 1979.

[26] Buzzard, G. and Mudge, T. "Object-Based Computing and the ADA Programming Language". *IEEE Computer.* March, 1985.

[27] Cameron, J.R. "Two Pairs of Examples in the Jackson Approach to Systems Development'" *IEEE Tutorial on Software Design Techniques.* Freeman, P. and Wasserman, A. (Editors). IEEE, 1983.

[28] Cameron, J.R. "An Overview of JSD". *IEEE Transactions on Software Engineering.* February, 1986.

[29] Carlson, W. "Business Information Analysis and Integration Technique (BIAIT) - The New Horizon". *Advanced System Development/Feasibility Techniques.* Couger, J., Colter, M., and Knapp, R. (Editors). John Wiley, 1982.

[30] Chen, P. "The Entity-Relationship Model: Toward a Unified View of Data". *ACM Transactions on Data Base.* March, 1976.

[31] Chung, C. "DATAPLEX: An Access to heterogeneous Distributed Databases". *CACM.* January, 1990.

[32] Codd, E. "Extending the Database Relational Model to Capture More Meaning". *ACM Transactions on Data Base.* December, 1979.

[33] Cox, Brad. *Object-Oriented Programming.* Addison-Wesley, 1986.

[34] Cox, Brad. "Message/Object Programming: An Evolutionary Change in Programming Technology." *IEEE Software.* January, 1984.

[35] Cox, B. and Hunt, B. "Objects, Icons and Software-IC's". *BYTE.* August, 1986.

[36] Danforth, S. and Tomlinson, C. "Type Theories and Object-Oriented Programming". *ACM Computing Surveys.* March, 1988.

[37] Diederich, J. and Milton, J. "Experimental Prototyping in Smalltalk." *IEEE Software.* May, 1987.

[38] Digitalk. *Smalltalk/V286: Tutorial and Programming Handbook.* 1988.

[39] Fischer, G. "Cognitive View of Reuse and Redesign". *IEEE Software.* July, 1987.

[40] Fishman, D., Beech, D., Cate, H., Chow, E., Connors, T., Davis, J., Derrett, N., Hoch, C., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M., Ryan, T., and Shan, M. "Iris: An Object-Oriented Database Management System". *ACM Transactions on Office Information Systems.* January, 1987.

[41] Flint, R. "An Activity Model of the Working Conference Problem". *Information Systems Design Methodologies: Improving the Practice.* Olle, T., Sol, H., & Verrjin-Stuart, A (Eds). Elsevier Science Publishers, 1986.

[42] Flint, R. and Leveson, N. "The PAMS Approach top Modelling Database Activity". *Information Systems: Theoretical and Formal Aspects.* Sernadas, A., Bubenko, J., & Olive, A. (Eds). North-Holland, 1985.

[43] Floyd, C. "A Comparative Evaluation of System Development Methods". *Information Systems Design Methodologies: Improving the Practice.* Olle, T., Sol, H., & Verrjin-Stuart, A (Eds). Elsevier Science Publishers, 1986.

[44] Foley, J. "Interfaces for Advanced Computing". *Scientific American.* October, 1987.

[45] Gibbs, S. "Conceptual Modelling and Office Information Systems". *Office Automation.* Tsichritzis, D. (Editor) Springer-Verlag, 1985.

[46] Gilmore, P. *Concepts and Methods for Database Design.* Technical Report 87-31. Department of Computer Science. University of British Columbia. 1987.

[47] Goguen, J., Thatcher, J., and Wagner, E. "An Initial Algebra Approach to the Specification, Correctioness, and Implementation of Abstract Data Types". *Current Trends in Programming Methodology. Volume IV, Data Structuring.* Yeh, R. (Ed). Prentice-Hall, 1978.

[48] Goldberg, A. "Programmer as Reader". *IEEE Software.* September, 1987.

[49] Goldberg, A. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[50] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[51] Greenspan, S. *Requirements Modelling: A Knowledge Representation Approach to Software Requirements Definition* Computer Systems Research Group, University of Toronto, 1984.

[52] Greenspan, S. and Mylopoulos, J. *A Knowledge Representation Approach to Software Engineering: The Taxis Project.* CIPS Proceedings, Ottawa, 1983.

[53] Hacking, I. *Representing and Intervening.* Cambridge University Press, 1983.

[54] Halbert, D. and O'Brien, P. "Using Types and Inheritance in Object-Oriented Programming". *IEEE Software.* September, 1987.

[55] Harel, D. "On Visual Formalisms". *CACM.* May, 1988.

[56] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[57] Hughes, C. and Clark, J. "The Stages of CASE Usage." *Datamation.* February 1, 1990.

[58] Hull, R. and King, R. "Semantic Database Modelling: Survey, Applications and Research Issues". *ACM Computing Surveys.* September, 1987.

[59] IBM. "Business Systems Planning". *Advanced System Development/Feasibility Techniques.* Couger, J., Colter, M., and Knapp, R. (Editors). John Wiley, 1982.

[60] Isner, J. "A Fortran Programming Methodology Based on Data Abstraction". *CACM.* October, 1982.

[61] Jackson, M. *Principles of Program Design* Academic Press, 1975.

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

[63] Jacky, J. and Kalet, I. "An Object-Oriented Programming Discipline for Standard Pascal". *CACM.* September, 1987.

[64] Jansson, C-G. "A Framework for Representation". *Information Systems: Theoretical and Formal Aspects.* Sernadas, A., Bubenko, J., & Olive, A. (Eds). North-Holland, 1985.

[65] Kent, W. "Entities and Relationships in Information". *Architecture and Models in Data Base Management Systems.* Nijessen, G. (Editor). North Holland, 1977.

[66] Kent, W. *Data and Reality.* North Holland, 1978.

[67] Kent, W. "The Realities of Data: Basic Properties of Data Reconsidered". *Data Semantics (DS-1).* Steel, T. and Meersman, R. (Editors). North Holland, 1986.

[68] Kilov, H. "Reviews of Object-Oriented Papers". *SIGMOD Record.* December, 1989.

[69] Kim, W. and Lochovsky, F. *Object-Oriented Concepts: Databases and Applications.* ACM Press, Addison-Wesley, 1989.

[70] Kimura, G. "A Structure Editor for Abstract Document Objects". *IEEE Transactions on Software Engineering.* March, 1986.

[71] Konsynski, B. "Databases for Information Design". *New Directions for Database Systems.* Ariav, G. and Clifford, J. (Eds). Ablex, 1986.

[72] Kroenke, D. and Dolan, K. *Database Processing.* Science Research Associates, 1988.

[73] Kung, C. and Solvberg, A. "Activity Modelling and Behaviour Modelling". *Information Systems Design Methodologies: Improving the Practice.* Olle, Sol & Verrijn-Stuart (Ed). North-Holland, 1986.

[74] Lakoff, G. *Women, Fire, and Dangerous Things.* University of Chicago Press, 1987.

[75] Ledbetter, L. and Cox, B. "Software-IC's". *BYTE.* June, 1985.

[76] Ledgard, H. and Taylor, R. "Two Views of Data Abstraction". *CACM.* June, 1977.

[77] Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development.* McGraw-Hill, 1986.

[78] Liskov, B. and Zilles, S. "Specification Techniques for Data Abstractions". *IEEE Transactions of Software Engineering.* March, 1975.

[79] Maier, D. and Stein, J. "Development and Implementation of an Object-Oriented DBMS". *Research Directions in Object-Oriented Programming.* Shriver, B. and Wegner, P. (Editors). MIT Press, 1987.

[80] Mattessich, R. "Axiomatic Representation of the Systems Framework: Similarities and Differences Between Mario Bunge's World of Systems and My Own Systems Methodology". *Cybernetics and Systems.* 13:51-71, 1982.

[81] Mattessich, R. "An Applied Scientist's Search for a Methodological Framework". *Proceedings of the 11th International Wittgenstein Symposium.* August, 1986.

[82] Mattessich, R. "Prehistoric Accounting and the Problem of Representation: On Recent Archeological Evidence of the Middle-East from 8000 B.C. to 3000 B.C.". *The Accounting Historians Journal.* Vol 14, No 2, 1987.

[83] Mattessich, R. "Social and Physical Reality in Accounting: Onion model vs. Pygmalion Syndrome". *UBC Working Paper.* 1989.

[84] Mendelson, E. *An Introduction to Mathematical Logic.* Princeton: Van Nostrand. 1963.

[85] Meyer, B. "Reusablity: The Case for Object-Oriented Design". *IEEE Software.* March, 1987.

[86] Meyer, B. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[87] Minsky, N. *Law-Governed Systems.* Technical Report LCSR-TR-101. Department for Computer Science. Rutgers University. 1989.

[88] Minsky, N. and Rozenstein, D. "A Law-Based Approach to Object-Oriented Programming." *OOPSLA '87 Proceedings.* 1987.

[89] Nierstrasz, O. "An Object-Oriented System". *Office Automation.* Tsichritzis, D. (Editor) Springer-Verlag, 1985.

[90] Nierstrasz, O. "What is the 'Object' in Object-Oriented Programming?". *Objects and Things.* Tsichritzis, D. (Editor) 1987.

[91] Nilsson, N. *Principles of Artificial Intelligence.* Tioga Publishing. 1980.

[92] Olle, T., Sol, H., and Tully, C., Editors. *Information Systems Design Methodologies: A Feature Analysis.* North-Holland. 1983.

[93] Orr, K. "Structured Systems Design". *Advanced System Development/Feasibility Techniques.* Couger, J., Colter, M., and Knapp, R. (Editors). John Wiley, 1982.

[94] Pascoe, G. "Elements of Object-Oriented Programming". *BYTE.* August, 1986.

[95] ParkPlace Systems, Inc. *The Smalltalk-80 Programming System. Version 2.4.* ParkPlace Systems, Inc, 1989.

[96] Parsons, T. *Non-existent Objects.* Yale University Press. 1980.

[97] Peckham, J. and Maryanski, F. "Semantic Data Models". *ACM Computing Surveys.* September, 1988.

[98] Quillian, M. "Semantic Memory". *Semantic Information Processing.* Minskey (Ed). MIT Press, 1968.

[99] Reiss, S. "An Object-Oriented Framework for Conceptual Programming". *Research Directions in Object-Oriented Programming.* Shriver, B. and Wegner, P. (Editors). MIT Press, 1987.

[100] Robson, D. and Goldberg, A. "The Smalltalk-80 System". *BYTE.* August, 1981.

[101] Rosenquist, C. "Entity Life Cycle Models and their Applicability to Information Systems Development Life Cycles". *The Computer Journal.* Vol 25, No3, 1983.

[102] Ross, R. *Entity Modelling: Techniques and Application.* Database Research Group, 1987.

[103] Royce, W. "Managing the Development of Large Software Systems: Concepts and Techniques." *Proceedings of the Ninth International Conference on Software Engineering.* IEEE Press, 1987.

[104] Rosenquist, C. "Entity Life Cycle Models and their Applicability to Information Systems Development Life Cycles". *The Computer Journal.* Vol 25, No 3, 1982.

[105] Rozenshtein, D. and Minsky, N. "A Law-Governed Object-Oriented System". *Journal of Object-Oriented Programming.* Vol 1, No 6, 1989.

[106] Rzevski, G. "On the Comparison of Design Methodologies". *Information Systems Design Methodologies: A Feature Analysis.* Olle, T., Sol, H., and Tully, C., Editors. North-Holland. 1983.

[107] Sciore, E. "Object Specialization". *ACM Transactions on Information Systems.* April, 1989.

[108] Shaw, M. "Abstraction Techniques in Modern Programming Languages". *IEEE Software.* October, 1984a.

[109] Shaw, M. "The Impact of Modelling and Abstraction Concerns on Modern Programming Languages". *On Conceptual Modelling.* Brodie, Mylopoulos & Schmidt (Ed), Springer-Verlag, 1984b.

[110] Shneiderman, B. "Direct Manipulation: A Step Beyond Programming Languages". *IEEE Computer.* August, 1983.

[111] Sibley, E. "The Evolution of Approaches to Information Systems Design Methodology". *Information Systems Design Methodologies: Improving the Practice.* Olle, T., Sol, H., & Verrjin-Stuart, A (Eds). Elsevier Science Publishers, 1986.

[112] Shlaer, S. and Mellor, S. *Object-Oriented Systems Analysis: Modelling the World in Data.* Yourdon Press, 1988.

[113] Smith, R., Barth, P. and Young, R. "A Substrate for Object-Oriented Interface Design". *Research Directions in Object-Oriented Programming.* Shriver, B. and Wegner, P. (Editors). MIT Press, 1987.

[114] Smith, J. and Smith, D. "Database Abstractions: Aggregation and Generalization". *ACM Transactions on Data Base.* June, 1977.

[115] Sol, H. "A Feature Analysis of Information Systems Design Methodologies: Methodological Considerations". *Information Systems Design Methodologies: A Feature Analysis.* Olle, T., Sol, H., and Tully, C., Editors. North-Holland. 1983.

[116] Stamper, R. "A Logic of Social Norms for the Semantics of Business Information". *Data Semantics (DS-1).* Steele, T. and Meersman, R. (Eds). North-Holland, 1986.

[117] Stamper, R. "Towards a Semantic Normal Form". *Database Architecture.* Bracchi and Nijssen (Eds). North-Holland, 1979.

[118] Stamper, R. "Aspects of Data Semantics: Names, Species and Complex Physical Objects". *Information Systems Methodologies.* Bracchi and Lockemann (Eds). Springer-Verlag, 1978.

[119] Stamper, R. "Physical Object, Human Discourse and Formal Systems". *Models in Database Management Systems.* Nijssen (Ed). North-Holland, 1977.

[120] Stefik, M. and Bobrow, D. "Object-Oriented Programming: Themes and Variations". *AI Magazine.* Winter, 1986.

[121] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley. 1986.

[122] Tarski, A. *Introduction to Logic.* Oxford University Press. 1941.

[123] Teorey, T., Yang, D. & Fry, J. "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model". *ACM Computing Surveys.* June, 1986.

[124] Tsichritzis, D. (Editor). *Office Automation.* Springer-Verlag, 1985.

[125] Tsichritzis, D. "Objectworld". *Office Automation.* Tsichritzis, D. (Editor) Springer-Verlag, 1985.

[126] Tsichritzis, D., Fiume, E., Gibbs, S. and Nierstrasz, O. "KNOs: KNowledge Acquisition, Dissemination, and Manipulation Objects". *ACM Transactions of Office Information Systems.* January, 1987.

[127] van Frassen, B. *The Scientific Image.* Clarendon Press, 1980.

[128] Verheijen, G. and van Bekkum, J. "NIAM: An Information Analysis Method". *Proceedings of the IFIP WG 8.1 Working Conference.* Olle, T et al. (Editors). North-Holland, 1982.

[129] Wand, Y. *An Ontological Foundation for Information Systems Design Theory.* Working Paper. Faculty of Commerce, University of British Columbia.

[130] Wand, Y. "A Proposal for a Formal Model of Objects". *Object-Oriented Concepts: Databases and Applications.* Kim, W. and Lochovsky, F., Editors. ACM Press, Addison-Wesley, 1989.

[131] Wand, Y. and Weber, R. *Formalization of Information Systems Design.* Working Paper. Faculty of Commerce, University of British Columbia. May, 1987.

[132] Wand, Y. and Weber, R. *An Ontological Analysis of Some Fundamental Information Systems Concepts.* Working Paper 88-MIS-002. Faculty of Commerce, University of British Columbia. March, 1988.

[133] Wand, Y. and Weber, R. *A Deep Structure Theory of Information Systems.* Working Paper 88-MIS-003. Faculty of Commerce, University of British Columbia. March, 1988.

[134] Warnier, J. *Logical Construction of Systems.* Van Nostrand Reinhold, 1981.

[135] Weber, R. *Life's Complexities: The Decomposition of Systems.* Working Paper. Faculty of Commerce, University of British Columbia. August, 1987.

[136] Wegner, P. "The Object-Oriented Classification Paradigm". *Research Directions in Object-Oriented Programming.* Shriver, B. and Wegner, P. (Editors). MIT Press, 1987.

[137] Winston, R. *Artificial Intelligence.* Addison-Wesley. 1984.

[138] Woo, C. *An Object-Oriented Model for Supporting Office Work.* Technical Report 88-MIS-025. MIS Division, Faculty of Commerce. University of British Columbia. 1988.

[139] Woodfield, S., Embley, D. and Scott, D. "Can Programmers Reuse Software?". *IEEE Software.* July, 1987.

[140] Young, J. and Kent, H. "Abstract Formulation of Data Processing Problems". *Journal of Industrial Engineering.* Nov.-Dec., 1958.

[141] Zachman, J. "A Framework for Information Systems Architecture". *IBM Systems Journal.* V. 26, N. 3, 1987.

[142] Zalta, E. *Abstract Objects.* Reidel, Dordrecht. 1983.

[143] Zilles, S. "Types, Algebras and Modelling". *On Conceptual Modelling.* Brodie, Mylopoulos & Schmidt (Ed), Springer-Verlag, 1984.

Appendix A

Bunge's Theorems

A.1 THEOREM 1.1

The association of any two individuals is the supremum (least upper bound or l.u.b.) for them with respect to the part-whole ordering:

$$\text{If } x, y \in S \text{ then } sup\{x, y\} = x \circ y.$$

Proof. By associativity and idempotence $x \circ (x \circ y) = (x \circ x) \circ y = x \circ y$. By Definition 1.2[1] the last formula is the same as: $x \sqsubset (x \circ y)$. Likewise $y \circ (x \circ y) = x \circ y$, whence $y \sqsubset (x \circ y)$. Hence $x \circ y$ is an upper bound of x and y. It is also their least upper bound. In fact call z an upper bound of x and y, ie. $x, y \sqsubset z$. Then $x \circ y \circ z = z$. Thus $x \circ y \sqsubset z$.

A.2 THEOREM 1.2

The world is the aggregation of all individuals:

$$\diamond = [S] = sup\ S.$$

Proof. By Postulate 1.2,[2] $\diamond$ exists and is the last individual, ie. for every $x \in S$, $x \sqsubset \diamond$. But this individual fits the conditions of Definition 1.7,[3] ie. $\diamond = sup\ S$.

A.3 THEOREM 1.3

The ordered quadruple $< S, \circ, \square, \diamond >$ is a sup-semilattice with least element $\square$ and last element $\diamond$ with respect to the part-whole relation $\sqsubset$.

Proof. By Theorem 1.1 (above) there exists a supremum for any two individuals, namely their association.

---

[1] DEFINITION 1.2. If x and y are substantial individuals, then x is a *part* of y iff $x \circ y = y$. Symbol: $x \sqsubset y$.

[2] POSTULATE 1.2. There exists an individual such that every other individual is part of it. Ie. $(\exists x)[x \in S \& (y)(y \in S \Rightarrow y \sqsubset x)]$.

[3] DEFINITION 1.7. Let $T \subseteq S$ be a set of substantial individuals. Then the *aggregation* or *association* of T, or $[T]$ for short, is the supremum of T. Ie., $[T] = sup\ T$.

Besides, $\Box$ is part of every individual, so it lies at the bottom of the net. Dually, $\Diamond$ contains every individual, so it perches on top. These conditions, together with the commutativity of association, define a sup-semilattice.

Appendix B

Smalltalk OBIS Prototype Shell

B.1 OBCM/IS Primitives - OBIS

(Object) subclass: #OBIS
instanceVariableNames: surrogate
stateFunctions
history
pOrder
classVariableNames: SystemObject
poolDictionaries:

OBIS class methods

OBIS methods

asSet
"Answers a set with self"
↑ Set with:self.

change:sf **to:**value
"Generic method to change value of :sf
to :value"
|result oldValue|

CursorManager execute change.
sf = #fAComp ifFalse:[
oldValue := stateFunctions at:sf ifAbsent:[nil].
stateFunctions at:sf asSymbol put:value].
(result := self validateLaws)
ifTrue:[self cHistory:sf to:value]
ifFalse:[
sf = #fAComp ifTrue:[self surrogateRemove:value]
ifFalse:[stateFunctions at:sf put:oldValue].

```
    self validateLaws ifFalse:[Menu message:'System Error'.
                               self inspect]].
self sfUpdate.
CursorManager normal change

changeVar:sf to:value
"Generic method to change value of :sf
        to :value.  No validation performed."

|result oldValue|

CursorManager execute change.
oldValue := stateFunctions at:sf ifAbsent:[nil].
stateFunctions at:sf asSymbol put:value.
self cHistory:sf to:value.
CursorManager normal change

checkSurrogate:anObject
   "Find all other objects with same surrogate as :anObject."
 | surrogates |

↑ SystemObject fComp select:[:each|
    each surrogate= self surrogate]

checkSurrogate:anObject with:surrogate
"Verifies :anObject has surrogate :surrogate"
 |surrogateLattice|

anObject surrogate isNil
  ifTrue:[surrogateLattice := (anObject lattice collect:[:each |
            each surrogate]) asSet]
  ifFalse:[surrogateLattice := Set with:anObject surrogate].
surrogate do: [:each |
    surrogateLattice remove:each ifAbsent:[↑ false]].
surrogateLattice size = 0
   ifTrue:[↑ true]
   ifFalse:[ ↑ false].

cHistory:sfName to:sfValue
"Update History"

history addLast:(Association
```

```
    key:Date today asSeconds + Time now asSeconds
    value:(Array with:sfName with:sfValue)).

composition
"Primitive function.
Answer all objects in composition of current object"
|objects|

objects := Set new.
(self surrogate composition) do:[:each |
    objects addAll:each objects].
↑ objects

debug
"Systems testing method"

   self inspect

initialize
"Initialize all base state functions"

self class = System ifTrue:[SystemObject := self].
history:= OrderedCollection new.
stateFunctions:= Dictionary new.
stateFunctions at:#fKinds put:(Set with:self class);
            at:#fComp put:self fComp;
            at:#fAComp put:self fAComp;
            at:#fPartOf put:self fPartOf.
pOrder := Set new.
↑ nil.

isChange:aChange
 "Verifies if change method"

    (aChange printString size >= 2 and: [
    (aChange printString at: 1) == $c and: [
    (aChange printString at: 2) isUpperCase ]])
          ifTrue:[↑ true]
          ifFalse:[↑ false].

isFunction:aFunction
"Verifies if State Function method"
```

```
    (aFunction printString size >= 2 and: [
    (aFunction printString at: 1) == $f and: [
    (aFunction printString at: 2) isUpperCase ]])
          ifTrue:[↑ true]
          ifFalse:[↑ false].

isLaw:aLaw
"Verifies if Law method"

    (aLaw printString size >= 2 and: [
    (aLaw printString at: 1) == $l and: [
    (aLaw printString at: 2) isUpperCase ]])
          ifTrue:[↑ true]
          ifFalse:[↑ false].

isView:aMethod

    (aMethod  printString size >= 2 and: [
    (aMethod  printString at: 1) == $v and: [
    (aMethod  printString at: 2) isUpperCase ]])
          ifTrue:[↑ true]
          ifFalse:[↑ false].

lattice:aComp
    "Primitive method.
     Answers the recursive composition of :aComp"

|collection lattice aKinds comp|

aKinds := self fKinds.
collection := OrderedCollection with:self.
comp := aComp select:[:each | aKinds includes:each class].
comp remove:self.  comp size = 0 ifTrue:[↑ collection asSet].
comp do:[:each|
    lattice := each composition.
    collection addAll:(each lattice:lattice)].
↑ collection asSet

objectName
"Answer the objectName and
      its Model Object name"
```

```
|name|

(self respondsTo:#fName)
  ifTrue:[name := self fName]
  ifFalse:[name := stateFunctions at:#fName ifAbsent:['object']].
↑ name,'<',self printString,'>'.

objectOfKind:k in:objectSet
  "Returns first object ofKind:k in :objectSet"
|aSet|

aSet := (self perform:objectSet asSymbol) select:[:each |
        each class = k].
aSet size = 0
  ifTrue:[↑ nil]
  ifFalse:[↑ (aSet asArray) at:1]

objectsOfKind:k
  "Returns objects ofKind:k in System"

↑ (SystemObject fComp) select:[:each |
        each class = k].

objectsOfKind:k in:objectSet
  "Returns objects ofKind:k in :objectSet"

↑ (self perform:objectSet asSymbol) select:[:each |
        each class = k].

objectsOfKind:k inSet:objectSet
  "Returns objects ofKind:k in :objectSet"

↑ objectSet select:[:each |
        each class = k].

partOf:instances
   "Primitive.
    Answers all part-of"
|collection|

instances size = 0 ifTrue:[↑ Set new].
collection := OrderedCollection new.
```

```
instances do:[:each|
    (each fComp includes:self)
        ifTrue:[collection add:each]].
↑ collection asSet

same:v1 as:v2
"Primitive.
  Answer true if v1 same as v2"

v1 = v2
  ifTrue:[↑ true].
((v1 isKindOf:Collection) and:[
 v2 isKindOf:Collection])
   ifFalse:[↑ false].
v1 do:[:each | (v2 includes:each) ifFalse:[↑ false]].
v2 do:[:each | (v1 includes:each) ifFalse:[↑ false]].
↑ true

sfUpdate
"Primitive.
  Updates stateFunctions and History"
|sFunctions baseFunctions |

baseFunctions := ModelObj selectors select:[:aSelector|
  self isFunction:aSelector].
SystemObject fComp do:[:each |
  sFunctions := each class selectors select:[:aSelector |
                        self isFunction:aSelector].
  sFunctions addAll:baseFunctions.
  sFunctions do:[:sf | each sfvChange:sf]].

sfvChange:function
"Primitive.
   Updates stateFunctions and History"
|oldValue newValue |

newValue:=self perform:function asSymbol.
function asSymbol = #fHistory
  ifTrue:[↑ newValue].
oldValue := stateFunctions at:function asSymbol ifAbsent:[nil].
(self same:oldValue as:newValue)
  ifFalse:[self cHistory:function to:newValue.
```

```
        stateFunctions at:function asSymbol put:newValue].
↑ newValue

sumValuesOf:state fromObjectsOfKind:k in:objectSet
  "Answers the sum of values of :state from objects of kind :k
   in :objectSet"
|sum|

sum := 0.
(self valuesOf:state fromObjectsOfKind:k in:objectSet) do:[:each |
  sum := sum + each].
↑ sum

surrogate
"Primitive.
  Answers surrogate"

↑ surrogate

surrogateAdd:anObject
  "Primitive.
    Add :anObject to composition of current surrogate."

self surrogate compositionAdd:anObject surrogate.
self surrogate objectsAdd:anObject

surrogateRemove:anObject
  "Primitive.
   Remove :anObject from composition of current surrogate."

self surrogate compositionRemove:anObject surrogate.
self surrogate objectsRemove:anObject

surrogateSet
 "Primitive.
    Assign surrogate to current object and check consistency"
      |object|

(Menu oneOf:#('New Surrogate?' 'Same as Existing Surrogate?') and:#(1 2)) = 1
 ifTrue:[
   surrogate := Surrogate new:self]
 ifFalse:[
```

```
    (object := MoMenu selectFrom:SystemObject fComp) isNil
      ifTrue:[↑ nil]
      ifFalse:[surrogate := object surrogate.
            (object surrogate) objectsAdd:self]].

valuesOf:state fromObjectsOfKind:k in:objectSet
  "Answers vector of values of :state from objects of kind :k
   in :objectSet"
|return|

return := OrderedCollection new.
(self objectsOfKind:k in:objectSet) do:[:each |
    return add:(each perform:state asSymbol)].
↑ return

view:anObject as:kind
"Answers object of :kind with same surrogate
  as :anObject"
|set|

set := (self checkSurrogate:anObject) select:[:object |
      object class = kind].
(set size = 0)
   ifTrue:[↑ nil].
(set size = 1)
   ifTrue:[↑ set asArray at:1].
```

B.2 Windowing and Display Management - OBISForm

```
(OBIS) subclass:              #OBISForm
instanceVariableNames:        graphForm
                              graphRect
classVariableNames:           ActiveList
poolDictionaries:

OBISForm class methods

OBISForm methods


changeList
"List all change methods"
```

```
|changeList|

changeList := (self class selectors select:[:each | self isChange:each])
      asArray.
changeList := changeList reject:[:each | (pOrder includes:each)].
↑ changeList asSortedCollection

display:buffer
 "Primitive.
  Displays the buffer in Graph Pane"
    |textForm aStream y|

textForm := CharacterScanner new initialize:Display boundingBox
      font:Font eightLine
      dest:graphForm.
textForm blankRestFrom:0.
self changed:#displayGraph:.
aStream := ReadStream on:buffer withCrs.
y := 0.
 [aStream atEnd] whileFalse: [
   textForm display:aStream nextLine at:0@y.
   y := y+8].
self changed:#displayGraph:.

displayGraph:aRect
"Primitive.
  Sets the display Form"

graphRect := aRect.
graphForm displayAt: aRect origin clippingBox:aRect.
↑ graphForm

doNothing
"Do nothing"
↑ nil

doNothing:parm
"Answer nil"
↑ nil

graphRect
"Primitive.
```

```
    Answer GraphPane rectangle"

↑ graphRect

listMenu
"Primitive.  Service Menu"

↑ Menu
  labels:'ofKind\compOf\partOf\stateValue' withCrs
  lines:#()
  selectors:#(mKinds\mComp\mPart\mState).

listObjects
"Primitive.
  Answer collection of objects"

↑ ActiveList asOrderedCollection collect:[:eachObject |
    eachObject objectName].

listSelect:selection
"Primitive.  Control window."

Scheduler topDispatcher closeWindow.
Scheduler systemDispatcher redraw.
(Scheduler dispatchers select:[:each | each pane model isKindOf:OBIS] )
   do:[:each | each closeWindow].  "kill all object panes"
Scheduler systemDispatcher redraw.
(ActiveList asOrderedCollection at:selection) openOn.
Scheduler resume.

objectMenu
    "Primitive.
      Bring up the Service Menu "


↑ Menu
 labels:'States\Changes\Laws\fComp\fAComp\fPartOf\fSurrogates\Scan \A-Comp\Scan\
   Part-Of\debug' withCrs
 lines:#(3 7)
 selectors:#(sStates\sChanges\sLaws\scanfC\scanfAC\ scanfP\scanSurrogates\mAComp\mPartOf\de

open
```

```
  | index aDispatcher|
"Primitive.  Open window"

(Scheduler topDispatcher pane model isKindOf:OBIS) and:[
  (Scheduler topDispatcher pane model) ≐ SystemObject
    ifTrue:[Scheduler topDispatcher closeWindow]].
  index := (Scheduler dispatchers collect:[:each | each pane model] )
     indexOf:self ifAbsent:[↑ self openOn].
  aDispatcher := Scheduler dispatchers at:index.
  Scheduler remove:aDispatcher;
          add: aDispatcher.
  (aDispatcher pane collapsed)
      ifTrue:[aDispatcher collapse].
   aDispatcher openWindow.

openBack
"Primitive.
  Re-open window"

(Scheduler topDispatcher pane model) = self
  ifFalse:[Scheduler topDispatcher closeWindow.
          Scheduler systemDispatcher redraw.
          self open. Scheduler resume].
↑ nil

openOn
  "Primitive.
   Create a browser window for ModelObjects"
   |aTopPane pixelHeight|

   pixelHeight := ListFont height + 4.
   aTopPane := TopPane new
      model: self;
      label: self objectName;
      minimumSize: 20 * SysFontWidth
          @ (10 * SysFontHeight);
      rightIcons: #( resize collapse);
      yourself.

   graphForm := Form width:Display width height:Display height.
   graphForm white.
   self fGForm isNil
```

```
        ifFalse:[graphForm copy:self fGForm boundingBox
                        from:self fGForm to:0@0 rule:Form over].
    aTopPane addSubpane:
        (GraphPane new
            model: self;
            menu: #objectMenu;
            change:#test:;
            name: #displayGraph:;
            framingRatio: (0 @ 0 extent: 1 @ 1)).
    self fForm isNil
      ifTrue:[ Scheduler add:(aTopPane dispatcher open)]
      ifFalse:[Scheduler add:(aTopPane dispatcher
                openIn:((self fForm at:1)  extent:
                    (self fForm at:2)))].
      aTopPane displayWindow. aTopPane dispatcher openWindow.

openOnList
    "Primitive.
     Create a List window for Objects"
    |aTopPane|

Scheduler topDispatcher closeWindow.

    aTopPane := TopPane new
        model: self;
        label: 'List Objects';
        minimumSize: 20 * SysFontWidth
            @ (10 * SysFontHeight);
        rightIcons: #( resize collapse);
        yourself.
    aTopPane addSubpane:
        (ListPane new
            model: self;
            menu: #listMenu;
            name: #listObjects;
            change:#listSelect:;
            returnIndex:true;
            framingRatio: (0 @ 0 extent: 1 @ 1)).
     Scheduler add:(aTopPane dispatcher open).
     Scheduler resume.

printObjects:objects
```

```
    "Primitive.
      Return string of object names in
      :objects"
    |collection|

collection := OrderedCollection new.
objects do:[:each | collection add:each objectName].
↑ collection

sChanges
"Primitive.
  Select a change function"
|changes selection|

changes := self changeList asArray.
changes size < 20
  ifTrue:[selection := Menu oneOf:changes]
  ifFalse:[selection := Menu oneOfMany:changes].
selection isNil
  ifTrue:[↑ nil].
(self isChange:selection)
    ifFalse:[Terminal bell. ↑ nil].
(pOrder includes:selection asSymbol)
    ifTrue:[Terminal bell. ↑ nil].
↑ self perform:selection asSymbol.

sLaws
"Primitive.
  Display Law Statement"

LogicBrowser new openOn:(Array with:
    (Smalltalk at:(self class printString,'Laws') asSymbol))

sStates
"Primitive.
  Display State Functions"
|states selection textBuffer|

states := self stateList asArray.
states size < 20
  ifTrue:[selection := Menu oneOf:states]
  ifFalse:[selection := Menu oneOfMany:states].
```

```
selection isNil
  ifTrue:[↑ nil].
(textBuffer:=self perform:selection asSymbol) isNil
  ifTrue:[↑ nil].
(textBuffer isMemberOf:String)
    ifFalse:[textBuffer := textBuffer printString].
SysFont := Font eightLine.
Menu oneOf:(self sStatesString:textBuffer).
SysFont := Font fourteenLine.
self display:textBuffer.

sStatesString:aString
      "Primitive.
       Answer an array of substrings from the
       receiver. The receiver is divided into
       substrings at the occurrences of $
which
       is converted to space character"
   | aStream answer index a|
   answer := OrderedCollection new.
   aStream := ReadStream on: aString.
   index := 1.
   [aStream atEnd]
      whileFalse: [
         (aStream skipTo: $ ) ifTrue:[
           answer add:
            (aString copyFrom: index to: (aStream position - 1)).
           index := aStream position+1]].
    answer add:
       (aString copyFrom: index to: (aStream position)).
   ↑ answer asArray

stateList
"Primitive.
  Answer list of state functions"

↑ ((self class selectors asArray,ModelObj selectors asArray)
        select:[:each | self isFunction:each]) asSortedCollection.

test:something
"Primitive. System testing function"
```

```
CursorManager crossHair change.
 [Terminal read == (FunctionKeys at:'EndSelectFunction')]
    whileFalse: [].
CursorManager normal change.

view:buffer
    "Primitive.
     Displays the buffer in Graph Pane"
     |textForm aStream y|

textForm := CharacterScanner new initialize:Display boundingBox
      font:Font eightLine
      dest:graphForm.
textForm blankRestFrom:0.
self changed:#displayGraph:.
aStream := ReadStream on:buffer withCrs.
y := 0.
 [aStream atEnd] whileFalse: [
   textForm display:aStream nextLine at:0@y.
   y := y+8].
self changed:#displayGraph:.
```

B.3 User Interface - OBISInterface

(OBISForm) subclass: #OBISInterface
instanceVariableNames:
classVariableNames:
poolDictionaries:

OBISInterface class methods

OBISInterface methods

mAComp
"Manipulate A-composition of current object"

```
ActiveList := self fAComp.
self openOnList
```

mComp
"Manipulate composition of current object"

```
|newList|

newList := OrderedCollection new.
ActiveList do: [:object |  newList addAll:object fAComp].
ActiveList := newList asSet.
self changed:#listObjects.
```

mKinds

```
"Scope by kinds"
|index selection|

index := Dictionary new.
ActiveList do: [:object |  index at:object class symbol put:object class].
(selection := Menu oneOfMany:index keys asOrderedCollection) isNil
   ifTrue:[↑ nil].
selection := index at:selection.
ActiveList := ActiveList select:[:object | object isMemberOf:selection].
self changed:#listObjects.
```

mPart

```
"Answer part-of"
|newList|

newList := OrderedCollection new.
(ActiveList size = 0)
  ifTrue:[ActiveList := SystemObject composition].
ActiveList do: [:object | newList addAll:object fPartOf].
ActiveList := newList asSet.
self changed:#listObjects.
```

mPartOf

```
   "Manipulate objects part-of current object"

ActiveList := self fPartOf.
self openOnList
```

mState

```
"Answer state functions"
|index states selection value|

(ActiveList collect: [:object |  object class]) asSet size = 1
   ifFalse:[↑ nil].
```

```
states :=
   ActiveList asArray first class selectors select:[:each | self isFunction:each].
(selection := Menu oneOf:states asOrderedCollection) isNil
   ifTrue:[↑ nil].
value := Prompter prompt:'Value?' default:''.
ActiveList :=
 ActiveList select:[:object | (object perform:selection) = value].
self changed:#listObjects.

remove
   "Removes an object from composition"
   |selection|

selection := MoMenu selectOne:self fComp.
(selection isNil or:[selection = self])
  ifTrue:[↑ nil].
self surrogateRemove:selection

scan:objects
   "Primitive.
      Scans objects"
   |selection|

objects size = 0
  ifTrue:[↑ nil].
selection := MoMenu selectOne:objects.
(selection isNil or:[selection = self])
  ifTrue:[↑ nil].
selection open.
Scheduler resume.

scanfAC
   "Scanner for A-composition of current object"

self scan:self fAComp

scanfC
   "Scanner for composition of current object"

self scan:self fComp

scanfP
```

```
    "Scanner for objects part-of current object"

self scan:self fPartOf

scanSurrogates
  "Scanner for surrogates of current object"

self scan:(self checkSurrogate:self)

validateLaws
    "Primitive.
    Validate laws of all related objects"
    |objects|

(SystemObject fComp select:[:each | each respondsTo:#lLaws])
    do:[:anObject | (anObject perform:#lLaws) ifFalse:[
                Menu message:('Law! ',anObject printString).
                ↑ false]].
↑ true
```

B.4 MOC Subsystem - ModelObject

```
(OBISInterface) subclass:     #ModelObj
instanceVariableNames:
classVariableNames:
poolDictionaries:

ModelObj class methods


new
|object|
"Create and initialize object."

object := super new.
object surrogateSet;
      initialize;
      cInit.
↑ object

ModelObj methods
```

```
addComp:kind
    "Generic method for adding object of kind to self."
  |object composition|

(Scheduler topDispatcher pane model) = self
  ifFalse:[self open].
(Menu confirm:'Adding ',kind printString)
  ifFalse:[↑ nil].
(Menu oneOf:#('New?' 'Existing?') and:#(1 2)) = 1
 ifTrue:[
   object := kind new]
 ifFalse:[
   object := MoMenu selectFrom:kind allInstancesMO].
object isNil
 ifTrue:[Terminal bell.
        self openBack. ↑ nil].
(Menu confirm: 'include ',object objectName,'?') = true
 ifFalse:[object become:String new.
         Terminal bell.
         ↑ self openBack].
self surrogateAdd: object.
(Scheduler topDispatcher pane model) = self
  ifFalse:[Scheduler topDispatcher closeWindow.
          Scheduler systemDispatcher redraw.
          self open].
↑ object

fAComp
  "Answers the A-composition of current object"

|aKinds a|

aKinds := self fKinds.
↑ self composition select:[:each |
    aKinds includes:each class ].

fComp
"Answer all objects in composition of current object"

↑ self lattice:(self composition).
```

fForm
"Answer fForm"

```
↑ stateFunctions at:#fForm ifAbsent:[↑ nil].
```

fGForm
"Answer fGForm"

```
↑ stateFunctions at:#fGForm ifAbsent:[↑ nil].
```

fHistory
"Answer fHistory"

```
|string|

string := String new.
history do:[:aHistory |
  string := string,
        (Date fromDays:(aHistory key)//86400) formPrint,' ',
        (Time fromSeconds:aHistory key) printString,' ',
        (aHistory value) printString,'].
↑ string
```

fKinds
*"Answer the Kinds allowed in
the composition of the current object."*

```
↑ stateFunctions at:#fKinds ifAbsent:[↑ self class].
```

fLaws
"Primitive. System testing"

```
CursorManager execute change.
(self respondsTo:#lLaws )
  ifTrue:[self lLaws].
CursorManager normal change.
↑ 'Done'
```

fName
"Answers name"

```
↑ stateFunctions at:#fName ifAbsent:['object'].
```

fPartOf
"Answers all objects partOf current object"

↑ self partOf:(SystemObject fComp)

fSizeAComp
"Answers size of A-composition"

↑ self fAComp size

fSizeComp
"Answers size of composition"

↑ self fComp size

Appendix C

Order Entry Example - OBIS Implementation

C.1 Customers

(ModelObj) subclass: #Customer

State Function Methods

fCreditLimit
"Answer credit limit"

```
↑ stateFunctions at:#fCreditLimit ifAbsent:[↑ 0].
```

fDebt
"Answers fDebt defined as law lDebt"

```
|debt |

(CustomerLaws new :? lDebt(x,self)) do: [:each |
    debt := each at:1].
↑ debt.
```

fName
"Answer customer name define as law lName."

```
|result name|

(result := CustomerLaws new :? lName(x,self)) notNil
    ifTrue:[result do:[:each | name := each at:1].
            ↑ name]
    ifFalse:[↑ stateFunctions at:#fName ifAbsent:[↑ nil]].
```

Change Function Methods

cCreditLimit
"Change fCreditLimit"

```
|creditLimit|
```

```
creditLimit := self fCreditLimit.
creditLimit := (Prompter prompt: 'Credit Limit?'
                    default: (creditLimit printPaddedTo:1)) asInteger.
self change:#fCreditLimit to:creditLimit
```

cGForm
"Change fGForm"

```
self changeVar:#fGForm to:('Customer' magnifyBy:2@6).
pOrder add:#cGForm
```

cInit
"Initialize new object."

```
self cName;
    cGForm;
    cCreditLimit.
pOrder add:#cInit.
```

cName
"Change the customer name."

```
|name oldName|

oldName := self fName.
(name := CustomerLaws new :? lName(x,self)) notNil
    ifTrue:[name do:[:each | name := each at:1]].
name isNil ifTrue:[name := (Prompter prompt: 'Name?'
                    default: oldName)].
self change:#fName to:name.
```

Law Statements

lLaws
"Validate all laws"

```
    |fDebt|

"Validate lCreditLimit - Credit Limit Law "
fDebt := self fDebt. fCreditLimit := self fCreditLimit.
(CustomerLaws new :? lCreditLimit(fDebt,fCreditLimit)) notNil
   ifFalse:[↑ false].
```

```
"Validate lDebt - Debt Law "
(CustomerLaws new :? lDebt(fDebt,self)) notNil
   ifFalse:[↑ false].

"Validate lComp - Composition Law"
(CustomerLaws new :? lComp(self)) notNil
   ifFalse:[↑ false].

↑ true
```

(Prolog) subclass: #CustomerLaws

```
"Composition Law - simple object"
lComp(aCustomer) :-
   is(cardinality,
      aCustomer ctObjectsIn:#composition),
   eq(cardinality,1).

"Debt is <= Credit Limit"
lCreditLimit(fDebt, fCreditLimit) :-
   le(fDebt,fCreditLimit).

"Customer debt = sum of order values"
lDebt(debt, aCustomer) :-
  is(orderValues,
     aCustomer sumValuesOf:#fValue
               fromObjectsOfKind:Order
               in:#fPartOf),
  eq(debt,orderValues).

"Check for Person with same Surrogate"
lName(name,aCustomer) :-
   consult(name(x,aCustomer),PersonLaws new),
   eq(name,x).
```

C.2 Products

(ModelObj) subclass: #Product

State Function Methods

fValue
"Answer value"

```
↑ stateFunctions at:#fValue ifAbsent:[↑ 0].
```

Change Function Methods

cGForm
"Change fGForm"

```
self changeVar:#fGForm to:('Product' magnifyBy:6@10).
pOrder add:#cGForm
```

cInit
"Initialize new object."

```
self cName;
    cGForm;
    cValue.
pOrder add:#cInit.
```

cName
"Change name"

```
self changeVar:#fName
    to:(Prompter prompt:'Product Name?'
        default:'').
```

cValue
"Change value from external source"

```
|oldValue value|

oldValue := self fValue.
value := (Prompter prompt: 'Value?'
            default: (oldValue printPaddedTo:1)) asInteger.
self change:#fValue to:value.
```

Law Statements

lLaws
"Validate laws"

```
"Order Law: Product can only be partOf one Order"
(ProductLaws new :? lPartOf(self)) notNil
   ifFalse:[↑ false].

"Composition Law"
(ProductLaws new :? lComp(self)) notNil
   ifFalse:[↑ false].

↑ true
```

(Prolog) subclass: #ProductLaws

```
"Composition Law - simple object"
lComp(aProduct) :-
   eq(cardinality,1),
   is(cardinality, aProduct composition size).

"Product can only be part-of one Order"

lPartOf(aProduct) :-
  is(orders,
     (aProduct objectsOfKind:Order
           in:#fPartOf)size),
  le(orders,1).
```

C.3 Orders

(ModelObj) subclass: #Order

State Function Methods

fValue

```
   "Answer fValue defined as law lValue"
   |value|

(OrderLaws new :? lValue(x,self)) do: [:each |
    value := each at:1].
↑ value
```

Change Function Methods

cACompCustomer

```
    "Add a customer to Order."
```

```
self change:#fAComp to:(self addComp:Customer).
pOrder add:#cACompCustomer.
```

cACompProduct

```
    "Add a product to Order."
|continue|

continue := true.
 [continue] whileTrue:[
 self change:#fAComp to:(self addComp:Product).
 continue := Menu confirm:'Add another product?']
```

cInit

```
      "Initialize new object name."

self cName;
    cGForm;
    cKinds;
    cACompCustomer;
    cACompProduct.
pOrder add:#cInit.
```

cKinds

"Initialize fKinds to Customer, Product"

```
self change:#fKinds to:(Set with:self class
                        with:Customer
                        with:Product).
pOrder add:#cKinds
```

cName

"Change name from external source"

```
self changeVar:#fName to:(Prompter prompt:'Order No?'
                  default:'').
```

Law Statements

lLaws

"Validate laws"

```
|fValue|
```

```
"Validate lComp - cardinality law"
(OrderLaws new :? lComp(self)) notNil
   ifFalse:[↑ false].

"Validate lValue - value law"
fValue := self fValue.
(OrderLaws new :? lValue(fValue,self)) notNil
   ifFalse:[↑ false].

↑ true
```

(Prolog) subclass: #OrderLaws

```
"Cardinality Law - each order has one customer and
 0 or more products"
lComp(anOrder) :-
   is(crdCustomer,
      (anOrder objectsOfKind:Customer
            in:#fAComp) size),
   is(crdProduct,
      (anOrder objectsOfKind:Product
            in:#fAComp) size),
   eq(crdCustomer,1),
   ge(crdProduct,0).

"Order Value Law - Order value = sum of values of
 products in its composition"
lValue(orderValue,anOrder) :-
   eq(orderValue,productValues),
   is(productValues,
      anOrder sumValuesOf:#fValue
            fromObjectsOfKind:Product
            in:#fAComp).
```

C.4 Order Entry System

(ModelObj) subclass: #OrderEntry

Change Function Methods

cACompCustomer

"Add a customer to OrderEntry."

↑ self change:#fAComp to:(self addComp:Customer).

cACompOrder
"Add a Order to Order Entry."

↑ self change:#fAComp to:(self addComp:Order).

cACompProduct
"Add a Product to OrderEntry."

↑ self change:#fAComp to:(self addComp:Product).

cForm
"Change the fForm parameters"

self change:#fForm to:(OrderedCollection with:0@0
with: Display width * 3//4 @ (Display height * 3//4)).

cGForm
"Change **the cGForm parameters"**

self changeVar:#fGForm to:('O/E' magnifyBy:6@10).
pOrder add:#cGForm

cInit
"Initialize"

self cKinds;
cGForm;
cForm.
pOrder add:#cInit.
↑ nil

cKinds
"Kinds are Customer, Order, Product"

self change:#fKinds to:(Set with:self class
with:Customer
with:Order

```
with:Product).
```

Appendix D

The IFIP Working Conference Problem

This appendix details the OBCM specifications for the IFIP Working Conference problem. Portions of a current implementation of this case study using the prototype OBCM/IS shell is found in Appendix E.

D.1 IFIP Working Conference Problem

The IFIP Working Conference Problem is presented in its entirety below. The problem statement is relatively brief but it has been observed that it supports a wide range of interpretations and perceptions [Essink, 1986; Flint, 1986].

D.1.1 Background

"An IFIP Working Conference is an international conference intended to bring together experts from all IFIP countries to discuss some technical topic of specific interest to one or more IFIP Working Groups. The usual procedure, and that to be considered for the present purposes, is an invited conference which is not open to everyone. For such conferences it is something of a problem to ensure that members of the involved IFIP Working Group(s) and Technical Committee(s) are invited even if they do not come. Furthermore, it is important to ensure that sufficient people attend the conference so that the financial break-even point is reached without exceeding the maximum dictated by the facilities available.

IFIP Policy on Working Conferences suggest the appointment of a Program Committee to deal with the technical contents of the conference and an Organizing Committee to handle financial matters, local arrangement, and invitations and/or publicity. These committees clearly need to work together closely and have a need for common information and to keep their recorded information consistent and up to date.

D.1.2 Information System to be Designed

The information system which is to be designed is that necessary to support the activities of both a Programme Committee and an Organising Committee involved in arranging an IFIP Working Conference. The involvement of the two committees is seen as analogous to two organizational entities within a corporate structure using some common information. The following activities of the committee should be supported.

Programme Committee

1. Preparing a list to whom the call for papers is to be sent.
2. Registering the letters of intent received in response to the call.
3. Registering the contributed papers on receipt.
4. Distributing the papers among those undertaking the refereeing.
5. Collecting the referees' reports and selecting the papers for inclusion in the programme.
6. Grouping selected papers into sessions for presentation and selecting chairman for each session.

Organizing Committee

1. Preparing a list of people to invite to the conference.
2. Issuing priority invitations to National Representatives, Working Group members and members of associated working groups.
3. Ensuring all authors of each selected paper receive an invitation.
4. Ensuring authors of rejected papers receive an invitation.
5. Avoiding sending duplicate invitations to any individual.
6. Registering acceptance of invitations.
7. Generating final list of attendees.

D.1.3 Boundaries of System

It should be noted that budgeting and financial aspects of the Organizing Committees's work, meeting plans of both committees, hotel accommodation for attendees and the matter of preparing camera ready copy of the proceedings have been omitted from this exercise, although a submission may include some or all of these extra aspects if the authors feel so motivated."

D.2 OBCM Interpretation of the IFIP Case

The IFIP case study is described as an OBCM model in the following terms.

- Things which can be represented in OBCM by *simple* objects. These are described by the following model objects.

  PERSON
  AUTHOR
  ATTENDEE
  REFEREE
  TEXT
  FACILITY

- Things which can be represented as compositions of the simple objects implied above. These can be described by the following models.

  INVITATIONLETTER
  INTENTLETTER
  PAPER
  REFEREEREPORT.

- Further components of the IFIP case viewed as aggregations of objects as described above. A Programme Committee, for example, is an aggregation of persons, invitation and intent letters, papers, referee reports and so forth. A Working Conference Session is organized around a number of selected papers and a facility (or facilities) in which to meet. A Working Conference is a

collection of sessions. The IFIP system itself is an aggregation of all the objects of interest in the system. These components can be described by the models

PROGRAMME COMMITTEE

ORGANIZING COMMITTEE

SESSION

WORKING CONFERENCE

IFIP.

Each of the above models are specified in detail below.

D.3 Person

The model object PERSON describes all human beings (or individuals) of interest to the IFIP problem. A collection of surrogates S_P is assumed, each of which corresponds to some person of interest.

State Functions

In addition to the standard state functions common to all OBCM objects[1] a useful property is the person's identity. This is represented by the conventional attribute

$$\mathsf{fName} : S_P \rightarrow \mathbf{N}_S,$$

where $\mathbf{N}_S$ is the set of names.

Law Statements

Objects of type PERSON are simple. The law statement lComp enforces this idea

$$\begin{aligned} &\mathsf{lComp}: \\ &\quad \mathsf{fComp} = \mathsf{self}. \end{aligned}$$

[1]Recall from Chapter 5: fComp, fAComp, fKinds, fPartOf, fForm and fHistory.

Change Functions

The relevant change functions are as follows. First, to set the value of fForm,

cForm :

fForm = *external source.*

The person's name is set as

cName :

fName = *external source.*

The following change function initializes the value of fKinds:

cKinds :

fKinds := {PERSON}.

The compound[2] change function cInit initializes new objects as follows:

cInit :

cForm * cKinds.

As expected, cInit can be invoked only once.

The partial ordering of these change functions can be depicted as

cInit $\prec$ (cForm, cName).

D.4 Author

An author is an individual who is involved in producing a paper. The model object AUTHOR is a view of persons. That is, the surrogates for AUTHOR is a subset $S_{Pa} \subseteq S_P$.

State Functions

The identity of an author is represented by the state function fName

fName : $S_{Pa} \rightarrow \mathbf{N}_S$.

[2]Recall from Chapter 5 the notion of *composite change.*

Law Statements

Objects of type AUTHOR are simple. The law statements lComp and lComp enforce this idea

$$\begin{aligned} &\mathsf{lComp}: \\ &\quad \mathsf{fComp} = \mathsf{self}. \end{aligned}$$

The law of *strict generalization* is applied to objects of type AUTHOR. That is, authors are *specializations* of persons.[3] In Chapter 4, this law was given as

$$\begin{aligned} &\mathsf{lIsA}: \\ &\quad \mathsf{qSize}[O_{S_g}(\mathsf{self})/\sim_{\text{PERSON}}] = 1. \end{aligned}$$

In addition, the state function fName must take its value from the PERSON object with which the AUTHOR object shares its surrogate, ie.

$$\begin{aligned} &\mathsf{lName}: \\ &\quad \mathsf{fName} = [O_{S_g}(\mathsf{self})/\sim_{\text{PERSON}} .\mathsf{fName}]. \end{aligned}$$

Change Functions

The relevant change functions include cKinds:

$$\begin{aligned} &\mathsf{cKinds}: \\ &\quad \mathsf{fKinds} := \{\text{AUTHOR}\}. \end{aligned}$$

Change function cInit is described as[4]

$$\begin{aligned} &\mathsf{cInit}: \\ &\quad \mathsf{cForm} * \mathsf{cKinds}. \end{aligned}$$

Note that since fName depends upon the person sharing the same surrogate as the author, there is no change function to change the author's name.

The partial ordering for change functions is

$$\mathsf{cInit} \prec (\mathsf{cForm}).$$

[3]This version of the IFIP case ignores situations where authors might be deemed to be corporate or other entities which may not be considered to be human beings.

[4]Assume cForm defined for all model objects as

$$\begin{aligned} &\mathsf{cForm}: \\ &\quad \mathsf{fForm} = \textit{external source}. \end{aligned}$$

D.5 Attendee

An attendee is an individual who has declared an intent to attend a Working Conference. The model object ATTENDEE is a view of persons. That is, the surrogates for ATTENDEE are a subset $S_{Pt} \subseteq S_P$.

State Functions

The identity of an attendee is represented by the state function fName

$$\mathsf{fName} : S_{Pt} \rightarrow \mathbf{N}_S.$$

Law Statements

Objects of type ATTENDEE are simple and enforced by

$$\mathsf{lComp}:$$
$$\mathsf{fComp} = \mathsf{self}.$$

The law of *strict generalization* is applied to objects of type ATTENDEE. That is, attendees are *specializations* of persons, ie.

$$\mathsf{lIsA}:$$
$$\mathsf{qSize}[O_{S_g}(\mathsf{self})/\sim_{\mathrm{PERSON}}] = 1.$$

This allows the state function fName to take its value from the PERSON object with which the ATTENDEE object shares its surrogate, ie.

$$\mathsf{lName}:$$
$$\mathsf{fName} = O_{S_g}(\mathsf{self})/\sim_{\mathrm{PERSON}} .\mathsf{fName}.$$

Change Functions

The relevant change functions include

$$\mathsf{cKinds}:$$
$$\mathsf{fKinds} := \{\mathrm{ATTENDEE}\},$$

and

$$\mathsf{cInit}:$$
$$\mathsf{cForm} * \mathsf{cKinds}.$$

The partial ordering is

$$\mathsf{cInit} \prec (\mathsf{cForm}).$$

D.6 Referee

A referee is an individual who has agreed to referee papers for a Working Conference. The model object REFEREE is a view of persons. That is, the surrogates for REFEREE are a subset $S_{Pr} \subseteq S_P$.

State Functions

The identity of a referee is represented by the state function fName

$$\mathsf{fName} : S_{Pt} \rightarrow \mathrm{N}_S.$$

Law Statements

REFEREE objects are simple:

$$\mathsf{lComp}:$$
$$\mathsf{fComp} = \mathsf{self}.$$

The law of *strict generalization* is also applied to objects of type REFEREE. Referees are *specializations* of persons, ie.[5]

$$\mathsf{lIsA}:$$
$$\mathsf{qSize}[O_{S_g}(\mathsf{self})/\sim_{\mathrm{PERSON}}] = 1.$$

The state function fName must take its value from the PERSON object with which the REFEREE object shares its surrogate, ie.

$$\mathsf{lName}:$$
$$\mathsf{fName} = O_{S_g}(\mathsf{self})/\sim_{\mathrm{PERSON}} .\mathsf{fName}.$$

Change Functions

The following change functions are relevant to REFEREE. First, the change function cKinds.

$$\mathsf{cKinds}:$$
$$\mathsf{fKinds} := \{\mathrm{REFEREE}\},$$

[5]Again, this ignores the possibility that a referee may not be an individual. Consider, for example, that chess players are no longer necessarily humans. Given the current state of art, these laws above still apply. However, they may change at some future date. In OBCM, law statements can be added or changed at any time (any change, of course, must not violate the current state of the OBIS).

The function cInit initializes new objects of type REFEREE.

cInit :

cForm * cKinds.

Partial ordering is

cInit $\prec$ (cForm).

D.7 Text

A text is an organized collection of symbols expressing ideas, suggestions, and other information.[6] Assume a collection of surrogates S_{Tx} each corresponding to some unique text.

State Functions

The state function fCopy represents the contexts of a text as follows

$$\text{fCopy} : S_{Tx} \rightarrow 2^{\mathbf{T}_S},$$

where $\mathbf{T}_S$ is the set of relevant characters and symbols.

Law Statements

TEXT objects are simple:

lComp :

fComp = self.

Change Functions

The following change functions are defined for TEXT. First, the function cKinds:

cKinds :

fKinds := {TEXT}.

[6]It can be argued that in Bunge's ontology, text is *not* a substantial thing and cannot be represented by a model thing (see also [Bunge, 1977, pp. 116-117]). This thesis assumes, however, that a text has some physical form which can be described by a model object.

The value of fCopy comes from an external source as follows:

cCopy :

fCopy := *external source.*

The function cInit is defined as

cInit :

cForm * cKinds * cCopy.

D.8 Facility

A facility is some building, place or location at which a conference activity can occur. Assume the set of surrogates S_F, each corresponding to a facility of interest.

State Functions

The state function fName represents the identify of the facility.

$$\mathsf{fName} : S_F \rightarrow \mathbf{N}_S.$$

The *cost* of using the facility can be represented by the state function fCost

$$\mathsf{fCost} : S_F \rightarrow \mathbf{N}.$$

Law Statements

FACILITY objects are simple:

lComp :

fComp = self.

Change Functions

The change function cKinds is defined as

cKinds :

fKinds := {FACILITY}.

The value of fCost is set from an external source

cCost :

fCost := *external source.*

The usual cInit is defined as

$$\begin{aligned}&\mathsf{cInit}:\\&\quad \mathsf{cForm} * \mathsf{cKinds} * \mathsf{cCost}.\end{aligned}$$

D.9 Invitation Letter

An invitation letter or call for a paper is a piece of text addressed to some individual inviting that individual to submit a paper on some topic. It is represented by the model object INVITATION. The possible surrogates for INVITATION are defined as

$$S_C = S_P \times S_{Tx},$$

that is, an invitation letter is composed of a PERSON object and a TEXT object.[7]

State Functions

An invitation letter has the property of being directed or addressed to some individual. This is represented by the state function fName

$$\mathsf{fName} : S_C \rightarrow \mathbf{N}_S.$$

Law Statements

The cardinality of INVITATION is enforced by the law statement

$$\begin{aligned}&\mathsf{IComp}:\\&\quad [\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{PERSON}}) = 1 \ \&\\&\quad \mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{TEXT}}) = 1].\end{aligned}$$

The value of fName is defined to be the name of the person in the composition of the invitation. This is reflected in the law statement

$$\begin{aligned}&\mathsf{IName}:\\&\quad \mathsf{fName} = (\mathsf{fAComp}/\sim_{\mathrm{PERSON}}).\mathsf{fName}.\end{aligned}$$

[7] In the interest of simplicity, the possibility of addressing an invitation letter jointly to more than one individual is not considered.

Change Functions

Change functions are required to add PERSON and TEXT objects to the composition of an INVITATION object.

$$\begin{aligned}&\mathsf{cAddPerson}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ p,\end{aligned}$$

where p is an object of kind PERSON.[8] Similarly,

$$\begin{aligned}&\mathsf{cAddText}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ t,\end{aligned}$$

where t is an object of kind TEXT. Both the above changes can occur only once.[9]

The change function cKinds is

$$\begin{aligned}&\mathsf{cKinds}:\\ &\quad \mathsf{fKinds} := \{\text{PERSON,TEXT}\}.\end{aligned}$$

The initialization change function not only sets the values for fForm and fKinds but also adds a person and text to an invitation when it is first created:

$$\begin{aligned}&\mathsf{cInit}:\\ &\quad \mathsf{cForm} * \mathsf{cKinds} * \mathsf{cAddPerson} * \mathsf{cAddText}.\end{aligned}$$

This function can be invoked only once. Thus, the *partial ordering* of change functions can be expressed as

$$\mathsf{cInit} \prec \mathsf{cForm}.$$

D.10 Intent Letter

An intent letter is a text from some individual indicating an intent to submit a paper in response to an invitation. It is represented by the model object INTENT whose possible surrogates are defined as

$$S_I = S_P \times S_{Tx},$$

[8]Recall from Chapter 5 that the expression $\mathsf{f} := \widehat{\mathsf{f}} \circ x$ sets the new value of f to its previous value aggregated with x.

[9]Assuming that neither the addressee nor the text of an invitation letter should be changed after it is produced.

State Functions

The state function fName represents the *sender* of an intent letter and is defined as

$$\mathsf{fName} : S_I \rightarrow \mathbf{N}_S.$$

Law Statements

Law statements for INTENT are similar to invitation letters. First, cardinality is enforced[10]

$$\begin{aligned}&\mathsf{IComp}:\\ &\quad [\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{PERSON}}) = 1 \ \&\\ &\quad \mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{TEXT}}) = 1].\end{aligned}$$

That is, an intent letter is addressed to only one person and has only one text.

The value of fName is defined to be the name of the person in the composition of the intent letter. This is the law statement

$$\begin{aligned}&\mathsf{IName}:\\ &\quad \mathsf{fName} = (\mathsf{fAComp}/\sim_{\mathrm{PERSON}}).\mathsf{fName}.\end{aligned}$$

Change Functions

Change functions for INTENT are similar to those for INVITATION. The function cKinds is defined

$$\begin{aligned}&\mathsf{cKinds}:\\ &\quad \mathsf{fKinds} := \{\mathrm{PERSON,TEXT}\}.\end{aligned}$$

Functions are needed to add persons and text to intent letters.

$$\begin{aligned}&\mathsf{cAddPerson}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ p,\end{aligned}$$

where p is an object of kind PERSON. Similarly,

$$\begin{aligned}&\mathsf{cAddText}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ t,\end{aligned}$$

where t is an object of kind TEXT. The initialization of new INTENT objects is achieved by

$$\begin{aligned}&\mathsf{cInit}:\\ &\quad \mathsf{cForm} * \mathsf{cKinds} * \mathsf{cAddAuthor} * \mathsf{cAddText}.\end{aligned}$$

As expected, this function is invoked once only for each intent letter.

[10]Again, assuming the sender of an intent letter is a single individual.

D.11 Paper

A paper is a text produced by an author or authors, which has been submitted to the Working Conference in response to a call for paper. It is modelled by PAPER with possible surrogates[11]

$$S_{Pp} = 2^{S_{Pa}} \times S_{Tx}.$$

That is, for OBCM modelling purposes, a paper is an aggregate of one or more authors and a text.

State Functions

The state function fName is defined to represent the paper's identity.[12]

$$\mathsf{fName} : S_{Pp} \rightarrow \mathrm{N}_S.$$

Law Statements

An important law statement for PAPER is its cardinality law

$$\begin{aligned} &\mathsf{lComp}: \\ &\quad [\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{AUTHOR}}) \geq 1 \ \& \\ &\quad \mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{TEXT}}) = 1. \end{aligned}$$

That is, a paper is authored by at least one author and has only one text.

Change Functions

Change functions for PAPER include adding the author:

$$\begin{aligned} &\mathsf{cAddAuthor}: \\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ a, \end{aligned}$$

[11]This is a fairly liberal interpretation. A more restrictive version might assume, for example, that the only papers of interest are those (1) in response to a call (ie. no unsolicited papers are acceptable) and (2) for which an intent to submit was declared (ie. there is an intent letter on file). In this case, the surrogates for PAPER would be modified as $S_{Pp} = 2^{S_{Pa}} \times S_{Tx} \times S_C \times S_I$.

[12]A more elaborate model may include other state variables. For instance, a state function can be defined to map papers to sets of keywords or topic names which can then be matched to an equivalent state variable in REFEREE. In this way, the system can assist in identifying suitable referees for papers.

where a is an object of kind AUTHOR. Inclusion of the paper's text is as follows:

$$\begin{aligned}&\mathsf{cAddText}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ t,\end{aligned}$$

where t is an object of kind TEXT. Both these change functions are invoked once only.[13]

The value of fName is taken from an external source[14]

$$\begin{aligned}&\mathsf{cName}:\\ &\quad \mathsf{fName} := \textit{externalsource}.\end{aligned}$$

The change function cKinds is defined as

$$\begin{aligned}&\mathsf{cKinds}:\\ &\quad \mathsf{fKinds} := \{\text{AUTHOR,TEXT}\}.\end{aligned}$$

The initialization of a paper is defined as

$$\begin{aligned}&\mathsf{cInit}:\\ &\quad \mathsf{cForm} * \mathsf{cKinds} * \mathsf{cAddAuthor} * \mathsf{cAddText} * \mathsf{cName}.\end{aligned}$$

D.12 Referee Report

A Referee Report is a text written by a referee, commenting upon a paper. It is described by the model REFEREEREPORT. Assuming for simplicity that a paper is assessed by a single referee, the surrogate for a referee report is defined as

$$S_{RR} = S_R \times S_{Pp} \times S_{Tx}.$$

That is, for OBCM modelling purposes, a referee report is an aggregation of a referee, a paper and a text commenting upon the paper.

State Functions

The following state function maps each report to *accept/ reject* values.

$$\mathsf{fEval} : S_{RR} \rightarrow \{\textit{accept, reject}\}.$$

Another useful state function identifies the referee report as follows

$$\mathsf{fName} : S_{RR} \rightarrow \mathrm{N}_S.$$

[13]The possibility of modifying the text is not considered in this version of the case (although not difficult to add).

[14]This is the simplest alternative. The name by which a paper is called can be changed in many ways.

Law Statements

A cardinality law applies to REFEREEREPORT.

$$\begin{aligned}&\mathsf{lComp}:\\&\quad[\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{REFEREE}})=1\ \&\\&\quad\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{PAPER}})=1\ \&\\&\quad\mathsf{qSize}(\mathsf{fAComp}/\sim_{\mathrm{TEXT}})=1].\end{aligned}$$

Change Functions

Change functions for REFEREEREPORT include the addition of the paper:

$$\begin{aligned}&\mathsf{cAddPaper}:\\&\quad\mathsf{fAComp}:=\widehat{\mathsf{fAComp}}\circ p,\end{aligned}$$

where p is an object of kind PAPER. This is invoked once only.

The following adds the referee:

$$\begin{aligned}&\mathsf{cAddReferee}:\\&\quad\mathsf{fAComp}:=\widehat{\mathsf{fAComp}}\circ r,\end{aligned}$$

where r is an object of kind REFEREE.

The comments or text of the report is added by

$$\begin{aligned}&\mathsf{cAddText}:\\&\quad\mathsf{fAComp}:=\widehat{\mathsf{fAComp}}\circ t,\end{aligned}$$

where t is an object of kind TEXT. The last two change functions can be invoked more than once (that is, it is possible that a referee may be unable to complete an evaluation and the substitution of a new referee and related comments are possible). This requires the change function

$$\begin{aligned}&\mathsf{cRemoveRef}:\\&\quad\mathsf{fAComp}:=\widehat{\mathsf{fAComp}}\odot r\odot t,\end{aligned}$$

where $\odot$ indicates that the object r of type REFEREE and/or t of type TEXT in A-composition of the referee report are removed.

A composite change function cChangeRef is defined as

$$\begin{aligned}&\mathsf{cChangeRef}:\\&\quad\mathsf{cRemoveRef}*\mathsf{cAddReferee}.\end{aligned}$$

Various techniques can be applied to assign a value for fEval. In this case study, the value is assigned manually from an external source. That is, the external source (eg. a human operator) reviews the referee's comments and decides whether the recommendation is to accept or reject the paper.[15] This requires the change function

cEval :

fEval = *external source.*

The change function cKinds is

cKinds :

fKinds := {REFEREE, PAPER, TEXT}.

The initialization of a new referee report is represented by

cInit :

cForm * cKinds * cAddReferee * cAddPaper.

The partial ordering is somewhat more complex than usual:

cInit $\prec$ (cChangeRef) $\prec$ cAddText $\prec$ cEval.

That is, the referee's comments are accepted only after a referee is assigned and the evaluation is made after the referee's comments are received. Note also that the referee can be changed only after he/she is assigned (in cInit).

D.13 Programme Committee

For OBCM modelling purposes, the Programme Committee is viewed as an aggregation of the invitation letters, intent letters, papers, and referee reports which are required to support the activities of the Committee. The Programme Committee is represented by the model object PROGCOM and its surrogate is defined as

$$S_{PC} = 2^{S_C} \times 2^{S_I} \times 2^{S_{PP}} \times 2^{S_{RR}}.$$

Conceptually, in the OBCM/IS model, as invitations are generated, they are added to the Programme Committee. As intent letters and papers are received, they are also added to the Programme Committee. And when a paper is distributed for refereeing, a referee report is generated and added to the Programme Committee.

[15] A more complex law statement might possibly be formulated which assigns the value resulting from some form of automated content analysis of the referee's comments.

State Functions

An important property of the Program Committee is its ability to generate lists of individuals to whom it can issue a call for papers. The state function fCallList maps a Programm Committee to sets of persons as follows:

$$\mathsf{fCallList} : S_{PC} \rightarrow 2^{S_P}.$$

Another property of interest is its ability to select a set of papers for the conference. The state function fSelectedPapers maps a Programme Committee to a set of papers.

$$\mathsf{fSelectedPapers} : S_{PC} \rightarrow 2^{S_{P_P}}.$$

Law Statements

Assuming that the Program Committee will want to issue invitations to the widest range of individuals as possible the law statement lCallList defines the value of fCallList as

$$\mathsf{lCallList}:$$
$$\mathsf{fCallList} = \mathit{IFIP}.\mathsf{fComp}/\sim_{\text{PERSON}}.$$

That is, the list to whom the call for papers is to be sent is the set of all objects of type PERSON in the IFIP system.[16]

The law statement lSelectPapers is defined as

$$\mathsf{lSelectPapers}:$$
$$\mathsf{fSelectedPapers} = (\mathsf{fAComp}/\sim_{(\text{REFEREEREPORT } \&\ \mathsf{fEval} = \mathit{accept})}).\mathsf{fAComp}/\sim_{\text{PAPER}}.$$

That is, given all the referee reports in the composition of the Programme Committee, the selected papers are the papers in the composition of all referee reports with fEval = *accept*.

Interestingly enough, for the purposes of this case study, the cardinalities of the components of a Programme Committee are not relevant. Therefore, no law statements concerning the composition of Programme Committees are specified.

[16]Recall that the IFIP system is the total aggregation of all objects of interest to the IFIP case study. It is also reasonable to modify this law statement to restrict invitations to some selected group or groups of individuals. In the absence of additional information in the problem description, however, lCallList remains as defined.

Change Functions

Change functions for PROGCOM include adding calls for papers (ie. invitations)

$$\begin{aligned}&\textsf{cAddCall}:\\ &\quad \textsf{fAComp} := \widehat{\textsf{fAComp}} \circ c,\end{aligned}$$

where c is an object of kind INVITATION. The function **cAddIntent** allows adding letters of intent as they are received by the committee.

$$\begin{aligned}&\textsf{cAddIntent}:\\ &\quad \textsf{fAComp} := \widehat{\textsf{fAComp}} \circ i,\end{aligned}$$

where i is an object of kind INTENT. Similarly, papers are added as they are received.

$$\begin{aligned}&\textsf{cAddPaper}:\\ &\quad \textsf{fAComp} := \widehat{\textsf{fAComp}} \circ p,\end{aligned}$$

where p is an object of kind PAPER. The following allows referee reports to be generated and added.

$$\begin{aligned}&\textsf{cAddRefereeReport}:\\ &\quad \textsf{fAComp} := \widehat{\textsf{fAComp}} \circ r,\end{aligned}$$

where r is an object of kind REFEREEREPORT.

The change **cKinds** is

$$\begin{aligned}&\textsf{cKinds}:\\ &\quad \textsf{fKinds} := \{\text{PERSON, INVITATION, INTENT, PAPER, REFEREEREPORT}\}.\end{aligned}$$

Initialization involves

$$\begin{aligned}&\textsf{cInit}:\\ &\quad \textsf{cForm} * \textsf{cKinds}.\end{aligned}$$

D.14 Organizing Committee

For OBCM modelling purposes, the Organizing Committee is viewed as a set of facilities and attendees.

$$S_{OC} = S_F \times S_{Pt}.$$

Conceptually, in the model, as facilities are located, they are added to the composition of the Organizing Committee. Similarly, as individuals confirm their intent to attend, they are added to the Organizing Committee.

Change Functions

Change functions for ORGCOM include the addition of facilities:

$$\begin{aligned}&\mathsf{cAddFacility}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ f,\end{aligned}$$

where f is an object of kind FACILITY. The addition of attendees is represented by

$$\begin{aligned}&\mathsf{cAddAttendee}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ a,\end{aligned}$$

where a is an object of kind ATTENDEE.

The change cKinds is defined as

$$\begin{aligned}&\mathsf{cKinds}:\\ &\quad \mathsf{fKinds} := \{\text{FACILITY, ATTENDEE}\}.\end{aligned}$$

Initialization is represented by the change function

$$\begin{aligned}&\mathsf{cInit}:\\ &\quad \mathsf{cForm} * \mathsf{cKinds}.\end{aligned}$$

D.15 Session

A session is viewed as an aggregation of attendees, papers and facilities. That is, attendees meet at one or more facilities to discuss a set of papers. The model for a session is SESSION and its surrogate is

$$S_S = 2^{S_{Pt}} \times 2^{S_F} \times 2^{S_{Pp}}.$$

State Functions

The cost of a session can be represented by the state function fCost

$$\mathsf{fCost} : S_S \rightarrow \mathrm{N}.$$

Law Statements

The law statement lCost defines the value of fCost as

$$\begin{aligned}&\mathsf{lCost}:\\ &\quad \mathsf{fCost} = \Sigma(\mathsf{fAComp}/\sim_{\text{FACILITY}}.\mathsf{fCost}).\end{aligned}$$

That is, the cost of a session is equal to the sum of the costs of its facilities.

Change Functions

Change functions for SESSION consist of the following. A change to add a facility.

$$\begin{aligned}&\mathsf{cAddFacility}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ f,\end{aligned}$$

where f is an object of kind FACILITY. A change to add an attendee.

$$\begin{aligned}&\mathsf{cAddAttendee}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ a,\end{aligned}$$

where a is an object of kind ATTENDEE. A change to add a paper.

$$\begin{aligned}&\mathsf{cAddPaper}:\\ &\quad \mathsf{fAComp} := \widehat{\mathsf{fAComp}} \circ p,\end{aligned}$$

where p is an object of kind PAPER.

Other changes include cKinds

$$\begin{aligned}&\mathsf{cKinds}:\\ &\quad \mathsf{fKinds} := \{\text{FACILITY, ATTENDEE, PAPER}\}.\end{aligned}$$

Initialization is defined as

$$\begin{aligned}&\mathsf{cInit}:\\ &\quad \mathsf{cForm} * \mathsf{cKinds}.\end{aligned}$$

D.16 Working Conference

A Working Conference consists of a Programme Committee, an Organizing Committee and a number of sessions. It is described by the model WORKINGCONFERENCE. Its surrogate is

$$S_{WC} = 2^{S_S} \times S_{PCOM} \times S_{OCOM}$$

State Functions

The state function fCost represents the total cost of a Working Conference:

$$\mathsf{fCost} : S_{WC} \rightarrow \mathrm{N}.$$

A Working Conference can be identified by a name:

$$\mathsf{fName} : S_{WC} \rightarrow \mathrm{N}_S.$$

Law Statements

A cardinality law applies to WORKINGCONFERENCE.

$$\begin{aligned}&\mathsf{lComp}:\\&\quad[\mathsf{qSize}(\mathsf{fAComp}/\sim_{\text{PROGCOM}}) = 1 \;\&\\&\quad\mathsf{qSize}(\mathsf{fAComp}/\sim_{\text{ORGCOM}}) = 1.\end{aligned}$$

The law statement lCost defines the value of fCost as

$$\begin{aligned}&\mathsf{lCost}:\\&\quad\mathsf{fCost} = \Sigma[(\mathsf{fAComp}/\sim_{\text{SESSION}}).\mathsf{fCost})].\end{aligned}$$

That is, the cost of a Working Conference is equal to the sum of the costs of its sessions.

Change Functions

The following change functions apply to WORKINGCONFERENCE. A change to add its Program Committee.

$$\begin{aligned}&\mathsf{cAddProgCom}:\\&\quad\mathsf{fAComp} := \mathsf{fA}\widehat{\mathsf{Comp}} \circ p,\end{aligned}$$

where p is an object of kind PROGCOM. Next, a change to add its Organizing Committee.

$$\begin{aligned}&\mathsf{cAddOrgCom}:\\&\quad\mathsf{fAComp} := \mathsf{fA}\widehat{\mathsf{Comp}} \circ o,\end{aligned}$$

where o is an object of kind ORGCOM. Both cAddProgCom and cAddOrgCom can only be invoked once. A change cAddSession adds sessions to the Working Conference:

$$\begin{aligned}&\mathsf{cAddSession}:\\&\quad\mathsf{fAComp} := \mathsf{fA}\widehat{\mathsf{Comp}} \circ s,\end{aligned}$$

where s is an object of kind SESSION.

The fKinds of a Working Conference are changed by

$$\begin{aligned}&\mathsf{cKinds}:\\&\quad\mathsf{fKinds} := \{\text{SESSION, PROGCOM, ORGCOM}\}.\end{aligned}$$

A Working Conference is initialized by

$$\begin{aligned}&\mathsf{cInit}:\\&\quad\mathsf{cForm} * \mathsf{cKinds} * \mathsf{cAddProgCom} * \mathsf{cAddOrgCom}.\end{aligned}$$

D.17 IFIP

The IFIP is the collection of all Working Conferences and all individuals of interest to the IFIP system.[17] It is described by the model IFIP which has the surrogates

$$S_{IFIP} = 2^{S_{WC}} \times 2^{S_P}.$$

Change Functions

The following change functions are found in IFIP. New Working Conferences are added by

cAddConference :

fAComp := $\widehat{\text{fAComp}} \circ c$,

where c is an object of kind WORKINGCONFERENCE.

The state variable fKinds is changed by

cKinds :

fKinds := {WORKINGCONFERENCE, PERSON}.

Initialization of the IFIP system itself is achieved by

cInit :

cForm * cKinds.

[17] This is an appropriate place to populate with objects of type PERSON since individuals can participate in several Working Conferences.

Appendix E

IFIP Working Conference Problem - OBIS Implementation

E.1 Person

(ModelObj) subclass: #Person

cForm
```
self change:#fForm to:(OrderedCollection with:100@50
        with: Display width * 1//2 @ (Display height * 3//4)).
pOrder add:#cForm.
```

cGForm
```
self changeVar:#fGForm
    to:((FreeDrawing pictureDictionary at:'aGraph') offset:0@0).
pOrder add:#cGForm
```

cInit
```
  "Initialize the new object"
self cForm;
    cGForm;
    cName.
pOrder add:#cInit.
```

cName
```
      "Change the Person name."
|name|
name := (Prompter prompt: 'Name?'
                default: self fName).
self changeVar:#fName to:name.
```

lLaws
```
|compSize|
"Composition Law"
compSize := self fAComp size.
(PersonLaws new :? lComp(self)) notNil
```

```
    ifFalse:[↑ false].
↑ true
```
(Prolog) subclass: #PersonLaws

```
"Composition Law - simple object"
lComp(aPerson) :-
    is(cardinality,
       aPerson fComp size),
    eq(cardinality,1).
"If a Person view exists of :anObject, return its name"
lName(name,anObject) :-
    is(x,anObject view:anObject as:Person),
    eq(x,nil),!,fail().
lName(name,anObject) :-
    is(x,anObject view:anObject as:Person),
    is(name,x fName).
```

E.2 Author

(ModelObj) subclass: #Author

cGForm
```
self changeVar:#fGForm to:('Author' magnifyBy:6@10).
pOrder add:#cGForm
```

cInit
```
  "Initialize the new object"
self cGForm;
    cName.
pOrder add:#cInit.
```

cName
```
      "Change the author name."
|name oldName|
oldName := self fName.
(name := AuthorLaws new :? lName(x,self)) notNil
    ifTrue:[name do:[:each | name := each at:1]].
name isNil ifTrue:[name := (Prompter prompt: 'Name?'
                    default: oldName)].
self change:#fName to:name.
```
(Prolog) subclass: #AuthorLaws

```
"Composition Law - simple object"
lComp(compositionSize) :-
   eq(compositionSize,1).

"Name from Person View if any"
lName(name,anAuthor) :-
   consult(name(x,anAuthor),PersonLaws new),
   eq(name,x).
```

E.3 Attendee

(ModelObj) subclass: #Attendee

cGForm

```
self changeVar:#fGForm to:('Attendee' magnifyBy:6@10).
pOrder add:#cGForm
```

cInit

```
  "Initialize the new object"
self cGForm;
    cName.
pOrder add:#cInit.
```

cName

```
      "Change the attendee name."
|name oldName|
oldName := self fName.
(name := AttendeeLaws new :? lName(x,self)) notNil
   ifTrue:[name do:[:each | name := each at:1]].
name isNil ifTrue:[name := (Prompter prompt: 'Name?'
                  default: oldName)].
self change:#fName to:name.
```

E.4 Referee

(ModelObj) subclass: #Referee

cGForm

```
self changeVar:#fGForm to:('Referee' magnifyBy:6@10).
pOrder add:#cGForm
```

```
cInit
  "Initialize the new object"
self cGForm;
    cName.
pOrder add:#cInit.

cName
      "Change the Referee name."
|name oldName|
oldName := self fName.
(name := RefereeLaws new :? lName(x,self)) notNil
    ifTrue:[name do:[:each | name := each at:1]].
name isNil ifTrue:[name := (Prompter prompt: 'Name?'
                    default: oldName)].
self change:#fName to:name.
(Prolog) subclass:          #RefereeLaws

"Composition Law - simple object"
lComp(compositionSize) :-
   eq(compositionSize,1).

"Name from Person View if any"
lName(name,aReferee) :-
   consult(name(x,aReferee),PersonLaws new),
   eq(name,x).
```

E.5 Text

```
(ModelObj) subclass:        #Text

cCopy
"Change text copy"

|text|
text := (Prompter prompt: 'Text Copy?'
          default:").
self change:#fCopy to:text.
pOrder add:#cCopy.

cInit
      "Initialize Text object.
       Set its text. "
```

```
self cCopy.
pOrder add:#cInit.
```

fCopy
"Answer Text Copy"
```
↑ stateFunctions at:#fCopy ifAbsent:[↑ nil].
```

E.6 Facility

(ModelObj) subclass: #Facility

cCost
```
|oldCost cost|
oldCost := self fCost.
cost := (Prompter prompt: 'Cost?'
          default: (oldCost printPaddedTo:1)) asInteger.
self change:#fValue to:cost.
```

cGForm
```
self changeVar:#fGForm to:('Facility' magnifyBy:6@10).
pOrder add:#cGForm
```

cInit
"Initialize new object"
```
self cGForm;
    cCost.
pOrder add:#cInit.
```

cName
"Change the Facility name."
```
|name|
name := (Prompter prompt: 'Name?'
                  default: self fName).
self changeVar:#fName to:name.
```

fCost
"Answer the cost of facility"
```
↑ stateFunctions at:#fCost ifAbsent:[↑ 0].
```
(Prolog) subclass: #FacilityLaws

"Composition Law - simple object"
```
lComp(compositionSize) :-
```

```
eq(compositionSize,1).
```

E.7 Invitation Letter

(ModelObj) subclass: #Invitation

cACompPerson

"Add a Person to Invitation."

```
self change:#fAComp to:(self addComp:Person).
pOrder add:#cACompPerson.
```

cACompText

"Add a text to Invitation."

```
self change:#fAComp to:(self addComp:Text).
pOrder add:#cACompText.
↑ self
```

cInit

"Initialize new object"

```
self cKinds;
    cACompPerson;
    cACompText.
pOrder add:#cInit.
```

cKinds

"Kinds are Person, Text"

```
self change:#fKinds to:(Set with:self class
                        with:Person
                        with:Text).
pOrder add:#cKinds
```

cName

```
|name|
(InvitationLaws new :? lName(x,self)) do: [:each |
    name := each at:1].
self changeVar:#fName to:name.
```

(Prolog) subclass: #InvitationLaws

"Name Law: identify Invitation as name of addressee"

```
lName(name,anInvitation) :-
   is(name,
      (anInvitation objectOfKind:Person in:#fAComp) fName).
```

E.8 Intent Letter

(ModelObj) subclass: #Intent

cACompPerson
"Add a Person to Intent Letter."
```
self change:#fAComp to:(self addComp:Person).
pOrder add:#cACompPerson.
```

cACompText
"Add a text to Intent Letter."
```
self change:#fAComp to:(self addComp:Text).
pOrder add:#cACompText.
```

cInit
"Initialize new object "
```
self cKinds;
    cACompPerson;
    cACompText;
    cName.
pOrder add:#cInit.
```

cKinds
"Kinds are Person, Text"

```
self change:#fKinds to:(Set with:self class
                        with:Person
                        with:Text).
pOrder add:#cKinds
```

cName
```
|name|
(IntentLaws new :? lName(x,self)) do: [:each |
    name := each at:1].
self changeVar:#fName to:name.
pOrder add:#cName.
```
(Prolog) subclass: #IntentLaws

"Name Law: identify as name of sender"
```
lName(name,anIntent) :-
   is(name,
      (anIntent objectOfKind:Person in:#fAComp) fName).
```

E.9 Paper

(ModelObj) subclass: #Paper

cACompAuthor
"Add an author."
```
self change:#fAComp to:(self addComp:Author).
pOrder add:#cACompAuthor.
```

cACompText
"Add Text."
```
self change:#fAComp to:(self addComp:Text).
pOrder add:#cACompText.
↑ self
```

cInit
"Initialize new object "
```
self cKinds;
    cACompAuthor;
    cACompText.
pOrder add:#cInit.
```

cKinds
"Kinds are Author, Text"
```
self change:#fKinds to:(Set with:self class
                        with:Author
                        with:Text).
pOrder add:#cKinds
```

cName
"Change the Paper title."
```
|name|
name := (Prompter prompt: 'Paper Title?'
                  default: self fName).
self changeVar:#fName to:name.
```

E.10 Referee Report

(ModelObj) subclass: #RefereeReport

cACompPaper
"Add a Paper to RefereeReport."

```
self change:#fAComp to:(self addComp:Paper).
pOrder add:#cACompPaper
```

cACompReferee

```
    "Add a Referee to RefereeReport."
self change:#fAComp to:(self addComp:Referee).
pOrder add:#cACompReferee
```

cACompText

```
    "Add a Text to RefereeReport."
self change:#fAComp to:(self addComp:Text).
pOrder add:#cACompText
```

cChangeRef

```
"Change a Referee"
pOrder remove:#cAddReferee.
self cRCompReferee;
    cACompReferee.
```

cEval

```
self changeVar:#cEval
    to:(Prompter prompt:'Evaluation?'
        default:'').
pOrder add:#cChangeRef.
```

cInit

```
"Initialize"
self cKinds;
    cACompPaper;
    cACompReferee.
pOrder add:#cInit.
↑ nil
```

cKinds

```
"Kinds are Referee, Paper, Text"
self change:#fKinds to:(Set with:self class
                        with:Referee
                        with:Text
                        with:Paper).
pOrder add:#cKinds
```

cRCompReferee

"Remove a Referee from RefereeReport."
self change:#fAComp to:(self removeComp:Referee).
pOrder add:#cACompReferee

fEval
"Answer Evaluation"
↑ stateFunctions at:#fEval ifAbsent:[↑ 0].

fName
"Report name = title of Paper"
|paper|
paper := self objectOfKind:Paper in:#fAComp.
↑ paper fName.
(Prolog) subclass: #RefereeReportLaws

"Cardinality Law - one Referee, one Paper and one Text"
lComp(aReport) :-
is(referee,
(aReport objectsOfKind:Referee
in:#fAComp) size),
is(paper,
(aReport objectsOfKind:Paper
in:#fAComp) size),
is(text,
(aReport objectsOfKind:Text
in:#fAComp) size),
eq(referee,1),
eq(paper,1),
eq(text,1).

E.11 Programme Committee

(ModelObj) subclass: #ProgCom

cACompIntent
"Add a Letter of Intent."
self change:#fAComp to:(self addComp:Intent).

cACompInvitation
"Add an Invitation."
self change:#fAComp to:(self addComp:Invitation).

cACompPaper

"Add a Paper."

```
self change:#fAComp to:(self addComp:Paper).
```

cACompRefereeReport

"Add a RefereeReport."

```
self change:#fAComp to:(self addComp:RefereeReport).
```

cGForm

```
self changeVar:#fGForm to:('ProgCom' magnifyBy:6@10).
pOrder add:#cGForm
```

cInit

"Initialize new object name"

```
self cGForm;
    cKinds.
pOrder add:#cInit.
```

cKinds

```
|kinds|
kinds := Set with:self class.
kinds add:ProgCom;
      add:Invitation;
      add:RefereeReport;
      add:Paper;
      add:Intent.
self change:#fKinds to:kinds.
pOrder add:#cKinds.
```

fCallList

*"Answers a list to whom the call for papers can be sent.
Determined by law lCallList"*

```
|list|
(ProgComLaws new :? lCallList(x,self)) do: [:each |
    list := each at:1].
↑ list
```

fSelectedPapers

"Answers a list of Selected Papers"

```
|list|
(ProgComLaws new :? lCallList(x,self)) do: [:each |
    list := each at:1].
```

```
↑ list
(Prolog) subclass:          #ProgComLaws

"Call list is all Persons in the IFIP System"
lCallList(callList,aProgCom) :-
  is(callList,
    aProgCom objectsOfKind:Person).

"Selected papers all have Referee Reports with fEval = 'accept"'
lSelect(sPapers,aProgCom) :-
  is(reports,
    aProgCom objectsOfKind:RefereeReport
          with:#fEval equal:'accept' in:#fAComp),
  is(papers,
    aProgCom objectsOfKind:Paper in:reports).
```

E.12 Organizing Committee

```
(ModelObj) subclass:        #OrgCom

cACompAttendee
    "Add an Attendee."
self change:#fAComp to:(self addComp:Attendee).

cACompFacility
    "Add an Facility."
self change:#fAComp to:(self addComp:Facility).

cGForm
self changeVar:#fGForm to:('OrgCom' magnifyBy:6@10).
pOrder add:#cGForm

cInit
      "Initialize new object"
self cGForm;
    cKinds.
pOrder add:#cInit.

cKinds
|kinds|
kinds := Set with:self class.
kinds add:OrgCom;
```

```
    add:Facility;
    add:Attendee.
self change:#fKinds to:kinds.
pOrder add:#cKinds.
```

E.13 Session

(ModelObj) subclass: #Session

cACompAttendee

```
    "Add an Attendee."
self change:#fAComp to:(self addComp:Attendee).
```

cACompChairman

```
    "Add a Chairman."
self change:#fAComp to:(self addComp:Chairman).
pOrder add:#cACompChairman
```

cACompFacility

```
    "Add a Facility."
self change:#fAComp to:(self addComp:Facility).
```

cACompPaper

```
    "Add a Paper."
self change:#fAComp to:(self addComp:Paper).
```

cGForm

```
self changeVar:#fGForm to:('Session' magnifyBy:6@10).
pOrder add:#cGForm
```

cInit

```
"Initialize"
self cKinds;
    cGForm.
pOrder add:#cInit.
↑ nil
```

cKinds

```
"Kinds are Attendees, Facilities, Papers"
self change:#fKinds to:(Set with:self class
                        with:Facility
                        with:Attendee
```

```
                    with:Paper).
pOrder add:#cKinds

cName
    "Change the Session name."
|name|
name := (Prompter prompt: 'Session name?'
                  default: self fName).
self changeVar:#fName to:name.
(Prolog) subclass:          #SessionLaws

"Session cost = sum of costs of facilities"
lCost(cost,aSession) :-
  is(cost,
    aSession sumValuesOf:#fCost
        fromObjectsOfKind:Facility
        in:#fAComp).
```

E.14 Working Conference

```
(ModelObj) subclass:        #Conference

cACompOrgCom
    "Add a OrgCom to Conference."
self change:#fAComp to:(self addComp:OrgCom).
pOrder add:#cACompOrgCom

cACompProgCom
    "Add a ProgCom to Conference."
self change:#fAComp to:(self addComp:ProgCom).
pOrder add:#cACompOrgCom

cACompSession
    "Add a Session to Conference."
self change:#fAComp to:(self addComp:Session).

cGForm
self changeVar:#fGForm to:('Conference' magnifyBy:9@15).
pOrder add:#cGForm

cInit
self cKinds;
```

```
    cGForm;
    cName;
    cACompProgCom;
    cACompOrgCom.
pOrder add:#cInit.

cKinds
|kinds|
kinds := Set with:self class
          with:Session
          with:ProgCom
          with:OrgCom.
self change:#fKinds to:kinds.
pOrder add:#cKinds.

cName
      "Change the Conference name."
|name|
name := (Prompter prompt: 'Conference Name?'
                  default: self fName).
self changeVar:#fName to:name.

fCost
  "Answer fCost defined as law lCost"
  |value|
(ConferenceLaws new :? lCost(x,self)) do: [:each |
   value := each at:1].
↑ value
(Prolog) subclass:          #ConferenceLaws

"Cardinality Law - one Referee, one Paper and one Text"
lComp(aConference) :-
  is(progCom,
    (aConference objectsOfKind:ProgCom
          in:#fAComp) size),
  is(orgCom,
    (aConference objectsOfKind:OrgCom
          in:#fAComp) size),
  eq(progCom,1),
  eq(orgCom,1).

"Cost is sum of cost of all Sessions"
```

```
lCost(cost,aConference) :-
   is(cost,
     aConference sumValuesOf:#fCost
          fromObjectsOfKind:Session
          in:#fAComp).
```

E.15 IFIP

(ModelObj) subclass: #IFIP

cACompConference
"Add a Conference to IFIP."
```
self change:#fAComp to:(self addComp:Conference).
```

cACompPerson
"Add a Person to IFIP."
```
self change:#fAComp to:(self addComp:Person).
```

cForm
"Change the fForm parameters"
```
self changeVar:#fForm to:(OrderedCollection with:0@0
          with: Display width * 3//4 @ (Display height * 3//4)).
pOrder add:#cForm
```

cGForm
```
self changeVar:#fGForm to:('IFIP' magnifyBy:9@15).
pOrder add:#cGForm
```

cInit
"Initialize IFIP object"
```
self cForm;
   cGForm;
   cKinds.
pOrder add:#cInit.
```

cKinds
"Kinds"
```
self change:#fKinds to:(Set with: self class
                    with:Conference
                    with:Person).
pOrder add:#cKinds.
```

Appendix F

Semantic Data Bases

F.1 Kroenke and Dolan

Kroenke & Dolan [1988] describe object orientation as a means of unifying database concepts. They view an object as a "stored data representation" of an *entity* which is loosely defined as "something a user perceives as an independent unit" in the world of interest. Entities are perceptions and may or may not be physical. More specifically, in their view, an *object* is "a named collection of properties that sufficiently describes an entity in the user's work environment".[1] A *property* is defined as a characteristic of the corresponding entity which is deemed important to one or more users of the database. The notion of *sufficiency* emphasizes that not all aspects of an entity need be represented to meet the needs of users. Both entities and objects are viewed as *classes* or *types*. Individuals are called *entity instances* or *object instances*.

F.1.1 Object Properties

Kroenke & Dolan use *object diagrams* to visually describe objects (Figure 2.2 adapted from [Kroenke & Dolan, 1988]). An object diagram is drawn as a box within which properties are listed. The *name* of an object instance is the value of one of the properties of the object. Names need not be unique. Object properties can be single or multi-valued (multi-valued properties are marked MV in the diagram). There are two types of properties, *non-object* and *object* properties.

Non-object properties have domains such as numbers or strings. The domain of an object property is a set of object instances. Not all the properties of an object instance need be accessible to a user. A particular application may have access to only a subset of the properties of an object instance. The portion of an object accessible to an application is called a *view*.

F.1.2 Object Categories

The specification for a database consists of object definitions, ie. the names and properties of the objects, and domain specifications. Heuristics and rules of thumb are presented for identifying the objects.

Kroenke and Dolan distinguish among various categories of objects such as simple, compound, composite, association and aggregation objects. Simple objects contain only single-valued, non-object properties. Composite objects contain one or more non-object multi-valued properties. Compound

[1] Ie. *object* appears to be defined as a Cartesian aggregation of properties.

objects contain at least one object property. Association objects represent relationships between two (or more) objects. For example, a FLIGHT-SCHEDULE object relates PILOT and AIRPLANE objects. Finally, aggregated objects represent *groups* of entities. For example, a FACULTY object includes as one of its properties, multi-valued occurrences of TEACHER objects. Aggregated objects can be arranged in hierarchies and inheritance can be defined over these hierarchies.

F.1.3 Objects and DataBase

The notion of object is used to support a user-oriented perspective of databases.

Kroenke and Dolan claim that users perceive their environment in terms of entities and associated objects. However, DBMS do not directly support objects. Hence, they provide specific methodologies whereby the different categories of objects are transformed into their equivalent relational database representations. The database design task involves the integration of different user views of some entity into a single object in a process they call *object materialization.* This object is then transformed into a relational representation using their methodologies. Database languages, menus, forms design software, and display screens are viewed as tools and techniques to aid the materialization process.

F.2 ACM/PCM

Most semantic data models have focused almost exclusively on the structural aspects of representing information. However, only part of the semantics of an application are described by structural descriptions. Although behavioural properties are sometimes implied by structural properties, primitives and procedures outside the data model are usually required to complete the semantics and describe the application completely.

Brodie & Ridjanovic [1984] present a data modelling methodology which attempts to integrate both structural and behavioural aspects of conceptually modelling in the database context. They present both a data model (SHM+) and a methodology (ACM/PCM), which draw heavily from concepts in database and programming languages such as data abstraction, abstract data types, procedural abstractions, and specification techniques. As such, their approach has a strong "object-oriented" flavour.

F.2.1 Structure Modelling

The modelling approach, SHM+, uses one construct, *object*, and four abstraction mechanisms to structurally organize objects: *classification, aggregation, generalization,* and *association.* The process of structure modelling at the conceptual level involves identifying all objects and their relationships in an application.

Classification is used to achieve the notion of *object classes* in terms of the properties shared by all object instances in a class. This allows the world to be described in terms of object classes (referred to from this point as simply "objects") rather than in terms of individual object instances. Aggregation is used to describe a relationship between component objects and a higher level aggregate object in a *part-of* relationship. Generalization relates object classes into hierarchical structures as *is-a* relationships. Association expresses the relationship between member objects and a higher level *set* object, ie. a

member-of relationship. Brodie [1982] and others have defined these abstractions in terms of predicate axioms and set-theoretic functions.

Repeated application of these abstractions result in aggregation, association and generalization *hierarchies.* Brodie and Ridjanovic see these abstractions as orthogonal, so objects can participate simultaneously in many hierarchies. Further, to reduce the complexity of monolithic database design, a principle of *localization* is invoked whereby only one object and those objects immediately related are considered at any one time.

A graphical representation called an *object scheme* is used for describing overall, gross properties of the conceptual model. Object schemes are directed graphs in which nodes are strings denoting objects and edges identify the abstraction mechanisms related objects. Algorithms are available for converting object schemes to traditional database designs (eg. relational). Further, a concept of *semantic relativism* is developed whereby object properties can be viewed as attributes, entities or relationships, depending upon the needs of the user. A specification language, Beta, is provided for a more detailed description.

F.2.2 Behaviour Modelling

Brodie and Ridjavonic note that most traditional DB models offer only primitive operations and little or no support for higher conceptual levels of modelling dynamics. SHM+ is a descriptive language which provides for both high-level structure and behavioral modelling.

Behavioral modelling involves three concepts at three different levels of conceptualization. First, *primitives* (INSERT, DELETE, UPDATE, FIND, CREATE and REQUEST) operate on the database level. Second, two forms of procedural abstractions, *transactions* and *actions*, are provided for describing operations on an object level. Finally, *control abstractions* (sequence, choice and repetition) are provided for the user interface.

In SHM+, an *action* is a behavioural property of an object defined in terms of a primitive operation and an invocation context. The invocation context consists of *pre-conditions, postconditions,* and *exception handling.* Actions define completely the behaviour of objects and provide the only means for their alteration. Further, the principle of localization is invoked whereby designers model properties of application object independently then integrate them to complete the design. *Transactions* fulfill specific user requirements, are composed of actions along with an invocation context, and operate over an arbitrary collection of objects. A graphical representation called *behaviour schemes* is developed in which nodes are object names and edges are operations.

Brodie and Ridjanovic make an analogy between the structure of objects and the control structures of operations upon the objects. Briefly, they relate aggregation, association and generalization to the control abstractions of sequencing, repetition and choice. To explain further, an operation over an aggregate presumably required individual operations over each component object in the aggregate. An operation over a type hierarchy, ie. resulting from generalization, requires choice for each category in the hierarchy. Finally, an operation over an association involves operating over a set of similar objects, ie. iteration. This relationship between structure and behaviour is an analogy only but is claimed to give insight into the relationship between structural and behavioural modelling.

The authors claim that since the three forms of control abstraction above can be used to define all computable functions, this suggests the completeness of SHM+ with respect to database transactions.

They further note the intuitive appeal in how the "deep structure" of complex objects described in terms of association, generalization, and aggregation hierarchies is reflected in a "deep structure" of the operations over these objects.

F.2.3 Methodology

The ACM/PCM methodology is characterized by three principles: (1) abstraction of object structure and behaviour; (2) localization; and (3) refinement.

Since both structure and behaviour are integrated, designers can start with either structure or behaviour first. In fact, it should be easy to iterate between the two. Localization helps to control the complexity of design. Several levels are offered, resulting in incremental design and different levels of precision as appropriate. For example, object and behaviour schemes are useful for gross, overall design.

Scheme diagrams can be expressed in terms of SHM+, resulting in descriptions which are abstract and should be easy to use since they are expressed in application terms. However, they are not precise. Formal techniques to support analysis and verification are precise but not easy to use. An intermediate specification language is described as a compromise. However, the authors note that where critical operations are concerned, more formal and precise mathematical techniques will be necessary. The point is that ACM/PCM accommodates various levels of analysis, providing for smooth transitions between these levels and thus facilitating incremental design.

F.3 Prototype Activity Modelling System

Flint [1986] and Flint and Leveson [1985] present a modelling system which integrates both objects and operations in a consistent and homogeneous fashion. They note the large number of modelling constructs and complexity of most semantic data models. Their goal, therefore, is to develop a modelling system with both good representation power and an easy to understand, intuitive interpretation (such as the table abstraction of Codd's original Relational Model). In addition to integrating objects and operations, their Prototype Activity Modelling System (PAMS) attempts to integrate design, specification and implementation into a single modelling system in order to facilitate *evolutionary growth* of the IS.

F.3.1 Containment

Flint notes that the concept of *object types* is found under a variety of different names in all data modelling approaches. Data models, however, differ considerably in how the types of relationships among object types are supported. PAMS offers only one form of data aggregation, the *bundle*, defined as:

> ..a homogeneous cover aggregation of Cartesian aggregations of objects. The cover aggregation may be ordered by one or more total ordering functions and may be constrained to any particular number of cardinalities. The objects contained in the Cartesian aggregation may be simple instances of data values or may be other bundles of objects [Flint, 1986, p. 250].

For example, aggregation hierarchies can be developed as

US GOVERNMENT {(branch)}
BRANCH {(name, department)}
DEPARTMENT {(name, agency)}
AGENCY {(name, division)}, etc.

(Upper case denotes bundle names, parentheses surround Cartesian aggregates, braces surround cover aggregations. In the examples above, each bundle is a cover aggregation of a single Cartesian aggregation.)

Such nested bundle descriptions constitute the PAMS model. Object hierarchies are open-ended. Hence, the model is dynamic in the sense that one can extend the upper and lower limits of the hierarchy as appropriate to the level of modelling desired. Further, a change can be made to the definition of a bundle by ordinary insertion/deletion/update operations on the contents of a bundle. This, according to Flint, forms the basis for PAMS to serve both as a database description and implementation simultaneously. Design evolution (by updating bundles) and database manipulation (presumably by updating object instances) are treated homogeneously.

F.3.2 Conditional Abstraction

In PAMS, the definition or *structure* of one object may depend on the *value* of another. This is consistent with its open-ended and dynamic nature. For example, the structure of a seemingly simple object such as TELEPHONE number is a Cartesian aggregate of AREA CODE, EXCHANGE, and NUMBER in the US and Canada. In some other countries, telephone numbers are aggregates of COUNTY CODE, ROUTING CODE, and NUMBER. Still other countries omit ROUTING CODE. Therefore, the structure of TELEPHONE will vary depending upon other objects (eg. COUNTRY) with which TELEPHONE is associated. Flint calls this *conditionally defined abstraction.*

Flint notes that some semantic modelling systems include limited support for conditional abstraction (eg. Codd's RM/T). PAMS implements conditional abstraction by augmenting the aggregation (bundling) notation with a specification for operations. Eg.

TELEPHONE:
SELECT FROM structure bundle
VIA country.

Flint calls this the principle of *non-procedural thunk.*[2] Pre- and post-conditions can also be implemented as thunks. Flint also considers the possibility of including *scripts* as an object type in the model.

F.3.3 Abstraction Support

Aggregation and classification abstraction support is provided as a consequence of the definition of bundles. Generalization is not directly supported. However, Flints suggests that generalization is not necessarily orthogonal to aggregation as is the usual claim.

Basically, PAMS sees generalization from the *specialization* point of view. An entity can be specialized from a more general counterpart by specifying an *operation* which results in the specialized object.

[2]In programming, a *thunk* is a procedure passed within parameters [Ralston, 1976].

Flint suggests that the usual notion of generalization is overly permissive, since (1) it applies to all instances of the generalized object type and (2) it encourages false generalizations (eg. all birds can fly). By focusing upon specialization as a specific object-operation pair, PAMS supports what Flint calls a *pessimistic* form of generalization, since no instance of an object can be generalized unless the original object-operation pair is known.

Appendix G

Figures

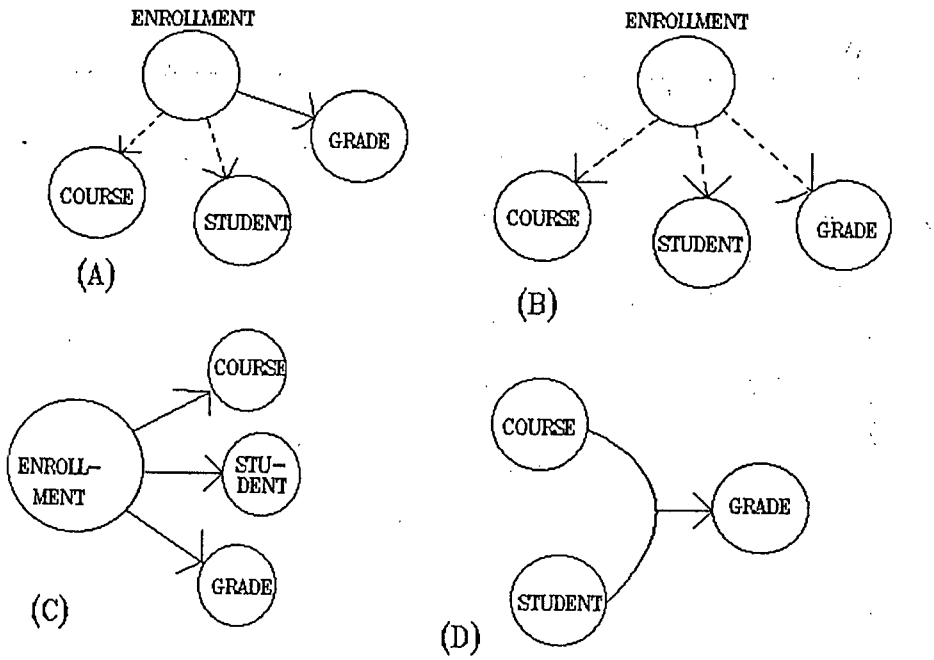

Figure 2.1: Different mechanisms for interrelating types.

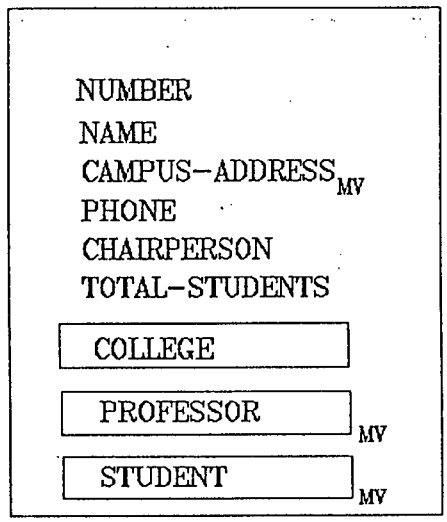

Figure 2.2: Object diagram.

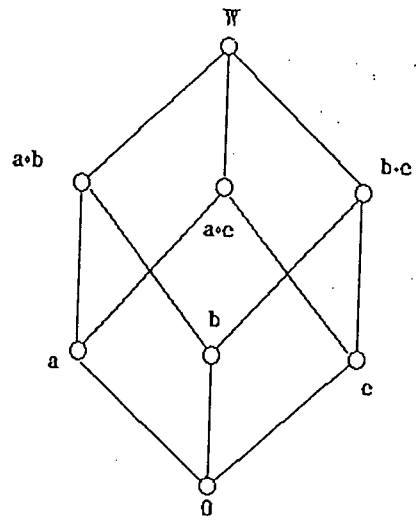

Figure 3.1: Semilattice generated by $< S, \circ, \Box, \diamond >$. In the diagram, W is $\diamond$ and O is $\Box$.

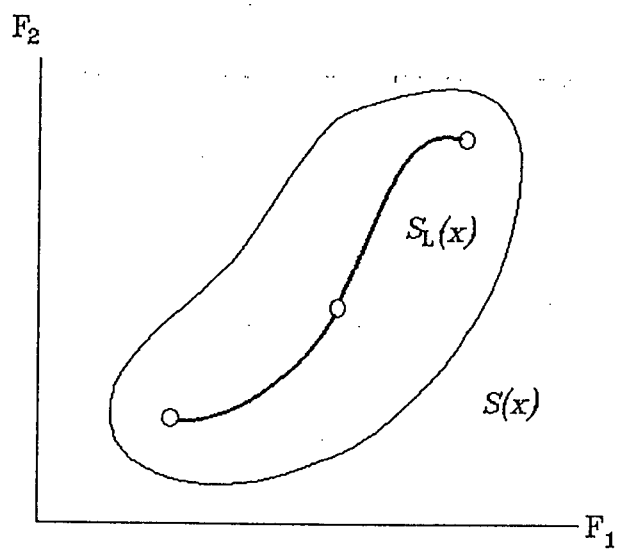

Figure 3.2: Trajectory of a thing undergoing change.

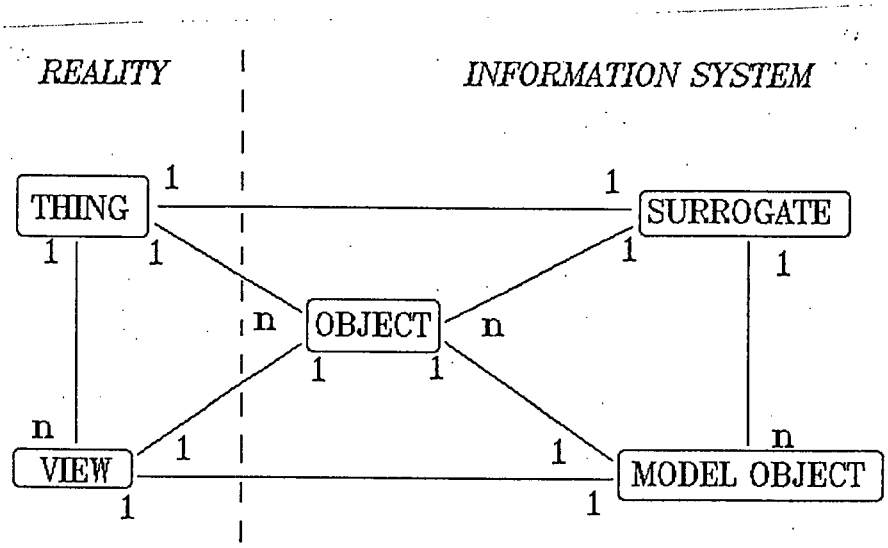

Figure 4.1: Relationships between the Reality being represented and the Information System.

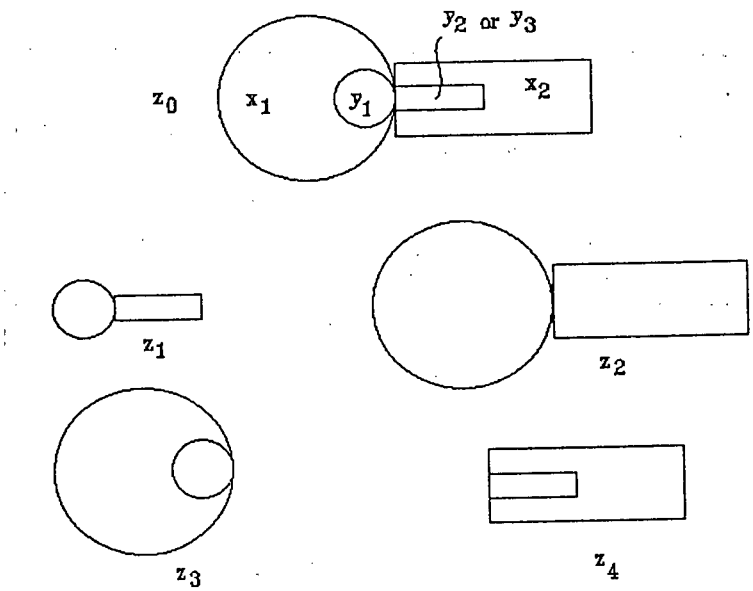

Figure 4.2: Different forms of composition. See Section 4.5.3.

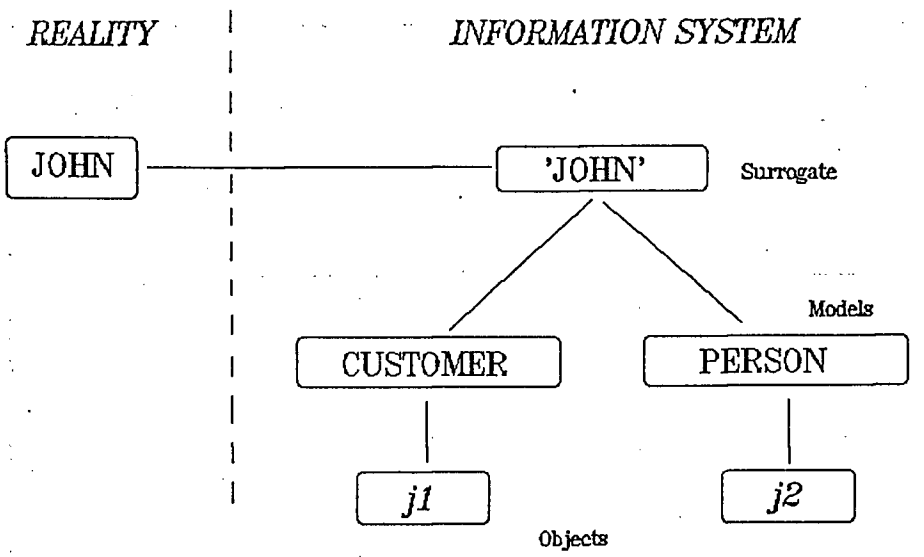

Figure 4.3: Two views of the (real) thing JOHN.

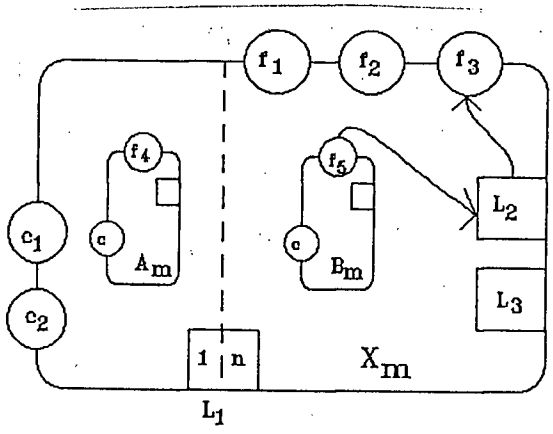

Figure 4.4: A visual notation for objects. The rounded rectangles represent objects, the circles labeled c_i represent change functions, f_i represent state functions, and rectangles labeled l_i represent law statements.

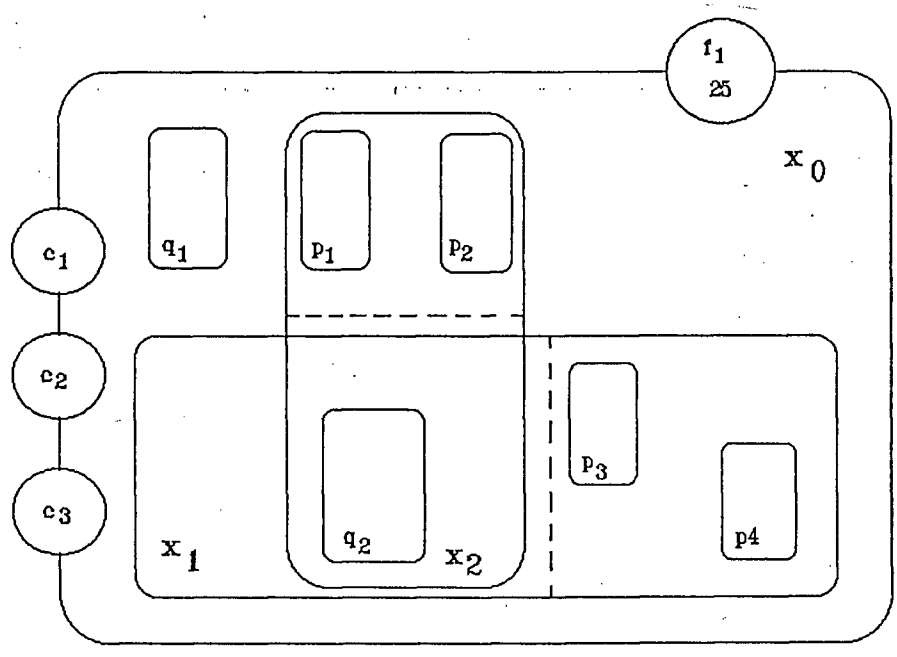

Figure 4.5: A specific object x_0. Note that $q_2 \sqsubset x_2$, $q_2 \sqsubset x_1$, and $q_2 \sqsubset x_0$.

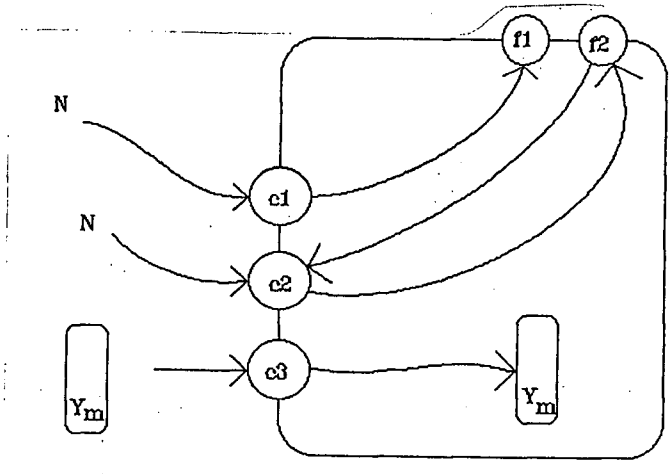

Figure 4.6: A simple notation depicting change.

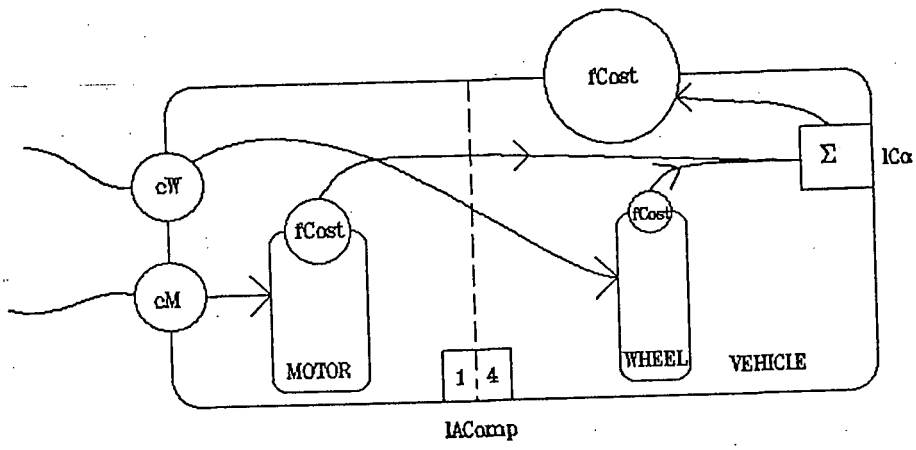

Figure 4.7: The vehicle example in the visual notation of Chapter 4.

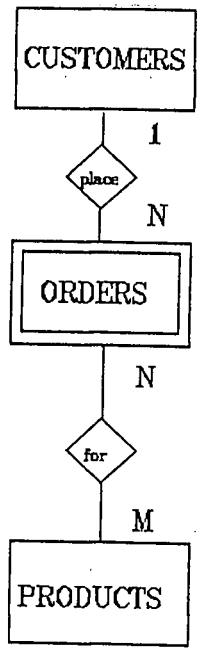

Figure 4.8: Typical Entity-Relationship diagram.

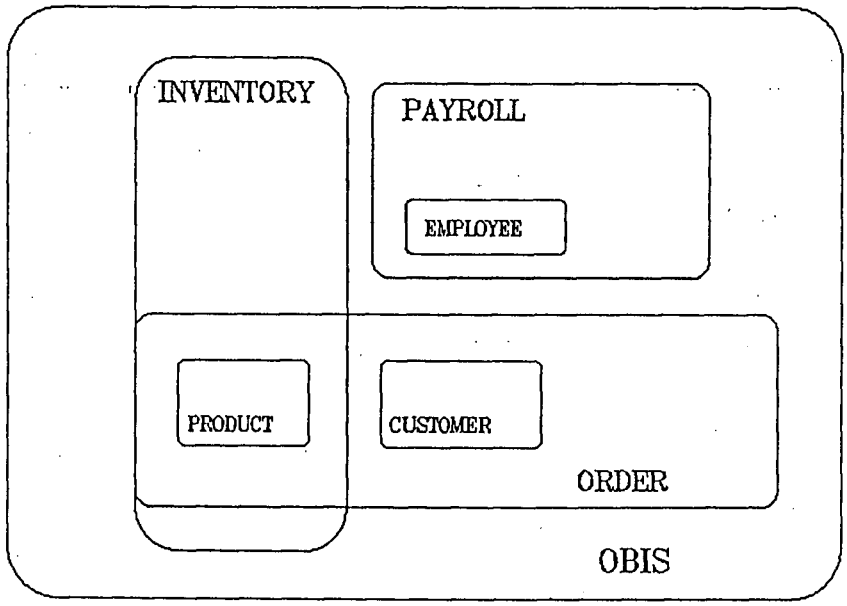

Figure 5.1: OBIS as composition of three objects of type INVENTORY, PAYROLL and ORDER.

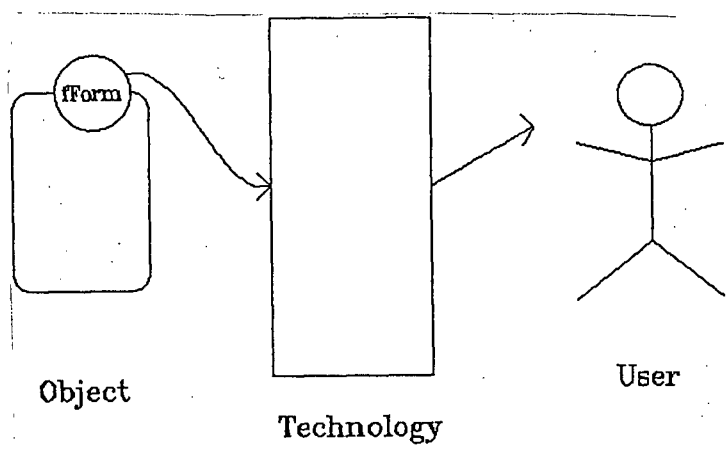

Figure 5.2: Display of objects via a technology.

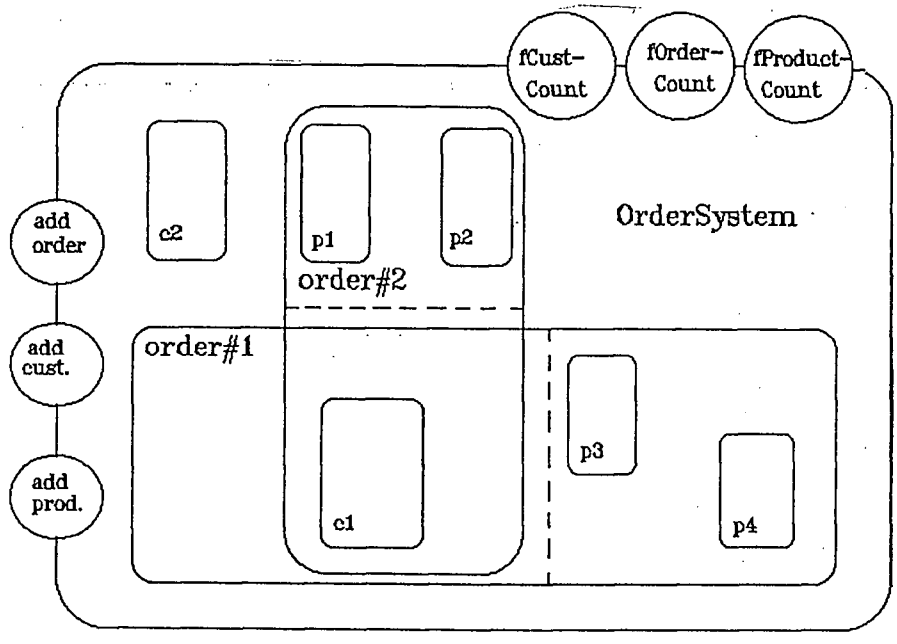

Figure 5.3: *OrderSystem* as composition of orders, customers, and products.

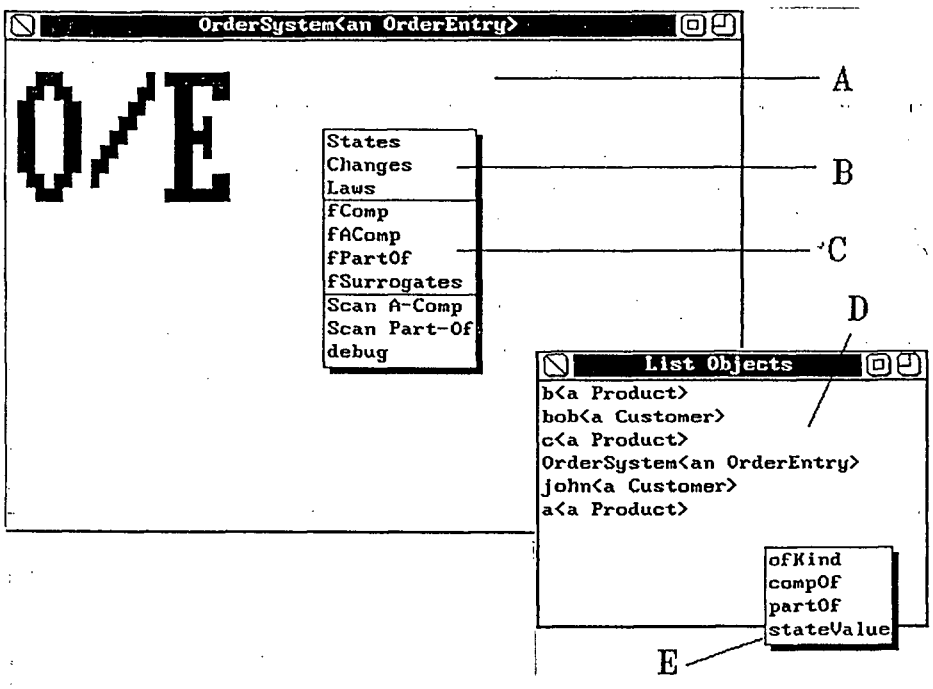

Figure 6.1: A Smalltalk-based OBIS implementation.

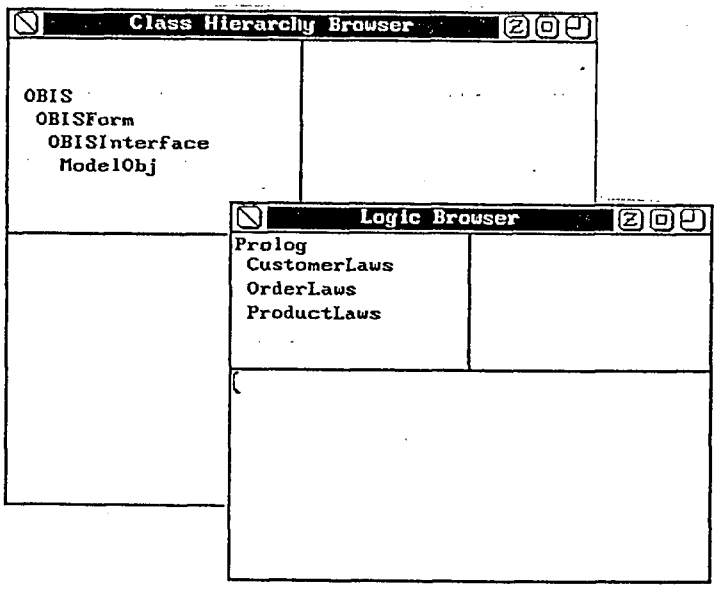

Figure 6.2: The Smalltalk Prototype Shell.

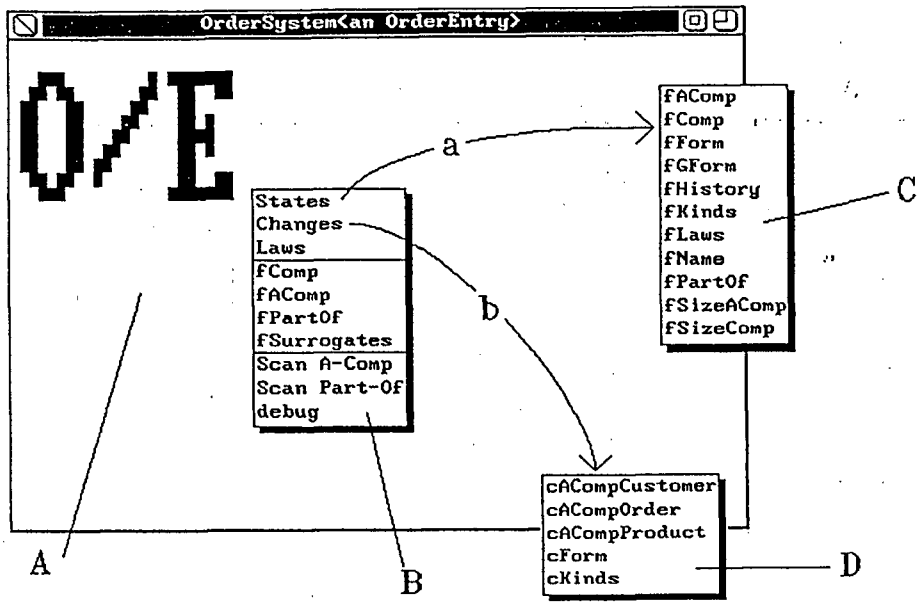

Figure 6.3: A window onto a Smalltalk implementation of the object *OrderSystem*. State functions common to all objects (eg. fComp, fPartOf, etc.) are redundantly listed in a separate service menu for user convenience.

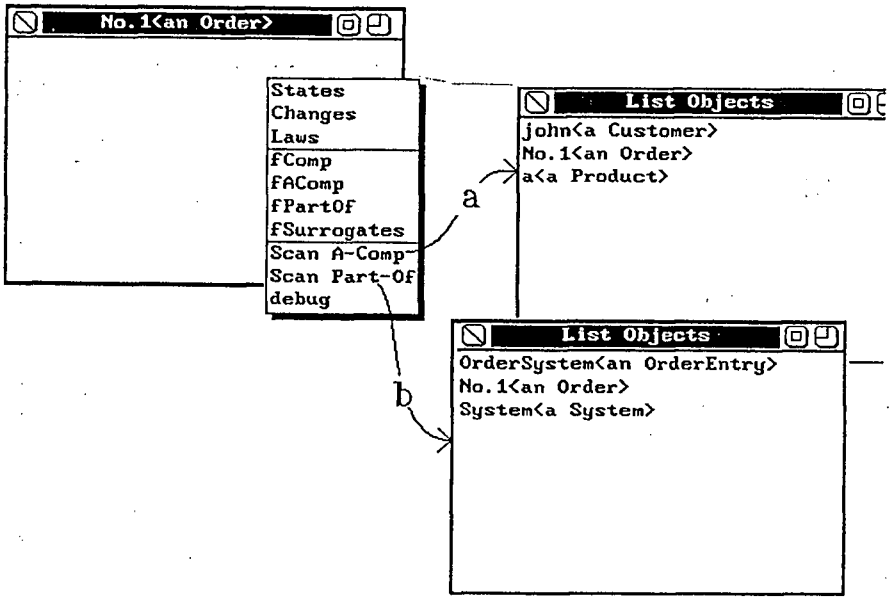

Figure 6.4: Scanning fAComp and fPartOf.

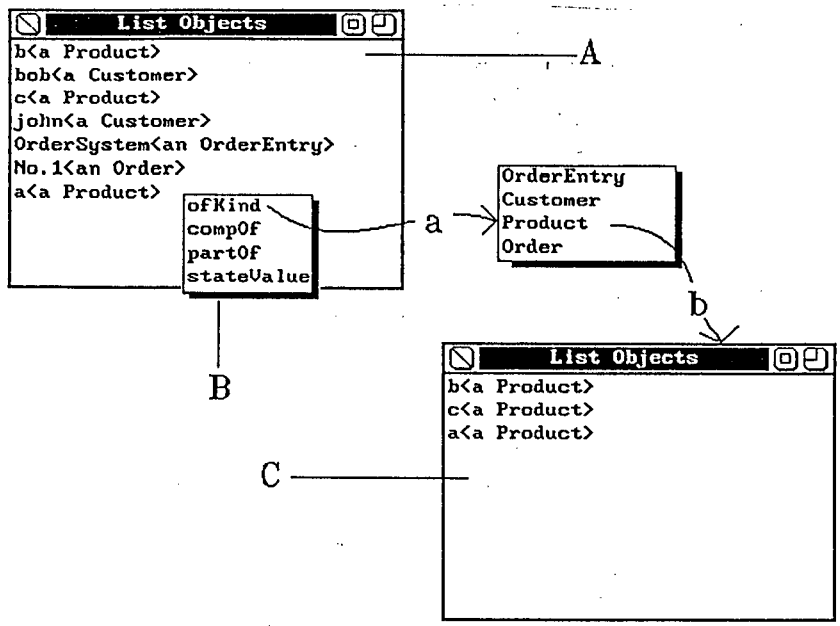

Figure 6.5: Scanning all objects of kind PRODUCT in the composition of *OrderEntry*.

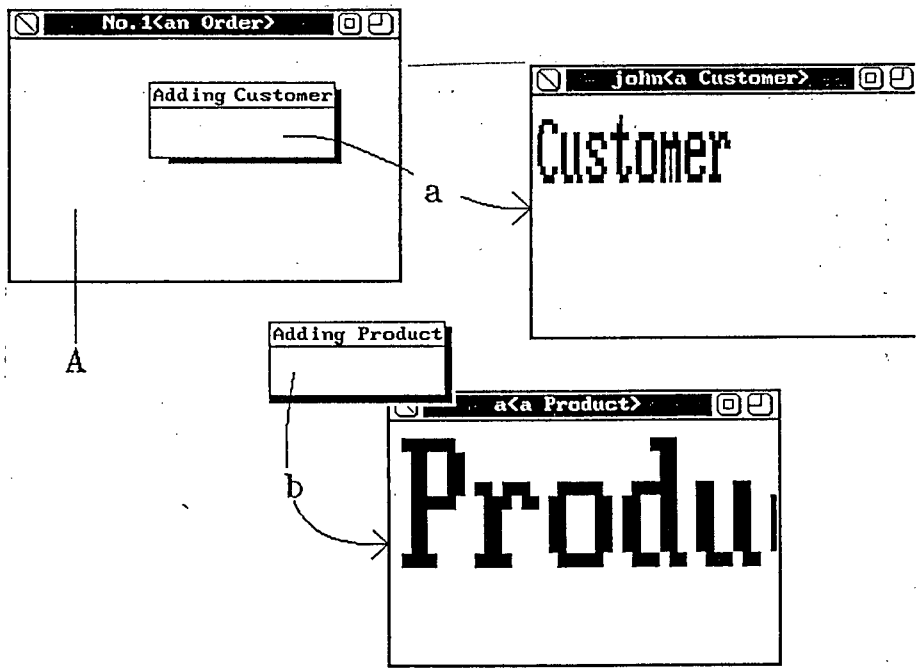

Figure 6.6: Adding customer and product to an order.

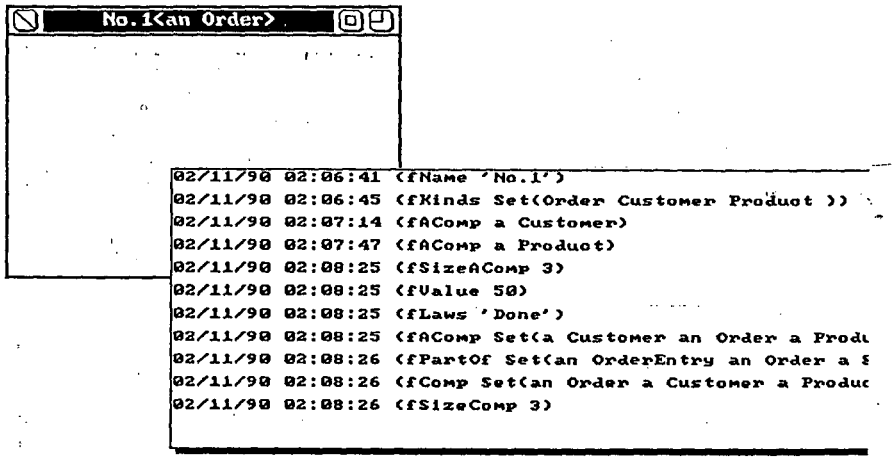

Figure 6.7: History of an object.

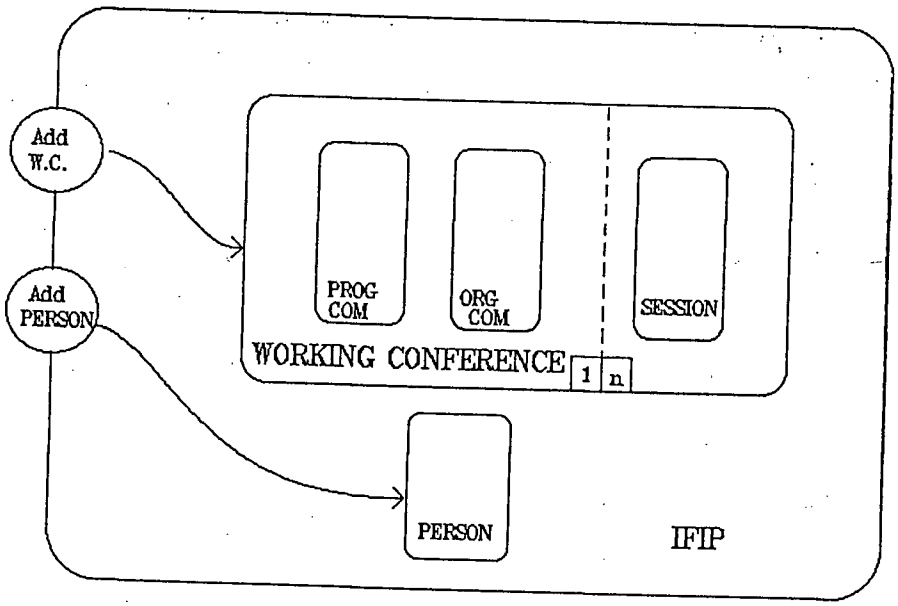

Figure 7.1: Visual notation of IFIP Working Conference Problem.

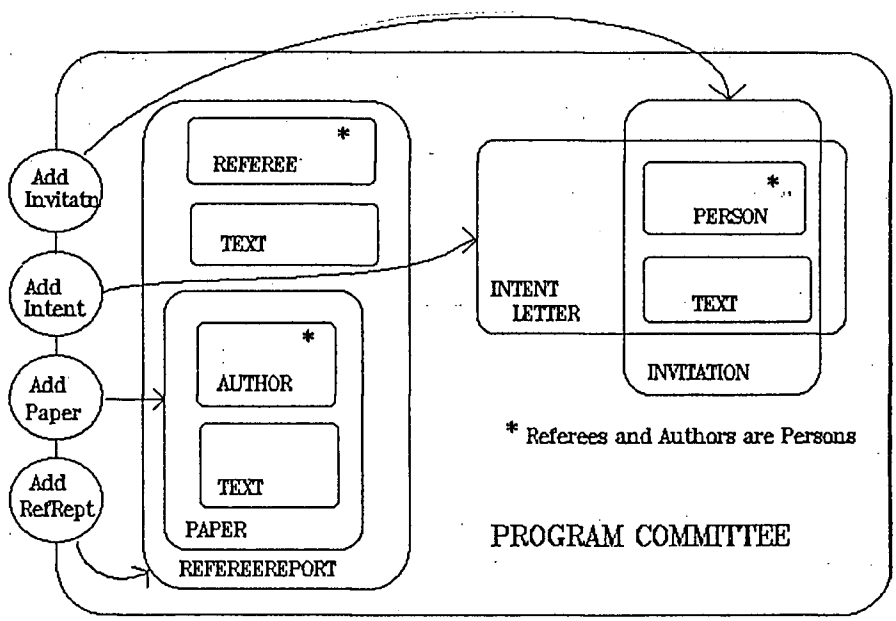

Figure 7.2: Visual notation of PROGRAM COMMITTEE.

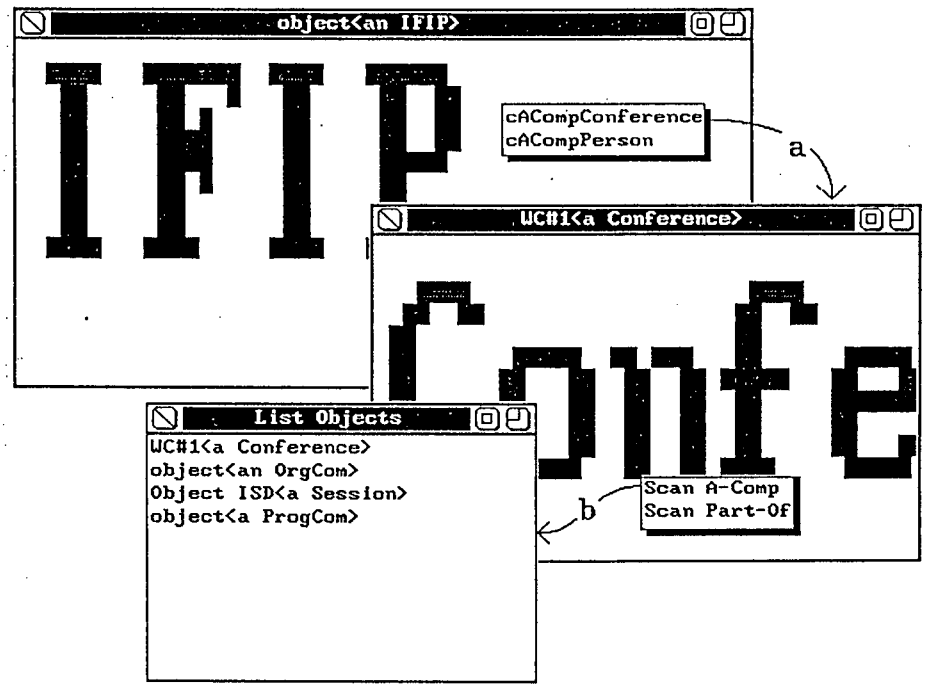

Figure 7.3: Smalltalk implementation of IFIP.

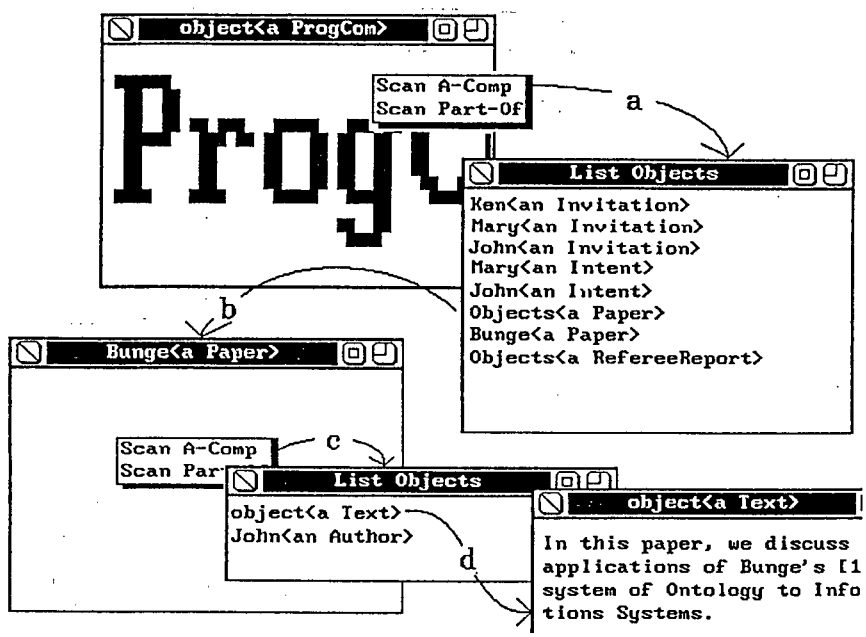

Figure 7.4: Accessing a paper and its text submitted to the Program Committee.

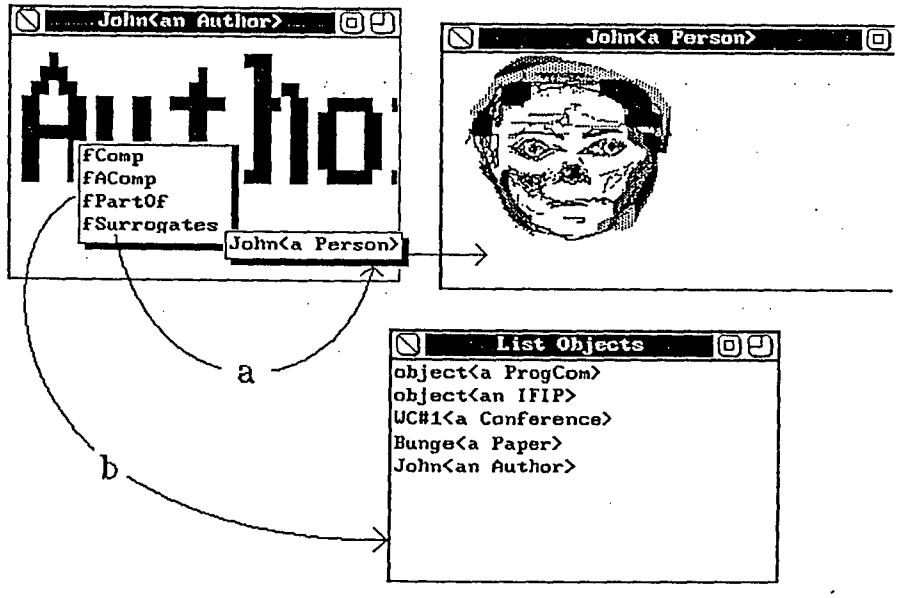

Figure 7.5: Two views of the real world thing JOHN, as AUTHOR and as PERSON.