SEMANTICS OF BEHAVIORAL INHERITANCE IN DEDUCTIVE OBJECT-ORIENTED DATABASES

Hasan M. Jamil

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

May 1996

ISBN 0-612-18404-8

Abstract

Semantics of Behavioral Inheritance in Deductive Object-Oriented Databases

Hasan M. Jamil, Ph.D.
Concordia University, 1996

We argue that powerful models for supporting next generation database and knowledge-base applications can be built by extending semantic data models in the direction of Object Oriented modeling. It is clear that modeling such new applications will require concepts like *modularity, behavioral abstraction, derived schema components*, and *database knowledge* (e.g. *constraints*). Most of the required concepts are already present in these two seemingly parallel models in the form of the notions like object identity, inheritance, encapsulation, methods, virtual objects and classes. but have important differences. We present a conceptual *Abstract Data Model* called the *Object Relationship* (OR) model which reconciles semantic and object-oriented data models and extends the modeling capability with additional concepts like *withdrawal, stratified constraints, methods in relationships*, and *encapsulation.*

We then propose a novel semantics for object-oriented deductive databases as a formalization of the OR model in the direction of F-logic to *logically* account for *behavioral inheritance*, *conflict resolution* in multiple inheritance hierarchies, and *overriding.* We introduce the ideas of *withdrawal*, *locality*, and *inheritability* of *properties* (i.e., methods and signatures). Exploiting these ideas, we develop a declarative semantics of behavioral inheritance and overriding without having to resort to non-monotonic reasoning. Conflict resolution in our framework can be achieved both via specification and by detection. The possibility of specification based conflict resolution through withdrawal allows users to state inheritance preference. We present a formal account of the semantics of our language by defining a model theory, proof theory and a fix-point theory. We also prove that the different characterizations of our language are equivalent.

We finally present an elegant technique to reduce inheritance to deduction based on the idea of *constrained deduction*, called the *i-completion*. The reduction technique makes it possible to implement object-oriented databases with inheritance, overriding and conflict resolution in a purely deductive system. An ORLog prototype implementation on Coral deductive database system is discussed based on this reduction technique. We are able to exploit the rich set of query optimization techniques available in Coral since the implementation does not require meta-interpretation.

Dedication

To My Parents

and

To My Grandfather Abdul Hai

Acknowledgments

I have been fortunate to have Dr. Laks V. S. Lakshmanan and Dr. Fereidoon Sadri as my thesis supervisors. They always encouraged me to pursue the best and strive for excellence. I was introduced to the challenging field of logic and object-orientation by Dr. Lakshmanan and he gave me the foundation I needed to advance in my research. Despite a busy schedule, he always had the time for me, and his guidance and our numerous discussions paved the way for the completion of this thesis. I thank him for taking the time to read the manuscript of this thesis and fixing several bugs. I would like to express my indebtedness to him for his exceptional guidance and inspiration.

Fereidoon gave me the basics of databases and rudimentary details of deductive databases at the early stage of my research. The interest he developed in me later triggered my fascination in deductive object-oriented databases. He then introduced me to uncertainty management in databases, and always encouraged me to explore new frontiers in research. It was a real delight doing research with him in these areas. He always had faith in me, took interest in me and helped me in shaping my professional life. I am grateful to him for everything he did for me.

I was also blessed by two great souls, Dr. V. S. Alagar and Dr. Peter Grogono, during my stay at Concordia. They always gave me the much needed encouragement, and helped me change my views towards teaching, research and life as a whole. I received so much appreciation from them that I remain ever thankful.

I am glad that I came to know Michele Bugliesi from the University of Padova, Italy. We shared interesting ideas and worked as colleagues. I had an exciting and enlightening time working with him and doing insightful research. He presented to me an entirely new dimension of logical systems and helped me develop a deeper understanding of logic programming as a whole. This was instrumental in shaping the research contained in this thesis. I owe to him all that I learned from him and I am thankful that we are friends.

Thanks are also due to Michael Kifer (State University of New York, USA), Rodney Topor (Griffith University, Australia), Kenneth Smith (University of Illinois at Urbana-Champaign, USA), Laura Giordano (University of Torino Italy), and Paul Johannesson (Stockholm University, Sweden) for the many interesting discussions we have had from time to time.

My first year of study in Concordia was made easy by Rajwant Kohli. He took the time to upgrade me to Unix itroff, and explained to me every necessary details I needed to know to survive in a new environment. Ramesh Achuthan, Basudeb Das, Khaled Jababo, Alexander Lakher, Nematollah Shiri, and Iyer Subramanian have been excellent friends. I thank them all for being extremely helpful in so many ways. I thank my dearest friend Mahbubur Rahman for helping me during the worst times of my life in Montreal.

I would like to express my sincere gratitude to the analyst pool of the Computer Science Department and the department secretaries Edwina Bowen, Terry Czernienko, Halina Monkiewicz and Stephanie Roberts for being so helpful and cooperative. Special thanks are due to Stan Swiercz for helping me with Unix, Coral, and WWW. Thanks are also due to Ianina Orenman for working with me for the implementation of ORLog, for which she did an excellent job.

My graduate study at Concordia has been supported by several grants which I would like to gratefully acknowledge. I received financial support from the Canadian Commonwealth Scholarship and Fellowship Plan and from the University of Dhaka, Bangladesh for my entire graduate study at Concordia. I also received stipends for one year and partial support for conference travel from NSERC (Canada) and FCAR (Quebec).

Finally, I would like to thank my family - my wife Nusrat, my daughter Nishat, my parents and my brothers (specially my brother Hasan Zaman) for allowing me all the time I needed to complete this thesis, for forgiving me for not being able to be there when they needed me most, and for supporting me during my thesis work in every way they could - a debt which I will never be able to repay.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

"Object orientation" is both a language feature and a design methodology. The design methodology is likely to have evolved from the simulation approach that was used in Simula [14]. Among others, an important characteristic of objects is that they provide a uniform interface to all components of a system. In particular the interactions between objects occur via a single and simple concept of *message passing* independently of the size or structure of the objects.

Object-oriented systems also provide a form of modularity, independence, abstraction, scope for sharing, etc. These are the properties that are vital for the survival and effectiveness of modern software systems. These systems also provide a "black box" capability by hiding implementation details which is useful for practical large software development and their maintenance. Inspired by these ideas, several successful procedural languages, such as loosely typed Smalltalk [16] and strongly typed C^{++} [72], have emerged.

Motivated by the success of these languages, the logic programming community has expended their best effort in devising an object-oriented logic programming language in the recent years. Despite all the efforts, only limited success has been achieved in this area. However, researchers have identified two major issues that need to be addressed. The first issue is developing a logical characterization of *inheritance* in object-oriented systems. Although the idea of inheritance is simple, it has been found to be one of the most difficult challenges towards the development of an object-oriented logic programming language. The second issue is capturing *encapsulation* in the intended language and this has been found to be another difficult concept to formalize. The concept of encapsulation, however, is in the heart of "abstract data types" on which object-oriented systems rely. Together they, inheritance and encapsulation, form the backbone of object-oriented languages in general.

The investigation into object-oriented issues in a closely related field of logic programming, the deductive databases, is fairly recent. Traditionally, databases are mostly concerned with persistence, storage management, concurrency, recovery and ad hoc querying facility. In the recent years, researchers in the deductive database community have successfully added declarative style of programming and querying the databases to this paradigm. In this declarative environment, users are now free to design and query the database in any fashion without worrying about the cost or efficiency of the system. This means that in deductive databases, this responsibility is delegated to the system.

Researchers in databases widely believe that next generation database systems will require the data modeling capability of object-oriented systems, and they agree in principle that an object-oriented database system must satisfy two criteria – it should be a database management system and it should be an object-oriented system. The second criterion translates into the following eight mandatory features that we need to accommodate in databases – (i) complex structured objects, (ii) object identity, (iii) encapsulation, (iv) types or classes, (v) inheritance, (vi) overriding and late binding, (vii) extensibility, and (viii) computational completeness [9]. Clearly, a deductive object-oriented database would also be required to accommodate all eight of these features.

In this thesis we address the issue of the integration of object-orientation with deductive databases. Clearly part of the challenge is to provide linguistic instruments to capture the eight golden properties mentioned above in a declarative database language. Fortunately, for deductive databases, there has been a commonly agreed data model, namely first-order logic, that nicely maps on to the relational data model. But unfortunately, there is no commonly agreed upon data model for object-oriented databases. Hence, we are required to devise a data model that incorporates features to support these golden properties. Furthermore, we will have to take up the issue of inheritance and encapsulation as faced by the logic programming community, since the problems are identical in these two paradigms.

1.2 Scope of this Thesis

The broad scope of this thesis is to develop a loosely typed logic based object-oriented query language for deductive databases. This loosely typed language would allow static typing and type safety yet allow flexible programming a la Smalltalk.

The specific goal of this thesis, however, is to develop a query language called ORLog and a logical semantics for the seven out of eight mandatory object-oriented features, namely - complex objects, object identity, types or classes, inheritance, overriding and late binding, extensibility and computational completeness. The only component that is not covered by this work is encapsulation. We will, however, revisit the issue of incorporating encapsulation in ORLog at the end of this thesis.

Although we develop ORLog to model the seven features outlined above, we pay particular attention to inheritance, overriding and late binding, and completeness. The other features are captured as a by-product.

But before we proceed with the development of ORLog, we must also develop a data model that can be used as an underlying conceptual tool for ORLog schema design and conceptual modeling. To that end, we develop a conceptual model, called the OR model, that combines features from SDMs and object-oriented programming, giving rise to the notion of abstract data models. Finally, we also present a prototype implementation of ORLog into Coral deductive database system. To this end, we first develop an encoding technique to translate ORLog to Coral and an inheritance reduction technique to be able to implement inheritance in a purely deductive system.

1.3 Organization of this Thesis

We organize the presentation of this thesis as follows. We present a complete theme in a single chapter. Since we address several unique issues that have distinct properties and features, we delegate the discussion on their background and the related research in that area in the corresponding chapters. Hence we do not include a separate chapter on related works and motivations in this thesis. Whenever appropriate, we include our comments at the end of a particular chapter.

Keeping the above in mind, in Chapter 2, we develop and present our abstract data model called the Object Relationship model, or the OR model. Then in Chapter 3 we present our language ORLog that embodies the semantics of all the seven mandatory features, and behavioral inheritance in particular. ORLog is a logical formalization of the OR abstract data model. We discuss a translation technique of ORLog to a first order language in Chapter 4. In this chapter, we also present an inheritance reduction technique based on completion that together with first-order translation enables us to implement ORLog in an existing deductive database system, such as Coral, as an object-oriented front-end. Finally we also report and discuss a prototype implementation in Chapter 5 based on the reduction technique presented in Chapter 4. In Chapter 6 we present a critical review of contemporary research in logical object-oriented paradigm in view of ORLog and discuss issues related to ORLog implementation. We then summarize and give our conclusion in Chapter 7 and discuss future research.

1.4 Credits

Different parts of the work reported in this thesis have been carried out in collaboration with various people. The purpose of this section is to acknowledge their contributions and attribute credits.

The conceptual OR data model described in Chapter 2 is a joint work with Dr. Lakshmanan and Mona Soliman. The logical foundations of ORLog, its syntax, declarative, fixpoint and proof-theoretic semantics presented in Chapter 3, and the reduction of ORLog inheritance to deduction discussed in Chapter 4 as well as the results in these chapters were established in collaboration with Dr. Lakshmanan. Finally, implementation of ORLog in Coral was done jointly with Dr. Lakshmanan and Ianina Orenman.

Chapter 2

OR Model: An Abstract Data Model

In this chapter we shall present our *Object Relationship* (OR) model. We will first outline the broad objectives of this new data model. Then we will describe various building blocks, or semantic constructs, we use in our model. While some of the constructs are adapted from other semantic data models, several new ones are introduced to increase the modeling capability of the OR model. We then discuss global restrictions on the building blocks in order to be able to use these constructs in a coherent manner for database schema design. We also propose a notion of *admissible databases* which captures the intended semantics of a meaningful database by imposing further restrictions on the semantic behavior of the model. We will also identify its strengths and weaknesses by comparing it with contemporary data models as we describe our model. But before we proceed, we present a survey of existing data models for object-oriented databases.

2.1 Related Research and Motivation

Programming with taxonomically organized data which are encapsulated with operations has led to object-oriented programming. The object-oriented programming paradigm is not only able to handle and manage complex and hierarchically structured data, it is also able to express the dynamic aspects of data with the introduction of the concept of behavior and methods. The notion of inheritance in object-oriented programming is intimately related to re-usability and sharing. The notion of encapsulation brings modularity and implementation independence to this paradigm.

It is believed that combining object-oriented concepts with databases will result in clear benefits [77]. There is currently significant interest in object-oriented database systems. Much of the research on OODBs was sparked by the need to provide adequate database support for next generation database/knowledge-base applications such as engineering design, software development, VLSI design, etc. In general, such applications necessitate features such as complex objects, object identity, behavioral aspects of objects, re-usability (of designs or software code), and modularity and implementation independence of subparts of the (software) design objects.

With the advent of new database applications such as geographical, engineering design, multi-database, office automation, etc., the database modeling requirements have dramatically changed. These modeling applications demand sophisticated abstraction mechanisms to capture complex organization of data, intricate inter-object relationships, means for variety of constraint specifications, etc.

Similar to object-oriented systems, semantic data models[1] (SDM) are very powerful in specifying database structures and schema at varying levels of abstractions, and hence have a strong potential to be the object-oriented data model. SDM features have been found to be extremely useful in the modeling and development of data and knowledge based systems, as a design technique and documentation tool. SDMs are also very powerful in expressing the relationships among data and their semantics at a higher level of abstraction. Several forms of constraints can also be expressed in these models. As opposed to behavioral encapsulation in object-oriented paradigm, semantic models tend to encapsulate structural aspects of objects. In a different way they are also capable of modeling complex structures, derived schema components, type constructors, inheritance, etc. SDMs tend to hide the implementation specific details while focusing on the logical organization of the databases. In that way they serve as a great conceptual design tool as required by the three level architecture of databases.

Although current SDMs offer a rich set of modeling features, they lack some of the desired properties for an object-oriented model. Since the object-oriented paradigm already possess some of the useful features essential for next generation data

[1]See [63] for a detailed survey of SDMs. A detailed reading and a tutorial on SDMs may also be found in [39].

and knowledge base applications, it is believed that semantically rich and expressive conceptual models can be built for future applications by extending the functionality of the SDMs in the direction of object-oriented programming. The resulting model would likely become similar to abstract data types giving rise to the concept of *abstract database models* (ADM).

The idea of abstract data types (ADT) has been extensively used by the object-oriented (OO) programming and database community during the recent years. The need for programming with taxonomically organized data that are encapsulated with operations, and are arbitrarily complex structured has led to the concept of object-oriented programming. The OO paradigm is not only able to manage complex and hierarchically structured data, it is also able to express the dynamic aspects of data with the introduction of the concept of behavior and methods. The notion of inheritance in OO programming made it possible to maximize sharing of objects and re-usability. The notion of encapsulation has given this paradigm the strength of modularity and implementation independence, and hence provided a higher level of abstraction. The introduction of behavior and methods provided another dimension of logical independence and allowed the user to view objects in terms of the characteristics exhibited through their interfaces. Inheritance in OO languages allow sharing, and re-usability of software components, while specificity is facilitated by the idea of overriding. While encapsulation of methods enhances the idea of physical independence by hiding the internal structure, data, behavioral implementation, etc. from the outside world, inheritance provides implementation independence as an orthogonal mechanism. In all, the ADT is the key to all these nice features.

We, among others, believe that combining OO concepts with databases will result in clear benefits. There is currently significant interest in various forms of OO database (OODB) systems. Researchers have made a significant stride during the last few years in combining these two concepts. Among others, the works reported in [38, 37, 27, 35, 5, 34, 17, 70] are some of the representative ones to which we will focus our attention.

The Data and Knowledge (DK) model [38] attempts to extend the ER like models in the direction of OO paradigm and tries to capture dynamic behavior, meta-knowledge about data and constraints on the data in a simple setting and appears to be the right approach towards a rich modeling tool. But it lacks quite a few features that are essential for the purpose of abstract database modeling. Namely, it only uses aggregation [2] type objects and is thus value based. Hence it lacks the strength of modeling complex objects and obviously other important features. The DERDL model [37] also captures a few features in our direction. Particularly, it proposed the idea of dynamic objects and relationships, and similar other dynamic properties of database that is close to the spirit of derived schema components in SDM [36, 63, 39]. Since the underlying concept is value based, it also has the shortcomings of the DK model.

The OBJECTModeler presented in [27] attempts to capture user's view of the data and takes an OO approach. Similar to ER model, it emphasizes on type constructors instead of attributes and functions. It uses aggregation and grouping as the abstraction mechanism. However, it does not capture the true spirit of OO modeling and the idea of object based entities. That is the underlying model is still value based. It somehow comes close to the spirit of network data model. Although it allows multi-valued attributes, it fails to capture data relationships in such attributes (m-m relationships). Other shortcomings include the dynamic schema components, parametric attributes, and other several OO features.

In [35], Object Behavior diagrams are proposed to model objects and their behaviors and mostly concerns itself with behavior specification of object models. The schema representation in this model seems to suffer from similar draw backs as it was in the case with OBJECTModeler. Overall, the model seems complex, and description of methods and its associated behavior seems to have no clear mapping.

The model proposed in [5] is heavily biased towards OO modeling, and concerns mostly with constraint management, and so does [34]. We do not consider them as general modeling tools since they do not capture several essential features we deem necessary for an abstract database model. But they do introduce a rich set of constraint specification and maintenance tools. The CASE based model proposed in [17] is a semantic network based model developed specifically for the OO language O_2 [53] and Morse. It has limitations, in our opinion, even as an OO modeling tool. It is an example of DBMS dependent data model. It also proposes an automated design procedure of OO database schema from abstract specifications of the models.

To model a wide range of future applications such as engineering design, biological and geographic database, etc. a generalization of several data modeling tools that are currently available is required. The model should also be capable of providing support for *modular schema design*, and as an aside, should support *schema evolution* in a natural way. The experience with semantic and object-oriented data modeling and the needs for future database modeling applications suggests that an abstract data model should particularly possess the following properties: (i) it should be simple, expressive and versatile, (ii) should serve as a schema specification, documentation and communication tool among various levels of users, (iii) it should be able to capture user's view of the enterprise data, (iv) it should naturally support a user-friendly graphical user interface for schema description and query processing, (v) it should have a simple but rich set of high level abstraction mechanisms, (vi) it should be able to model abstract data types, inter-object relationships and their associated constraints, (vii) encapsulation should include both the behavioral and structural aspect of data, (viii) the model should emphasize on type constructors rather than attributes for modeling inter-object relationships, (ix) should allow value as well as and id based objects, etc.

The semantic models in their present form are capable of capturing *structural encapsulation*, notions of *abstract objects, derived schema components, type constructors* e.g., set objects, relationships, complex objects, etc.), *is-a hierarchy* of object classes, several forms of *constraints*, etc. The OO models on the other hand, support objects of *abstract types*, and *classes*, *methods*. *messages*, *object hierarchy*, *(non)monotonic* and *multiple inheritance*, *behavioral encapsulation*, etc. OO models also incorporate a number of constraints similar to update constraints which are unique to OO models (e.g. *update dependency relationships*). The similarity in the set of concepts in these two class of models suggest that an integration of this two concepts is possible.

In the next sections, we present our *OR* model. We start by defining the goals of this model that includes a balanced mix of the features available in SDMs and OO models. It also incorporates a limited form of security features [12] in the form of privatization of methods and attributes and the notion of legal access to objects and methods.

2.2 Objectives of the OR Model

We extend the concepts of the current SDMs in the direction of OO paradigm to facilitate modeling of taxonomical and hierarchical data that are encapsulated with operations, and thus enrich SDMs with modeling features available in the OO paradigm.

We incorporate the notions of objects (defined shortly), entities and relationships (in extended ER sense), and values in our model as basic building blocks. We stress type constructor based inter object associations (as in extended ER models) - called *relationships* as opposed to attribute based associations (as in FDM) – called *inversions* [36]. We extend the framework of inheritance in OO models by introducing the notion of *withdrawal* which allows us to model several applications very naturally that we could not model using current approaches. We adopt in our model a subclass instance relationship scheme different from many others. In principle we do not organize classes of objects on the basis of their types, rather on the basis of the semantic relationships of the classes as viewed by the user (user defined subclass relationships). This is consistent with the applications in the OO domains - such as engineering design, and biological databases, etc. This idea has its roots in the definition of SDM classes. The existing notion of *derived schema components* in SDMs is extended to incorporate the notion of methods and messages in OO paradigm. The notion of encapsulation is also incorporated in our model, and we give a clear definition of this notion in the context of abstract modeling. The notion of constraints in SDMs is extended by organizing them in strata (*global, local,* and *inherited*) that gives us a clearer semantic meaning of constraints. This also buys us the power of resolving constraint conflicts, easier constraint enforcement and maintenance schemes.

Finally, it is our intention to make the model expressive with a rich set of constructs, constraints and knowledge modeling tools while keeping it simple and easy to use. We take a diagrammatic approach to support a user-friendly schema design interface with a small number of constructs. We view our model as a documentation and communication tool among people in an enterprise at different levels. We also believe that our model is also capable of capturing *user view* of the data(base). Due to this uniformity it eliminates the need for a mapping from the user view of the data to the corresponding conceptual view.

2.3 Basic Constructs

In this section we describe the basic constructs we use in our model and give their local semantics. In the next section we will then impose restrictions on the use of these constructs as well as define their global semantics to be able to design meaningful database schema. Figure 1 shows a partial McDonnell Douglas Aircraft Design Database schema using *OR-diagrams.* The features in the OR model are represented using OR-diagrams, which play a role analogous to that played by ER-diagrams for the ER model. The legends in Figure 1 shows the meanings of each of the symbols used in the OR example.

2.3.1 Objects

An *object* is a distinct real world entity with a unique identity. All objects have intrinsic characterizing properties. Properties of an object help distinguish the object from other objects. An object may be *concrete* or *abstract.* An object is concrete if the object can be described solely by its own intrinsic properties. These objects are the so called *printable objects* in many semantic models and are strings of alphanumeric characters. In OR model these objects are called the basic objects.

Abstract objects on the other hand, are entities that are characterized by properties described using other basic or abstract objects. A notable distinction is that such an object is almost always an abstraction of the corresponding real world entity and is thus incomplete, while it possibly represents the object adequately for modeling purposes. Hence an abstract object can be viewed as a collection of representative properties of the real world counterpart. For example, *aircraft* and *seat* are abstract objects (types). Note that, in OR model a rectangle is used to denote an abstract object. The name of the object is written inside the rectangle above the horizontal line. Shortly we will see other types of objects.

We say that an object is *value based* if a change in properties changes its identity. In OR model we call them *entity.* On the other hand an object is *identity based* or *object based* if its identity is independent of its properties. Such objects are actually identified by a surrogate, called an *object id* or *oid.* We call them *semantic objects.* In most of the semantic models and object oriented models, it is one or the other types of objects that are allowed - i.e., the entities or the semantic objects. In OR model, we allow a free mixing of both kinds of objects, and will make no distinction between them unless it is necessary. This uniformity in the OR model, gives us the capability of modeling heterogeneous databases, and design applications that require multi-model database integration.

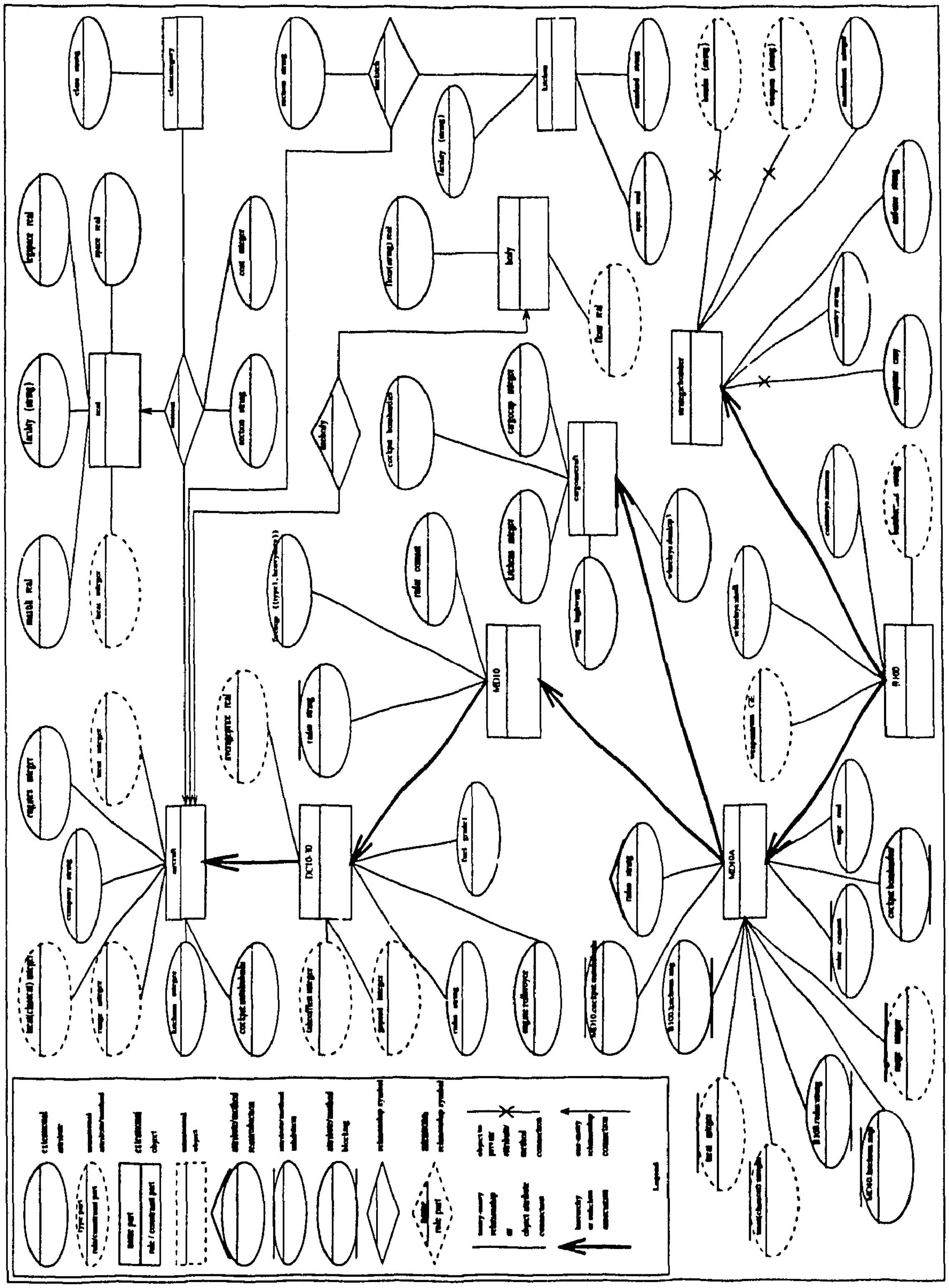

Figure 1: Partial McDonnell Douglas Aircraft Design Database Scheme.

The characterizing properties of abstract objects are called *attributes*. As opposed to the object-oriented practice, properties are grouped around objects and only semantically relevant properties are attached to an object as is the case in relational model. Objects store data in attributes in order to save their states, and respond to instructions for carrying out specific operations on their attributes when a request for an operation, a *message*, is received by an object. A message specification always includes a context (i.e. a recipient) saying which object should receive and respond to a message. The set of attributes that can be accessed or the set of operations that can be performed on an object are determined by the object. Thus an object exhibits its *behavior* in terms of these attributes and operations. In the example of Figure 1, *aircraft*, *DC10-30*, *seat*, etc. represent (classes of) objects. The physical identity of an object is hidden from the user if the object is a semantic object. In case of entities, a set of atomic attributes uniquely identifies an object in its class – a *key* and is thus available to users. Semantic objects in OR model exhibit a duality — as a *class* and as a simple object, called an *instance*. Similar kinds of objects may be grouped into classes of semantic objects or entity sets of entities. Unlike other SDMs (e.g., [36]), we do not distinguish between attributes that describe properties of class objects or member objects. We will take up this two issues again later in this se 'on.

2.3.2 Types of Objects

Similar to other semantic data models, OR model also has the provision for direct representation of object types distinct from their attributes and sub- or supertypes. Concrete objects in OR model are said to have *basic types* and abstract objects have *constructed types*. They both correspond to the *atomic types* in SDMs. However, we do not consider abstract types to be atomic, rather we view them as *complex*[2] since they may have arbitrary nested structures. Our model supports *aggregation* type [39], and *grouping* (or *association*) type [39] constructors to create new types of objects from the basic and constructed ones. An aggregation is a composite object constructed from other objects in the database. For example, each object associated with the aggregation type *useseat* in Figure 1, is an ordered triple of *aircraft*, *seat* and

[2]In this thesis we do not make a distinction between complex objects and composite objects, and thus use these terms interchangeably. However, Kim et al. [48, 49] makes a clear distinction between them by bringing in *is-part-of* relationship between objects – a construct that we do not incorporate.

classcategory type objects. Mathematically, an aggregation is an ordered $n-$tuple. An instance of an aggregation type will be a subset of the Cartesian product of the active domains assigned to the participating nodes in it. Note that the identity of an aggregation object depends on the component values of the object. Also note that in the sense of ER model, entities can be viewed as an aggregation of basic types, while a relationship is an aggregation of at least two or more constructed types. We also take the same view and treat relationship types to be different from the entity types and say that all entities are persistent objects while the objects in relationships are not. It is easy to see that aggregation types (entities and relationships) in OR model are value based. We denote semantic object types and entity types using a rectangle in the OR model whereas the relationships are denoted by diamonds.

The grouping constructs allow us to model sets of objects of the same type. Mathematically speaking, a grouping is a finite set. An instance of a grouping type will hold a finite subset of objects of the active domain of the node of the grouping type. A grouping object always has exactly one child (*facility* in *seat*). The identity of a grouping object is determined completely by that set.

Most of the type constructors of OR model may be applied recursively and numerous kinds of types can be constructed using these simple constructors. But soon we will see that OR model only allows certain kinds of types by imposing a global set of restrictions on the type of applications of the constructors.

2.3.3 Domains and Values

In the OR model, there are two types of *values* - *basic values* and *constructed values*. Basic values are essentially basic objects and are atomic elements of the corresponding basic domains such as **integer**, **real**, **string**, etc. Many of the existing OO models (e.g., see O_2 model [53] and F-logic [47]) treat basic values also as objects, some models (e.g., O_2) even allow oids for basic values. We believe this is unnatural and make a clear distinction between them. Basic values are just values, distinct from other objects, without any surrogates or ids, and are naturally treated as just values. Unlike objects they do not participate in any class hierarchy (described next). On the other hand, abstract objects (their surrogates or ids) are values of constructed types. Thus, the value that can be associated with an attribute may be basic, or structured. For example, the value of *engines* in *aircraft* is basic, the value of the attribute *weapons* in the object *strategicbomber* is set structured, the value of *engine* in *DC10-30* refers to another object of type rollsroyce and thus is of constructed type, while the attribute *fireeqp* in object *md10* is of multi-type constructed value. That means each of the values of the attribute *fireeqp* belongs simultaneously to the types *type1* and *heavyduty*.

2.3.4 Attributes and Methods

Objects (except basic objects) are described using attributes which are essentially named data items describing the properties of an object. For example, in Figure 1, *company* in *aircraft*, and *takeofflen* in *DC10-30* are attributes. There are two kinds of attributes in the OR model. The kind of attributes that store data explicitly in the database are known as *extensional*. *Intensional* attributes have a procedure associated with them in order to compute the value of the attribute at run time. The language of implementation of intensional attributes in OR model is a rule based language (e.g., first-order logic or predicate calculus) which has a clear syntax and formal semantics. For example, in the object class *aircraft*, *company* is an extensional attribute, while *tseat* is an intensional attribute. The latter represents the total number of seats in an aircraft. This is dependent on the floor space left after placing the *kitchens* in the aircraft and on the type of seats that are being used in different accommodation classes in the aircraft. Since the values of intensional attributes are computed using a procedure, in terms of functionality they capture the concept of *methods* in OO models and the *derived schema components* (derived attributes) in SDMs.

Methods have an underlying *context*, which is the object where they are operative, and a set of typed input arguments. Thus, a message should have a matching context and a set of arguments in order to be accepted by a method. An accepted message leads to a successful invocation of a method (procedure). The implementation of a method is hidden in the object where it is defined. E.g., *tseat* is a method in the object class *aircraft*. In OR model it is not required that an attribute name be unique in a family of classes as it is essential in some of the models (e.g., [36]). This allows OR model to capture the notion of method polymorphism that almost all the OO models have. We, however in general, do require that the *signature* (described shortly) of each method be unique in a family of classes that are related via inheritance or is-a association. The value of an attribute may be null, a basic value, a constructed value, or a multi-typed value. An attribute can be either single valued or set valued.

Speaking differently, attributes are the means of relating two or more types of objects. Hence, an attribute in the OR model is viewed as an n-argument partial function from an n-tuple of types to another type, where $n \geq 1$. The object in which the attribute is being defined is called the context and acts as the first argument of the n argument function. Unlike other semantic models, we impose a type restriction on the domain and range types of these attribute functions. We allow only basic and constructed types in the domain, and all but aggregation type are allowed in the range. The function may be single valued or set valued. SDM supports a feature called *type attribute* which is an attribute associated with grouping type that keeps a kind of count, or other statistics, on the grouping types (e.g., cardinality of the set). We do not support this feature because we support other feature that may augment the functionality of type attributes. We also do not anticipate any natural use for such feature in our framework. Moreover, we do not allow grouping type database objects in isolation other than as a value of attributes and methods.

As mentioned above, all attribute functions are said to have a type. The type of an attribute function f is of the form $\tau_1 \times \ldots \times \tau_k \rightarrow t$ if it is single valued, or $\tau_1 \times \ldots \times \tau_k \twoheadrightarrow \{t\}$ if it is set valued. If the function is single valued, then each tuple in the domain is assigned exactly one object in the range of the expected type, otherwise it is assigned a set of objects from the active domain of the range type. It is also possible to assign a type to f of the form $\tau_1 \times \ldots \times \tau_k \rightarrow \{t_1, \ldots, t_m\}$, where it means that f is single valued but the object in the range must belong to a set of domains $\{t_1, \ldots, t_m\}$ simultaneously. Ideally, t_1 through t_m must be constructed types, because a basic object can not belong to more that one domain at a time. This is called a multi-type single valued function. Similarly a multi-type set valued function has the type of the form $\tau_1 \times \ldots \times \tau_k \twoheadrightarrow \{t_1, \ldots, t_m\}$. For example the attribute *fireeqp* in *md10* is such a function. It says *fireeqp* is a multi-type set valued function of type md10 $\twoheadrightarrow$ {typel, heavyduty}.

Hence, unlike other SDMs [36], we allow an attribute name to accept arguments and thereby forming an aggregation of different types. In this way OR model supports *relation valued attribute* since these functions represents $n + 1$-ary relations. Alternatively, it is clear that there is a close relationship between a one-argument attribute whose range is an aggregation type and an n-argument attribute. By allowing constructed type in the range of an attribute function, OR model supports

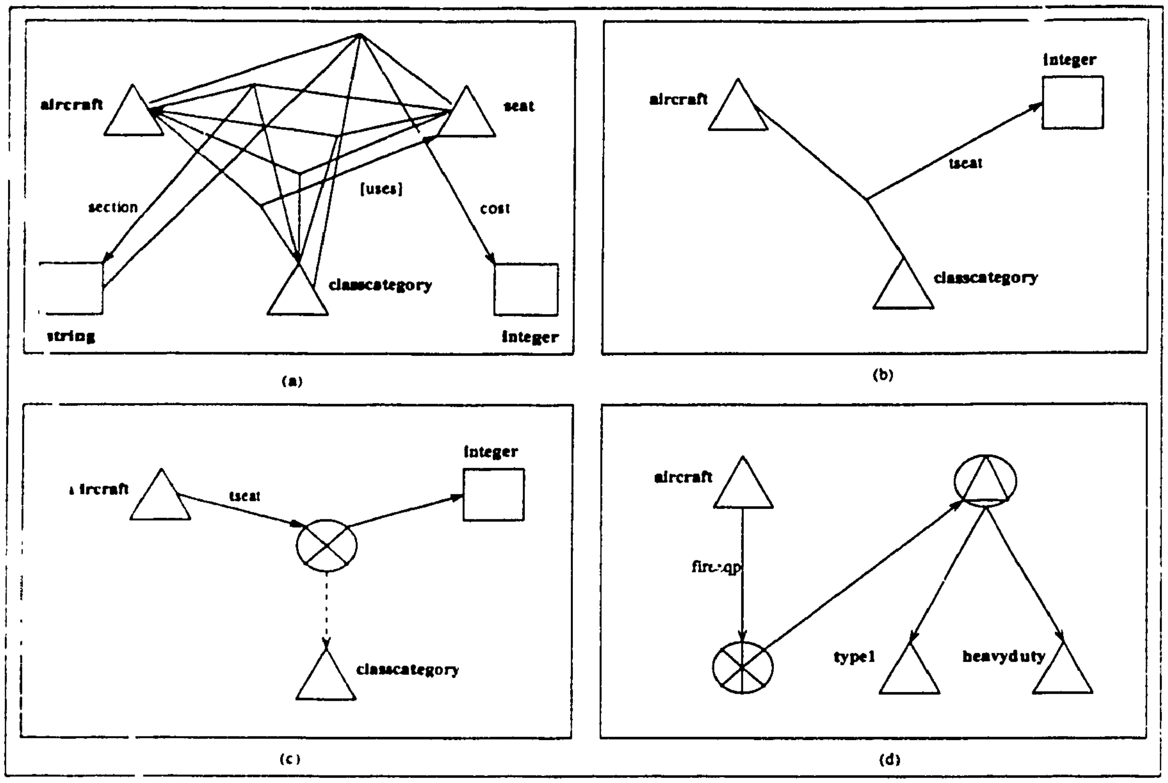

Figure 2: Alternative approaches to attribute based inter-object relationships.

complex typing (aggregation type) of objects.

Some models (e.g., SDM, FDM, F-Logic, etc.) capture inter-object relationships using object attributes based on the notions of *inversion* and *matching* [36]. Although inversion and matching is possible in OR model, we do not view an attribute as a relationship mechanism among objects or entities, and we do not emphasize inter-object relationships using either of these primitives. In OR model we emphasize type constructor based inter-object relationships. The use of type constructors allows information to be associated directly with schema abstractions. As an illustration, consider Figure 2 that uses standard SDM notations [36].

In Figure 2 (a) the relationship between *aircraft*, *seat* and *classcategory* is shown using inversion through the family of attributes [*uses*]. A version of *uses* is defined in each of these objects to capture the ternary relationship. Consider the same relationship implemented in Figure 1 using the aggregation type constructor – the relationship *useseat*. In models like OR, that stresses type constructors, relationships between types are essentially viewed as types and thus they are allowed to have attributes that further describe them [39]. In Figure 1, we have allowed attributes *cost* and *section* in *useseat*. Contrast this with its inversion counterpart in Figure 2 (a). In this case a family of attributes [*section*] are needed to be defined in each of the objects (shown partially). Similarly, to capture the relationship between *cost*, *section* and others, a similar set of attributes has to be defined. This is a major obstacle in models (e.g., SDM, FDM, F-logic, etc.) which stress inversion as inter-object relationship mechanism. The users either have to decide on the types of anticipated relationships or have to define all possible sets of inversions. Some models such as SDM and SmallTalk do not support an explicit type constructor and hence have no choice but to model inter-object relationships using attribute inversions only. It also implies that in such models, value based objects, or entities are not possible.

In [68] similar observations have been made which supports our view point of not stressing inversion as a relationship primitive. In [68] it is argued that enforcement of inversion and matching constraints are not efficient particularly if it is enforced using methods. This is because the constraint is spread over multiple objects and the objects concerned must communicate through messages back and forth to enforce the constraint or to take corrective measures in case the constraint is violated possibly after a long chain of method invocations. Moreover, such constraints must be coded outside the objects concerned to have a global view of the constraints or in the method implementations sacrificing modularity and abstraction. This result supports our position of not using inversion as an inter-object relationship primitive. Thus we stress that attributes should only be used to describe the object's intrinsic characteristics (data and operations).

In Figure 2 (b), *tseat* is shown as a binary function from *aircraft* and *classcategory* to *integer* which represents the method *tseat* in *aircraft*. Notice that it is not clear where the method is defined, and it may be regarded as it is defined either in *aircraft*, in *classcategory*, or in both. To make a distinction as to where a method is defined, we use a notation derived from standard SDMs as shown in Figure 2 (c), where *tseat* is viewed as an attribute of the aggregation of *aircraft* and *classcategory* objects. Note that in such an aggregation, only one attribute is allowed (i.e., one outward solid arrow) and the solid arrow from a type to the aggregation shows where the attribute is defined. In Figure 2 (c), *tseat* if defined in *aircraft*. Note that the same attribute *tseat* is represented in Figure 1 using an attribute in *aircraft* with arguments of type classcategory and a range type of integer.

Again consider the attribute *fireeqp* of Figure 1. To represent a multi-typed range we introduce a *multi-type* constructor, again derived from standard SDMs, as shown in Figure 2 (d) (a triangle inside a circle). Each child of these nodes must be of abstract object types (triangles) for the reasons described earlier, and such nodes may only be used in the range of an attribute or a grouping type.

In some models (e.g. [5]) attributes are not allowed to have objects as values (i.e., the range of the attribute function may not be an abstract type), and others do not allow arguments in attributes (e.g. [5, 36]). This limits the application of these models, weakens their modeling capability considerably, and parametric methods become impossible to capture. In the former case, the model lacks the strength of supporting complex objects that is the main purpose of OO models and in the latter case the methods become essentially an attribute computed only at run time.

2.3.5 Signatures

As discussed before, all attributes and methods (function) in an object have a *type*. A type of an attribute specifies what type of value an attribute can take. A type associated with a method, also called a *signature*, is a tuple of the types of its input arguments and the type of the computed value, of the form $\tau_1 \times \ldots \times \tau_n \rightarrow t$. The types associated with the method *tseat* in *aircraft* are **aircraft→integer**, and **aircraft×classcategory→integer**, where **aircraft** is the type of the context and is implicit, **classcategory** is the explicit input argument type, and **integer** is the type of the output value. Note that OR model allows *multi-typing* of methods and in this example the same method has two distinct signatures. This is essential for supporting method polymorphism described later. When a method *tseat* is invoked, which method type will be activated will now depend on the specification of the message. The type of an abstract object is, however, composed of its constituent property types – attribute and method types. Hence the type of the object *aircraft* is <**company** : **string, tseat** : **integer, tseat(classcategory)** : **integer, range** : **integer, kitchens** : **integer, engines** : **integer, cockpit** : **mitshubishi**>.

2.3.6 Classes and Instances

In OR model, objects (entities) are grouped into classes and classes are organized in a *specialization-generalization (SG) hierarchy*[3]. In the spirit of the extended ER model, this leads to specialization (in most of the cases) where the classes lower in the SG hierarchy inherit the properties of those higher up. In our example of Figure 1, class *DC10-30* is a specialization of the class *aircraft*. Class *DC10-30* has all the properties[4] of the class *aircraft*, and some more. Classes have objects as their instances, which may have specialized properties. The SG hierarchy naturally induces (super)class/subclass associations among classes. For example, with respect to class *DC10-30*, *aircraft* is a superclass and *MD10* is a subclass. It is possible for an object class or an object to be part of multiple (super)classes at the same time, if it shares properties with each of these classes. In our example class *MD10A* is a subclass of the classes *cargoaircraft* and *MD10*.

Note that in OR model we do not incorporate the object type called *free* [2], since we do not impose the restriction that a superclass has to be defined before a subclass (free) is defined, nor do we require that all the subclasses should be removed from the database when a *base* class or the highest class is removed. In OR model each class has its own existence. But we do require that a class should be *well typed* or *well structured*. By well typing we mean that if an object has a property, it must have a signature. Alternately an object can not have properties without a structure. We, however, do not distinguish between class objects and an instance objects. An instance object may be viewed as a (sub)class object, if it has some new properties compared to its (super)class, some properties are *withdrawn* (described next) or it has its own instances. This view point has advantages with respect to a logic based query language design for the model, and with respect to the update semantics that we will discuss later. This is also because of the reason that the object hierarchy in OR model is not based on type inclusion semantics. This view point is justified because the objects in semantic models are abstract anyway; that is their types are distinct

[3]We do not distinguish between specialization and generalization as in other semantic models (e.g., IFO [2], SDM etc). Also note that the hierarchy is not based on the *sub-type* relationships as it is in most of the OO and semantic models where a subclass inherits its superclass' structure and properties, and thus has a superset of properties of all its superclasses. The class hierarchy is defined by the user and is decided by the application need. This gives OR model a definite advantage over other models which we will discuss later in the section on update semantics.

[4]In this particular instance.

from their attributes and sub- or supertypes [39], and their relationships with other objects in the database, and their identity does not depend on their attribute values or types. This uniform view point of classes and instances can be justified from yet another perspective and we will take up this issue again in the next section and when we discuss inheritance.

2.3.7 Inter-object Associations

In the OR model, inter object associations or relationships are of three kinds, namely - *aggregation, hierarchy*, and *relationship* associations. Aggregation association results when objects (oids) are used as values (in the range) of attributes or methods, thereby forming an attribute to object association. Since the method has the handle on the object through its id, it can now refer to its properties, and those properties may arbitrarily refer to properties of other objects in turn, which are essentially aggregations of properties. Using aggregation association, it is possible to have nested data structures of any depth, even recursive structures. This is the counterpart of the *composite objects* in OO data models. In a later section we will see that the update semantics of composite objects in the OR model is very different from that of those in many OO models. An instance of aggregation in our example is *refuelsys* in the class *B100* which refers to an object `shell8` of constructed type shell (for want of space, the type shell and its instance `shell8` are not shown in Figure 1).

The second kind of association is the hierarchy or *is-a* association between objects and classes, and between two classes, resulting from the SG hierarchy. Recall that the SG hierarchy in OR model is somewhat different from the hierarchy definition in other SDMs. We allow only one kind of is-a association and we do not incorporate the *generalization* or *exclusive subclass* [73, 36] is-a in OR model. It is however possible to capture these notions in OR model by specifying a constraint in the schema that says that the domain of each of the subclasses of a generalized class is disjoint. This will then imply that no multiple subclass relationships are possible among the classes under the generalized class. The OR subclass relationship is based on a *common set*[5] of attributes that the objects share among the classes rather than a subset-superset relationship of attributes as required by many semantic and OO models.

[5]Loosely speaking. In some cases the intersection may be empty. This kind of situation in real life is very unlikely to happen, which however may be attributed to as ill structured hierarchy or as a consequence of bad schema design.

This, surprisingly, is consistent with the SDM approach [36] in which classes can have special attributes and these attributes are not inherited down the SG hierarchy. That is class attributes are local to a class and are not shared by any subclass or instance. Thus we may view the sets of attributes that are not common to pairs of classes as class attributes. Note that we do not distinguish between class attributes and member attributes. To avoid inheritance of unwanted attributes (e.g. class attributes), we resort to a notion called *withdrawal* (described next). However, the set of instances in the superclass is a super set of the set of instances in the subclass[6].

The third kind of association is the *relationship* association among classes of objects (or entities). Relationships among classes of objects may be n-ary in general. We contend that relationships have a character distinct from classes of objects and are aggregations of constructed types. Thus they are *relations* involving different classes of objects, and do not participate in any SG hierarchy. As in the relational model, the objects in the participating classes form the key of the relation. Relationships can be one-one, one-many, or many-many. This is called the *maximum cardinality* of the relationship. In our example in Figure 1, *useseat* is a 3-ary relationship involving the classes *aircraft*, *seat* and *classcategory*, which says that an aircraft uses a type of seat in possibly more than one accommodation class (first class, club class, etc.). Relationships may have *minimum cardinalities* too. A minimum cardinality is shown by a hash mark on the side of the participating object class. A minimum cardinality specifies whether the objects of the class are required to take part in the relationships or not.

2.3.8 Methods in Relationships

Many OO models do not support relationships (or aggregations) as a first class component, and most of the models which do, do not support methods in the relationships. In the OR model, relationships may have descriptive attributes/methods in exactly the same way as objects. The only difference is that in the case of methods of objects, the context is the object (or the class) and the type of the object/object class takes part in the signature of the method. In the case of relationships, the context is the key of the relationship and, hence, the type of this key takes part in the method

[6]This means that if an object o is a subclass of p then all instances of o are also instances of p. So the subclass relationship is based on a monotonic set inclusion relationship of instances from subclass to superclass.

signature instead.

2.3.9 Inheritance

The objective of organizing objects in a hierarchy of classes is to share properties of the objects in useful, economical, and meaningful ways. Properties of superclasses can be *inherited* by their various subclasses. The notion of inheritance in OR model is influenced by the viewpoint we take in respect to objects and classes. Recall that we do not distinguish between class attributes and member attributes and allow subclasses to inherit methods and attributes from superclasses uniformly. This suggests that there is no distinction between the way an instance of a class inherits structure and properties from a class and a subclass inherits the same from the superclass. Hence from the inheritance point of view subclasses and instances can be viewed uniformly. Hence keeping a distinction between class and instance objects do not serve any useful purpose, and hence can be removed in OR model. This explains the dual nature of objects in OR model - object as a class and object as an instance. Which role an object takes at a given time now depends on how it is viewed. It is easy to see that using the constructs provided in OR model. we can, however, nicely model the conventional view of classes and instances.

The class *DC10-30* in our example inherits all the attributes in the class *aircraft*. In other words, the attributes and methods that are available in *aircraft* are also available in *DC10-30*. In this case, the inheritance is called *monotonic*. However. it is possible that a subclass or instance may not inherit all the properties from a superclass. because they do not make any sense in the subclass. For example, suppose that the class *MD10* is a redesigned version of *DC10-30* with almost no changes in the basic design of *DC10-30*, except perhaps a minor change in body structure and a radio replaced by a computerized radar (see Figure 1). Due to the merger of McDonnell and Douglas. the company name is changed to McDonnell Douglas. The *DC10-30*s are still manufactured and maintained by the Douglas, but the *MD* family belongs to the new company. Clearly the company name in the attribute *company* in *aircraft* is not applicable to *MD10*. Somehow, the inheritance of the attribute *company* should be made ineffective in *MD10*. The way this situation is handled in the OR model is by using a *non-monotonic* inheritance. There are three mechanisms for achieving non-monotonic inheritance - *inhibitance*. *blocking* and *overriding*. Methods or attributes may be overridden in a subclass or instance object. If a subclass or instance object redefines a method implementation or the value of an attribute, then the corresponding method or attribute is not inherited and is said to be overridden in the subclass object. That is more specific and recent definition takes effect. This is called specificity of properties. In our example, the value of *company* attribute in *DC10-30* is "Douglas" while the value for this attribute in *MD10* is "McDonnell Douglas". We say that *company* = "McDonnell Douglas" in *MD10* overrides *company* = "Douglas" in *DC10-30.*

In terms of method (intensional attribute) inheritance, we take the view that the implementation of the method is inherited in the subclass, not the value that is computed in the superclass. This implies that the procedure that implements the method in the superclass will now be evaluated in the context of the subclass by instantiating all *self references* in the procedure to the current class. It is important to observe here that the overriding semantics for methods definitions in OR model is different from logic based models such as F-logic [47]. In OR model, even if the inherited procedure fails to compute anything in the subclass (null value), the subclass will not inherit the corresponding method evaluation (value) in the superclass, if any. This is in agreement with the method inheritance principles in most of the OO systems. A similar notion of inheritance is not available in SDMs.

An inherited method in a subclass can be *withdrawn* by inhibiting it. On the other hand, inheritance of a method can be *prevented* by blocking it in the superclass. Withdrawal of signature implies withdrawal of data corresponding to a method type, but the converse is not true. Also, inhibition and blocking is symmetric and one implies the other but in the opposite direction. For instance, in *MD10* we *inhibit* or *withdraw* the method *radio* (which would otherwise be inherited) from the superclass *DC10-30.* The methods *radio* and *kitchens* are *selectively* blocked in *MD10A* for *B100*, while *cockpit* in *MD10A* is blocked for all subclasses of *MD10A* by qualifying the method name with the intended class name. *Re-introduction* is a mechanism for inheriting a method from an ancestral superclass as long as it is not blocked. In *MD10A*, the *radio* from the ancestor *DC10-30* is reintroduced as the communication instrument in place of the costly radar system. Since an object class can be a subclass of multiple superclasses, *multiple inheritance* is possible. In that event the subclass inherits properties from all its superclasses in a way described above.

The notion of non-monotonic inheritance is not captured in any of the contemporary SDMs. However in OO data models only overriding is supported. Withdrawal is handy in models like F-logic, OR model, etc. where there is no distinction between class attributes and instance attributes and classes and instances have no apparent distinctions. Consider for example, the attribute *averageprice* in *DC10-30*. This is meant to be a class property and does not make any sense in an instance object of the class *DC10-30*. We have two choices - we may withdraw this method in *DC10-30* by blocking it, or we may withdraw it in each instance by inhibiting the method as described earlier. SDM on the other hand can handle this situation nicely by defining the attribute *averageprice* as a class attribute and hence no instance will share it. However, now a subclass in SDM has to redefine the same attribute if it needs it. In OR model, a new definition is not necessary in a subclass in a similar situation, and can be nicely modeled using withdrawal. Note that the notion of withdrawal is unique to OR model.

Withdrawal is also useful to resolve schema conflicts in case of multiple inheritance. In our example of Figure 1, in absence of withdrawal, *MD10A* would have inherited **cockpit : mitshubishi** from *MD10* and **cockpit : bombardier** from *cargoaircraft*, resulting in a clear conflict of types. To resolve the crisis in our example, object *MD10A* favors the signature **cockpit : bombardier** from *cargoaircraft* by selectively inhibiting the inheritance of signature **cockpit : mitshubishi** from *MD10*. This is a major problem in a dynamically evolving database where schema and class membership is constantly changing, and OR model can handle this kind of situation nicely.

2.3.10 Inheritance Conflict Resolution

Since we allow objects to be modeled as a subclass or instance of multiple superclasses, inheritance conflicts may arise as discussed above. OR model incorporates a simple conflict resolution rule to handle such conflicts by avoiding inconsistent schema design. The first rule is, inheritance is allowed from a unique source (the point of definition) in the SG hierarchy. If this condition is violated, inheritance of the property in conflict is automatically inhibited in the class where it is in conflict. By defining suitable inhibition and/or blocking of properties, the designers may resolve this conflict by ensuring a unique source for the property. Hence, OR model supports two modes of conflict resolution - (i) user specified conflict resolution, and (ii) automatic or default conflict resolution.

2.3.11 Encapsulation

In the OR model the implementation of all the intensional attributes, or methods, is private to the object/class or to the relationships where they are defined. Users only see the interface to the objects or relationships, through which the set of methods applicable, as well as their signatures, are visible. It is possible to define some attributes or methods to be *private* to the object/class or relationship. Private attributes and methods are not visible from outside, and are only accessible by the object/class or the relationship in which they are defined, for the implementation of other methods in it. This provides a limited form of security addressed in some research works [12]. The concept of providing an interface to objects and relationships through which all legal methods are visible, and of localizing the implementation of the methods to the objects or relationships, is known as *encapsulation.* Encapsulation enhances implementation independence, modularity, and abstraction. In our example, the attributes *computer, weapons* and *bombs* are private to the class *strategicbomber* and are not visible from outside. Suppose the method *weapons* makes use of the attribute *airforce* to decide the set of weapons and bombs that the aircraft will carry. The methods *bomberunit* and *weapontransport* in *B100* then use the methods *weapons, bombs* and *missileunits* to decide the kind of bomb transport and launching system to be installed in the aircraft. Here the methods *weapons, bomberunit* and *weapontransport* use several private methods to respond to a message. Although these attributes and methods are inheritable, no other object or relationship can access these methods except via inheritance.

Notice that the notion of encapsulation in OR model is somewhat different than in semantic models and OO models. Essentially, SDMs encapsulate structural aspects of objects, whereas OO models encapsulate behavioral aspects of objects which are the method implementations. OR model extends the notion of encapsulation of structural aspects of objects by allowing private attributes and methods that are invisible to the users and to other objects. In OR model every attribute is visible by default unless specified as private. In contrast, in most of the OO models, state variables are private by default and methods have to defined to access them.

2.3.12 Method Polymorphism

Methods in the OR model are *polymorphic.* Let us consider the family of methods "*tseat*". The method *tseat* in *aircraft* and in all its subclasses wherever it is applicable, computes the total number of seats in the aircraft. Yet again, *tseat(classcategory)* *refines* the method *tseat*, by *overloading* it by giving it the argument *classcategory*, which then computes the total number of seats in certain classes in a given aircraft. On the other hand, the method *tseat* in the class *seat* represents the total number of seats that are in use in different aircraft, and, hence, has a completely different meaning than the previous versions. All these forms of overloading of method names are known as method polymorphism. Polymorphism aids refinement and extension of behavior through overloading, and enhances re-usability. *Redefinition* is also possible in the OR model. For example, in Figure 1, the attribute *range* in *MD10A* is first inhibited (withdrawn), and then redefined with a different type, i.e. real. In general, redefinition only shares the attribute or method name in immediate superclasses. It can completely change the type or the signature of the corresponding attribute or method, as well as its implementation. Contrast this with a re-definition of signatures using overriding. Note that if overriding is used, the data component of the method will still be inherited (if not overridden), while if withdrawal is used, both signature and data will be withdrawn if the signature is withdrawn.

2.3.13 Constraints

Type definition of attributes and signature of the methods can be viewed as constraints on the type of value that attributes can have and the type of input/output arguments the methods may accept and return. Since the type of objects, or relationships, is the composite of their attribute types and method signatures, the type constraints above also impose corresponding constraints on the scheme or type of the objects or relationships. Other forms of constraints include integrity constraints (IC), inclusion dependency, etc. Some ICs may only involve attributes/methods applicable to a class or relationship, or only to classes participating in a relationship. E.g., functional dependencies and referential ICs fit into this category. Other ICs may involve several classes, methods, and relationships. For example, we may define the following constraints: (i) a global constraint saying all aircraft should be able to cross the Atlantic (*range* > width of the Atlantic); and a set of local constraints such as (ii) the total number of seats in *aircraft* should be more than 100, and (iii) no passenger seats are allowed in a *cargoaircraft*, etc.

Similar to attributes and methods, the local constraints in objects get inherited down the SG hierarchy. There can be conflicts among different constraints that the objects and relationships must satisfy. For instance, if in our example we move constraint (ii) to the global level, the cargo liners will not be able to satisfy that constraint. In the example, constraint (ii) is meaningful for all passenger aircraft. But since *MD10A* is basically a passenger aircraft that is being redesigned as a cargo carrier, constraint (ii) does not make sense here. The situation can be more complex. Constraint (iii) defined for cargo aircraft is in clear conflict with constraint (ii) if they are both monotonically inherited in *MD10A*. Such conflicts are resolved by attaching priority to constraints. In the OR model constraints are placed in different strata. The lower the stratum the higher the priority. Global constraints are placed in stratum 0 and, hence, have the highest priority. The local constraints are in stratum 1 and the inherited constraints in stratum 2. Thus, if an inherited constraint is in conflict with the local one, the local one is given preference. E.g., to solve the conflict in our running example, we copy constraint (iii) from *cargoaircraft* to *MD10A*, which makes it local, giving it a higher priority than constraint (ii). In the case of multiple inheritance, conflicts may also arise between inherited attributes because of differences in types, values, implementations or the semantic meaning. This form of conflict can be effectively resolved using inhibition and blocking. In our example, the method *cockpit* which is inherited from *cargoaircraft* and *MD10* is in conflict in *MD10A*. So we selectively inhibit *MD10.cockpit* (notice the prefixing of the class name in front of the attribute name) in *MD10A* to favor the inheritance of *cockpit* from *cargoaircraft*. Similarly, selective blocking shown in the class *MD10A* and discussed earlier, can also be used to resolve such conflicts.

2.3.14 Virtual Objects and Relationships

An object in the OR model may be derived from the database contents. Its type and values (or properties) may also be derived. Such an object is not stored in the database. Only the rule that describes how to derive the object is stored. Constraints may be specified on such objects. All the properties that might be needed to describe the object are derived from other database contents. Since these objects are virtual, all their properties are also virtual. SDM supports virtual objects in the form of derived schema components, but requires that the objects be of type free (discussed earlier). We do not impose any such restrictions.

Virtual objects may be used to model user views of the database using *classification* and *association* [19]. Classification is a form of abstraction in which a collection of objects is considered a higher level object class, and association is a relationship between member objects that is considered as a higher level set object. OR model captures these abstractions through an attribute of the virtual object in the former case, and by *populating* a set of pertinent objects as the member of the virtual object (class) in the latter case. Populating a virtual object (class) is a process that identifies the set of instances of the class and require some rule evaluation. The set of instance objects in the population may be ground or virtual. Except the fact that the virtual object is derived, it has no distinction with a stored object. Such an object will be only created using the rule when some reference is made to it, and will be removed from the database after the reference is completed.

Similar to virtual objects, virtual relationships are also possible. The associations between objects may now be derived at run time. *Pumping* in the tuples of a virtual relationship will also require a rule evaluation which will derive the associations between objects. It is also similar in every respect with the usual relationships. If in a relationship, at least one of the participating object (class) is virtual and the object participates in the key of the relationship, then the relationship must also be virtual.

2.4 Global considerations

In a broader perspective, we now discuss how we develop database schemes from the local constructs and building blocks described in the previous section. First we see the structural nature of OR schemas, that is we discuss how constructs can be meaningfully combined to form database schemas. Then we investigate dynamic aspects of the schema and consistency issues. We have already specified some of the restrictions we impose on the local constructs of the OR model. We restate several of them in this section from global perspective. Such restrictions are motivated by the underlying philosophy of the model and its objectives.

We require that grouping objects be only used in attribute ranges because such objects are rarely of interest in isolation. Note that some models (e.g., IFO, SDM, O_2, etc.) allows such objects in the database to exist. We also do not allow aggregation or grouping type objects in attribute domains, and aggregation type objects in attribute ranges. Unlike IFO, SDM and O_2, we do not allow complex objects in the attribute range, we use *values* in the range instead. Recall that a value may be basic or an object id which represents an id based object. Although the underlying structure of an object remains arbitrarily complex, the immediate structure is simple and is only one level deep. This approach is close to the spirit of O_2 model. Note that aggregation association is only defined for id based objects, not for value based objects. That means, the navigation in value based objects are similar to relational model. We also require that all the types mentioned in all the attribute domain and ranges are database objects except the basic types and grouping types.

The restrictions on ISA constructs are simple. Though it is not required all the time, it is in general assumed that the database objects will be defined top-down. That is certain base types will be defined first and then its subclasses, and so on. We also require that the ISA relations be acyclic. Consider the ISA definition in Figure 3 (a). The three classes form a cycle and intuitively implies that the types are redundant, that is in every instance, the three types will contain the same set of objects. Furthermore, none of them can be considered as the base type and hence the underlying type of each of the objects are not deterministic. The definition of Figure 3 (b) is also not allowed if it is assumed that the domains of the superclass objects *aircraft* and *boat* are disjoint. However, if *someobject* is an amphibian aircraft, we might want to lift the domain disjointness restriction and allow the definition. Value based objects (entities) and id based objects (semantic objects) can not be mixed in a subclass relationship. That is an entity class can not be a sub- or superclass of an id based object class. If these restrictions are satisfied, it can be shown that every node will have an unambiguous underlying type, no pair of nodes will be redundant, and every node will be satisfiable in the sense that some instance will assign a nonempty active domain to that node.

Furthermore, we require that the intended type of an object in the hierarchy be type conflict safe after inheritance. By that we mean that, conflict free structure of an object/class should be guaranteed if necessary by using the method of withdrawal.

Derived schema components are one of the fundamental mechanisms in semantic models for data abstraction and encapsulation. Such a component has two elements:

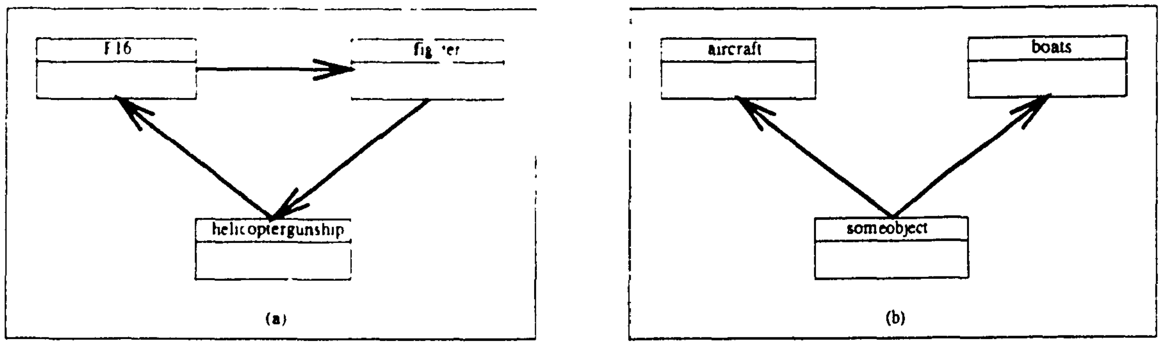

Figure 3: ISA restrictions.

a structural specification and a derivation rule. It thus allows to incorporate dynamic information directly into the database schema. Known semantic models only allow derived subtypes and attributes. In OR model we impose no restriction on the type of derived schema components. That is all permissible components may be derived. Derived components are readily identifiable in an OR schema. Such components are represented using dashed symbols. The horizontal line in each component separates the structural specification and the derivation rule; the derivation rules are specified in the lower half of the symbol. The rule part may also contain constraints relevant to that component. The language for the derivation rule in OR model is *first-order predicate calculus*. In the next section we introduce the notion of legal access which imposes a set of restrictions on the dynamic components of the database.

2.5 Accessibility

In the OR model the only way to reference the properties of the objects is by invoking the attributes/methods using messages. The set of methods that are visible from outside the objects are the only ones that can be invoked. There can be methods in objects and relationships which are private to them, and thus are invisible from the interface. Also, not all methods that are visible through the interface are always available. We require that only *relevant* methods can be invoked. A method is relevant if the objects or relationships involved are *related*. Two classes/objects are *related* if they are involved in a hierarchy association, a relationship association, or an aggregation association. For example, the object class *B100* or its instances are related to *DC10-30* through hierarchy association (but not to the specific instances of

DC10-30). The class *B100* is related to *seat* through relationship association *useseat*. The attribute *engine* in *DC10-30* is of type rollsroyce (aggregation), hence *DC10-30* and *rollsroyce* are related. The related objects or relationships are allowed to refer to the public properties of each other, and such accesses are called *legal accesses*. The notion of legal access is thus influenced by the notions of non-monotonic inheritance, associations and encapsulation.

2.6 Update Semantics

In this section we will investigate the issue of update semantics in OR model. The semantics is very intuitive, and we will take a very simplistic approach to describe it. There can be several forms of updates, namely change in attribute values, change in object schemas or types, change in method implementations, change in class memberships, change in relationships among objects, change in constraints, change in inheritance, addition/deletion of objects and classes, etc. Each of these changes has consequences in the OR schema. We will explain them very briefly.

Change in attribute values are permitted if the value refers to a basic value, or an existing database object at any time without any restriction. In some models (e.g. Orion) composite objects have a ownership relation. For example, if object *cathy* has an attribute called *doctor* with a value john meaning the object identified by the id john is the family doctor of cathy, then it is regarded that *cathy* owns object john. This is called a dependency relationship. In models like Orion, this dependency means that if the object *cathy* is deleted, then the object john must also be deleted. In OR model we do not insist on this dependency relationship. Hence if the attribute value is changed, or the so called owner object is deleted, the owned object will continue to persist in the database. This is also because of the view we take with respect to the uniformity of the class and instance duality of objects discussed earlier. If we incorporate the notion of dependency relationship in our model, then unwanted results may happen. For example, if the owned object is removed then it is possible that all the instances and subclasses of the object has to be removed due to type safety requirements discussed earlier (since we allow arbitrary subtyping). If some other objects have references to any of the instance objects that are removed then they will become dangling and result in an "inadmissible" database.

However, if it is so desired, such dependency relationship may be implemented via a suitable constraint specification. This approach gives us the flexibility of modeling the database semantics at the users' choice.

Change in object schema is not that simple, but is a lot more flexible than contemporary SDMs and OO systems. Since the class hierarchy is not based on types, simple type changes and withdrawal can be accommodated with minor changes in the schema and without the restructuring of the classes and instances provided the database still remains well typed. Change in method implementations have no side effects since it is local to the object where it is defined. Since the is-a semantics is different than most of the SDMs and OO models, the semantics of hierarchy association update is also quite different. Note that an object in OR model has a unique id and has only one copy that is sliced in many different components. That means if an object o is in class q and q is a subclass of p, o does not appear as an explicit instance of p as it is of q, rather it is implicitly an instance of p. This is not the case in many SDMs and OO models. Hence if o is removed from q it is implicitly removed from all the superclasses of q automatically. Similarly if o is an explicit instance of p and q (multiple subclass), then it is the case that o is a single object as an instance of these two classes by definition. This semantics also means that, if a property is deleted from or added to o, o does not require to be relocated in a class in the hierarchy where it fits best since the association is not type based.

Chapter 3

ORLog: A Logic Based Object-Oriented Language

3.1 Related Research and Motivation

In the recent years the database research community has made significant strides in combining the declarative aspects of deductive databases and the superior modeling capabilities of the object-oriented paradigm. Although the appeal of a combined model seems overwhelming, its logical rendition is not so easy. This is largely because of the presence of the concepts such as inheritance of behaviors, code reuse and overriding that are related to non-monotonic reasoning. Although overriding complicates the underlying theory to a great extent, it is a notion that makes the object-oriented paradigm interesting and useful. The issue gets further complicated when we allow multiple inheritance. We are then required to deal with inheritance conflicts and as a consequence may have to settle for multiple minimal models at the declarative level, e.g., as in [47]. Current solutions that are proposed in the literature are based on non-monotonic reasoning [74], stratification [22], program composition [20, 61, 62], etc. These proposals have been criticized due to the skepticism about their feasibility as an efficient computational platform.

Behavioral inheritance has been studied in deductive formalisms like the *Ordered Theories* of [50], in modular languages such as Contextual Logic Programming [61, 62] and several others. However, the framework in which they accomplish this are quite narrow compared to the needs of object-oriented databases and languages.

Logic languages like LOGIN [3] and LIFE [4] incorporate structural inheritance by means of an extended unification algorithm for ψ-terms, complex typed structures that are used for data representation. Kifer et al. [47] proposed an elegant logic, called F-logic, as a logical foundation for object-oriented databases. Only structural inheritance is captured in its semantics and proof theory, and for this component of the language it was shown to be sound and complete. However, completeness is achieved by taking a monotonic view of inheritance. Behavioral inheritance with overriding in F-logic is captured *indirectly* by a pointwise iterated fixpoint construction. F-logic *directly* accounts for ground molecules (corresponding to values computed from method execution) being inherited down an is-a hierarchy. *Code* inheritance is left outside the language and has to be *simulated* by the programmer using higher-order features of the language and F-logic's pointwise overriding in a clever manner.

Dobbie and Topor [29, 30] have developed a language called Gulog, inspired by a restricted fragment of F-logic. Gulog is function-free. For a restricted class of programs which are stratified with respect to inheritance and deduction, and where schema and data are separated, and the schema is predetermined, they propose a sound and complete query evaluation procedure. They account for (conflict-free) multiple inheritance with dynamic overriding, by forcing the is-a hierarchy to be static. However, they only account for value inheritance and do not capture behavioral inheritance.

There are some proposals that address the issue of behavioral inheritance but rely on a translation of an object-oriented logic program into a value-based conventional logic program. In this approach the insight into the meaning of inheritance and its interaction with deduction and overriding is lost, and there is little control on how the translated program would behave. OOLP+ [25] and *L&O* [58] are two such proposals which capture a few features of inheritance in a limited way.

3.2 Objectives of ORLog

The goal of this chapter is to develop a simple logical account of behavioral inheritance (in the sense of *code reuse*) with overriding, providing for conflict resolution in the event of multiple inheritance. This we achieve by enriching the syntax of the logic so that the *locality* (i.e., point of definition) and *inheritability* of properties (i.e., methods and signatures) can be asserted and inferred. In addition, our language allows for a syntactic instruction for *withdrawing* property definitions to prevent subclasses or instances from inheriting them. A special aspect of this latter feature is that it allows the programmer to influence the logic's inheritance mechanism to suit her preferred needs.

The above features were first introduced in the context of ORLog (for *Object Relationship Logic*) [40]. In [40], we developed a declarative semantics for the higher-order features of ORLog, not including inheritance. In a preliminary version of the work contained in this chapter [42], we presented a declarative characterization of behavioral inheritance in ORLog and showed that conflict resolution in ORLog is possible in a relatively simple way. In [22], a stable model semantics for behavioral inheritance was proposed using notions of locality and inheritability similar to those proposed in [40]. By contrast, in the present discussion, we account for behavioral inheritance *within the logic*, by capturing it within a sound and complete proof theory. We also develop a model-theoretic and fixpoint semantics and establish the equivalence of all three semantics.

Before closing this section, we note that a majority of the works on inheritance rely on a form of non-monotonic reasoning. Of these, [29, 30, 47] capture value inheritance while [50, 22, 18] handle a form of behavioral inheritance. While a non-monotonic account of inheritance is natural and interesting, it relies on extra-logical features to capture inheritance. In addition, most of these proposals incur a high cost for query processing, which can be significant for database applications. In comparison, a complete proof theoretic and declarative account of inheritance is at once intellectually more satisfying and offers a greater promise as an efficient computational platform.

Interested readers are, however, referred to [47] for a more detailed survey and an eloquent discussion on the issues related to logic based object-oriented languages in general.

The rest of this chapter is organized as follows. In section 3.3 we present the syntax and an informal semantics of inheritance. We then present a formal model-theoretic semantics of ORLog in section 3.4, an Herbrand semantics in section 3.5, a proof procedure in section 3.6, and finally a fixpoint semantics in section 3.7. In section 3.7 we also prove that (i) the fixpoint semantics and the model-theoretic semantics are equivalent, and (ii) the proof theory is sound and complete with respect to this semantics.

3.3 Overview of ORLog

In this section we discuss the salient features of ORLog, its syntax and inheritance semantics using an example. We also discuss, on intuitive grounds, the concepts of *clause locality* and *inheritability* that are unique to ORLog.

3.3.1 Syntax

The alphabet of the language $\mathcal{L}$ of ORLog is a tuple $\langle \mathcal{C}, \mathcal{V}, \mathcal{M}, \mathcal{T}, \mathcal{P} \rangle$, where (1) $\mathcal{C}$ is a denumerable set of constants playing the role of basic values and also object id's, (2) $\mathcal{V}$ is an infinite set of variables, (3) an infinite set of method symbols $\mathcal{M}$ for method names, (4) $\mathcal{T}$ is an infinite set of symbols for type names, and finally (5) $\mathcal{P}$ is an infinite set of predicate symbols. The set of terms $\mathcal{I}_o = \mathcal{C} \cup \mathcal{V}$ are called the *id-terms*. A *method denotation* is an expression of the form $m^k_{\mapsto}$, where m is a method symbol, k is a natural number and $\mapsto$ is one of $\rightarrow, \twoheadrightarrow, \Rightarrow$, and $\Rrightarrow$[1]. It says that m is a method of arity k and it takes on the incarnation $\rightarrow, \twoheadrightarrow, \Rightarrow$, or $\Rrightarrow$ as indicated by $\mapsto$. Intuitively, $\rightarrow$ and $\twoheadrightarrow$ are used to denote data expressions and $\Rightarrow$ and $\Rrightarrow$ are used for type or signature expressions, just as in F-logic. Throughout the paper we use the uppercase letters for variables and lowercase letters for constants. We use bold lowercase letters to denote arbitrary terms.

Atomic and Complex Formulas

There are nine types of atomic formulas in ORLog. If $\boldsymbol{a}_k$s, $\boldsymbol{p}$ and $\boldsymbol{q}$ are id-terms in $\mathcal{I}_o$, t_is are type names in $\mathcal{T}$, r is a predicate symbol of arity n in $\mathcal{P}$, and $m^k_{\mapsto}$ is a method denotation as defined above, then the set of atomic formulas of ORLog is defined as follows:

1. the *id-atoms* are of the form $\boldsymbol{p}[\,]$ which assert the existence of objects or types;

2. the *is-a atoms* are of the forms $\boldsymbol{p} : \boldsymbol{q}$ and $\boldsymbol{p} :: \boldsymbol{q}$ that state that object $\boldsymbol{p}$ is an instance or subclass of object $\boldsymbol{q}$, where $:$ ($::$) means *immediate* (*transitive*) instance/subclass;

[1]When a distinction is not important, we shall also use the notations $\mapsto^s$ and $\mapsto^d$ to stand for an element in $\{\Rightarrow, \Rrightarrow\}$ and $\{\rightarrow, \twoheadrightarrow\}$ respectively.

3. the *data atoms* or *d-atoms* are of the form $p[m(a_1,\ldots,a_k) \rightarrow a]$ or $p[m(a_1,\ldots,a_k) \twoheadrightarrow b]$ which means when method m is invoked in the context of object p with arguments $a_1,\ldots,a_k$, it returns a functional output a, or similarly a set valued[2] output of which b is a member;

4. the *signature atoms* or *s-atoms* are of the form $p[m(t_1,\ldots,t_k) \Rightarrow \{t'_1 \ldots t'_n\}]$ or $p[m(t_1,\ldots,t_k) \Rrightarrow \{t'_1 \ldots t'_n\}]$ which have a similar meaning as the d-atoms when invoked in object p with argument types $t_1,\ldots,t_k$, the method m returns an output (functional or set valued) which simultaneously belongs to the types $\{t'_1 \ldots t'_n\}$;

5. the *locality atoms* or *l-atoms* are of the form $p[m^k_{\mapsto}]$ which states that the data or signature (as determined by whether $\mapsto$ is one of $\rightarrow$, $\twoheadrightarrow$ or one of $\Rightarrow$, $\Rrightarrow$) of method m whose (input) arity is k is locally *defined* in object p;

6. the *inheritability atoms* or *i-atoms* are of the form $p[q@m^k_{\mapsto}]$ that specifies that the object p may use (or inherit) the method $m^k_{\mapsto}$ from object q;

7. the *participation atoms* or *r-atoms* are of the form $r \diamond t_1,\ldots,t_n$ that states that r is a relationship (relation) involving object types $t_1,\ldots,t_n$.

8. the *predicate atoms* or *pred-atoms* are of the form $r(a_1,\ldots,a_n)$ whose meaning is exactly as in classical logic.

9. and finally, the *withdrawal atoms*[3] or *w-atoms* are of the form $p[m^k_{\mapsto} \rhd\!\circ q]$ and $p[m^k_{\mapsto} \circ\!\lhd q]$ that captures the fact that the inheritance of the method $m^k_{\mapsto}$ is *blocked* ($\rhd\!\circ$) from the object p to q, or is *inhibited* ($\circ\!\lhd$) in p from q. Blocking is used when a superclass prevents an immediate subclass/instance from inheriting a property, whereas inhibition is used when a subclass/instance rejects the inheritance of a property from an immediate superclass.

The formulas of $\mathcal{L}$ are defined as usual. A literal is either an atom ($\mathcal{A}$) or its negation ($\neg\mathcal{A}$). Every literal is a well formed formula. If $\mathcal{F}$ and $\mathcal{G}$ are well formed

[2]Following F-logic, we also permit molecules as a syntactic abbreviation for conjunctions of atoms. E.g., $p[m_1(a_1,\ldots,a_k) \rightarrow b_1; \ldots; m_n(c_1,\ldots,c_m) \rightarrow b_n] \equiv p[m_1(a_1,\ldots,a_k) \rightarrow b_1] \wedge \ldots \wedge p[m_n(c_1,\ldots,c_m) \rightarrow b_n]$. Similarly, $p[m(a_1,\ldots,a_k) \twoheadrightarrow \{b_1 \ldots b_n\}] \equiv p[m(a_1,\ldots,a_k) \twoheadrightarrow b_1] \wedge \ldots \wedge p[m(a_1,\ldots,a_k) \twoheadrightarrow b_n]$. A difference between a set valued method and a functional method is that we enforce functionality requirement for the latter but not for the former.

[3]It turns out that one of blocking or inhibition is sufficient for our purposes. However, we keep both the constructs for data modeling convenience.

formulas and X is a variable, then so are $\mathcal{F} \wedge \mathcal{G}$, $\mathcal{F} \vee \mathcal{G}$, $\neg\mathcal{F}$, $(\mathcal{F})$, $\forall X(\mathcal{F})$, $\exists X(\mathcal{F})$, and $\mathcal{F} \leftarrow \mathcal{G}$. A formula that contains no variables is called a ground formula. We follow the standard practice in logic programming and consider only the definite Horn clause fragment of our language for the rest of the paper. Hence clauses in an ORLog program are of the form $\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$, where $\mathcal{A}$ is called the head atom and $\mathcal{B}_i$s are called the body literals. We assume that all the variables in a clause are universally quantified. If the head atom $\mathcal{A}$ of a clause $\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$ is a d- or s-atom, then we call it a d- or s-clause respectively. When convenient, we use the term *property* as a neutral term which refers to data or signature of methods. Thus, a *property-clause* (*p-clause*) is either a d-clause or an s-clause. Furthermore, in a p-clause of the form $o[m(\mathbf{a}_1, \ldots, \mathbf{a}_k) \mapsto \mathbf{a}] \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$, we refer to $\mathbf{o}$ as the *descriptor*, or the *context*, of the p-clause. In an analogous way we define is-a, l-, id-, i-, r-, pred- and w-clauses based on the head atoms of the form is-a, l-, id-, i-, r-, pred- and w-atoms respectively. Recall that variables and constants are written in upper and lowercase strings respectively. We use the letters O, P, Q, R (respectively, o, p, q, r or $\mathbf{o}, \mathbf{p}, \mathbf{q}, \mathbf{r}$) for variables (respectively constants, or arbitrary terms) representing object ids; m, s, t, u, v, etc. for method names, a, b, c, d, 1, 2, 3 for constants and basic values; W, X, Y, Z, etc. for variables; and so on.

Programs in ORLog specify objects, their structures and behaviors through signature and method definitions and organize objects in inheritance hierarchies through is-a specifications. Relationships among objects are specified using the relationships and predicates.

Definition 3.1 (Programs) An ORLog program **P** is a triple of the form $\langle \Lambda, \Upsilon, \Pi \rangle$ where,

- Λ is a (possibly empty) set of is-a, l-, id-, and w-clauses where the body literals are either is-a, l-, id-, or w-atoms.

- Υ is a (possibly empty) set of i-clauses with body literals from is-a, l-, id-, w-atoms or i-atoms, and

- Π is a (possibly empty) set of p-, pred- and r-clauses whose body literals are arbitrary atoms in $\mathcal{L}$. □

Intuitively, the clauses in Λ define the is-a hierarchy of classes and instances, the locality of clauses, objects, and the inheritance control information in the form of withdrawal definitions. The clauses in Υ define the inheritability of clauses and finally the clauses in Π define properties for objects, and relationships among classes of objects. Note that in the above definition, we have introduced an implicit dependency relation "$\leftarrow$" such that $\Upsilon \leftarrow \Delta$ and $\Pi \leftarrow \Upsilon$, which means that Υ depends on Δ and Π depends on Υ. The relation $\leftarrow$ is actually a partial order. It should be clear that we do not allow, in particular, the is-a hierarchy to depend on inheritability or properties of objects. This restriction helps make the is-a hierarchy static and keep the inheritability of properties deterministic. On the other hand, this still allows a large class of meaningful programs.

3.3.2 Informal Semantics of Inheritance

The objects in our programs are described by a set of d-clauses and a matching set of s-clauses. Here, s-clauses define method signature and d-clauses define method behavior. Objects also inherit and reuse these descriptions (of structures and behaviors) with other objects that are defined as their subclasses or instances, and they are subject to possible overriding by local definitions. Consequently, the state of an object depends on the structure of the object hierarchy defined using the is-a definitions.

In our conceptual model, we identify two types of p-clauses – *local* and *inherited clauses*. Intuitively, a p-clause is local to an object o, if it is defined in o. On the other hand a p-clause defining a method (data or signature) is inherited in o from q, if o does not define a similar p-clause and q is an ancestor of o such that q defines the p-clause, and there is no other ancestor of o from which such a p-clause is inheritable by o. The following definitions make these notions precise:

Definition 3.2 (Locality) Let cl be a p-clause such that its descriptor is $\boldsymbol{p}$, and $\boldsymbol{r}_1 \vdots \boldsymbol{q}_1, \ldots \boldsymbol{r}_n \vdots \boldsymbol{q}_n$ are all the is-a atoms[4] in its body. Then cl is *local* to every object $\boldsymbol{p}$ such that $\boldsymbol{r}_i \vdots \boldsymbol{q}_i$ holds for $1 \leq i \leq n$. That is, $\boldsymbol{p}[m^k_{\mapsto}] \leftarrow \boldsymbol{r}_1 \vdots \boldsymbol{q}_1, \ldots \boldsymbol{r}_n \vdots \boldsymbol{q}_n$ holds. $\square$

The notion of inherited clauses depends on the notion of inheritability of p-clauses, defined next.

Definition 3.3 (Inheritability) Let S be a set of (ground) id-, l-, w-, and is-a atoms, $m^k_{\mapsto}$ be a method denotation, and o be an object. Then the inheritability of

[4]In this definition and in the sequel of the paper $\vdots$ stands for either "." or ":" in places where the distinction between the two is unimportant.

$m^k_{\mapsto}$ in the object o is defined by the *context function* ∇ as follows:

$$\nabla(S, m^k_{\mapsto}, o) = \begin{cases} p & \begin{array}{|l} \text{if } o[m^k_{\mapsto}] \notin S \text{ and } [\exists q \text{ such that } o{\updownarrow}q \in \\ S,\ \nabla(S, m^k_{\mapsto}, q) = p,\ p[m^k_{\mapsto}] \in S \text{ and } (\forall r, \text{ such that} \\ o{\updownarrow}r \in S, \text{ one of the following holds.} \\ \quad \bullet\ \nabla(S, m^k_{\mapsto}, r) = r, \text{ and } r[m^k_{\mapsto}] \notin S, \text{ or} \\ \quad \bullet\ \nabla(S, m^k_{\mapsto}, r) = p, \text{ or } o[m^k_{\mapsto} \circ\!\!\lhd r] \in S, \text{ or} \\ \quad\ \ r[m^k_{\mapsto} \rhd\!\!\circ o] \in S.)] \end{array} \\ o, & \text{in all other cases.} \end{cases}$$ □

The operator ∇ works as follows. Let o be any object and $m^k_{\mapsto}$ any method denotation. Then $\nabla(S, m^k_{\mapsto}, o) = p$ only if $o{\updownarrow}p$ and $p[m^k_{\mapsto}]$, for some object p. Suppose o does not have its own definition of method $m^k_{\mapsto}$, as indicated by the absence of an atom $o[m^k_{\mapsto}]$ in S. Also suppose for some superclass p of o (i.e., $o{\updownarrow}p \in S$), $p[m^k_{\mapsto}] \in S$. Then $\nabla(S, m^k_{\mapsto}, o) = p$, provided o has an immediate superclass q (which is possibly p) which inherits $m^k_{\mapsto}$ from p, and for every other immediate superclass r of o satisfies *one* of the conditions: (i) r inherits $m^k_{\mapsto}$ from p (same source), (ii) r does not have a local definition of $m^k_{\mapsto}$ ($r[m^k_{\mapsto}] \notin S$), neither does it inherit from any of its superclasses as indicated by $\nabla(S, m^k_{\mapsto}, r) = r$, or finally (iii) either o inhibits $m^k_{\mapsto}$ from r, or r blocks $m^k_{\mapsto}$ for o. In all other cases, $\nabla(S, m^k_{\mapsto}, o) = o$, indicating either o has a local definition of $m^k_{\mapsto}$ and *overrides* all other definitions of $m^k_{\mapsto}$, or it cannot legitimately inherit $m^k_{\mapsto}$ from any of its superclasses, if at all they exist and define such a method. It is possible to show that ∇ is a total function.

Definition 3.4 (Inherited Clauses) Let $\mathbf{P} = \langle \Lambda, \Upsilon, \Pi \rangle$ be a program, $cl \equiv \mathcal{A} \leftarrow \mathcal{G} \in \Pi$ be a p-clause local to an object o, and let $meth(\mathcal{A}) = m^k_{\mapsto}$ be a method denotation. Let S be a set of ground is-a, l-, w- and id-atoms that are entailed by Λ[5]. Then cl is *inheritable* in an object q if $\nabla(S, m^k_{\mapsto}, q) = o$. The inherited clause cl' in q is obtained by replacing every occurrence of o in cl by q, i.e., $cl' \equiv (\mathcal{A} \leftarrow \mathcal{G})[o/\!/q]$, where $[o/\!/q]$ denotes the replacement of the term o by q. □

[5]For our language, this set is always finite and entailment for this part is in the classical sense, as will be seen in Section 3.5. This set can be determined using classical proof theory or fixpoint theory. We present the notion of inheritance here rather than in a later section, to give an intuitive feel for the semantics of inheritance, in advance.

We remark that context switch captured by the *term replacement* of the form $\rho = \{o//q\}$ should not be confused with the usual substitutions (e.g., $\theta = \{X/p\}$) where only variables are replaced by terms. We next illustrate with couple of examples our notion of behavioral inheritance (for code reuse) as well as indicate how the various notions developed above play a role in shaping the semantics of inheritance in ORLog.

Example 3.1 Consider an aircraft design database scheme in OR notation [40] (introduced earlier in Chapter 2) in Figure 4. We use this example to intuitively understand the underlying meaning of ORLog programs in terms of the introduced concepts in this section in a close to real life application. In the next example we will explain the working of ORLog in greater detail with a more abstract application.

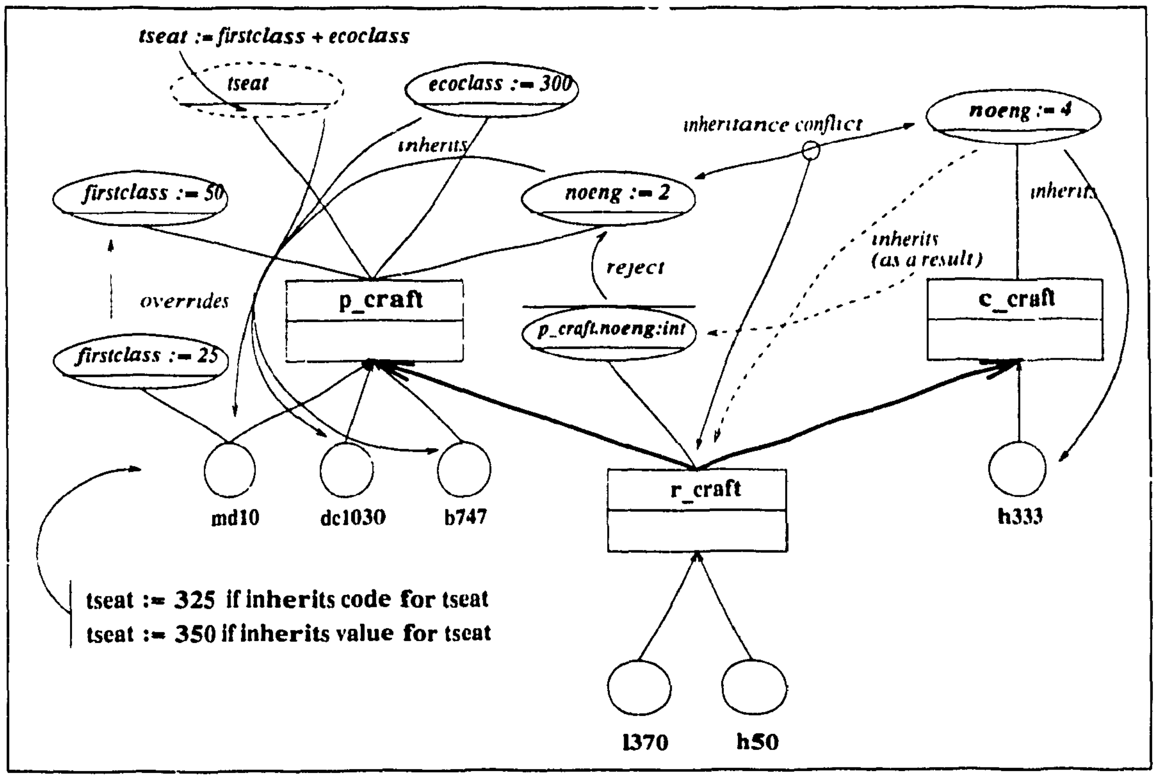

Figure 4: An Aircraft Design Database Scheme.

In Figure 4, we abuse the OR notation and in most cases show the values and codes corresponding to methods in the scheme instead of their signatures for brevity and simplicity. The ORLog program $\mathbf{P}' = \langle \Lambda', \Upsilon', \Pi' \rangle$[6] corresponding to this scheme

[6]We used molecular formula in this example a la [47] as follows: $p\_craft[crew \rightarrow 4, noeng \rightarrow 2] \equiv p\_craft[crew \rightarrow 4] \wedge p\_craft[noeng \rightarrow 2]$.

is given in Figure 5. In the diagram of Figure 4, *p_craft*, *r_craft* and *c_craft* refers to passenger aircraft, rescue aircraft and cargo aircraft respectively.

In the diagram of Figure 4, the attribute *firstclass* in instance object *md10* overrides the corresponding attribute value in its class *p_craft*. Hence, the value of *tseat* (total seats) at the *p_craft* level is 350 while the value at *md10* will be 325 since *md10* inherits the code for *tseat*, not the value (350) from *p_craft*. But *dc1030*, for example, will have 350 as the value for *tseat* since it inherits the value for *firstclass* and *ecoclass* from *p_craft* and hence inheriting the code for *tseat* from *p_craft* does not make any difference even when the code is evaluated at the *dc1030* level. Furthermore, since *r_craft* inhibits *noeng* from *p_craft*, *noeng* in *c_craft* can now be inherited in *r_craft* (and onward) since $\nabla([A'].noeng^{0}_{-}.r\_craft) = c\_craft$.

$$A' := \left|\begin{array}{ll}
(1) & md10 : p\_craft. \\
(2) & r\_craft : p\_craft. \\
(3) & r\_craft : c\_craft. \\
(4) & h50 : r\_craft. \\
(5) & l370 : r\_craft. \\
(6) & h333 : c\_craft. \\
(7) & dc1030 : p\_craft. \\
(8) & b747 : p\_craft. \\
(9) & r\_craft[noeng^{0}_{-} \propto\!\lhd p\_craft].
\end{array}\right. \qquad \Upsilon' := \emptyset$$

$$\Pi' := \left|\begin{array}{ll}
(10) & p\_craft[ecoclass \rightarrow 300; firstclass \rightarrow 50; \\
 & \quad crew \rightarrow 4; noeng \rightarrow 2]. \\
(11) & p\_craft[tseat \rightarrow T] \leftarrow p\_craft[ecoclass \rightarrow E; \\
 & \quad firstclass \rightarrow F], T = E + F. \\
(12) & md10[firstclass \rightarrow 25]. \\
(13) & c\_craft[noeng \rightarrow 4; tseat \rightarrow 4]. \\
(14) & X[makeen \rightarrow p\&h] \leftarrow X :: c\_craft.
\end{array}\right.$$

Figure 5: Example program **P'**

Now the question is what **P'** entails? Leaving the details to be explained in example 3.2, we expect the following from program **P'** among several others.

1. $h50 :: c\_craft$

2. $p\_craft[tseat \rightarrow 350]$

3. $md10[ecoclass \rightarrow 300]$ - inherited from p_craft.

4. $md10[tseat \rightarrow 325]$ - using inherited $tseat$ code in p_craft.

5. $h50[makeen \rightarrow p\&h]$ - by deduction.

It is interesting to observe how we expect $md10[tseat \rightarrow 325]$ rather than $md10[tseat \rightarrow 350]$ from program $\mathbf{P}'$. Since $\nabla([A'], tseat^{0}_{\rightarrow}, md10) = p\_craft$, the code for $tseat^{0}_{\rightarrow}$ is inheritable in $md10$. Then we have the clause

$$md10[tseat \rightarrow T] \leftarrow md10[ecoclass \rightarrow E; firstclass \rightarrow F], T = E + F.$$

by context switching from p_craft to $md10$ in rule 11 (i.e., apply $[p\_craft /\!/ md10]$) as

$$\begin{aligned}(p\_craft[tseat \rightarrow T] \;\; \leftarrow \;\; & p\_craft[ecoclass \rightarrow E; firstclass \rightarrow F], \\ & T = E + F.)[p\_craft /\!/ md10]\end{aligned}$$

which now correctly yields $md10[tseat \rightarrow 325]$. □

Example 3.2 Consider the following program $\mathbf{P}_1 = \langle \Delta_1, \Upsilon_1, \Pi_1 \rangle$ with objects o, p, q and r. Usually, the l- and i-clauses are not specified by the users. Rather the users implicitly assume the locality and inheritability of clauses according to the Definitions 3.2 and 3.4.

$$\Delta_1 := \left| \begin{array}{ll} (1) & o[\,]. \\ (2) & q[\,]. \\ (3) & q : o. \\ (4) & p : o. \\ (5) & r : p. \\ (6) & r : q. \\ (7) & r[t^{0}_{\rightarrow} \infty\!\!\lhd p]. \end{array} \right. \qquad \Upsilon_1 := \emptyset \qquad \Pi_1 := \left| \begin{array}{rl} (8) & o[m \rightarrow X] \leftarrow o[s \rightarrow X], \\ & \quad o[v \rightarrow g]. \\ (9) & o[s \rightarrow 5]. \\ (10) & P[u \rightarrow d] \leftarrow P : o. \\ (11) & p[s \rightarrow 2]. \\ (12) & p[t \rightarrow a]. \\ (13) & q[t \rightarrow c]. \\ (14) & Q[v \rightarrow g]. \end{array} \right.$$

Figure 6: Example program $\mathbf{P}_1$

From Definition 3.2 it follows that clauses (8) and (9) are local to object o only. Similarly clauses (11) and (12) are local to p, and clause (13) is local to q only. However, clause (10) is local to p and q since these are the two objects $\mathbf{P}$ in $\mathbf{P}_1$ for which $P : o$ holds true. Note that (10) is *not* local to o since $o : o$ never holds in ORLog. In fact, it is local to only p and q. If we change $P : o$ to $P :: o$ in the body, then (10) becomes local to all four objects in $\mathbf{P}_1$ since :: is a partial order. Clause

(14) is clearly local to all four objects. This is simply because the descriptor of (14) is a variable and is not constrained by any is-a literal in the body.

Clause (8) is inherited in p since p does not define $m^0_{\rightarrow} = meth(o[m \rightarrow X])$. Similarly, (8) is inherited in q and r. However, clause (9) is only inherited in q since q does not define $s^0_{\rightarrow}$ whereas p has a local definition for $s^0_{\rightarrow}$, i.e., clause (11). It is interesting to note that neither clause (11) nor clause (9) is inherited in r. This is because r can access two distinct definitions of $s^0_{\rightarrow}$ - the one in o and the one in p - via two non-overlapping paths. On the other hand, clause (13) is inherited in r since the definition of $t^0_{\rightarrow}$ in p is inhibited by r through clause (7), and now there is a unique ancestor q of r that defines $t^0_{\rightarrow}$.

The "intended" model of $\mathbf{P}_1$, formalized in section 3.5.3, is the one in which these definitions of locality and inherited clauses are respected. Intuitively, (omitting some obvious atoms) we expect the "intended" model of program $\mathbf{P}_1$ to contain (among others) the following ground atoms. The complete intended model for the program $\mathbf{P}_1$ can be found in Example 3.3.
$\{o[\ ].q[\ ].p : o.q : o,r : p,r : q,.r :: \mathbf{o}(for\ \mathbf{o} = o,p.q.r), q[u \rightarrow d].o[m \rightarrow 5]. o[v \rightarrow g].p[m \rightarrow 2].p[s \rightarrow 2], p[t \rightarrow a].p[u \rightarrow d].p[v \rightarrow g], q[m \rightarrow 5]. q[s \rightarrow 5]. q[t \rightarrow c].q[v \rightarrow g].r[t \rightarrow c], r[v \rightarrow g]\}$.

It is interesting to see how we expect $p[m \rightarrow 2]$ in p. Since clause (8) is inherited in p, it follows by Definition 3.4 that the inherited clause in p is $(8') = (o[m \rightarrow X] \leftarrow o[s \rightarrow X], o[v \rightarrow g].)[o/\!/p] = p[m \rightarrow X] \leftarrow p[s \rightarrow X].p[v \rightarrow g]$. Since we have $p[s \rightarrow 2]$ in clause (11) and $p[v \rightarrow g]$ is an instance of clause (14), we now have $p[m \rightarrow 2]$. In contrast, (8) inherited in q will become $q[m \rightarrow X] \leftarrow q[s \rightarrow X], q[v \rightarrow g]$. In a similar way we can show that $q[s \rightarrow 5]$ follows from clause (9). Hence, we have $q[m \rightarrow 5]$. This is *code reuse* – the code for $m^0_{\rightarrow}$ in o is reused by p, q and r by dynamically interpreting the context o as "*self*", namely, the current context of the method. Nicely enough, we have overriding built into the definition of inherited clauses. Notice that the code for $m^0_{\rightarrow}$ is evaluated in p using the definition for $s^0_{\rightarrow}$ in p, not the definition of $s^0_{\rightarrow}$ in o.

We are now also able to resolve conflicts in inheritance in two different ways – by detection and by preference specification. Observe that clauses (9) and (11) define $s^0_{\rightarrow}$ at both o and p. Let $[A_1]$ denote the ground closure of A_1. According to the definition of $\nabla([A_1], s^0_{\rightarrow}, r)$, r inherits neither since intuitively uniqueness of inheritance of $s^0_{\rightarrow}$ in r is lost. Thus with respect to conflict resolution by detection, we take a conservative approach. In particular, even if p and q (or o in this case) define identical values for $s^0_{\_}$, our semantics would reject both. This is a conservative approach and can be justified by the following facts. Methods are usually defined using programs and are computed. Since testing for equivalence of programs in general is undecidable, we take the view that two definitions in different classes are potentially conflicting. Hence we reject all. For the sake of uniformity, we also take a similar view to a ground case, which is a special case of a method defined via a unit ground atom. However, we go a step further to lend a hand to the users to indicate a preferred inheritance from a particular superclass. In this way, users may resolve conflicts using *withdrawal*. For example, although clauses (12) and (13) define method $t^0_{\_}$ at p and q, r inherits $t^0_{\_}$ from q by inhibiting $t^0_{\_}$ from p through clause (7). This is conflict resolution by preference specification, where the user has full control of how the conflict is to be resolved. □

3.4 Interpretation Structures

A semantic structure $\mathcal{S}$ of the language $\mathcal{L}$ is a tuple of the form $\langle \mathcal{D}, \ll_O, \preceq_O, I_d, I_s, L_{\_}, C_{\_}, W_{\_} \rangle$, where $\mathcal{D}$ is the domain of interpretation, $\ll_O$ is an irreflexive binary relation among objects in $\mathcal{D}$, $\preceq_O$ is a partial order among objects in $\mathcal{D}$ which is induced by $\ll_O$, and $I_d, I_s, L_{\_}, C_{\_}, W_{\_}$ are interpretation functions that assign meaning to the symbols in $\mathcal{L}$. Intuitively, $\mathcal{D}$ includes the objects and basic values needed to interpret the object identities and constants in $\mathcal{L}$. The irreflexive relation $\ll_O$ is the semantic counterpart of the immediate is-a association of objects. The partial order $\preceq_O$ is induced by the $\ll_O$ relation since is-a associations are also non-trivially implied by the immediate is-a relation. Note that, for every object o in $\mathcal{D}$, $o \preceq_O o$ trivially holds, but $o \ll_O o$ does not. Thus, $\preceq_O$ is the reflective transitive closure of $\ll_O$.

Formally, the domain $\mathcal{D}$ of the semantic structure $\mathcal{S}$ includes (i) a set of objects $\mathcal{O}$, (ii) a set of elements $\mathcal{B}$ corresponding to basic values, and (iii) a set of elements, one each corresponding to each type name $\tau \in \mathcal{T}$. For simplicity and clarity, we suppose that the set (iv) above is $\mathcal{T}$ itself. Clearly, there is no loss of generality in doing so. So we can suppose that $\mathcal{D} = \mathcal{B} \cup \mathcal{O} \cup \mathcal{T}$. Notice that $\mathcal{D}$ includes the domain of elements D_τ corresponding to each type $\tau \in \mathcal{T}$.

In order to keep the semantics first-order, we treat objects and classes as well as subclasses and instances in a uniform manner. Thus, an object could be viewed as an instance of its superclass object as well as a subclass of instances below it in the object hierarchy. This mirrors a standard trick in mathematical logic [31] which is adopted here in order to provide a first-order semantics for a language which is syntactically higher order. Models such as "Higher-Order" logic [57] and (earlier) F-logic [47] follow a similar approach.

We now define the interpretation functions of $\mathcal{S}$. The function I_d interprets each symbol in $\mathcal{L}$ as follows[7]:

- I_d associates each object id i in $\mathcal{I}_O$ with a unique object in $\mathcal{O}$.

- I_d maps each constant c of sort τ in $\mathcal{C}$ to an element of the corresponding basic domain D_τ, i.e. $I_d(c) \in D_\tau \subset \mathcal{B}$.

- Each basic type name τ in $\mathcal{T}_B$ is associated with a basic domain of the same type, and each constructed (object) type name τ is associated with an object $o \in \mathcal{O}$ that encodes the domain of objects of type τ in $\mathcal{O}$. That is, $I_d(\tau) = D_\tau \subset \mathcal{B}$ for $\tau \in \mathcal{T}_B$, and $I_d(\tau) = o \in \mathcal{O}$ for $\tau \in \mathcal{T}_O$, where o is the semantic counterpart of the object type[8] τ.

- Each symbol r in $\mathcal{P}$ of type $\tau_1 \times \ldots \times \tau_n$ is assigned a relation on $D_{\tau_1} \times \ldots \times D_{\tau_n}$, i.e. $I_d(r) \subseteq D_{\tau_1} \times \ldots \times D_{\tau_n}$.

- Each symbol m in $\mathcal{M}$ is associated with a set[9] of partial functions of the form $f : D_{\tau_{o_1}} \times \ldots D_{\tau_{o_k}} \times D_{\tau_1} \times \ldots \times D_{\tau_j} \rightsquigarrow [D_\tau \cup 2^{D_\tau}]$, where there is exactly one such function in the set for a fixed type associated with m. That is, $I_d(m) \subseteq \cup_{k=1,j=0,\tau}^{\infty}[D_{\tau_{o_1}} \times \ldots D_{\tau_{o_k}} \times D_{\tau_1} \times \ldots \times D_{\tau_k} \rightsquigarrow [D_\tau \cup 2^{D_\tau}]]$. Here $\tau_{o_i} \in \mathcal{T}_O$, $i = 1, \ldots, k$ and $r_l \in \mathcal{T}$, $l = 0, \ldots, j$, and $\tau_o \in \mathcal{T}_O$.

Note that the set of partial functions associated with m by I_d relates m to a unique function for each type associated with m. This helps us *overload* method names and *refine* them by allowing them to accept any number of arguments, and

[7]Note that in what follows we use the notation $[A \rightsquigarrow B]$ and $[A \longmapsto B]$ to denote the set of all partial and total functions respectively from A to B.

[8]Note that we use object ids $\tau \in \mathcal{T}_O \subseteq \mathcal{I}_O$ to represent object types that are associated with their semantic counterpart – the actual object $o \in \mathcal{O}$ that the id τ represents. It is useful to think of o as a concise representation of the domain corresponding to all its instances in the structure $\mathcal{S}$

[9]Recall that each method has a set of types associated with it.

use different implementations for each type associated with the same method name. Furthermore, we also allow one function per type. For example, we can define a method *child* with a signature $person[child(year) \Rightarrow \{person\}]$ which refers to all the *children* of a *person* in a given *year*, and another method *child* with a signature $person[child(spouse) \Rightarrow \{person\}]$ refers to the set of *children* with a given *spouse*. (Here *person* is the context of the method.) Note that (i) for methods m defined in objects the index $k = 1$ and $D_{\tau_{o_k}}$ refers to the implicit argument (the object acting as the context for the method). (ii) for methods in relationships[10], $k \geq 1$ and in this case $D_{\tau_{o_1}} \times \ldots D_{\tau_{o_k}}$ refers to the (explicit) key of the relation r of types $\tau_1 \times \ldots \times \tau_k$. In this case, the key acts as the context of the method. Also, note that a method may return a single or a set structured value, depending on its signature.

We next consider the function I_s.

- I_s acts as the identity function on each oid $i \in \mathcal{I}_O$, i.e. $I_s(i) = i \in \mathcal{T}_O$.
- Each constant c in $\mathcal{C}$ is associated with its intended type, i.e, $I_s(c) = \tau$, for some type $\tau \in \mathcal{T}_B$.
- I_s acts as the identity function on type names, i.e. $I_s(\tau) = \tau$, $\forall \tau \in \mathcal{T}$.
- Each predicate symbol $r \in \mathcal{P}$ is assigned its type, i.e, $I_s(r) \in \mathcal{T}_O^k$, where k is the arity of r.
- Each symbol m in $\mathcal{M}$ is associated with a set of type tuples of the form $< \tau_{o_1}, \ldots, \tau_{o_j}, \tau_1, \ldots, \tau_k, \{t_1, \ldots, t_n\} >$, which represent the (function) type $\tau_o \times \ldots \times \tau_{o_j} \times \tau_1 \times \ldots \times \tau_k \rightarrow \{t_1, \ldots, t_n\}$. That is, $I_s(m) \subseteq \cup_{j=1,k=0}^{\infty}[\mathcal{T}_O^j \times \mathcal{T}^k \times 2^{\mathcal{T}}]$.

The functions $L_{\mapsto d}$ and $L_{\mapsto s}$ associate with each symbol $m \in \mathcal{M}$ a set of partial functions of the form $f : \mathcal{N} \rightsquigarrow 2^{\mathcal{O}}$, where there is exactly one such function in the set for a fixed arity associated with m, i.e., $L_{\mapsto} : \mathcal{M} \rightarrow [\mathcal{N} \rightsquigarrow 2^{\mathcal{O}}]$. That is, $L_{\mapsto d}(m)(k) \subseteq \mathcal{O}$, and similarly for $L_{\mapsto s}(m)(k) \subseteq \mathcal{O}$. Here k is a natural number in $\mathcal{N} \subset \mathcal{B}$ and corresponds to an arity of the method m. Intuitively, it means that a method definition (similarly a method signature) of arity k is locally available at each object $o \in L_{\mapsto d}(m)(k)$ (similarly, $o \in L_{\mapsto s}(m)(k)$).

[10]In ORLog, we allow relations to have methods, or virtual attributes, as much the same way the objects do. This follows from the underlying semantic data model, the OR model, on which ORLog is based.

The functions $C_{\mapsto d}$ and $C_{\mapsto s}$ associate with each method symbol $m \in \mathcal{M}$ of arity k and object $o \in \mathcal{O}$, a unique object $p \in \mathcal{O}$. Formally $C_{\mapsto} : \mathcal{M} \longrightarrow [\mathcal{N} \longrightarrow [\mathcal{O} \mapsto \mathcal{O}]]$, where each m is associated with a total function $\mathcal{N} \longrightarrow [\mathcal{O} \mapsto \mathcal{O}]$. Intuitively, whenever $C_{\mapsto}(m)(k)(o) = p$, it says that object o may use the definition or signature of m of arity k from the object p. We will later see that defining suitable additional properties for these functions, it is possible to achieve conflict free inheritance of methods and signatures in the object hierarchies and to capture the semantics of withdrawal in a clean declarative manner.

Finally, the functions $W_{\mapsto d}$ and $W_{\mapsto s}$ associate with each method symbol $m \in \mathcal{M}$ of arity k and object $o \in \mathcal{O}$, a set of objects in $\mathcal{O}$. Formally $W_{\mapsto} : \mathcal{M} \longrightarrow [\mathcal{N} \longrightarrow [\mathcal{O} \rightsquigarrow 2^{\mathcal{O}}]]$. That is, whenever $p \in W_{\mapsto}(m)(k)(o)$, it means that method m (both data and signature) if defined in the immediate superclass p of o, is *withdrawn* from o (for inheritance purposes). As a result o can not inherit this method from p.

A variable assignment $\nu : \mathcal{V} \longrightarrow \mathcal{D}$ is a function that assigns each basic domain variable X an element of $\mathcal{B}$ of appropriate sort, and assigns each id variable X an object from $\mathcal{O}$. Variable assignments can be extended recursively to all formulas in the obvious manner.

Let $\mathcal{S}$ be a semantic structure and ν be a variable assignment. An atom $\mathcal{A}$ in $\mathcal{L}$ is true under the semantic structure $\mathcal{S}$ with respect to the variable assignment ν, denoted $\mathcal{S} \models_\nu \mathcal{A}$, iff $\mathcal{S}$ has an object, relationship, or a value with properties specified in $\nu(\mathcal{A})$. Formally:

- For an is-a atom $\boldsymbol{q} :: \boldsymbol{p}$, $\mathcal{S} \models_\nu \boldsymbol{q} :: \boldsymbol{p}$ iff $\nu(\boldsymbol{q}) \preceq_{\mathcal{O}} \nu(\boldsymbol{p})$. This says that object $\nu(\boldsymbol{q})$ is a subclass of object $\nu(\boldsymbol{p})$, equivalently an instance of $\nu(\boldsymbol{p})$, in the semantic structure $\mathcal{S}$[11].

- For an is-a atom $\boldsymbol{q} : \boldsymbol{p}$, $\mathcal{S} \models_\nu \boldsymbol{q} : \boldsymbol{p}$ iff $\nu(\boldsymbol{q}) \ll_{\mathcal{O}} \nu(\boldsymbol{p})$. This says that object $\nu(\boldsymbol{q})$ is an immediate subclass of object $\nu(\boldsymbol{p})$, equivalently an immediate instance of $\nu(\boldsymbol{p})$, in the semantic structure $\mathcal{S}$.

- For a functional d-atom of the form $\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \rightarrow \boldsymbol{v}]$, $\mathcal{S} \models_\nu \boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \rightarrow \boldsymbol{v}]$ iff $I_d(m)(\nu(\boldsymbol{p}), \nu(\boldsymbol{a}_1), \ldots, \nu(\boldsymbol{a}_n)) = \nu(\boldsymbol{v})$. Similarly, for a set-valued d-atom of the form $\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \twoheadrightarrow \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}]$, $\mathcal{S} \models_\nu \boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \twoheadrightarrow \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}]$ iff $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\} \subseteq I_d(m)(\nu(\boldsymbol{p}), \nu(\boldsymbol{a}_1), \ldots, \nu(\boldsymbol{a}_n))$.

[11]Recall the dual treatment of instances and subclasses. This is important for keeping the semantics first-order.

- For an s-atom of the form $p[m(t_1, \ldots, t_n) \overset{s}{\mapsto} \{type_1, \ldots, type_k\}]$, $\mathcal{S} \models_\nu p[m(t_1, \ldots, t_n) \overset{s}{\mapsto} \{type_1, \ldots, type_k\}]$ iff $< \nu(p), t_1, \ldots, t_n, \{type_1, \ldots, type_k\} > \in I_s(m)$.

- For a p-atom $r \diamond t_1, \ldots, t_n$, $\mathcal{S} \models_\nu r \diamond t_1, \ldots, t_n$ iff r is a relation of type $t_1 \times \ldots \times t_n$.

- For an l-atom of the form $p[m^k_{\llcorner}]$, $\mathcal{S} \models_\nu p[m^k_{\llcorner}]$ iff $\nu(p) \in L_{\llcorner}(m)(k)$.

- For an i-atom of the form $p[o@m^k_{\llcorner}]$, $\mathcal{S} \models_\nu p[o@m^k_{\llcorner}]$ iff $C_{\llcorner}(m)(k)(\nu(p)) = \nu(o)$.

- For a w-atom of the form $p[m^k_{\llcorner} \bowtie\!\lhd o]$ (or $o[m^k_{\llcorner} \rhd\!\bowtie p]$), $\mathcal{S} \models_\nu p[m^k_{\llcorner} \bowtie\!\lhd o]$ (or $\mathcal{S} \models_\nu o[m^k_{\llcorner} \rhd\!\bowtie p]$) iff $\nu(o) \in W_{\llcorner}(m)(k)(\nu(p))$.

Satisfaction of complex formulas can be defined inductively in terms of atomic satisfaction. Let ϕ and ψ be any formulae. Then,

- $\mathcal{S} \models_\nu \phi \wedge \psi$ iff $\mathcal{S} \models_\nu \phi$ and $\mathcal{S} \models_\nu \psi$.

- $\mathcal{S} \models_\nu \phi \vee \psi$ iff $\mathcal{S} \models_\nu \phi$ or $\mathcal{S} \models_\nu \psi$.

- $\mathcal{S} \models_\nu \neg\phi$ iff $\mathcal{S} \not\models_\nu \phi$.

- $\mathcal{S} \models_\nu \psi \leftarrow \phi$ iff $\mathcal{S} \not\models_\nu \phi$ or $\mathcal{S} \models_\nu \psi$

- $\mathcal{S} \models_\nu (\forall X)\phi$ if for every μ that agrees with ν everywhere, except possibly on X, $\mathcal{S} \models_\mu \phi$.

- $\mathcal{S} \models_\nu (\exists X)\phi$ if for some μ that agrees with ν everywhere, except possibly on X, $\mathcal{S} \models_\mu \phi$.

Satisfaction of other formulae can be obviously derived from the above. If ψ is a closed formula, its meaning is independent of a variable assignment, and its satisfaction in $\mathcal{S}$ can be simply written as $\mathcal{S} \models \psi$.

Since our goal is to account for inheritance directly into the semantic structures of ORLog, we impose additional restrictions on our interpretation structures. It is essential that every ORLog structure assign unique inheritability of properties in the objects in the structure. We now define the concept of *unique inheritability* of properties in ORLog structures in the light of Definition 3.3.

Definition 3.5 An ORLog structure $\mathcal{S}$ satisfies unique inheritability property if the following holds:
For every $o \in \mathcal{O}$, method symbol m of arity k, the following condition is satisfied:

> if $o \not\in L_{\mapsto}(m)(k)$ & $\exists q$ such that $o :: q$ & $C_{\mapsto}(m)(k)(q) = p$ & $p \in L_{\mapsto}(m)(k)$ &
> (for all other $r \in \mathcal{O} : (o :: r \rightarrow (C_{\mapsto}(m)(k)(r) = r$ &$r \not\in L_{\mapsto}(m)(k)), \vee$
> $C_{\mapsto}(m)(k)(r) = p, \vee o \in W_{\mapsto}(m)(k)(r)))$
> **then** $C_{\mapsto}(m)(k)(o) = p$.
> **In all other cases** $C_{\mapsto}(m)(k)(o) = o$. □

The above definition asserts that for every object o and method denotation $m^k_{\mapsto}$. o can inherit the code (the clasues that define the method m of arity k of the type $\mapsto$) from another object p iff p has a local definition for $m^k_{\mapsto}$, o does not locally define $m^k_{\mapsto}$. some object q (not necessarily distinct from p) which is a superclass of o inherits $m^k_{\mapsto}$ from p, and all other superclasses r of o (if exists) inherits $m^k_{\mapsto}$ either from p. or they do not inherit it from any object, or $m^k_{\mapsto}$ is withdrawn from r and o. Otherwise. o must use its own definition for $m^k_{\mapsto}$. The essence of this difinition is that there must exist a unique path from an object o to a superclass object p for o to be able to inherit a locally defined method $m^k_{\mapsto}$ at p and no object q has a local definition of the same method that is a superclass of o but a subclass of p. Furthermore, all other paths to any superclass r of o which has a local definition for the same method $m^k_{\mapsto}$ are either withdrawn (blocked or inhibited) or are blocked due to inheritance conflict.

The functions I_d and I_s in ORLog structures assign meaning to each symbol in the language independently of each other. Similarly the functions $L_{\mapsto}, C_{\mapsto}$ and $W_{\mapsto}$ assign meaning to symbols in isolation. For example. the data concepts into which symbols are interpreted by I_d should be consistent with the types associated with these symbols by I_s. Hence we require that every ORLog structure satisfy some *goodness* property to be a candidate for a meaningful structure. The following *goodness* properties enforce this consistency[12]. We say that an ORLog structure $\mathcal{S}$ is *admissible* if it satisfies the following goodness properties.

- I_d and I_s satisfy the following properties:

[12]Some of our goodness conditions are reminiscent of the well typing conditions of F-logic [47]. However. since our language and semantic structure are quite different from those of F-logic, there are important differences between the two sets of well typing conditions.

  - For each constant c in $\mathcal{C}$, $I_d(c) \in I_d(I_s(c))$, where $\tau = I_s(c)$ is the type of c and $I_d(\tau) = D_\tau$ is the domain of τ. That is, each constant is mapped to an element of a domain of its intended type.
  - For a method m, an object $o \in \mathcal{O}$, and elements $p_1, \ldots, p_k \in \mathcal{D}$, if $I_d(m)(o, p_1, \ldots, p_k)$ is defined, then (i) there is a type tuple of the form $< \tau_o, \tau_1, \ldots, \tau_k, \{t_1, \ldots, t_l\} >$, in $I_s(m)$, where τ_o is the type of the context o and τ_i is the type of p_i, $i = 1, \ldots, k$, and (ii) if m is a single valued method, then the value $I_d(m)(o, p_1, \ldots, p_k)$ belongs to all types in $\{t_1, \ldots, t_n\}$, and if m is a set-valued method, then all the values in $I_d(m)(o, p_1, \ldots, p_k)$ belong to all types in $\{t_1, \ldots, t_n\}$.
  - For each r in $\mathcal{P}$, if the type of r is $I_s(r) =< \tau_1, \ldots, \tau_n >$, then $I_d(r) \subseteq D_{\tau_1} \times \ldots \times D_{\tau_n}$.

- If a structure $\mathcal{S}$ satisfies a clause of the form $\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \boldsymbol{r}_1 \sharp \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \sharp \boldsymbol{q}_n, \mathcal{G}$, where $\mathcal{G}$ is "is-a free", i.e., $\mathcal{S} \models_\nu \boldsymbol{p}[m(\boldsymbol{a}_1, \ldots \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \boldsymbol{r}_1 \sharp \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \sharp \boldsymbol{q}_n, \mathcal{G}$ then we require that $\mathcal{S} \models_\nu \boldsymbol{p}[m^k_{\mapsto}] \leftarrow \boldsymbol{r}_1 \sharp \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \sharp \boldsymbol{q}_n$ also.

- For every object $o \in \mathcal{O}$ and method denotation $m^k_{\mapsto}$, $C_{\mapsto}$ defines inheritability of $m^k_{\mapsto}$ in o and satisfies the unique inheritability property as defined in Definition 3.5.

- For every o, m, k and p, $o \in W_{\mapsto}(m)(k)(p)$ implies $o \ll_O p$.

ORLog structures also satisfy clauses due to structural and behavioral inheritance in a non-trivial and unique way. To this end, we require that every admissible ORLog structure also satisfy the following closure condition:

Definition 3.6 An admissible ORLog structure is inheritance closed, or *i-closed*, if the following holds.

For any clause $\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \mathcal{G}$,

if $\mathcal{S} \models_\nu \boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \mathcal{G}$ and $\mathcal{S} \models_\nu \boldsymbol{o}[\boldsymbol{p}@m^k_{\mapsto}]$,
then $\mathcal{S} \models_\nu (\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \mathcal{G})[\boldsymbol{p}//\boldsymbol{o}]$, where $[\boldsymbol{p}//\boldsymbol{o}]$ is a clause-wise context substitution that replaces every occurrence of $\boldsymbol{p}$ by $\boldsymbol{o}$. □

3.5 Model Theoretic Semantics

We now develop an Herbrand semantics for our language and introduce the notions of satisfaction and mod 's. Given an ORLog language $\mathcal{L}$, the *Herbrand universe* $\mathcal{U}$ of $\mathcal{L}$ is the set of all ground id-terms - in our case just the individual constants. The *Herbrand base* $\mathcal{H}$ of $\mathcal{L}$ is the set of all ground atoms that can be built from $\mathcal{U}$ and the vocabulary of $\mathcal{L}$. Let $\mathcal{H}_A$ denote the set of id-, r-, l-, is-a and w-atoms, $\mathcal{H}_\Upsilon$ denote the set of i-atoms, and finally $\mathcal{H}_\Pi$ denote the set of pred-, r- and p-atoms (i.e., d-atoms and s-atoms) in $\mathcal{H}$ such that $\mathcal{H} = \mathcal{H}_A \cup \mathcal{H}_\Upsilon \cup \mathcal{H}_\Pi$. An *Herbrand structure* $\mathbf{H}$ of $\mathcal{L}$ is a triple $\langle \mathbf{H}_A, \mathbf{H}_\Upsilon, \mathbf{H}_\Pi \rangle$, where $\mathbf{H}_A \subseteq \mathcal{H}_A$, $\mathbf{H}_\Upsilon \subseteq \mathcal{H}_\Upsilon$, and $\mathbf{H}_\Pi \subseteq \mathcal{H}_\Pi$.

3.5.1 Canonical Models

Ground instances of formulas are defined as in the classical case. We define satisfaction in a manner identical to the classical case.

Definition 3.7 (Herbrand Structures) Let $\mathbf{H}$ be an Herbrand structure. Then

- a ground atom, $\mathcal{A}$, is true in $\mathbf{H}$, denoted $\mathbf{H} \models \mathcal{A}$, iff $\mathcal{A} \in \mathbf{H}$[13];
- a ground negative literal, $\neg\mathcal{A}$, is true in $\mathbf{H}$, denoted $\mathbf{H} \models \neg\mathcal{A}$, iff $\mathcal{A} \notin \mathbf{H}$;
- a ground conjunction of literals is true in $\mathbf{H}$, denoted $\mathbf{H} \models \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_m$, iff $\mathbf{H} \models \mathcal{B}_i, i = 1, \ldots, m$;
- a ground clause cl is true in $\mathbf{H}$, denoted $\mathbf{H} \models \mathcal{A} \leftarrow \mathcal{G}$, iff $\mathbf{H} \models \mathcal{G} \Longrightarrow \mathbf{H} \models \mathcal{A}$; and
- a clause cl is true in $\mathbf{H}$ iff all ground instances of cl are true in $\mathbf{H}$. □

To account for the notion of inheritance outlined in the Example 3.2, we require that every Herbrand structure be "*proper*" in the sense of Definition 3.8 below.

Definition 3.8 (Proper Structures) An Herbrand structure $\mathbf{H}$ of ORLog is called *proper* iff it satisfies the following properties, where o, p, q, r, t_is are arbitrary ground id-terms, m is any method symbol.

[13]An atom $\mathcal{A}$ is in $\mathbf{H}$ iff $\mathcal{A}$ is either in $\mathbf{H}_A$, $\mathbf{H}_\Upsilon$ or $\mathbf{H}_\Pi$.

1. $o[m^{k}_{\mapsto}] \in \mathbf{H}_{\mathcal{A}} \Longrightarrow o :: o \in \mathbf{H}_{\mathcal{A}}$. Also $p : o \in \mathbf{H}_{\mathcal{A}} \Longrightarrow o :: o \in \mathbf{H}_{\mathcal{A}}$, and $o : p \in \mathbf{H}_{\mathcal{A}} \Longrightarrow o :: o \in \mathbf{H}_{\mathcal{A}}$.

2. $o : q \in \mathbf{H}_{\mathcal{A}}$ and $q :: p \in \mathbf{H}_{\mathcal{A}} \Longrightarrow o :: p \in \mathbf{H}_{\mathcal{A}}$.

3. Whenever $\mathbf{H}$ satisfies a ground p-clause $o[m(t_1, \ldots, t_k) \mapsto t] \leftarrow r_1 \sharp q_1, \ldots, r_n \sharp q_n, \mathcal{G}$, where $\mathcal{G}$ is "is-a free", we require that $\mathbf{H}$ also satisfy the ground l-clause $o[m^{k}_{\mapsto}] \leftarrow r_1 \sharp q_1, \ldots, r_n \sharp q_n$.

4. $o[m^{k}_{\mapsto s} \triangleright\!\circ p] \in \mathbf{H}_{\mathcal{A}} \Longrightarrow o[m^{k}_{\mapsto d} \triangleright\!\circ p] \in \mathbf{H}_{\mathcal{A}}$. Similarly, $o[m^{k}_{\mapsto s} \propto\!\triangleleft p] \in \mathbf{H}_{\mathcal{A}} \Longrightarrow o[m^{k}_{\mapsto d} \propto\!\triangleleft p] \in \mathbf{H}_{\mathcal{A}}$.

5. Nothing else is in $\mathbf{H}_{\mathcal{A}}$.

6. Whenever (i) $p[m^{k}_{\mapsto}] \in \mathbf{H}_{\mathcal{A}}$, (ii) for some o, $o :: p \in \mathbf{H}_{\mathcal{A}}$, and (iii) $\nabla(\mathbf{H}_{\mathcal{A}}, m^{k}_{\mapsto}, o) = p$, we have $o[p \hat{@} m^{k}_{\mapsto}] \in \mathbf{H}_{\Upsilon}$.

7. Whenever $\mathbf{H}$ satisfies a ground p-clause $cl = o[m(t_1, \ldots, t_k) \mapsto t] \leftarrow \mathcal{G}$ and $p[o \hat{@} m^{k}_{\mapsto}] \in \mathbf{H}_{\Upsilon}$, we require that $\mathbf{H}$ also satisfy the ground p-clause $cl' = (o[m(t_1, \ldots, t_k) \mapsto t] \leftarrow \mathcal{G})[o /\!/ p]$, which denotes the clause obtained from cl by replacing every occurrence of o by p. (See also Definition 3.4.) $\square$

The purpose of a proper structure is intuitive and is a model-theoretic counterpart of the informal semantics of inheritance we discussed earlier. Conditions (1)-(2) establish "::" as the reflexive transitive closure of ":". Condition (3) says whenever $\mathbf{H}$ satisfies a ground p-clause, it also asserts the locality of the p-clause. (4) and (5) say signature withdrawal implies withdrawal of the corresponding method. (6) says the inheritability as defined by the operator ∇ (see Definition 3.3) must be respected. (7) correctly enforces what it means for a structure to respect behavioral inheritance. This is accomplished by the context switch $[o /\!/ p]$.

In practice, as in the case of F-logic, we only want those models of a program where the method data respects the signature definitions associated with that method. This notion of "well-typing" is left outside the proof theory again as in F-logic, but can be dealt with using an approach similar to that adopted in [47]. We do not elaborate on this issue further here, as our main interest here is in inheritance.

Definition 3.9 (Proper Models) An Herbrand structure $\mathbf{M}$ is a *proper model* of a program $\mathbf{P}$ iff it is proper and for every ground instance cl of any clause $Cl \in \mathbf{P}$, $\mathbf{M} \models cl$. □

Definition 3.10 (Local Properness of Models) Let $\mathbf{P}$ be a program and $\mathbf{M}$ be any model such that $\mathbf{M} = \langle \mathbf{M}_\Lambda, \mathbf{M}_\Upsilon, \mathbf{M}_\Pi \rangle$. Then $\mathbf{M}_\Lambda$ is *locally proper* if it satisfies conditions (1) through (5) of definition 3.8. Likewise, $\mathbf{M}_\Upsilon$ and $\mathbf{M}_\Pi$ are locally proper if they satisfy condition (6) and (7) of definition 3.8 respectively. Furthermore, $\mathbf{M}$ is proper, if $\mathbf{M}_\Lambda, \mathbf{M}_\Upsilon$, and $\mathbf{M}_\Pi$ are locally proper. □

However, not every proper model is an "intended" model. An intended model must respect certain consistency criteria. These are formalized below. In Definition 3.11 below and the sequel, we treat equality as syntactic identity.

Definition 3.11 (I-consistent and Canonical Models) A proper structure $\mathbf{M}$ is *inheritance consistent*, or *i-consistent*, iff the following conditions are satisfied:

1. $o :: p \in \mathbf{M}_\Lambda$ and $p :: o \in \mathbf{M}_\Lambda \Longrightarrow o = p$ – no cycles in object hierarchy.
2. $o[p@m^k_{\mapsto}] \in \mathbf{M}_\Upsilon$ and $o[m^k_{\mapsto}] \in \mathbf{M}_\Lambda \Longrightarrow p = o$ – overriding is respected.
3. $o[m(a_1, \ldots, a_k) \rightarrow v] \in \mathbf{M}_\Pi$ and $o[m(a_1, \ldots, a_k) \rightarrow w] \in \mathbf{M}_\Pi \Longrightarrow v = w$ – functionality of methods is not violated.
4. $o[p@m^k_{\mapsto}] \in \mathbf{M}_\Upsilon$ and $o[r@m^k_{\mapsto}] \in \mathbf{M}_\Upsilon \Longrightarrow p = r$ – uniqueness of inheritability is respected.

A *canonical model* of a program $\mathbf{P}$ is a model of $\mathbf{P}$ that is i-consistent as an Herbrand structure. □

A program is *i-consistent* if it has a canonical model. In the sequel we only consider i-consistent programs.

Example 3.3 The table below in Figure 3.3 shows a canonical model $\mathbf{M}_{\mathbf{P}_1}$ of the program $\mathbf{P}_1$ of Example 3.2, where we use molecular abbreviations like $o[m^0_{\mapsto}; s^0_{\mapsto}; v^0_{\mapsto}] \equiv o[m^0_{\mapsto}] \wedge o[s^0_{\mapsto}] \wedge o[v^0_{\mapsto}]$, $o[m \rightarrow 5; s \rightarrow 5] \equiv o[m \rightarrow 5] \wedge o[s \rightarrow 5]$, etc. The model $\mathbf{M}_{\mathbf{P}_1}$ below formally captures the ideas informally discussed in Example 3.2 concerning locality and inheritability. We shall see later in section 3.5.3 that $\mathbf{M}_{\mathbf{P}_1}$ above is indeed the *intended* model of $\mathbf{P}_1$, in a sense to be made precise (see also Example 3.2).

	Object o	Object p	Object q	Object r
immediate is-a		$p : o$	$q : o$	$r : p, r : q$
transitive is-a	$o :: o$	$p :: p, p :: o$	$q :: q, q :: o$	$r :: r, r :: p,$ $r :: q, r :: o$
locality	$o[m^{0}_{\rightarrow}; s^{0}_{\rightarrow};$ $v^{0}_{\rightarrow}]$	$p[s^{0}_{\rightarrow}; t^{0}_{\rightarrow};$ $u^{0}_{\rightarrow}; v^{0}_{\rightarrow}]$	$q[t^{0}_{\rightarrow}; u^{0}_{\rightarrow};$ $v^{0}_{\rightarrow}]$	$r[v^{0}_{\rightarrow}]$
inheritability	$o[o@m^{0}_{\rightarrow};$ $o@s^{0}_{\rightarrow};$ $o@v^{0}_{\rightarrow}]$	$p[o@m^{0}_{\rightarrow};$ $p@s^{0}_{\rightarrow}; p@t^{0}_{\rightarrow};$ $p@u^{0}_{\rightarrow}; p@v^{0}_{\rightarrow}]$	$q[o@m^{0}_{\rightarrow};$ $o@s^{0}_{\rightarrow}; q@t^{0}_{\rightarrow};$ $q@u^{0}_{\rightarrow}; q@v^{0}_{\rightarrow}]$	$r[o@m^{0}_{\rightarrow};$ $q@t^{0}_{\rightarrow};$ $r@v^{0}_{\rightarrow}]$
withdrawal				$r[t^{0}_{\rightarrow} \propto\!\!\lhd p]$
data	$o[m \rightarrow 5;$ $s \rightarrow 5;$ $v \rightarrow g]$	$p[m \rightarrow 2; s \rightarrow 2;$ $t \rightarrow a; u \rightarrow d;$ $v \rightarrow g]$	$q[m \rightarrow 5; s \rightarrow 5;$ $t \rightarrow c; u \rightarrow d;$ $v \rightarrow g]$	$r[t \rightarrow c;$ $v \rightarrow g]$

Figure 7: Intended model $\mathbf{M}_1$ for the example program $\mathbf{P}_1$.

3.5.2 Correspondence between Herbrand Structures and ORLog Structures

The correspondence between Herbrand structures and general ORLog structures can be stated in a way similar to [47] as follows : Given a general structure for a set of clauses $\mathbf{S}$, the corresponding Herbrand structure is the set of ground atoms that are true in the general structure. Conversely, for an Herbrand structure, $\mathbf{H}$, the corresponding general ORLog structure, $\mathbf{I}_H = \langle \mathcal{D}, \ll_O, \preceq_O, I_s, I_d, L_{\mapsto}, C_{\mapsto}, W_{\mapsto} \rangle$, is defined as follows:

- The domain $\mathcal{D}$ is identical to $\mathcal{U}$.
- The orderings $\preceq_O$ and $\ll_O$ are derived from the transitive and immediate is-a assertions in $\mathbf{H}_A$: For all $p, q \in \mathcal{D}$, we assert $q \preceq_O p$ iff $q :: p \in \mathbf{H}_A$, and $q \ll_O p$ iff $q : p \in \mathbf{H}_A$.
- $I_s(m)(o, t_1, \ldots, t_k) = \begin{cases} \{t\} & \text{if } o[m(t_1, \ldots, t_k) \overset{s}{\mapsto} t] \in \mathbf{H}_H \\ \text{undefined} & \text{otherwise} \end{cases}$
- $I_d(m)(o, t_1, \ldots, t_k) = \begin{cases} t & \text{if } o[m(t_1, \ldots, t_k) \rightarrow t] \in \mathbf{H}_H \\ \text{undefined} & \text{otherwise} \end{cases}$
- $I_d(m)(o, t_1, \ldots, t_k) = \begin{cases} \{t\} & \text{if } o[m(t_1, \ldots, t_k) \twoheadrightarrow t] \in \mathbf{H}_H \\ \text{undefined} & \text{otherwise} \end{cases}$

- $L_{\mapsto}(m)(k) = \{\{o\} \mid o[m^k_{\mapsto}] \in \mathbf{H}_A\}$.
- $C_{\mapsto}(m)(k)(o) = \{p \mid o[p \trianglelefteq m^k_{\mapsto}] \in \mathbf{H}_T\}$.
- $W_{\mapsto}(m)(k)(o) = \{\{p\} \mid o[m^k_{\mapsto} \propto\!\lhd p] \in \mathbf{H}_A$ or $p[m^k_{\mapsto} \rhd\!\circ o] \in \mathbf{H}_A\}$.

It is easy to see that $\mathbf{I}_H = \langle \mathcal{D}, \ll_O, \preceq_O, I_s, I_d, L_{\mapsto}, C_{\mapsto}, W_{\mapsto} \rangle$ is well-defined and is indeed an ORLog structure. The following proposition, adapted from [47], follows from the above correspondence between the two structures. The proof for this proposition is similar to the corresponding proof in [47] and hence omitted.

Proposition 3.1 Let $\mathbf{S}$ be a set of clauses. Then $\mathbf{S}$ is unsatisfiable iff $\mathbf{S}$ has no Herbrand model.
Proof It is easy to verify that for every Herbrand structure $\mathbf{H}$, the entailment $\mathbf{H} \models \mathbf{S}$ takes place if and only if $\mathbf{I}_H \models \mathbf{S}$, where $\mathbf{I}_H$ is the ORLog structure corresponding to $\mathbf{H}$ as defined above. □

3.5.3 Intended Models

We define the declarative semantics of an ORLog program $\mathbf{P}$ as the least canonical model $\mathbf{M_P}$ of $\mathbf{P}$. An ordering elation $\sqsubseteq$ over Herbrand structures can be derived from the partial order $\subseteq$ (set inclusion) as in the classical case. For any program $\mathbf{P}$, if $I_1 = \langle I_{1_A}, I_{1_T}, I_{1_\Pi} \rangle$ and $I_2 = \langle I_{2_A}, I_{2_T}, I_{2_\Pi} \rangle$ are two structures, then

$$I_1 \sqsubseteq I_2 \Longleftrightarrow I_{1_A} \subseteq I_{2_A},\ I_{1_T} \subseteq I_{2_T},\ I_{1_\Pi} \subseteq I_{2_\Pi}$$

Consequently, for every program, the set of associated structures $\mathcal{P}(\mathcal{H})$ is a complete lattice L with join and meet operators defined respectively as follows.

$$\begin{array}{llcl} \text{(join)} & I_1 \sqcup I_2 & = & \langle I_{1_A} \cup I_{2_A}, I_{1_T} \cup I_{2_T}, I_{1_\Pi} \cup I_{2_\Pi} \rangle \\ \text{(meet)} & I_1 \sqcap I_2 & = & \langle I_{1_A} \cap I_{2_A}, I_{1_T} \cap I_{2_T}, I_{1_\Pi} \cap I_{2_\Pi} \rangle \end{array}$$

$I_\perp = \langle \emptyset, \emptyset, \emptyset \rangle$ and $I_\top = \langle \mathcal{H}_A, \mathcal{H}_T, \mathcal{H}_\Pi \rangle$ denote respectively the bottom and top elements of this lattice. $I_\top$ may be verified to be a proper model (see Definition 3.9) of any i-consistent program. However, it is not necessarily an i-consistent model itself (see Definition 3.11) and hence not canonical. Clearly, our interest is only in canonical models of programs. First we show that the model intersection property holds for proper models.

Theorem 3.1 (Model Intersection Property) Let $\mathbf{M}_1$ and $\mathbf{M}_2$ be two proper models of a program $\mathbf{P}$. Then $\mathbf{M}_1 \sqcap \mathbf{M}_2$ is also a proper model of $\mathbf{P}$.

Proof: We show that for any ground clause cl, if $\mathbf{M}_1 \models cl$ and $\mathbf{M}_2 \models cl$, then $\mathbf{M}_1 \sqcap \mathbf{M}_2 \models cl$, and that if $\mathbf{M}_1 \sqcap \mathbf{M}_2$ is not proper then either $\mathbf{M}_1$ or $\mathbf{M}_2$ is not proper as well. We prove this proposition in two steps.

$\mathbf{M}_1 \sqcap \mathbf{M}_2$ *is a Model*:

If cl is a unit clause, then it must be the case that $cl \in (\mathbf{M}_1 \sqcap \mathbf{M}_2)$ since $\mathbf{M}_i \models cl \Longrightarrow cl \in \mathbf{M}_i$, for $i = 1,2$. Hence the claim follows immediately. If cl is a clause of the form $\mathcal{A} \leftarrow \mathcal{G}$, then the implications $\mathbf{M}_i \models \mathcal{G} \Longrightarrow \mathbf{M}_i \models \mathcal{A}$, for $i = 1,2$ hold true by hypothesis. We then proceed by induction on the structure of $\mathcal{G}$.

Basis: If $\mathcal{G}$ is atomic, then $\mathbf{M}_i \models \mathcal{G} \Longrightarrow \mathcal{G} \in \mathbf{M}_i$, for $i = 1,2$, and the proof is identical as for the unit clause case.

Inductive Step: Assume that the claim holds true for any goal $\mathcal{G}$. Consider now that $\mathcal{G}' = \mathcal{G}_1, \mathcal{G}_2$, and assume that $\mathbf{M}_i \models \mathcal{G}'$ holds for $i = 1,2$. Then by definition $\mathbf{M}_i \models \mathcal{G}_1$ and $\mathbf{M}_i \models \mathcal{G}_2$, $i = 1,2$. This implies that $\mathcal{G}_1 \in \mathbf{M}_i$ and $\mathcal{G}_2 \in \mathbf{M}_i$, $i = 1,2$. Then by inductive hypothesis $(\mathbf{M}_1 \sqcap \mathbf{M}_2) \models \mathcal{G}_1$ and $(\mathbf{M}_1 \sqcap \mathbf{M}_2) \models \mathcal{G}_2 \Longrightarrow (\mathbf{M}_1 \sqcap \mathbf{M}_2) \models \mathcal{G}_1, \mathcal{G}_2$.

Properness of $\mathbf{M}_1 \sqcap \mathbf{M}_2$:

Observe that a canonical model is always proper but the converse is not always true. Hence a proper model $\mathbf{M}$ corresponding to a program $\mathbf{P}$ is not required to be consistent. By definition 3.8, properness of $\mathbf{M}$ only ensures (i) the partial ordering of the is-a hierarchy, (ii) locality of p-clauses, (iii) withdrawals implied by $\mathbf{P}$, (iv) minimality of $\mathbf{M}_\mathcal{A}$, (v) exact inheritability of p-clauses implied by the program and finally (vi) inheritance of p-clauses captured by the inheritability assertions in (iv). Note that for any program $\mathbf{P}$, and for any two proper models $\mathbf{M}_1$ and $\mathbf{M}_2$ of $\mathbf{P}$, $\mathbf{M}_{1_\mathcal{A}} = \mathbf{M}_{2_\mathcal{A}}$ holds true by condition (5) of definition 3.8.

From the minimality of $\mathbf{M}_\mathcal{A}$, the property of inheritability function ∇, and the condition (6) of proper structures, it follows that every proper model of $\mathbf{P}$, $\mathbf{M}_\mathcal{T}$ to be precise, satisfies a minimum set of i-atoms depending on the $\mathbf{M}_\mathcal{A}$ component of the model that is common to every proper model of $\mathbf{P}$.

Now since $\mathbf{M}_1$ and $\mathbf{M}_2$ are two proper models by hypothesis, then $\mathbf{M}_{1_\mathcal{A}} = \mathbf{M}_{2_\mathcal{A}}$. Let us assume that $\mathbf{M}_1 \sqcap \mathbf{M}_2$ is not proper Then there are three possible cases: either (i) $\mathbf{M}_{1_\mathcal{A}} \cap \mathbf{M}_{2_\mathcal{A}}$, (ii) $\mathbf{M}_{1_\mathcal{T}} \cap \mathbf{M}_{2_\mathcal{T}}$, or (iii) $\mathbf{M}_{1_\Pi} \cap \mathbf{M}_{2_\Pi}$ is not locally proper.

Case (i): Assume $(\mathbf{M}_1 \sqcap \mathbf{M}_2)_\mathcal{A} = \mathbf{M}_{1_\mathcal{A}} \cap \mathbf{M}_{2_\mathcal{A}}$ is not locally proper. Note that from condition (5) of definition 3.8, $\mathbf{M}_{1_\mathcal{A}} \cap \mathbf{M}_{2_\mathcal{A}} = \mathbf{M}_{1_\mathcal{A}} = \mathbf{M}_{2_\mathcal{A}}$. It can now be verified that $(\mathbf{M}_1 \sqcap \mathbf{M}_2)_\Lambda$ satisfies each of the conditions (1) through (5) of definition 3.8. and hence is locally proper.

Case (ii): Assume that $(\mathbf{M}_1 \sqcap \mathbf{M}_2)_\Upsilon = \mathbf{M}_{1_\Upsilon} \cap \mathbf{M}_{2_\Upsilon}$ is not locally proper. Then it must be the case that for some objects o and p, and a method denotation $m^k_\sqcup$, either (i) $\nabla(\mathbf{M}_{1_\Lambda}, m^k_\sqcup, o) = p$, and $o[p@m^k_\sqcup] \in \mathbf{M}_{1_\Upsilon}$ but $o[p@m^k_\sqcup] \notin \mathbf{M}_{1_\Upsilon} \cap \mathbf{M}_{2_\Upsilon}$, or (ii) $\nabla(\mathbf{M}_{2_\Lambda}, m^k_\sqcup, o) = p$, and $o[p@m^k_\sqcup] \in \mathbf{M}_{2_\Upsilon}$ but $o[p@m^k_\sqcup] \notin \mathbf{M}_{1_\Upsilon} \cap \mathbf{M}_{2_\Upsilon}$. But this is not possible since $\mathbf{M}_{1_\Lambda} = \mathbf{M}_{2_\Lambda}$, and for objects o and p, and method denotation $m^k_\sqcup$, $\nabla(\mathbf{M}_{1_\Lambda}, m^k_\sqcup, o) = \nabla(\mathbf{M}_{2_\Lambda}, m^k_\sqcup, o) = p$, and $o[p@m^k_\sqcup]$ must be in both $\mathbf{M}_{1_\Upsilon}$ and $\mathbf{M}_{2_\Upsilon}$. hence in $\mathbf{M}_{1_\Upsilon} \cap \mathbf{M}_{2_\Upsilon}$ making $\mathbf{M}_{1_\Upsilon} \cap \mathbf{M}_{2_\Upsilon}$ a proper model. A contradiction.

Case (iii): Suppose $(\mathbf{M}_1 \sqcap \mathbf{M}_2)_\Pi = \mathbf{M}_{1_\Pi} \cap \mathbf{M}_{2_\Pi}$ is not locally proper, i.e., it violates condition (7) of definition 3.8. So, let $\mathbf{M}_{1_\Pi} \cap \mathbf{M}_{2_\Pi}$ satisfy a ground p-clause $cl = o[m(t_1, \ldots, t_k) \mapsto t] \leftarrow \mathcal{G}$ and suppose that $p[o@m^k_\sqcup] \in \mathbf{M}_{1_\Upsilon} \cap \mathbf{M}_{2_\Upsilon}$ but $\mathbf{M}_{1_\Pi} \cap \mathbf{M}_{2_\Pi} \not\models cl[o/\!/p]$. This implies that $\mathbf{M}_{i_\Pi} \models cl$, and $p[o@m^k_\sqcup] \in \mathbf{M}_{i_\Upsilon}$. $i = 1, 2$. By the properness of $\mathbf{M}_i$, we also have $\mathbf{M}_i \models cl[o/\!/p]$, $i = 1, 2$. which implies $\mathbf{M}_{1_\Pi} \cap \mathbf{M}_{2_\Pi} \models cl[o/\!/p]$. contradicting the above. □

We have already seen that every program has at least one proper model. namely $I_\top$. In view of Theorem 3.1, we can conclude that every program $\mathbf{P}$ has also a *least* proper model defined as $\mathbf{M_P} = \sqcap\{\mathbf{M} \mid \mathbf{M} \text{ is a proper model of } \mathbf{P}\}$. But what can we say about canonical models of $\mathbf{P}$. since we would like to declare the intended meaning of a program as its least canonical model? The answer is given in Theorem 3.2.

Theorem 3.2 Let $\mathbf{P}$ be an i-consistent program and $\mathbf{M}$ be its least proper model. Then $\mathbf{M}$ is i-consistent. Furthermore, $\mathbf{M}$ is the least canonical model of $\mathbf{P}$.

Proof: Let $\mathbf{P}$ be an i-consistent program, and $\mathbf{M}$ be its least proper model. First. recall that every canonical model. by definition, is proper. This implies, for every ground atom $\mathcal{A} \in M$, $\mathcal{A}$ is in every proper model of $\mathbf{P}$, and hence in every canonical model of $\mathbf{P}$. Let $\mathbf{N}$ be any canonical model. The above argument shows that $\mathbf{M} \subseteq \mathbf{N}$. This implies that $\mathbf{M}$ is canonical, since no superset of an i-inconsistent model can be i-consistent. Since $\mathbf{M}$ is included in every canonical model, it follows that $\mathbf{M}$ is the least canonical model of $\mathbf{P}$. □

From Theorem 3.2, it follows that every i-consistent program $\mathbf{P}$ has a least canonical model $\mathbf{M_P}$. We call $\mathbf{M_P}$ the *intended model* of $\mathbf{P}$ and say that the declarative semantics of a program is given by its intended model. It can be verified that the canonical model $\mathbf{M}_{\mathbf{P}_1}$ in Example 3.3 is the intended model of $\mathbf{P}_1$.

Definition 3.12 (Logical Entailment) Let $\mathbf{P}$ be an ORLog program and $\mathcal{A}$ any ground atom. We say that $\mathbf{P}$ *logically implies* $\mathcal{A}$, denoted $\mathbf{P} \models \mathcal{A}$, provided $\mathcal{A}$ is true in every canonical model of $\mathcal{A}$. We say that two ORLog programs $\mathbf{P}_1$ and $\mathbf{P}_2$ are *equivalent* provided they have the same class of canonical models. □

3.6 Proof Theory

In this section, we develop a proof theory for ORLog and establish that it is sound and complete with respect to the *intended model* semantics developed in Section 3.5. Intuitively, our approach to proof theory is summarized by saying that we pre-compute the "closure" (see Definition 3.6.1 for a formal definition) of an ORLog program $\mathbf{P}$ with respect to the set of all is-a, l-, w-, id- and i-atoms that are entailed by $\mathbf{P}$ in the sense of Definition 3.12. The rationale for this approach is as follows. (1) We can show that a program is equivalent to its closure. (2) Because of the separation between the Λ, Υ and Π components of a program, and the static nature of the is-a hierarchy, the closure of an ORLog program can be *effectively* pre-computed. (3) This approach makes the determination of inheritability simpler, and helps keep the proof theory modular and cleaner. Besides, we are able to ensure that the inheritability is deterministic.

3.6.1 Closure of a Program

Let $\mathbf{P} = \langle \Lambda, \Upsilon, \Pi \rangle$ be a definite ORLog program. We first define a pre-closure of $\mathbf{P}^* = \langle \Lambda^*, \Upsilon^*, \Pi^* \rangle$ as follows. Recall that the locality of clauses is not usually supplied by the programmers. By taking the pre-closure, we account for the locality of clauses in $\mathbf{P}$. The *pre-closure* of a program $\mathbf{P}$ is the smallest set of clauses $\mathbf{P}^*$ satisfying $\mathbf{P} \subseteq \mathbf{P}^*$, and the conditions below.

1. $\Lambda \subseteq \Lambda^*, \Upsilon = \Upsilon^*, \Pi = \Pi^*$.

2. Whenever a p-clause $\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \boldsymbol{r}_1 \mathbin{\natural} \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \mathbin{\natural} \boldsymbol{q}_n, \mathcal{G} \in \Pi^*$, where $\mathcal{G}$ is "is-a free", we have $\boldsymbol{p}[m^k_{\mapsto}] \leftarrow \boldsymbol{r}_1 \mathbin{\natural} \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \mathbin{\natural} \boldsymbol{q}_n \in \Lambda^*$.

3. $\boldsymbol{p}[m^k_{\mapsto}, \rhd\!\circ \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^* \Longrightarrow \boldsymbol{p}[m^k_{\mapsto_d} \rhd\!\circ \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^*$.

4. $\boldsymbol{p}[m^k_{\mapsto}, \circ\!\lhd \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^* \Longrightarrow \boldsymbol{p}[m^k_{\mapsto_d} \circ\!\lhd \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^*$.

5. $P :: P \leftarrow P[m^k_\llcorner] \in \Lambda^*$.

6. $P :: P \leftarrow P : Q \in \Lambda^*$.

7. $P :: P \leftarrow Q : P \in \Lambda^*$.

8. $P :: Q \leftarrow P : R, R :: Q \in \Lambda^*$.

For a set of clauses Σ, we let $[\Sigma]$ denote its ground closure, which is the set of ground atoms derivable in the classical sense from its Herbrand instantiation. Let $\mathbf{P}^* = \langle \Lambda^*, \Upsilon^*, \Pi^* \rangle$ be the closure of a program $\mathbf{P}$ and let $\mathbf{P}^*_g = \langle [\Lambda^*], [\Upsilon^*], \Pi^* \rangle$. It is important to note here that we do not require a ground closure of the component Π^* of $\mathbf{P}^*$, which presumably is the largest component of $\mathbf{P}^*$. We refer to $\mathbf{P}^*_g$ as the partial ground closure of $\mathbf{P}^*$.

Finally, the *closure* $\mathbf{P}_c$ of a program $\mathbf{P}^*$ is obtained from $\mathbf{P}^*_g$ by applying the operator ∇ as follows.

1. $\Lambda_c = [\Lambda^*], \Upsilon_c \supseteq [\Upsilon^*]$ and $\Pi_c = \Pi^*$.

2. for every $o[m^k_\llcorner] \in \Lambda_c$ and $p :: o \in \Lambda_c$, $\nabla(\Lambda_c, m^k_\llcorner, p) = o \Longrightarrow p[o \hat{@} m^k_\llcorner] \in \Upsilon_c$.

3. Nothing else is in Υ_c.

Theorem 3.3 below establishes that a program is equivalent to its closure.

Theorem 3.3 Let $\mathbf{P}$ be a program, $\mathbf{P}_c$ be its closure. Then $\mathbf{P}$ and $\mathbf{P}_c$ are logically equivalent.

Proof: We show that an Herbrand structure is a proper model of $\mathbf{P}$ iff it is a proper model of $\mathbf{P}_c$ by showing that there is a one-one relationship between the conditions that an Herbrand structure must satisfy to be proper and the axioms in the set $(\mathbf{P}_c - \mathbf{P})$.

Conditions (1) through (4) in definition 3.8 are satisfied by an Herbrand model $\mathbf{M}$ of a program $\mathbf{P}$ iff $\mathbf{M}$ satisfies the rules (1) through (8) that are added to the pre-closure of $\mathbf{P}$. Rules (9) and (10) added to the closure of $\mathbf{P}$ exactly capture condition (6) of definition 3.8. Since Λ and Λ^* are logically equivalent, in the sense that considered as programs by themselves, their classes of (classical) Herbrand models are the same, the operator ∇ behaves identically on Λ and Λ^*. From this, it follows that $\mathbf{P}$ and $\mathbf{P}_c$ have the same classes of proper models, which implies the theorem. □

3.6.2 Proof Rules

In this section we develop a goal directed sequent style proof system for ORLog programs. We adopt this style along the lines of [59, 60]. In Figure 8 below, we present four inference rules which define the properties of the proof predicate $\vdash$. We use the notation $\mathbf{P}_c \vdash_\theta \mathcal{G}$ to represent the fact that the goal $\mathcal{G}$ is derivable from the closed program $\mathbf{P}_c$ with a substitution[14] θ, i.e., $\mathbf{P}_c \vdash \mathcal{G}\theta$. Following [60], the structure of the proof rules are as shown below where we read them from bottom up. Note that the application of a proof rule is contingent upon the satisfaction of the conditions specified at the right hand side of eacn rule. The inference figures or proof rules for ORLog are shown in Figures 8 and 9. We use θ, ϕ, etc. to represent most general unifiers, and ϵ to represent the empty substitution. $\Box$ denotes the empty goal, which is always true. Our proof theory consists of four inference rules - EMPTY, AND, DEDUCTION and INHERITANCE. For the sake of clarity, we present the first three inference rules first and explain their intuition and our conventions. Then, we present the last inference rule.

$$\text{(EMPTY)} \qquad \frac{}{\mathbf{P}_c \vdash_\epsilon \Box}$$

$$\text{(AND)} \qquad \frac{\mathbf{P}_c \vdash_\theta \mathcal{G}_1 \qquad \mathbf{P}_c \vdash_\phi \mathcal{G}_2[\theta]}{\mathbf{P}_c \vdash_{\theta\phi} \mathcal{G}_1 \wedge \mathcal{G}_2}$$

$$\text{(DEDUCTION)} \qquad \frac{\mathbf{P}_c \vdash_\sigma \mathcal{G}[\theta]}{\mathbf{P}_c \vdash_{\theta\sigma} \mathcal{A}} \qquad \left(\begin{array}{l} C = \mathcal{A}' \leftarrow \mathcal{G} \in \mathbf{P}_c, \\ \theta = mgu(\mathcal{A}, \mathcal{A}') \end{array}\right)$$

Figure 8: Classical inference rules.

As in [59], a proof for $\mathbf{P}_c \vdash \mathcal{G}\theta$ is a tree rooted at $\mathbf{P}_c \vdash_\theta \mathcal{G}$ with internal nodes that are instances of one of the four inference rules and with the leaf nodes that are labelled with the figure EMPTY. The *height* of a proof is the maximum of the number of nodes in all the branches in the proof tree, and the *size* of a proof is the number of nodes in the proof tree.

The interpretation of the provability relation $\vdash$ is straightforward. The first three rules capture the operational semantics of classical logic with the only difference that

[14]The classical notions of substitution and unification can be adapted to ORLog with minor modifications.

our language has a different syntax, and an elaborate set of atoms. The rule EMPTY says that the empty goal can be proved with the empty substitution. The rule AND for conjunctive goal is intuitive: to prove $\mathcal{G}_1, \mathcal{G}_2$, we have to prove $\mathcal{G}_1$ and $\mathcal{G}_2$ from left to right as in Prolog. The rule DEDUCTION for logical inference (or back-chaining) is as expected. To prove $\mathcal{A}$, we need to prove the body $\mathcal{G}$ of a clause $\mathcal{A}' \leftarrow \mathcal{G}$ such that $\mathcal{A}$ and $\mathcal{A}'$ unify with θ. In this context, we would like to make the following remark about the composition of substitutions in our proof rules.

Consider, for example, the deduction rule. Initially the lower sequent is $\mathbf{P}_c \vdash \mathcal{A}$ that states that $\mathcal{A}$ is derivable from $\mathbf{P}_c$. Now to prove $\mathcal{A}$ from the clause $\mathcal{A}' \leftarrow \mathcal{G}$, we incur a substitution θ on $\mathcal{A}$ through which $\mathcal{A}$ and $\mathcal{A}'$ unify. Then we are left to prove $\mathcal{G}\theta$, since θ should be applied to the body as well. To prove $\mathcal{G}\theta$ we may similarly incur another substitution σ. Note that similar to classical logic it is customary that $\mathcal{A}$ must inherit the substitutions that are applied to $\mathcal{G}$ at a later proof. Thus σ must be applied to $\mathcal{A}$ after we apply θ since θ was incurred first. This explains our convention of right composing the substitutions in the proof rules. We denote by $\theta\sigma$ the composition of substitutions θ with σ as defined in [55].

Finally, the rule INHERITANCE in Figure 9 is unique to ORLog. This rule accounts for the structural and behavioral inheritance in our language. The key idea is that to prove a ground p-goal (a p-atom) of the form $o[m(a_1, \ldots, a_k) \mapsto a]$ via inheritance, we must find a unit i-clause $o[p@m^k_{\mapsto}]$ in $\mathbf{P}_c$ and a p-clause cl such that cl defines $m^k_{\mapsto}$ at p. Then we apply a clause-wise term replacement $\{p/\!/o\}$ and prove the body of the clause. Note that, the term replacement $\{p/\!/o\}$ captures the idea of *context switching* which is central to the realization of code reuse in ORLog. This is the basic idea. However, it is cumbersome and inefficient to force inheritance to the ground level. Consequently, our INHERITANCE inference rule in Figure 9 incorporates this idea on general (non-ground) p-clauses, by using the context switch judiciously.

$$\text{(INHERITANCE)} \quad \frac{\mathbf{P}_c \vdash_\pi \mathcal{G}[\phi\psi\varrho\theta]}{\mathbf{P}_c \vdash_{\phi\psi\theta\pi} o[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}]} \quad \left(\begin{array}{l} o'[\boldsymbol{p}'@m^k_{\mapsto}] \in \Upsilon_c, \\ \boldsymbol{p}[m(\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_k) \mapsto \boldsymbol{a}'] \leftarrow \mathcal{G} \in \Pi_c, \\ \phi = mgu(\boldsymbol{o}, \boldsymbol{o}'), \quad \psi = mgu(\boldsymbol{p}'[\phi], \boldsymbol{p}), \\ \varrho = \{\boldsymbol{p}\psi /\!/ \boldsymbol{o}\phi\psi\}, \\ \theta = mgu(< \boldsymbol{o}, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_k, \boldsymbol{a} > [\phi\psi], \\ \quad < \boldsymbol{p}, \boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_k, \boldsymbol{a}' > [\psi\varrho]) \end{array} \right)$$

Figure 9: Inheritance rule of ORLog.

Recall that $\varrho = \{p\psi//o\phi\psi\}$ is a term replacement implementing *context switching*, not a substitution. The way this inference rule works is as follows. In order to prove a p-atom, determine by looking in the closure $\mathbf{P}_c$ where the descriptor of the p-atom is supposed to inherit the property from. Once this is established, locate any clause defining this property at the source of the inheritance, and prove the body of this clause after performing the appropriate context switch (ϱ). All this is accomplished modulo unification since we are dealing with non-ground clauses. Below is an example of a proof using our proof theory.

Example 3.4 Revisit Example 3.2. Let us add the is-a definition $k : p$ as clause (15). $k : p$ then belongs in the component Λ_1 in program $\mathbf{P}_1$ of Example 3.2. Let $\mathbf{P}_{1_c}$ be the closure of $\mathbf{P}_1$. Now consider proving the goal $k[m \rightarrow W]$ from the program $\mathbf{P}_{1_c}$ using the proof rules above. A proof tree corresponding to $\mathbf{P}_{1_c} \vdash k[m \rightarrow W]$ is given below where ϱ is the context switch as shown. The proof returns the answer substitution $\{W/2\}$.

$$\dfrac{\dfrac{\dfrac{}{\mathbf{P}_{1_c} \vdash_\epsilon \Box}\ (\text{EMPTY})}{\mathbf{P}_{1_c} \vdash_{\{Q/k\}} k[v \rightarrow g]}\ \begin{bmatrix}(\text{DEDUC})\ (14),\\ \theta = \{Q/k\}\end{bmatrix} \quad \dfrac{\dfrac{}{\mathbf{P}_{1_c} \vdash_\epsilon \Box}\ (\text{EMPTY})}{\mathbf{P}_{1_c} \vdash_{\{X/2\}} k[s \rightarrow X][\theta]}\ \begin{bmatrix}(\text{INHERIT})\ (9),\\ \varrho_2 = \{p//k\}\end{bmatrix}}{\dfrac{\mathbf{P}_{1_c} \vdash_{\{Q/k\}\{X/2\}} k[s \rightarrow X], k[v \rightarrow g]}{\mathbf{P}_{1_c} \vdash_{\{W/X\}\{Q/k\}\{X/2\}} k[m \rightarrow W]}\ \begin{bmatrix}(\text{INHERITANCE})\ (8),\\ \varrho_1 = \{o//k\}\end{bmatrix}}\ (\text{AND})$$

3.7 Fixpoint Semantics

We now define a constructive way of defining the least model for a program $\mathbf{P}$. Notice that in any ORLog program $\mathbf{P} = \langle \Lambda, \Upsilon, \Pi \rangle$, there is an implicit stratification among its components. Namely, for a program $\mathbf{P} = \langle \Lambda, \Upsilon, \Pi \rangle$, the dependencies $\Upsilon \leftarrow \Lambda$, and $\Pi \leftarrow \Upsilon$ hold, as stated in Definition 3.1. Note that to compute inheritability of property clauses, as required by the algorithm for ∇, we need to have a complete knowledge of the object hierarchy and the locality of clauses defined in program $\mathbf{P}$.

These observations and the depends on relation "$\leftarrow$" suggest a way of constructing a model for any program $\mathbf{P}$ in general. We can compute a "local model" $\mathbf{M}_\Lambda$ for the component Λ, and then compute another local model for $\mathbf{M}_\Upsilon$ for the component Υ

from the "local program" $\Upsilon \cup \mathbf{M}_A$. This approach will give us the required knowledge for computing inheritability of p-clauses. The closure $\mathbf{P}_c$ of a program $\mathbf{P}$ embodies the knowledge required for this purpose. Because of this and since $\mathbf{P}_c$ is equivalent to $\mathbf{P}$, we base our fixpoint theory on the closed program $\mathbf{P}_c$.

We have already established that every ORLog program $\mathbf{P}$ has a unique intended model $\mathbf{M_P}$ which is obtained as the intersection of all proper models of $\mathbf{P}$. Recall that $\mathbf{M_P}$ is always canonical (see Theorem 3.1 and Theorem 3.2). We will show in this section that it is possible to obtain the intended model $\mathbf{M_P}$ of a program $\mathbf{P}$ (which by Theorem 3.2 is the same as the intended model $\mathbf{M}_{\mathbf{P}_c}$ of the closure $\mathbf{P}_c$ of $\mathbf{P}$), by means of a bottom-up least fixpoint computation, based on the immediate consequence operator $\mathbf{T}_{\mathbf{P}_c}$, defined below.

Definition 3.13 Let $\mathbf{P}_c$ be the closure of a definite ORLog program and let $\widehat{\mathbf{P}_c} = \langle \widehat{\Delta_c}, \widehat{\Upsilon_c}, \widehat{\Pi_c} \rangle$ be its Herbrand instantiation defined as usual. Let I be an Herbrand structure for $\mathbf{P}_c$. We define $\mathbf{T}_{\mathbf{P}_c}$ to be the immediate consequence operator that transforms Herbrand structures to Herbrand structures, i.e., $\mathbf{T}_{\mathbf{P}_c} : \mathcal{P}(\mathcal{H}) \longmapsto \mathcal{P}(\mathcal{H})$, such that

$$\begin{aligned}\mathbf{T}_{\mathbf{P}_c}(I) \;=\; & \{o[m(a_1,\ldots,a_k) \mapsto a][o/\!/p] \mid o[m(a_1,\ldots,a_k) \mapsto a] \leftarrow \mathcal{B} \in \widehat{\Pi}_c, \\ & p[o@m^k_{\mapsto}] \in I, I \models \mathcal{B}[o/\!/p]\} \cup \{\mathcal{A} \mid \mathcal{A} \leftarrow \mathcal{B} \in \widehat{\Pi}_c, I \models \mathcal{B}\}\end{aligned}$$

Furthermore, since ORLog captures (multiple) inheritance with overriding, not surprisingly, the operator $\mathbf{T}_{\mathbf{P}_c}$ is *not* monotone in general, as suggested by the following example.

Example 3.5 Let $\mathbf{P}_2 = \langle \Delta_2, \Upsilon_2, \Pi_2 \rangle$ be a program such that

$$\Delta_2 := \left| \begin{array}{ll} (1) & p :: p. \\ (2) & o :: o. \\ (3) & q :: q. \end{array} \right. \qquad \Upsilon_2 := \emptyset \qquad \Pi_2 := \left| \begin{array}{ll} (4) & p[m \rightarrow a]. \end{array} \right.$$

Now, consider two interpretations I_1 and I_2 such that

$$\begin{aligned} I_1 &= \{p :: p, o :: o, q :: q, o : p\} \\ I_2 &= I_1 \sqcup \{o : q\} \end{aligned}$$

While $o[m \rightarrow a] \in \mathbf{T}_{\mathbf{P}_c}(I_1)$, $o[m \rightarrow a] \notin \mathbf{T}_{\mathbf{P}_c}(I_2)$. So, in general, $\mathbf{T}_{\mathbf{P}_c}$ is not monotone. □

Thus a *fundamental challenge* is, how can we build a "fixpoint semantics" using an operator that is not monotone. The following lemma is important in this respect.

Lemma 3.1 Suppose $I_1 \sqsubseteq I_2$ and furthermore, (i) $(I_1)_{\mathcal{A}} = (I_2)_{\mathcal{A}}$, (ii) $(I_1)_{\Upsilon} = (I_2)_{\Upsilon}$, and (iii) both I_1 and I_2 satisfy all the clauses in $(\mathbf{P}_c)_{\mathcal{A}} \cup (\mathbf{P}_c)_{\Upsilon}$. Then $\mathbf{T}_{\mathbf{P}_c}(I_1) \sqsubseteq \mathbf{T}_{\mathbf{P}_c}(I_2)$.

Proof: The proof is almost identical to the classical case. The only observation required are: (1) since I_1 and I_2 agree on their $\mathcal{A}$ and Υ-components and since they satisfy the $\mathcal{A}$ and Υ-components of $\mathbf{P}_c$, (a) the operator ∇ will behave identically with respect to I_1 and I_2, and (b) all atoms in $\mathbf{T}_{\mathbf{P}_c}(I_1) - I_1$ (as also those in $\mathbf{T}_{\mathbf{P}_c}(I_2) - I_2$) are p-, pred-, and r-atoms. (2) Whenever $I_1 \models \mathcal{F}[o/\!/p]$, we have $I_2 \models \mathcal{F}[o/\!/p]$ for any formula $\mathcal{F}$, since $I_1 \sqsubseteq I_2$. □

Remarks: Lemma 3.1 simply says that on the class of interpretations which satisfy the is-a hierarchy and inheritability requirements of $\mathbf{P}_c$ and are in agreement in this regard, $\mathbf{T}_{\mathbf{P}_c}$ is indeed a monotone operator. This result is significant since $\mathbf{T}_{\mathbf{P}_c} \uparrow^1 =_{def} \mathbf{T}_{\mathbf{P}_c}(\mathbf{T}_{\mathbf{P}_c} \uparrow^0) = \mathbf{T}_{\mathbf{P}_c}(\emptyset)$, satisfies all the $\mathcal{A}$ and Υ-clauses in $\mathbf{P}_c$ and furthermore $\forall n > 1 : (\mathbf{T}_{\mathbf{P}_c} \uparrow^n)_{\mathcal{A}} = (\mathbf{T}_{\mathbf{P}_c} \uparrow^1)_{\mathcal{A}}$ and $(\mathbf{T}_{\mathbf{P}_c} \uparrow^n)_{\Upsilon} = (\mathbf{T}_{\mathbf{P}_c} \uparrow^1)_{\Upsilon}$. The latter simply follows from the definition of $\mathbf{P}_c$ and of $\mathbf{T}_{\mathbf{P}_c}$. This suggests the iterative applications of $\mathbf{T}_{\mathbf{P}_c}$ on $\mathbf{T}_{\mathbf{P}_c} \uparrow^1$ will indeed produce a monotonically increasing sequence of interpretations suggesting some hope of reaching a fixpoint.

The monotonicity of $\mathbf{T}_{\mathbf{P}_c}$ on the above class of interpretations guarantees the existence of a fixpoint. Now we have the following theorems that establish the equivalence of the model theoretic and the fixpoint theoretic semantics.

Proposition 3.2 Let $I_1, I_2, \ldots$ be an infinite directed sequence of interpretations such that $I_1 \sqsubseteq I_2 \sqsubseteq \ldots$. Then if $\sqcup_{i=1}^{\infty} I_i \models \mathcal{G}$, then $\exists k\ I_k \models \mathcal{G}$.

Proof: We proceed by inducing on the structure of the goal $\mathcal{G}$.

Basis: If $\mathcal{G}$ is atomic, then $\sqcup_{i=1}^{\infty} I_i \models \mathcal{G}$ if and only if $\mathcal{G} \in \sqcup_{i=1}^{\infty} I_i$. Therefore, there exists a k such that $\mathcal{G} \in I_k$ and then $I_k \models \mathcal{G}$.

Inductive step: Assume that the claim is true for any ground goal $\mathcal{G}$. Now consider a goal $\mathcal{G}' = \mathcal{G}_1, \mathcal{G}_2$. By inductive hypothesis there exists k_1 and k_2 such that $I_{k_1} \models \mathcal{G}_1$ and $I_{k_2} \models \mathcal{G}_2$. Now by choosing $k = max(k_1, k_2)$, it follows that $I_k \models \mathcal{G}_1$ and $I_k \models \mathcal{G}_2$, and therefore $I_k \models \mathcal{G}_1, \mathcal{G}2$. □

Theorem 3.4 Let $\mathbf{P}$ be a program, $\mathbf{P}_c$ be its closure, and I be a structure such that $(\mathbf{P}_c)_\Lambda \subseteq (I)_\Lambda$ and $(\mathbf{P}_c)_\Upsilon \subseteq (I)_\Upsilon$. Then I is a proper model of $\mathbf{P}$ iff $\mathbf{T}_{\mathbf{P}_c}(I) \sqsubseteq I$, and I is proper as a structure.

Proof: We show implications in both directions.

$\Rightarrow$: Suppose I is a proper model of $\mathbf{P}$. Then by Theorem 3.3, I is a proper model of $\mathbf{P}_c$. Since I, being proper, satisfies condition (7) of definition 3.2, it immediately implies that $\mathbf{T}_{\mathbf{P}_c}(I) \sqsubseteq I$.

$\Leftarrow$: Suppose $\mathbf{T}_{\mathbf{P}_c}(I) \sqsubseteq I$ and I is a proper structure. Since $(\mathbf{P}_c)_\Lambda \subseteq (I)_\Lambda$ and $(\mathbf{P}_c)_\Upsilon \subseteq (I)_\Upsilon$, I satisfies the Λ and Υ-clauses in $\mathbf{P}_c$. That it also satisfies the p-, r- and pred-clauses follows from the definition of $\mathbf{T}_{\mathbf{P}_c}$ and of satisfaction. Notice that since $(\mathbf{P}_c)_\Lambda \subseteq (I)_\Lambda$ and $(\mathbf{P}_c)_\Upsilon \subseteq (I)_\Upsilon$, all the atoms in $\mathbf{T}_{\mathbf{P}_c}(I)$ are p-, r- and pred-atoms. An inspection of the conditions (1) through (7) in definition 3.8 of proper structures immediately reveals that $\mathbf{T}_{\mathbf{P}_c}(I)$ is proper as long as I is. □

The bottom-up fixpoint iteration of $\mathbf{T}_{\mathbf{P}_c}$ is defined as follows:

$$\begin{aligned}
\mathbf{T}_{\mathbf{P}_c}\uparrow^0 &= I_\perp \\
\mathbf{T}_{\mathbf{P}_c}\uparrow^{n+1} &= \mathbf{T}_{\mathbf{P}_c}(\mathbf{T}_{\mathbf{P}_c}\uparrow^n) \\
\mathbf{T}_{\mathbf{P}_c}\uparrow^\omega &= \sqcup_{n<\omega}\mathbf{T}_{\mathbf{P}_c}\uparrow^n .
\end{aligned}$$

Note that owing to the monotonicity of $\mathbf{T}_{\mathbf{P}_c}$ it has a least fixpoint $lfp(\mathbf{T}_{\mathbf{P}_c})$ and since ORLog is function-free, we trivially have $lfp(\mathbf{T}_{\mathbf{P}_c}) = \mathbf{T}_{\mathbf{P}_c}\uparrow^\omega$. One of our main results is the following theorem, proved analogously to the classical case. The only subtlety is handling clause inheritance via context switch.

Theorem 3.5 Let $\mathbf{P}$ be a program and $\mathbf{P}_c$ be its closure. Then,

1. $\mathbf{M}_\mathbf{P} = \mathbf{M}_{\mathbf{P}_c} = lfp(\mathbf{T}_{\mathbf{P}_c})$, where $\mathbf{M}_\mathbf{P} = \mathbf{M}_{\mathbf{P}_c}$ is the least proper model of $\mathbf{P}$, and

2. $lfp(\mathbf{T}_{\mathbf{P}_c})$ can be computed in a finite number of bottom-up iterations.

Proof:

<u>(1)</u>.

$$\begin{aligned}
\mathbf{M}_{\mathbf{P}_c} &= \sqcap\{I \mid I \text{ is a proper Herbrand model of } \mathbf{P}_c\}, \\
&\quad \text{by Theorem 3.1}
\end{aligned}$$

Note that whenever I is a (proper) model of $\mathbf{P}$ (and hence of $\mathbf{P}_c$), $(\mathbf{P}_c)_A \subseteq (I)_A$ and $(\mathbf{P}_c)_T \subseteq (I)_T$. From this, we can see that

$$\begin{aligned}
\mathbf{M}_{\mathbf{P}_c} &= \sqcap\{\mathbf{M} \mid \mathbf{M} \text{ is a proper model of } \mathbf{P}_c\} \\
&= \sqcap\{\mathbf{M} \mid \mathbf{T}_{\mathbf{P}_c}(\mathbf{M}) \sqsubseteq \mathbf{M},\ \mathbf{M} \text{ is proper and } (\mathbf{P}_c)_A \subseteq (I)_A,\ (\mathbf{P}_c)_T \subseteq (I)_T\}, \\
&\quad \text{by proposition 3.4} \\
&= lfp(\mathbf{T}_{\mathbf{P}_c}), \text{ by the monotonicity of } \mathbf{T}_{\mathbf{P}_c} \text{ on the class of } \mathbf{P}_c \text{ satisfying the} \\
&\quad \text{conditions in Lemma 3.1, and Knaster-Tarski fixpoint theorem,} \\
&\quad \text{proving (1)}
\end{aligned}$$

<u>(2)</u>.
Since ORLog programs are function free, the least fixpoint above can be computed in a finite number of steps, just as for Datalog, which proves (2). □

Observation 3.1 Let $\mathbf{P}$ be an i-consistent program and $\mathbf{P}_c$ be its closure. Then $\mathbf{M}_{\mathbf{P}} = \mathbf{M}_{\mathbf{P}_c} = lfp(\mathbf{T}_{\mathbf{P}_c}) = \mathbf{T}_{\mathbf{P}_c} \uparrow^{\omega}$ is the intended model of $\mathbf{P}$. □

Theorem 3.5 establishes the equivalence between the declarative semantics based on intended models and the fixpoint semantics based on the least fixpoint of the operator $\mathbf{T}_{\mathbf{P}_c}$. It remains to establish their equivalence to the proof-theoretic semantics given in Section 3.6. As in the classical case, we accomplish this by relating the *stage* of a ground atom $\mathcal{A}$ – the smallest number of iteration k such that $\mathcal{A} \in \mathbf{T}_{\mathbf{P}_c} \uparrow^k$ – to the height of a proof tree for an atom more general than $\mathcal{A}$.

Theorem 3.6 (Soundness) Let $\mathbf{P}_c$ be a closed program, $\widehat{\mathbf{P}_c}$ be the Herbrand instantiation of $\mathbf{P}_c$, $\mathcal{G}$ be an atomic p-, r- or pred-goal, and $\widehat{\mathcal{G}}$ be all the ground instances of $\mathcal{G}$. If $\mathbf{P}_c \vdash_\alpha \mathcal{G}$ is provable then $\exists k$ such that $\widehat{\mathcal{G}\alpha} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^k$.
Proof: By induction on the height k of a proof tree. There are two cases, (i) p-goals and (ii) non p-goals (pred and r-goals). Note inheritance only applies to p-goals.
Basis: Suppose the proof tree has height 1. Then there are two possible cases – (i) either $\mathcal{G}$ is a p-clause local to some object, a pred-clause or an r-clause, or it is an inherited p-clause in some object from another object where it is local.
Case 1: There exists a unit clause $\mathcal{A} \in \mathbf{P}_c$ such that $mgu(\mathcal{G}, \mathcal{A}) = \theta$.
Case 2: There exist unit clauses $\mathcal{A}, \mathcal{B} \in \mathbf{P}_c$ such that $\mathcal{A}$ is a unit p-clause of the form $p[m(a'_1, \ldots, a'_k) \mapsto a']$ and $\mathcal{B}$ is a unit i-clause of the form $o'[p' @ m^k_{\mapsto}]$, and $\mathcal{G}$ is of

the form $o[m(a_1,\ldots,a_k) \mapsto a]$. Then the inheritance rule must be applied. Hence it must be the case that $\phi = mgu(o,o')$, $\psi = mgu(p'[\phi],p)$, $\varrho = [p\psi // o\phi\psi]$ and $\theta = mgu(< o,a_1,\ldots,a_k,a > [\phi\psi], < p,a'_1,\ldots,a'_k,a' > [\psi\varrho])$.

Let $\alpha = \theta$ for the first case and $\alpha = \phi\psi\theta$ for the second case. Since $\widehat{\mathcal{G}\alpha} \subseteq \widehat{\mathbf{P}_c}$ in both the cases, it follows from the definition of $\mathbf{T}_{\mathbf{P}_c}$ that $\widehat{\mathcal{G}\alpha} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^2$.

Inductive step: Assume that the claim is true for any proof of height $\leq k-1$ of a goal of the form $\mathbf{P}_c \vdash_\pi \mathcal{B}\beta$. Then we have again two cases.

Case (i): There must exist a clause of the form $\mathcal{A} \leftarrow \mathcal{B} \in \mathbf{P}_c$ such that $\theta = mgu(\mathcal{G},\mathcal{A})$, and $\beta = \theta$ and we have a proof of height k for $\mathbf{P}_c \vdash_\alpha \mathcal{G}$ where $\alpha = \beta\pi$ and the root node labelled DEDUCTION.

Case (ii): There must exist a p-clause $\mathcal{A} \longleftarrow \mathcal{B} \in \mathbf{P}_c$ and a unit i-clause $C \in \mathbf{P}_c$ such that $\mathcal{A}$ is of the form $p[m(a'_1,\ldots,a'_k) \mapsto a']$ and C is of the form $o'[p'@m^k_{\llcorner}]$, and $\mathcal{G}$ is of the form $o[m(a_1,\ldots,a_k) \mapsto a]$. We can now construct a proof tree of height k for $\mathbf{P}_c \vdash_\alpha \mathcal{G}$ such that the root node is labelled INHERITANCE, and $\phi = mgu(o,o')$, $\psi = mgu(p'[\phi],p)$, $\varrho = \{p\psi // o\phi\psi\}$ and $\theta = mgu(< o,a_1,\ldots,a_k,a > [\phi\psi], < p,a'_1,\ldots,a'_k,a' > [\psi\varrho])$. Also $\alpha = \phi\psi\theta\pi$ and $\beta = \phi\psi\varrho\theta$.

By inductive hypothesis, $\widehat{\mathcal{B}\beta\pi} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^{k-1}$ holds. Since $\widehat{\mathcal{G}\alpha} = \widehat{\mathcal{A}\beta\pi}$ in both the cases, it follows from lemma 3.1 that $\widehat{\mathcal{A}\beta\pi} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^k$ because either (i) $\widehat{\mathcal{B}\beta\pi} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^{k-1}$, or (ii) $\{\widehat{\mathcal{B}\beta\pi, o'[p'@m^k_{\llcorner}]\psi}\} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^{k-1}$, $o'[p'@m^k_{\llcorner}]$ being unit clauses and $\widehat{o'[p'@m^k_{\llcorner}]\psi} \subseteq \widehat{o'[p'@m^k_{\llcorner}]} \subseteq \widehat{\mathbf{P}_c}$. Hence $\widehat{\mathcal{G}\alpha} \sqsubseteq \mathbf{T}_{\mathbf{P}_c} \uparrow^k$. □

Theorem 3.7 (Completeness) Let $\mathbf{P}_c$ be a closed program and $\mathcal{G}$ be a ground p-, r- or pred-atom. For any k, if $\mathcal{G} \in \mathbf{T}_{\mathbf{P}_c} \uparrow^k$ then there is an atom $\mathcal{A}$ and a substitution α, such that there exists a proof tree of height k for $\mathbf{P}_c \vdash_\alpha \mathcal{A}$, and $\mathcal{G} \in \widehat{\mathcal{A}\alpha}$.

Proof: Again we proceed by induction on k.

Basis: Suppose $\mathcal{G} \in \mathbf{T}_{\mathbf{P}_c} \uparrow^1$. Then there must exist a unit clause $\mathcal{A} \in \mathbf{P}_c$ such that either (i) $\theta = mgu(\mathcal{G},\mathcal{A})$ and $\alpha = \theta$, or (ii) there exist unit clauses $\mathcal{A},\mathcal{B} \in \mathbf{P}_c$ such that $\mathcal{A}$ is a unit p-clause of the form $p[m(a'_1,\ldots,a'_k) \mapsto a']$, $\mathcal{B}$ is a unit i-clause of the form $o'[p'@m^k_{\llcorner}]$, and $\mathcal{G}$ is of the form $o[m(a_1,\ldots,a_k) \mapsto a]$ such that $\phi = mgu(o,o')$, $\psi = mgu(p'[\phi],p)$, $\varrho = \{p\psi // o\phi\psi\}$ and $\theta = mgu(< o,a_1,\ldots,a_k,a > [\phi\psi], < p,a'_1,\ldots,a'_k,a' > [\psi\varrho])$. Then there is a proof for $\mathbf{P}_c \vdash_\alpha \mathcal{A}$ of height 1 which is labelled either (i) by DEDUCTION where $\alpha = \theta$, or (ii) by INHERITANCE where $\alpha = \phi\psi\theta$ respectively. It immediately implies that $\mathcal{G} = \mathcal{A}\alpha$.

Inductive step: Assume that the claim holds true for $k-1$. From the definition of

$\mathbf{T}_{\mathbf{P}_c}$. if $\mathcal{G} \in \mathbf{T}_{\mathbf{P}_c} \uparrow^k$, then there are two cases.

Case (i): There must exist a clause $\mathcal{A} \leftarrow \mathcal{B} \in \mathbf{P}_c$ such that $\theta = mgu(\mathcal{G}, \mathcal{A})$, $\alpha = \theta$, $\mathcal{G} = \mathcal{A}\alpha$, and $\mathbf{T}_{\mathbf{P}_c} \uparrow^k \models \widehat{\mathcal{B}\alpha}$.

Case (ii): There exists a p-clause $\mathcal{A} \longleftarrow \mathcal{B} \in \mathbf{P}_c$ and a unit i-clause $\mathcal{C} \in \mathbf{P}_c$ such that $\mathcal{A}$ is of the form $\boldsymbol{p}[m(\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_k) \mapsto \boldsymbol{a}']$, $\mathcal{C}$ is of the form $\boldsymbol{o}'[\boldsymbol{p}'@m^k_{\sqcup}]$, and $\mathcal{G}$ is of the form $o[m(a_1, \ldots, a_k) \mapsto a]$ such that $\phi = mgu(o, \boldsymbol{o}')$, $\psi = mgu(\boldsymbol{p}'[\phi], \boldsymbol{p})$, $\varrho = \{\boldsymbol{p}\psi /\!/ o\phi\psi\}$ and $\theta = mgu(< o, a_1, \ldots, a_k, a > [\phi\psi], < \boldsymbol{p}, \boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_k, \boldsymbol{a}' > [\psi\varrho])$.

Since $\mathcal{G} \in \widehat{\mathcal{A}}$ (because $\mathcal{G}$ unifies with $\mathcal{A}$) and $\mathcal{G} \in \mathbf{T}_{\mathbf{P}_c} \uparrow^k$, by monotonicity of $\mathbf{T}_{\mathbf{P}_c}$ it must be the case that $\mathbf{T}_{\mathbf{P}_c} \uparrow^{k-1} \models \mathcal{B}$. Then by inductive hypothesis, we must have a proof for $\mathbf{P}_c \vdash_\pi \mathcal{B}\beta$ of height $k-1$. From this, we can then construct a proof for $\mathbf{P}_c \vdash_\alpha \mathcal{A}$ of height k from this whose root node is labelled either (i) by DEDUCTION where $\alpha = \theta\pi$ and $\beta = \theta$, or (ii) by INHERITANCE where $\alpha = \phi\psi\theta\pi$ and $\beta = \phi\psi\varrho\theta$ respectively. It immediately implies that $\mathcal{G} \in \widehat{\mathcal{A}\alpha}$. □

As a corollary, we have the following equivalence between the intended model semantics and the proof theory.

Theorem 3.8 Let $\mathbf{P}$ be a program, $\mathbf{P}_c$ be its closure, $\mathbf{M}_\mathbf{P}$ be the intended model for $\mathbf{P}$, and $\mathcal{G}$ be a ground goal. Then, we have that $\mathbf{P}_c \vdash_\epsilon \mathcal{G}$ is provable iff $\mathbf{M}_\mathbf{P} \models \mathcal{G}$.
Proof:

$$\begin{aligned} \mathbf{P}_c \vdash_\epsilon \mathcal{G} \text{ is provable} &\iff \mathcal{G} \in \mathbf{T}_{\mathbf{P}_c} \uparrow^\omega \text{ by Theorems 3.6 and 3.7} \\ &\iff \mathbf{M}_{\mathbf{P}_c} \models \mathcal{G} \text{ from Theorem 3.5} \end{aligned}$$ □

Chapter 4

Inheritance Reduction as an Aid to Implementation of ORLog

Language designers are often faced with situations where compromises must be made between competing requirements that are polarized in some way. Commercial viability and practicality of the system also plays an important role in the decision process that shapes its functionalities and characteristics. In the case of deductive object-oriented databases. the introduction of new features such as inheritance, object identity, encapsulation, signature, methods, etc. has increased the expressiveness, modeling capability and functionality of the database systems while the paradigm is now faced with seemingly unsurmountable complexity of the underlying semantics. The impact of this dilemma on the development of deductive object-oriented database systems has been undoubtably far reaching. This is evidenced by the multitude of opinions on an acceptable data model on which a declarative language can be built with semantic sufficiency.

As a consequence, there are several schools of thoughts with respect to application, theoretical rigor and implementation details of a declarative object-oriented language. This can be broadly classified into two groups in two orthogonal axes. (i) A predominant class of proposals attempt to capture object-oriented features in a well-known logical system such as Datalog or Prolog. In these approaches object-oriented features are captured by giving a relational interpretation to object-oriented concepts. These are the so called translation based approaches. (ii) The proponents of the other class of languages advocate a more direct semantics that does not require a translation.

These are the languages that have their own syntax and underlying semantics. Within each of these categories, they can be classified again into two classes. (a) Languages that are given a partial logical semantics and rely on a non-logical, meta-logical, or procedural semantics at varying degrees for some of their features. (b) The second class gives a complete logical interpretation to whatever features they embody in their model. Figure 8 shows some of the languages and their classification according to this scheme.

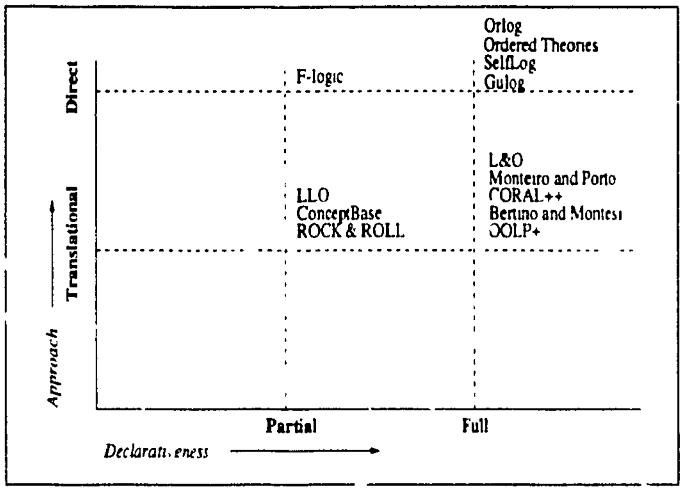

Figure 10: Classification of Languages According to their Semantics and Approach.

It is our thesis that a deductive language should be given a direct semantics which captures the actual logical meaning of most, if not all, of its features. In the translational approach, on the other hand, the insight is lost, and there is no control on how the translated program would behave. This results into a less than intuitive, and at times a complicated semantics that is hard to grasp. A direct semantics is "readily comprehensible" if the underlying model on which the language is based is so. This approach also preserves the declarativeness of the language and eliminates possible impedance mis-match, and has a greater appeal from a theoretical standpoint. Our contention is supported by a recent work by Kifer [46] where he argues that we can take whatever approach we would like to develop and implement a logic based language only after we fully understand the language from a logical standpoint and what it entails.

As we have demonstrated in the previous chapter and in the Figure 10 that ORLog has a direct and full first-order declarative semantics. It also has a sound and complete proof theory with respect to its model and fixpoint theory. In this chapter we, however, develop a technique to reduce inheritance to pure deduction. We also develop a first-order encoding algorithm to show that ORLog is first-order encodable. We prove that the encoding is sound and complete. This is not, in particular, indicative of a lack of semantics of the ORLog language itself in any way. This exercise, however, is crucial in several respects. First, we are about to make a design decision for the implementation of ORLog that requires us to investigate ways to implement object-oriented features in deductive database systems by purely deductive means. Secondly, it is essential that we show that ORLog has a first-order encoding to convince the critics that although ORLog has a higher order syntax that boosts its modeling capability, its semantics and expressibility are equivalent to first-order logic. Finally, the reduction technique would benefit a large population of systems that are proposed without a clear semantics of inheritance. Due to the non-monotonic nature of inheritance and associated notions of overriding and late binding, most of the translation based proposals do not address this issue or capture them procedurally in a not so intuitive way outside the logic. Some advanced proposals are based on negation semantics, and thus associate a large cost on query answering. It is our opinion that a technique similar to ours can be adapted in many systems that fall short of capturing inheritance logically.

4.1 Related Research and Motivation

In this section we briefly discuss few representative systems that rely on a translational semantics or capture features by non-logical means. The point we would like to stress here is that these systems, like several others, depend on a translational semantics and lack a logical interpretation of its features. Thereby they undermine their status as a purely declarative language.

The suit of research in [1, 23, 71] extend Datalog like languages to incorporate object-oriented features. These proposals are conservative in that they exploit what has been achieved in deductive systems and provide an object-oriented shell that eventually make use of the deductive layer underneath. These proposals support limited modeling facilities and provide a rather narrow vision of object-orientation.

Coral++ [71] is essentially an object-oriented interface to the deductive system

Coral [66] in which C++ objects can be accessed from Coral using linguistic extensions provided by Coral++. Clearly, this system introduces non-uniformity in the users view of the applications leading to the so called impedance mismatch problem. IQL+ [1] barely addresses the issue of inheritance. In particular, it does not cater for multiple inheritance and behavioral inheritance.

In contrast to the above proposals, OOLP+ [25], OIL [76], Logical-objects [13], and LLO [56], for example, rely on mappings to deductive systems to give a semantics to their language and for defining an implementation strategy as well. OOLP+ provides a mapping to Prolog but lacks a clear semantics for inheritance. Overriding seems to be a limiting factor for this language. Since it relies on Prolog, it is essentially incapable of query optimization for which deductive systems are well-known. OIL and Logical-objects are mapped to $\mathcal{LDL}$. LLO is inspired by F-logic and HiLog, and capitalizes on its higher-order syntax. LLO is built around the data model of O_2 and is also $\mathcal{LDL}$ translatable. Though inheritance is made part of this language, it is not at all clear how inheritance is handled in this framework. All these systems though appear to take a practical route, they suffer from complementary or similar drawbacks as in the extended Datalog approach.

Another interesting approach can be witnessed in proposals like ROCK & ROLL [10] and ConceptBase [45] where language integration is the key. ROCK & ROLL is more akin to translation based approach where it provides means for procedural (ROCK) as well as declarative (ROLL) specification of objects, methods, etc. The language is based upon a data model called the OM [10] where the two sub languages ROCK and ROLL, interact. ROCK allows schema declaration and data manipulation facilities and is implemented in $\hat{E}$ (extension of C++). ROLL is a function free typed Horn clause language. This language is strongly typed and allows mechanisms for static type checking. Although it gains from the integration of procedural and deductive languages through separation of logic and control, ROLL's dependence on ROCK is a real bottleneck. ConceptBase is based on the language Telos [45] and also features a co-existence of deductive and object-oriented layers in a single system. Very recently, the Peplom^d [26] has also taken this language integration approach to implementation.

4.2 Reducing Inheritance to Deductions

Although all the above proposals have advantages, none is absolutely better than the other. But the most striking distinction of ORLog with these languages is in its simplicity and naturalness. In the following sections, we discuss an encoding scheme of ORLog into first-order logic, and present a mechanism to reduce inheritance to deductions. We also prove that the encoding, and subsequently claim that the reduction, preserves the meaning of the original ORLog programs. This entails that introduction of non-logical constructs or procedurality is not essential for ORLog and a model for a program can still be computed by fixpoint computation of the reduced programs. To distinguish between ORLog and the encoded and reduced ORLog, we call them respectively ORLog and F-ORLog (for first-order ORLog). Notice that this reduction is possible only because ORLog has a clearly understood logical semantics of all its features.

We organize the rest of this chapter as follows. We first present an encoding scheme of ORLog in first-order logic in section 4.2.1. Then in section 4.3 we present the inheritance reduction technique relying on the idea of inheritance completion, or *i-completion* and show its equivalence by relating the intended model of ORLog programs to perfect models of reduced programs. We also give an algorithm for translating ORLog programs to Coral deductive database language. We defer our discussion on the design choices and decisions until after this chapter.

4.2.1 Encoding ORLog in Predicate Logic

Although ORLog enjoys a richer syntax than its predicate logic counterpart, it turns out that ORLog is no more expressive than predicate logic. In fact they are equivalent in terms of their expressive power. However, it is the case that ORLog syntax allows us to model and specify our domain of discourse in a very convenient way. Note that, in general higher-order syntax [24, 47] has been found to be useful to model complex objects and to reason about them.

In this section we develop an encoding algorithm to show that ORLog is first-order encodable, and the semantics of the encoded program is equivalent to the original program. This observation immediately indicates that it is possible to use a first-order language to implement ORLog as a viable alternative to a more direct implementation.

First we present the encoding scheme as follows.

Given an ORLog language $\mathcal{L} = \langle \mathcal{C}, \mathcal{V}, \mathcal{M}, \mathcal{T}, \mathcal{P} \rangle$, we define $\mathcal{L}_P^{encode} = \langle \mathcal{C}, \mathcal{V}, \mathcal{M}, \mathcal{T}, P, F \rangle$ to be a language in predicate logic such that P is a set of predicate symbols that includes $\{\mathsf{object}_1, \mathsf{local}_2, \mathsf{withdraw}_3, \mathsf{parent}_2, \mathsf{isa}_2, \mathsf{uses}_3, \mathsf{property}_4, \mathsf{participate}_2, \mathsf{relation}_2\}$ and F is a distinct set of function symbols apply_{n+1} one for each $n \geq 1$. Given an ORLog formula ϕ, its encoding into predicate logic, ϕ^*, is given as the following recursive transformation rules. In the following rules encode_a is a transformation that encodes ORLog atoms, and encode_t encodes ORLog terms appearing in atomic ORLog formulas.

- $\mathsf{encode}_t(X) = X$, for each variable $X \in \mathcal{V}$;
- $\mathsf{encode}_t(c) = c$, for each constants $c \in \mathcal{C}$;
- $\mathsf{encode}_t(s) = s$ for each symbol $s \in \mathcal{M} \cup \mathcal{T} \cup \mathcal{P} \cup \boldsymbol{p}$;
- $\mathsf{encode}_a(\mathcal{A} \vee \mathcal{B}) = \mathsf{encode}_a(\mathcal{A}) \vee \mathsf{encode}_a(\mathcal{B})$;
- $\mathsf{encode}_a(\mathcal{A} \wedge \mathcal{B}) = \mathsf{encode}_a(\mathcal{A}) \wedge \mathsf{encode}_a(\mathcal{B})$;
- $\mathsf{encode}_a(\neg\mathcal{A}) = \neg\mathsf{encode}_a(\mathcal{A})$;
- $\mathsf{encode}_a((QX)\mathcal{A}) = (QX)\mathsf{encode}_a(\mathcal{A})$, where Q is either $\exists$ or $\forall$.
- Encoding of atomic formulas are given case by case as follows:
  - $\mathsf{encode}_a(\boldsymbol{p}[\,]) = \mathsf{object}(\mathsf{encode}_t(\boldsymbol{p}))$;
  - $\mathsf{encode}_a(\boldsymbol{p} : \boldsymbol{q}) = \mathsf{parent}(\mathsf{encode}_t(\boldsymbol{p}). \mathsf{encode}_t(\boldsymbol{q}))$;
  - $\mathsf{encode}_a(\boldsymbol{p} :: \boldsymbol{q}) = \mathsf{isa}(\mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}))$;
  - $\mathsf{encode}_a(\boldsymbol{p}[m^k_{\mapsto}]) = \mathsf{local}(\mathsf{encode}_t(\boldsymbol{p}), m. k, type)$[1];
  - $\mathsf{encode}_a(\boldsymbol{p}[m^k_{\mapsto} \propto\!\lhd \boldsymbol{q}]) = \mathsf{withdraw}(\mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}), m, k, type)$;
  - $\mathsf{encode}_a(\boldsymbol{p}[m^k_{\mapsto} \rhd\!\circ \boldsymbol{q}]) = \mathsf{withdraw}(\mathsf{encode}_t(\boldsymbol{q}), \mathsf{encode}_t(\boldsymbol{p}), m, k. type)$;
  - $\mathsf{encode}_a(\boldsymbol{p}[\boldsymbol{q}@m^k_{\mapsto}]) = \mathsf{uses}(\mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}), m, k, type)$;

[1]Here and in what follows, *type* is mapped to one of the constants in $\{func\_val, func\_sig, set\_val, set\_sig\}$ depending on $\mapsto$, where $\rightarrow$ mapsto $func\_val$, $\twoheadrightarrow$ mapsto set_val. $\Rightarrow$ mapsto $func\_sig$, and finally $\Rrightarrow$ mapsto set_sig.

- $\mathsf{encode}_a(r \diamond t_1, \ldots, t_n) = \mathsf{participate}(\mathsf{encode}_t(r), \mathsf{apply}_n(\mathsf{encode}_t(t_1), \ldots, \mathsf{encode}_t(t_n)))$;
- $\mathsf{encode}_a(r(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)) = \mathsf{relation}(\mathsf{encode}_t(r), \mathsf{apply}_n(\mathsf{encode}_t(\boldsymbol{a}_1), \ldots, \mathsf{encode}_t(\boldsymbol{a}_n)))$;
- $\mathsf{encode}_a(\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}]) = \mathsf{property}(\mathsf{encode}_t(\boldsymbol{p}), m, k, type, \mathsf{apply}_k(\mathsf{encode}_t(\boldsymbol{a}_1), \ldots, \mathsf{encode}_t(\boldsymbol{a}_k)), \mathsf{encode}_t(\boldsymbol{a}))$.

Given an ORLog (Herbrand) Structure $\mathbf{H} = \langle \mathbf{H}_\Lambda, \mathbf{H}_\Upsilon, \mathbf{H}_\Pi \rangle$ with its Herbrand universe $\mathcal{U}$ and Herbrand base $\mathcal{H}$, the corresponding predicate logic structure, $\mathsf{encode}(\mathbf{M}) =< U, I_P >$, is defined as follows:

- $I_P(s) = s$ for every logical symbol $s \in \mathcal{U}$.
- $< \mathsf{encode}_t(\boldsymbol{p}) > \in I_P(\mathsf{object}) \iff \mathbf{H} \models \boldsymbol{p}[]$.
- $< \mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}) > \in I_P(\mathsf{parent}) \iff \mathbf{H} \models \boldsymbol{p} : \boldsymbol{q}$.
- $< \mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}) > \in I_P(\mathsf{isa}) \iff \mathbf{H} \models \boldsymbol{p} :: \boldsymbol{q}$.
- $< \mathsf{encode}_t(\boldsymbol{p}), m, k, type > \in I_P(\mathsf{local}) \iff \mathbf{H} \models \boldsymbol{p}[m^k_{\llcorner\!\lrcorner}]$.
- $< \mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}), m, k, type > \in I_P(\mathsf{withdraw}) \iff \mathbf{H} \models \boldsymbol{p}[m^k_{\llcorner\!\lrcorner} \propto\!\lhd \boldsymbol{q}] \vee \boldsymbol{q}[m^k_{\llcorner\!\lrcorner} \rhd\!\!\propto \boldsymbol{p}]$.
- $< \mathsf{encode}_t(r), \mathsf{apply}_n(\mathsf{encode}_t(t_1), \ldots, \mathsf{encode}_t(t_n)) > \in I_P(\mathsf{participate}) \iff \mathbf{H} \models r \diamond t_1, \ldots, t_n$.
- $< \mathsf{encode}_t(r), \mathsf{apply}_n(\mathsf{encode}_t(\boldsymbol{a}_1), \ldots, \mathsf{encode}_t(\boldsymbol{a}_n)) > \in I_P(\mathsf{relation}) \iff \mathbf{H} \models r(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)$.
- $< \mathsf{encode}_t(\boldsymbol{p}), \mathsf{encode}_t(\boldsymbol{q}), m, k, type > \in I_P(\mathsf{uses}) \iff \mathbf{H} \models \boldsymbol{p}[\boldsymbol{q}@m^k_{\llcorner\!\lrcorner}]$.
- $< \mathsf{encode}_t(\boldsymbol{p}), m, k, type, \mathsf{apply}_k(\mathsf{encode}_t(\boldsymbol{a}_1), \ldots, \mathsf{encode}_t(\boldsymbol{a}_k)), \mathsf{encode}_t(\boldsymbol{a}) > \in I_P(\mathsf{property}) \iff \mathbf{H} \models \boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}])$.

We can now state the following encoding theorem (adapted from [24]) that establishes the equivalence of the encoded program and the original ORLog program.

Theorem 4.1 Let **P** be any ORLog program and ϕ be an ORLog formula. Let M be any Herbrand structure, $\mathsf{encode}(M)$ be the semantic structure corresponding to the language $\mathcal{L}_P^{encode}$, and ν be a variable assignment function. Then

$$M \models_\nu \phi \iff \mathsf{encode}(M) \models_\nu \mathsf{encode}_a(\phi)$$

Proof: By a case by case structural induction of atom $\mathcal{A}$, $\nu(\mathcal{A}) = \nu(\mathsf{encode}_a(\mathcal{A}))$. By definition, $\mathsf{encode}_a(\mathcal{A}) = pred(\mathcal{A})(\mathsf{encode}_t(t(\mathcal{A})))$, where $pred(\mathcal{A})$ is a function that maps to the corresponding predicate symbol of $\mathcal{A}$ in $\mathcal{L}_P^{encode}$, and $t(\mathcal{A})$ is the list of arguments for $pred(\mathcal{A})$ constructed from the atom $\mathcal{A}$ according to the encoding algorithm. Therefore

$$\begin{aligned}
M \models_\nu \mathcal{A} &\iff \nu(\mathsf{encode}_t(t(\mathcal{A}))) \in I_P(pred(\mathcal{A})) \\
&\iff \mathsf{encode}(M) \models_\nu pred(\mathcal{A})(\mathsf{encode}_t(t(\mathcal{A})) \\
&\iff \mathsf{encode}(M) \models_\nu \mathsf{encode}_a(\mathcal{A})
\end{aligned}$$

It is now easy to show that the equivalence holds for arbitrary formula ϕ by induction on the structure of the formula. □

4.3 Reduction by Completion

As we discussed earlier, mapping object-oriented languages to a deductive language is an effective means to give a logical semantics to the source language. Although, the idea is simple, achieving completeness and capturing the intended features are not. Of the languages we have discussed at the outset of this chapter, most of them do not address the issue of inheritance. This is partly because the source languages do not have a semantics of their own, hence it becomes difficult to capture it in the target language.

The encoded databases, however, have a relational interpretation where probably every object is viewed as a theory consisting of a collection of Horn clauses. Messages are viewed as sub-goals which succeed iff they are entailed by the theory associated with the receiving objects. However, in a deductive setting, capturing inheritance as it stands, is a real challenge, because deductive systems are not equipped with the notion of inheritance and hence must be simulated. The question we need to ask is that is it possible to model inheritance in the form of deduction since all first-order systems are only capable of deduction? To put it another way, since inheritance is almost always deterministic, why not first compute *inheritability of clauses* and then compute the model by restricting evaluation of the rules based on computed inheritability in a purely deductive way?

Fortunately, the answers to these questions are favorable in the case of ORLog. As we shall see in the sequel of this chapter that the translation scheme of ORLog relies on the key idea of reducing inheritance to deduction using completion. The reduction enables us to implement ORLog on top of an existing deductive database system, namely Coral. We shall present the idea of *i-completion* that is the center point of the reduction technique. But first, we take a closure of ORLog programs, called the *l-closure*, to account for clause locality right in the closed programs. This is different from the closure discussed in section 3.6.1. The l-closure is essential since our goal is to implement ORLog on a deductive system that lacks the capability of detecting locality of clauses in programs in addition to its inability to handle inheritance. Finally we discuss a translation algorithm of i-completed ORLog programs into CORAL, called the F-ORLog programs and show that the F-ORLog programs preserve the semantics of every original ORLog program.

4.3.1 L-closure of Programs

Let $\mathbf{P} = \langle \Lambda, \Upsilon, \Pi \rangle$ be a definite ORLog program. We first define an l-closure $\mathbf{P}^l = \langle \Lambda^l, \Upsilon^l, \Pi^l \rangle$ of $\mathbf{P}$ as follows. Recall that the locality of clauses is not usually supplied by the programmers. By taking the l-closure, we account for the locality of clauses in $\mathbf{P}$. The *l-closure* of a program $\mathbf{P}$ is the smallest set of clauses $\mathbf{P}^l$ satisfying $\mathbf{P} \subseteq \mathbf{P}^l$, and the conditions below.

- $\Lambda \subseteq \Lambda^l, \Upsilon = \Upsilon^l, \Pi = \Pi^l$.
- Whenever a p-clause $\boldsymbol{p}[m(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k) \mapsto \boldsymbol{a}] \leftarrow \boldsymbol{r}_1 \natural \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \natural \boldsymbol{q}_n, \mathcal{G} \in \Pi^l$ we have $\boldsymbol{p}[m^k_{\mapsto}] \leftarrow \boldsymbol{r}_1 \natural \boldsymbol{q}_1, \ldots, \boldsymbol{r}_n \natural \boldsymbol{q}_n \in \Lambda^l$.
- $\boldsymbol{p}[m^k_{\mapsto} \rhd\!\circ \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^l \Longrightarrow \boldsymbol{q}[m^k_{\mapsto} \circ\!\lhd \boldsymbol{p}] \leftarrow \mathcal{G} \in \Lambda^l$.
- $\boldsymbol{p}[m^k_{\mapsto} \circ\!\lhd \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^l \Longrightarrow \boldsymbol{q}[m^k_{\mapsto} \rhd\!\circ \boldsymbol{p}] \leftarrow \mathcal{G} \in \Lambda^l$.

- $\boldsymbol{p}[m^k_{\mapsto_s} \rhd\!\circ \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^l \Longrightarrow \boldsymbol{p}[m^k_{\mapsto_d} \rhd\!\circ \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^l$.
- $\boldsymbol{p}[m^k_{\mapsto_s} \circ\!\lhd \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^l \Longrightarrow \boldsymbol{p}[m^k_{\mapsto_d} \circ\!\lhd \boldsymbol{q}] \leftarrow \mathcal{G} \in \Lambda^l$.
- $P :: P \leftarrow P[\,] \in \Lambda^l$.
- $P :: Q \leftarrow P : R, R :: Q \in \Lambda^l$.
- $\boldsymbol{p}[m^k_{\mapsto}] \leftarrow \mathcal{G} \in \Lambda^l \Longrightarrow \boldsymbol{p}[\,] \in \Lambda^l$.
- $\boldsymbol{p}[\boldsymbol{q} @ m^k_{\mapsto}] \leftarrow \mathcal{G} \in \Lambda^l \Longrightarrow \boldsymbol{p}[\,] \in \Lambda^l$.

The l-closure above exposes the properties of a program hidden in the definitions of clause locality. It also captures the meaning of withdrawal in its operational sense. Furthermore, the l-closure now explicitly captures the idea of is-a transitivity and reflexivity of objects in any program **P**. It is now a matter of simple exercise to show that the l-closed programs and the their corresponding ORLog programs admit the same class of canonical models and hence are equivalent.

4.3.2 I-completion

We now present the notion of inheritance completion, called the *i-completion*, of a program to capture inheritance in the form of *constrained deduction*. The idea is to constrain property clause evaluation such that the clauses are only fired in an object when the object is legitimately allowed to use the clause. Intuitively, given a program $\mathbf{P}^*$, for every p-clause in Π^l we replace the descriptor of the clause with a variable, and restrict the instantiation of the variable in a controlled and deterministic way. The idea of inheritability is the key. Recall that for a given program $\mathbf{P}^*$ and a given p-clause $Cl \in \Pi^l$, its inheritability is known. That is the set of objects in $\mathbf{P}^*$ that can inherit Cl is known. Only thing we have to do is restrict the instantiation of the replaced variable to the set of allowed objects. The following definition formally captures the idea of i-completion.

Definition 4.1 (I-completion) Let $\mathbf{P}^* = \langle \Lambda^l, \Upsilon^l, \Pi^l \rangle$ be an l-closed program. Then its i-completion, denoted $\mathbf{P}_i = \langle \Lambda_i, \Upsilon_i, \Pi_i \rangle$, is a minimal program defined as follows.

- $\Pi_i = \emptyset$. $\Lambda_i = \Lambda^l$ and $\Upsilon_i = \Upsilon^l$.

- for every p-clause $Cl = \boldsymbol{p}[m(t_1,\ldots,t_k) \mapsto t] \leftarrow \mathcal{B} \in \Pi^l \Longrightarrow (\boldsymbol{p}[m(t_1,\ldots,t_k) \mapsto t]) \leftarrow \mathcal{B})[\boldsymbol{p}/\!/V], V[\boldsymbol{p}@m^k_{\mapsto}] \in \Pi_i$ where V is a distinguished variable not occurring in Cl, and $\{\boldsymbol{p}/\!/V\}$ is a term replacement. □

The intention of the i-completion is clear and intuitive. We replace every self-reference in the clause to a variable V and decide evaluation of the clause in an object V in terms of its inheritability. If V is an instance or subclass of the object $\boldsymbol{p}$ where the clause was originally defined, and V can legitimately inherit $m^k_{\mapsto}$ from $\boldsymbol{p}$ (i.e., $V[\boldsymbol{p}@m^k_{\mapsto}]$ is true), then the clause evaluation is allowed. As discussed before, this is consistent with respect to the idea of code reuse and our notion of inheritance.

The i-completed p-clauses may be viewed as new kind of is-a constrained p-clauses. This is because for every such clause, $V[\boldsymbol{p}@m^k_{\mapsto}]$ is true only if $V :: \boldsymbol{p}$ holds true. We can now rely solely upon deduction provided we are able to compute the set of i-atoms of the form $\boldsymbol{p}[\boldsymbol{q}@m^k_{\mapsto}]$ for $\mathbf{P}$. Again, we can show that an i-completed program admits an identical class of canonical models with respect to its original ORLog program, and hence are equivalent. Let us turn to our running example to see how i-completion works.

Example 4.1 Consider program $\mathbf{P}_1$ in Example 3.2. The i-completion of $\mathbf{P}_1$ is as follows where $\Upsilon_{1_i} = \emptyset$:

$$\Delta_{1_i} := \left|\begin{array}{rl}
(1) & o[].\quad p[].\quad q[]. \\
(2) & r[].\quad X[] \leftarrow X : o. \\
(3) & Q[].\quad r : p. \\
(4) & r : q.\quad q : o.\quad p : o. \\
(5) & r[t^0_{\mapsto} \bowtie\!\!\lhd p].\quad p[t^0_{\mapsto} \rhd\!\!\bowtie r]. \\
(6) & X :: X \leftarrow X[]. \\
(7) & X :: Y \leftarrow X : Z, Z :: Y. \\
(8) & P[u^0_{\mapsto}] \leftarrow P : o. \\
(9) & o[m^0_{\mapsto}].\quad o[s^0_{\mapsto}]. \\
(10) & q[t^0_{\mapsto}].\quad p[s^0_{\mapsto}]. \\
(11) & p[t^0_{\mapsto}].\quad Q[v^0_{\mapsto}].
\end{array}\right.
\qquad
\Pi_{1_i} := \left|\begin{array}{rl}
(12) & V[m \rightarrow X] \leftarrow V[s \rightarrow X], \\
 & V[v \rightarrow g], V[o@m^0_{\mapsto}]. \\
(13) & V[s \rightarrow 5] \leftarrow V[o@s^0_{\mapsto}]. \\
(14) & V[u \rightarrow d] \leftarrow V : o, \\
 & V[P@v^0_{\mapsto}]. \\
(15) & V[s \rightarrow 2] \leftarrow V[p@s^0_{\mapsto}]. \\
(16) & V[t \rightarrow c] \leftarrow V[p@t^0_{\mapsto}]. \\
(17) & V[t \rightarrow c] \leftarrow V[q@t^0_{\mapsto}]. \\
(18) & V[v \rightarrow g] \leftarrow V[Q@v^0_{\mapsto}].
\end{array}\right.$$

□

4.3.3 Computing Inheritability of Clauses using ∇

Recall that our goal is to encode ORLog programs into a first-order language and use a bottom-up evaluator to compute the encoded intended model of the program. Since first-order bottom-up evaluators are only capable of deductions, we need to provide a means to compute the set of encoded i-atoms so that the evaluator can use them to evaluate the i-completed p-clauses that are encoded. So far, we have only provided a means to compute the locality of p-clauses and to derive the partial order of the is-a hierarchy through l-closure. Finally, by taking the i-completion we accounted for a transformation of clause inheritance into constrained deductions.

```
1. uses(Object, Object, Name, Arity, Method_type) ←
        local(Object, Name, Arity, Method_type).

2. uses(Source, Object, Name, Arity, Method_type) ←
        possible(Source, Object, Name, Arity, Method_type),
        ¬local(Object, Name, Arity, Method_type),
        local(Source, Name, Arity, Method_type), Source≠Object.

3. possible(Source, Object, Name, Arity, Method_type) ←
        parent(Object, Some_object), Object≠Some_object,
        uses(Source, Some_object, Name, Arity, Method_type),
        ¬withdraw(Object, Some_object, Name, Arity, Method_type),
        ¬conflict(Source, Object, Some_object, Name, Arity, Method_type).

4. conflict(Source, Object, Some_object, Name, Arity, Method_type) ←
        parent(Object, Another_object), Object≠Another_object,
        uses(Another_source, Another_object, Name, Arity, Method_type),
        ¬withdraw(Object, Another_object, Name, Arity, Method_type),
        Some_object≠Another_object, Another_source≠Source.
```

Figure 11: Implementation of ∇ using a stratified predicate logic program $\mathbf{P}_\nabla$.

Now, to be able to compute inheritability as stipulated in the definition of the ∇ function (Definition 3.3), we present the following set of encoded rules $\mathbf{P}_\nabla$ in Figure 11. Note that the rules involve negation and are *locally stratifiable* [67] since the underlying is-a hierarchy is acyclic by assumption. Observe the dependency relation of predicates in $\mathbf{P}_\nabla$ and in encode(A_i) as depicted in Figure 12. Intuitively, it is easy to see that the *parent* relation supplies new tuples to the rules for *conflict* and *possible*.

The rules for *uses*, *possible* and *conflict* traverse these tuples to decide inheritability. Hence a possible violation of local stratification will occur in the Herbrand instantiation of $\mathbf{P}_\nabla$ if the object hierarchy graph is cyclic. But since $\mathbf{P}$, and hence $\mathbf{P}_i$, is assumed to be i-consistent, the hierarchy graph is acyclic. Hence $\mathbf{P}_r$ must be locally stratified for every consistent ORLog program. Also note that the encoded ORLog program is still definite. The idea is to define a *perfect* model for the *reduced program* $\mathbf{P}_r$, which is the union of the encoded program and the set of axioms in $\mathbf{P}_\nabla$, i.e., $\mathbf{P}_r$ = encode($\mathbf{P}_i$) $\cup$ $\mathbf{P}_\nabla$.

Although the i-completed programs are conventional first-order definite programs, the reduced programs are normal Horn programs. Notice that the reduced program has all the machineries needed to compute a perfect model for $\mathbf{P}_r$. Also note that the depends on relation "$\leftarrow$" for ORLog programs still hold in $\mathbf{P}_r$. Our definitions of local stratification and perfect models coincide with [64] and hence are identical. Note that the structure of $\mathbf{P}_r = \langle \Lambda_r, \Upsilon_r, \Pi_r \rangle$ partly suggests a stratification in a broad sense where $\Pi_r < \Upsilon_r < \Lambda_r$. The goal of the stratification is to decompose the ground extension of a program $\mathbf{P}_r$ into different strata $P^1, \ldots, P^n$ such that $[\mathbf{P}_r]$ can be obtained as the disjoint union of these strata. The encoded intended models of reduced programs can now be computed by computing their perfect models using the standard recipes available for normal programs [7, 6].

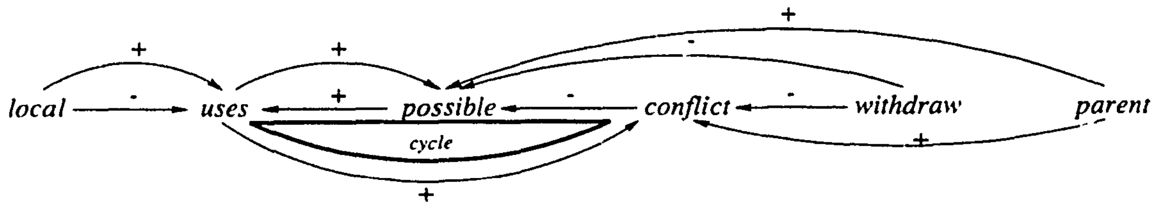

Figure 12: Dependency relation of predicates in $\mathbf{P}_\nabla$.

4.3.4 ORLog to Coral Translation Algorithm

We now present an algorithm that can be used to translate ORLog programs into F-ORLog programs that can be evaluated in Coral deductive database system using Coral query evaluator. Note that the actual implementation details are not an issue here, only the translation is.

The translation algorithm is presented in Figure 13. Its termination and correctness are indirectly captured in the discussion that precedes this section. We present a complete encoding of the i-completed program of Example 3.2 in Example 4.2.

```
Algorithm Translation;
Input: An ORLog program $P$;
Output: A reduced F-ORLog program $\mathbf{P}_r$;

begin
    1. Compute the l-closure $\mathbf{P}_p$ of $P$;
    2. Compute i-completion $\mathbf{P}_\imath$ of $\mathbf{P}_p$;
    3. Encode $\mathbf{P}_\imath$ into $\mathbf{P}_e$;
    4. Obtain $\mathbf{P}_r$ as $\mathbf{P}_e \cup \mathbf{P}_\nabla$;
end.
```

Figure 13: ORLog to Coral translation algorithm.

Example 4.2 Consider the i-completed program $\mathbf{P}_{1_\imath}$ in Example 4.1. The encoding encode($\mathbf{P}_{1_\imath}$) is the following first-order program.

(1) object(o).
(2) object(p).
(3) object(q).
(4) object(r).
(5) object(X) $\leftarrow$ parent(X, o).
(6) object(Q).
(7) parent(r, p).
(8) parent(r, q).
(9) parent(q, o).
(10) parent(p, o).
(11) withdraw($r, p, t, 0, func\_val$).
(12) withdraw($p, r, t, 0, func\_va'$).
(13) isa($X. X$) $\leftarrow$ object(X).
(14) isa(X, Y) $\leftarrow$ parent(X, Z). isa($Z. Y$).

(15) $\mathsf{local}(P, u, 0, \mathit{func\_val}) \leftarrow \mathsf{parent}(P, o)$.

(16) $\mathsf{local}(o, m, 0, \mathit{func\_val})$.

(17) $\mathsf{local}(o, s, 0, \mathit{func\_val})$.

(18) $\mathsf{local}(q, t, 0, \mathit{func\_val})$.

(19) $\mathsf{local}(p, s, 0, \mathit{func\_val})$.

(20) $\mathsf{local}(p, t, 0, \mathit{func\_val})$.

(21) $\mathsf{local}(Q, v, 0, \mathit{func\_val})$.

(22) $\mathsf{property}(V, m, 0, \mathit{func\_val}, \mathit{apply}(), X) \leftarrow \mathsf{property}(V, s, 0, \mathit{func\_val}, \mathit{apply}(), X)$,
$\mathsf{property}(V, v, 0, \mathit{func\_val}, \mathit{apply}(), g), \mathsf{uses}(V, o, m, 0, \mathit{func\_val})$.

(23) $\mathsf{property}(V, s, 0, \mathit{func\_val}, \mathit{apply}(), 5) \leftarrow \mathsf{uses}(V, o, s, 0, \mathit{func\_val})$.

(24) $\mathsf{property}(V, u, 0, \mathit{func\_val}, \mathit{apply}(), d) \leftarrow \mathsf{parent}(V, o), \mathsf{uses}(V, P, u, 0, \mathit{func\_val})$.

(25) $\mathsf{property}(V, s, 0, \mathit{func\_val}, \mathit{apply}(), 2) \leftarrow \mathsf{uses}(V, p, s, 0, \mathit{func\_val})$.

(26) $\mathsf{property}(V, t, 0, \mathit{func\_val}, \mathit{apply}(), a) \leftarrow \mathsf{uses}(V, p, t, 0, \mathit{func\_val})$.

(27) $\mathsf{property}(V, t, 0, \mathit{func\_val}, \mathit{apply}(), c) \leftarrow \mathsf{uses}(V, q, t, 0, \mathit{func\_val})$.

(28) $\mathsf{property}(V, v, 0, \mathit{func\_val}, \mathit{apply}(), g) \leftarrow \mathsf{uses}(V, Q, v, 0, \mathit{func\_val})$.

□

4.4 Implementability of ORLog

We conclude this section with the following remarks on the implementability of ORLog. It is clear from the encoding theorem 4.1 and the intuitive equivalence of reduction that for every i-consistent ORLog program **P**, there exists a perfect model for the corresponding reduced F-ORLog program $\mathbf{P}_r$ in the first-order predicate logic iff **P** has an intended model in ORLog. Furthermore, the encoding of the intended model is identical to the perfect model of the corresponding reduced program and vice-versa. This makes it possible to implement ORLog in any first-order database system that is capable of computing perfect models of programs. As it will be clear that no meta-interpretation is necessary to compute perfect models of reduced programs.

In [41], we explored the possibility of using $\mathcal{LDL}$ [32] as the host database system on which we built an ORLog interface. But its inability to support computation of perfect models based on local stratification was a major disadvantage. Since CORAL incorporates perfect model semantics as part of its negation semantics, we are inclined to use Coral at this point to maximize our design goals. In the next chapter we discuss issues pertinent to the implementation of ORLog in Coral.

Chapter 5

Design and Implementation of ORLog

Although the research into deductive object-oriented database systems are relatively new, experimental prototypes have already started to emerge. Again due to lack of a common data model, every system that came into existence have their unique characteristics. However, the approach to implementation can be classified into four distinct groups. (i) Implementation based on rewriting or translation into Datalog like languages, (ii) implementation based on extension to existing Datalog like languages, (iii) hybrid implementation, and (iv) systems based on direct implementation. In the first category some of the systems that are known to have been implemented are OOLP+ [25], OIL [76], Logical-objects [13], LLO [56]. In the second category are the systems CORAL++ [71], $\mathcal{LDL}++$ [69], Logres [23], IQL+ [1], among many others. ROCK & ROLL [10] and ConceptBase [45] are two systems that belong to the third category. We are not aware of any deductive object-oriented database systems that has been implemented directly that qualify as a member of the fourth category. We already briefly discussed about these systems in the previous chapter from a theoretical standpoint.

The development of ORLog was motivated by the need for a solid formal basis and a mathematical foundation for deductive object-oriented databases so that a commercially viable prototype can be built to meet the industrial demand for advanced database systems. It was also the goal that the language and the data model on which the language is built should be as natural and intuitive as possible. Except for few technical aspects we believe that we met these requirements. In practice, users of ORLog need not worry about these technical issues which were needed for the mathematical development of the language.

5.1 Related Research and Motivation

The number of deductive object-oriented database systems that emerged from the multitude of research done in this area is very small. We briefly discuss one representative system from each of the first three categories described above that are available in the literature. The goal of this discussion is to bring out the distinctions between the implementation approaches that are currently reported and practised and give a flavour of each of these styles. We hope that this discussion will also bring out few advantages and disadvantages of each of these alternatives approaches. It should be clear from the discussion that follows that every approach has their strengths and weaknesses and there is no absolute yardstick against which we can brand one approach to be the best.

5.1.1 CORAL++

Coral++ [71] is an integration of Coral with C++. The goal is to use an existing object model and perhaps an existing imperative language with its associated type system to extend the functionality of the Coral system. This allows Coral++ to exploit features in C++ and vice versa. Note that Coral is implemented in C++, hence an integration of the two is natural and practical.

Coral++ separates querying from creating, deleting and updating databases by providing separate sub-languages for the latter. It also relies on C++ for maintaining type safety, inheritance, encapsulation, and most other object-oriented features of the language. The overall goal is to use Coral run time system as much as possible and automatically invoke codes that handle object-oriented aspects of the language internally that are treated as external functions. Users need to define class definitions, methods. etc. in a C++ style sub-language, and can write C++ expressions in the rule bodies.

A database in Coral++ is required to be compiled along with the class definitions and basic Coral++ system so that the augmented Coral++ system becomes aware of the new user defined classes. Then a translation is applied to so that (i) for each method invocation and attribute access in Coral++ rule, external C++ predicates can be generated by the preprocessor that perform the relevant task at run-time. and (ii) that every method invocation can be replaced by these external predicates. Finally, the translated program is evaluated using the Coral interpreter. Note that the issues related to inheritance, encapsulation, etc. are resolved by C++ during compilation and translation, while Coral is responsible for deductions only.

5.1.2 OOLP+

The goal of this work is to implement an object-oriented logic programming language using only existing technologies. namely Prolog. The main idea is to define a suitable translation procedure for an object-oriented language. called the OOLP+ [25]. into Prolog. Hence, objects are clearly viewed as predicates or relations in the target language. OOLP+ provides constructs to define classes, objects and instances, methods. etc. in a key-word based language. In this language superclasses. instance objects. instance and class variables. and methods for an object class can be defined. Methods are defined using an extended syntax of Horn clauses. Limited amount of method ordering is possible in a specificational way. Method overriding is possible also in a limited way using Prolog cut.

Then given an OOLP+ program. the translation is applied and a Prolog program is obtained that captures the intended meaning of the OOLP+ program. The translated program is then evaluated in Prolog. Encapsulation appears to be not covered in this framework. and overriding seems to be limited. Persistence is not supported by OOLP+. Dynamic updating of instances and classes is possible vis-a-vis Prolog, and hence methods with side-effects are supported too. However, unlike many conventional object-oriented languages, OOLP+ does not allow invoking a specific method from a specific class, partly due to the translational approach to implementation.

It is, however, obvious from this discussion that this approach is the simplest and probably the cheapest. Whether the approach is desired in terms of flexibility, design. execution cost, etc. that remains an open question.

5.1.3 ROCK & ROLL

ROCK & ROLL [10]combines the capability of an imperative database programming language (ROCK) and a logic based first-order query language (ROLL). The imperative programming language is used for defining objects and classes, for writing method codes and as sole means of updating and changing the states of databases. The query language ROLL can be used to define rules and to express queries over extensional databases defined using ROCK. These two languages interact with and complement each other in a synergistic way. The fundamental principle behind its design is to use conventional language components judiciously and intelligently in complementary areas through language integration that does not require any extension of the components individually. It should be clear that Coral++ is a coupling between two languages through compilation and translation, while ROCK & ROLL is an integration of two languages.

Type system mismatch in this system is reduced since both the component languages are based on a common data model called the OM that makes it possible to perform strong type checking. Also since the data values accessed by one component can be accessed directly by the other. any special treatment or preprocessing of the data across platforms are not needed that further reduces the mismatch problem.

A particular strength of this system is that users can write programs in which ROLL may invoke any ROCK method as long as it is side-effect free, without any need for additional syntax. Whether a method call has a side-effect or not can be detected at compile time. In a complementary way, ROLL methods can be invoked in ROCK even with different binding patterns for their arguments. Overriding and overloading of ROLL methods are allowed and the process of (late) binding of a call to an implementation is handled in a way analogous to that for ROCK methods. Finally, the implementation includes a ROLL front-end for ROCK in which declarative component of programs can be defined, tested, type checked and compiled. In way, it can be said that this system is based on implementation of a deductive query language over an object-oriented database.

5.2 Objectives of the Interface

Several well-known deductive object-oriented database systems and research prototypes are known to have been built using other existing systems as back-ends. For example OOLP+ [25], OIL [76], Logical-objects [13], LLO [56]. Following the same direction, we propose a scheme for implementing ORLog in Coral [65] using a translational approach. Although a direct implementation of ORLog is obviously possible, we chose to follow the translational approach for the following reasons. Since the idea in a logic based language is to allow the user to program in a declarative way, query optimization then becomes a system level concern. However, the research into query optimization in deductive object-oriented databases is still in its infancy. While sophisticated optimization techniques are being researched, we try to take advantage of what has been successfully utilized in deductive databases for query optimization. This motivates our translation based approach where users perceive their applications naturally in an object-oriented way and never go through the mental exercise of mapping them into a non-object-oriented model. But still take advantage of the superior query optimization techniques available in the deductive paradigm.

We make use of the reduction technique we have developed in the previous chapter by customizing it for the Coral environment. We develop an user interface and command interpreter for ORLog. The interface can be viewed as an object-oriented front-end for Coral. For the moment, we only concentrate on the query processing aspect of the language and leave out persistence for simplicity.

5.3 Implementation of the ORLog System

In this section we discuss the design and prototype implementation of ORLog using Coral deductive database system as a back-end. We will first present an overview of the system, outline the system architecture, and discuss design considerations. Finally we review its performance and future enhancements.

5.3.1 Overview of the Interface

The primary goal of this prototype is (i) to provide a complete programming environment in ORLog that will reduce every ORLog program and answer queries by executing the reduced program in Coral, and (ii) to use Coral deductive database system as the backbone inference engine maintaining a complete transparency of Coral to the users of the interface. By making Coral user-transparent, we relieve the users from the burden of knowing Coral before being able to use ORLog. It has been designed to serve as an online interactive command interpretor as well as an ORLog interpretor. It allows users to write programs in ORLog, interpret them and answer queries. It also provides basic system services like viewing, editing and printing program files, saving in-memory programs, running a stored program, etc.

It is also one of our goals to let the users *eventually* exploit some of the programming features available in Coral. For example, execution tailoring through high level meta-annotations, modules, various query evaluation strategies (top-down, bottom-up, etc.), negation, persistent objects through Exodus storage manager, etc., and of course to take advantage of relational query optimization techniques, until we understand more about such issues in a purely object-oriented logic in general. We intend to support a subset of these features in our system by seamless integration of ours with Coral in a user-transparent way.

5.3.2 Systems Architecture

In Figure 14, the architecture of the system and its components are shown. The *user interface* module is an interactive shell and a command interpreter. Users are expected to interact with the system through this interface. At the command prompt users may request services, write programs interactively, or both. ASCII stored programs can also be brought into memory and executed using the **consult** command facility. Every clause that is part of a consulted file, or that was entered interactively, is parsed and translated into Coral by the *parser* and the *translator* modules respectively. The *translator* also determines the locality of each property clause and adds corresponding locality clauses to the program in real time. It also makes the user program clauses atomic[1]. Only syntactically correct clauses are accepted as active rules that are added to a system buffer by the *ORLog program buffer* module. A translated version is simultaneously maintained by the *translated Coral program* module. Users always see and refer to the program in the ORLog program buffer area.

[1]Recall that similar to F-logic [47], we also allow the users to write *molecular* formulas as a syntactic sugar. However, the molecules are broken down to atoms by the *atomizer* module, and subsequently all atomized clauses are converted to the usual clausal form (if not already)

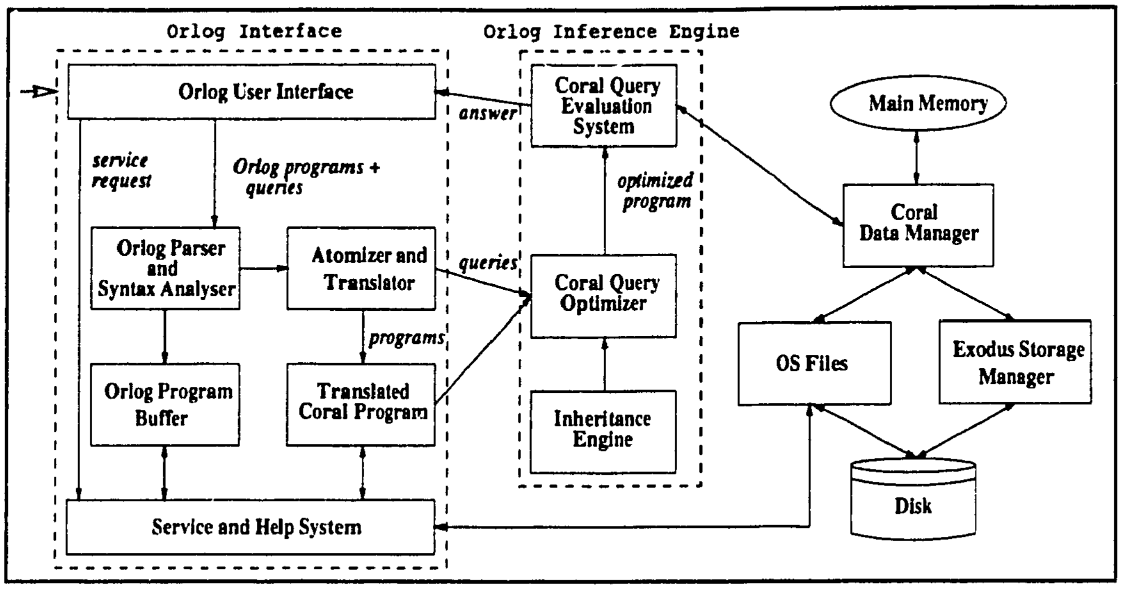

Figure 14: ORLog systems architecture.

Clauses can be added, or removed from the buffer area using the *service and help* module from command prompt. Queries (translated) are processed by the *Coral query optimizer* and the *Coral query evaluation system.* All queries are treated as ad hoc and processed the moment they are posed. The *Coral query optimizer* takes the translated program in the translated Coral program buffer, adds the set of rules (axioms) in the *inheritance engine* module to account for inheritability of properties, and then sends the optimized program to the *Coral query evaluation system.* Answers are then forwarded to the *user interface* by the *Coral query evaluator* as a set of bindings to the free variables in the query. Note that working in concert, the *Coral query optimizer*, *Coral query evaluation system* and the ORLog *inheritance engine* act as the *ORLog Inference Engine.*

5.3.3 Design Issues

Since the reduction is user-transparent, we maintain two versions of a program – one is in the input version to which the user refers, and the other is the translated version which is sent to Coral for execution. They are stored in two ASCII text files and are maintained by the program buffer and translated buffer modules respectively. Coral is invoked on the event of a user query as a system call from the interface and the encoded i-completed program is passed to it as an input parameter. Coral then adds the inheritability axioms from its inheritance engine and evaluates the query.

Completion and encoding take place interactively and instantly in the interface. This saves time. Since clauses can be added or taken away interactively, maintaining the set of locality information in the completed set is a bit involved. However, factoring out the inheritability computation in inheritance engine is viewed both as a convenience and saving. It also now lets us compute only the relevant part of the inheritability we need in order to answer a query.

5.3.4 Efficiency Issues

A considerable amount of time is being spent by the interface every time it calls Coral to evaluate a single query. The program files are being closed and opened several times, Coral environment is being created, data structures are built for the sole purpose of a single query and destroyed subsequently. Another source of inefficiency in our system is the modular stratification in Coral owing to our use of negation in implementing the inheritability function ∇ as a set of axioms in the *inheritance engine*. These axioms account for inheritance in Coral, and we found that the use of negation in the implementation of ∇ function is unavoidable.

Chapter 6

Comparison with Contemporary Research

In this chapter we compare ORLog with contemporary approaches to object-oriented logics in the literature. We show that overall ORLog has superior modeling capabilities compared to all representative logics. and it assigns meanings to every ORLog program which is not true for most other logics. We also show that, the concepts of *locality*. *withdrawal* and *inheritability* are useful in their own rights and they play a significant role in shaping the underlying semantics of inheritance in ORLog. Furthermore. we demonstrate that the use of *point of definition* (locality of clauses) based overriding gives us the edge over deduction based overriding in other logics. specially when implementation of logics are concerned. Besides, most procedural languages. e.g. C++. adopt definition based overriding approach such as ours. Finally. we discuss issues related to the ORLog interface to Coral.

6.1 Current Approaches to Behavioral Inheritance

There are several important differences between our language and others'. Behavioral inheritance has been studied in the context of Ordered Theories [50], Contextual Logic Programming [61, 62], artificial intelligence, non-monotonic reasoning, and a few others such as [20, 22], to name a few. Research in artificial intelligence and non-monotonic reasoning is mainly concerned with inheritance of properties and conflicts. and do not consider behaviors and structures of objects. Some of the proposals also sacrifice completeness, or are highly non-deterministic.

The proposals in [20, 50, 61, 62] are essentially modular languages that define collection of predicates in named modules. Inheritance is then captured by algebraic operations, called *program composition*, on the modules, to obtain new modules that inherit the predicates from a super module. In SelfLog [20], inheritance with overriding is captured by statically defining an ordering among the modules and using program composition to define inheritance. A similar approach is taken in [50, 61]. Although behavioral inheritance is captured in the semantics, program composition in itself is very costly in the context of object-oriented databases, where number of method definitions in the modules could be large and number of objects in practical databases may be much more than the static number of modules considered in these proposals. Besides, they achieve their functionality by giving up multiple inheritance and by avoiding the difficult issue of conflict resolution. Also the class of programs they allow is much smaller than a language such as F-logic [47], Gulog [29, 30], ORLog, etc.

The i-stratification in Gulog [29, 30] and the stable model for inheritance proposed in [22] are based on a preferred model construction. Clearly a framework such as ours, which captures behavioral inheritance within a sound and complete proof theory, has a greater intellectual appeal. Also, the approaches above have a very high computational complexity. In addition Gulog does not deal with inheritance conflicts and disallows programs with conflicts. On the other hand [22] does not recognize conflicts since it takes a relational approach to methods by treating them as set valued methods.

In the following sections we will take a closer look at the traditional approach to behavioral inheritance based on first-order (predicate) logic in artificial intelligence and knowledge representation and witness the difference with ORLog. We also examine a typical translation based logic such as OOLP+ [25] in the light of our logic. Finally, we compare our logic with best known logic in object-oriented paradigm, the F-logic [47].

6.1.1 Research in Artificial Intelligence and Knowledge Representation

We consider the canonical *Tweety* problem in this area. The logic program in Figure 15 states that (i) every bird flies, (ii) penguins do not fly, (iii) every penguin is a bird, and finally (iv) Tweety is a penguin. The issue is to decide whether the answer to the query $fly(tweety)$ is true or false, i.e., does Tweety fly? The answer to this query is not known, since the program does not have any model. In other words, we can prove both, $fly(tweety)$ and $\neg fly(tweety)$ by appropriately selecting the order of firing the rules in an attempt to prove $fly(tweety)$.

$$\begin{aligned}
r_1 &: fly(X) \leftarrow bird(X).\\
r_2 &: \neg fly(X) \leftarrow penguin(X).\\
r_3 &: bird(X) \leftarrow penguin(X).\\
r_4 &: penguin(tweety).\\
&\quad ?\ fly(tweety).
\end{aligned}$$

Figure 15: The *Tweety* problem in predicate logic.

It is easy to observe that the problems with this approach are that (i) implication ($\leftarrow$) is being used to model "is-a" as well as classical deduction, and (ii) negation ($\neg$) is being used to model exceptions or overriding. Hence the distinction between implication and is-a specification is lost. There is no constructs to specify is-a specificity and hence the intended rule ordering is also lost. Some researchers have proposed complicated extensions to this approach to achieve the desired rule ordering [28, 75]. But this ad hoc fixes makes it hard to use this approach to model object-oriented applications in general since these solutions are mostly application dependent.

As pointed out by Kifer [46], it is easier and perhaps desirable to develop a new language altogether for object-oriented data modeling by extending the current technology with special constructs to capture objects, methods, classes and instances, is-a hierarchy, overriding, etc. We now show that, with appropriate adaptation, we can easily give a semantics to the *Tweety* program above.

Consider the ORLog representation of the *Tweety* program of Figure 15 in Figure 16. The answer to the queries $tweety[locomotion \rightarrow fly]$ and $tweety[locomotion \rightarrow X]$

are shown. As can be seen that ORLog correctly assigns the intended meaning of the program and correctly computes the answers.

$$\begin{array}{lll}
r'_1 & : & bird[locomotion \rightarrow fly]. \\
r'_2 & : & penguin[locomotion \rightarrow walk]. \\
r'_3 & : & penguin : bird. \\
r'_4 & : & tweety : penguin. \\
 & & \\
 & & ?\ tweety[locomotion \rightarrow fly]. \\
 & & false \\
 & & \\
 & & ?\ tweety[locomotion \rightarrow X]. \\
 & & X = walk
\end{array}$$

Figure 16: The ORLog representation of the *Tweety* program.

It is easy to see that the structures present in ORLog (and other object-oriented languages such as F-logic [47], Gulog [29, 30], etc.) made it possible to capture the intended behavior of the program in Figure 16. Specially, the concepts such as locality, and inheritability played in important role in deciding which definition of *locomotion* the object *Tweety* will inherit. Although, F-logic and Gulog will assign the same meaning to this particular program, they do so at a very high computational cost since they take a model selection approach. Moreover, they are not able to assign meanings to every program, as we shall see later.

6.1.2 Semantics Based on Translation

We now take an OOLP+ [25] representation in Figure 17 of the *Tweety* program in Figure 15. OOLP+ is a language which does not have its own semantics and proof procedure. Instead, it relies on rewriting every OOLP+ program into Prolog and giving a relational interpretation to the source programs in OOLP+.

When rewritten in Prolog, the translated program looks like one in Figure 18. During translation, (i) every super class definition of the form **class** a **has super_class** b is translated into a Prolog clause of the form $b(X) \leftarrow a(X)$, (ii) every method definition of the form **class** a **has methods** p(Y) $\leftarrow$ body is translated into

```
class       bird has
            methods transport-mode(fly).

class       wingless-bird has
            super_class bird
            methodso transport-mode(walk).

instance    chipper
            is-a bird.

instance    tweety
            is-a wingless-bird.

            ? tweety ≪ transport-mode(fly).

            ? tweety ≪ transport-mode(X).
```

Figure 17: An OOLP+ representation of the *Tweety* program.

$p(X,Y) \leftarrow a(X), body$, (iii) every overridden method definition of the form **class** a **has methodso** p(Y) ← body is translated into a pair of clauses of the form $p(X,Y) \leftarrow a(X), !, p'(Y)$ and $p'(Y) \leftarrow body$, and finally (iv) every instance declaration of the form **instance** a **is-a** b translates simply into $b(a)$. A message call of the form **target** ≪ p(X), however, translates as $p(target, X)$ where **target** is an object id.

An override method of a class overrides any other methods of the same name from the superclasses. The translation and assertion of an override method is identical to the method translation scheme, except that a cut (!) is inserted in the body of the clause after the first sub-goal (which tests for class membership). The semantics of Prolog cut achieves the desired effect of overriding. The answers to the queries are also given in Figure 18.

The OOLP+ language uses keywords to override methods, and can not override class or instance variables. Hence it can not assign default values to instances. Furthermore, overriding is modelled via keywords and left as the responsibility of the users and hence is open to errors. Furthermore, it is not suitable for bottom-up computation due to the presence of the cut in overridden methods.

```
r1 : bird(X) ← wingless-bird(X).
r2 : transport-mode(X, fly) ← bird(X).
r3 : transport-mode(X, Y) ← wingless-bird(X), !, transport-mode'(Y).
r4 : transport-mode'(walk).
r5 : bird(chipper).
r6 : wingless-bird(tweety).

     ? transport-mode(tweety, fly).
     false

     ? transport-mode(tweety, X).
     X = walk
```

Figure 18: Prolog representation of the *Tweety* program in OOLP+.

6.1.3 Comparison with F-logic

F-logic does not *directly* capture behavioral inheritance as in code reuse. The only way to realize it is by *simulating* it via F-logic's pointwise overriding and deduction. But then, this makes it the programmer's responsibility. Consider Example 3.2. In our case clause (8) produces $p[m \rightarrow 2]$ by code reuse (as discussed in Example 3.2), while F-logic (for the program as it is in Example 3.2) will inherit the ground data expression $o[m \rightarrow 5]$ from o to produce $p[m \rightarrow 5]$ which is not behavioral inheritance as in code reuse. Furthermore, due to clauses (12) and (13), F-logic will have two minimal models – in one model it will inherit (12) in r and override (13); and in another model it will inherit (13) in r and override (12). By contrast, in ORLog we have only one model in which we inherit neither. By analogy with the literature on negation [33], we can say that F-logic's approach to multiple inheritance is *brave* while ours is *cautious*. Rather than debate on which is better, we would like to remark that a brave semantics for multiple inheritance destroys any hopes of a complete proof theory[1]. Besides, a nice feature of our framework is the ability to simulate the brave semantics at a "local" level, to a limited extent. This can be accomplished by using the withdrawal atoms. E.g., in Example 3.2, clause (7) withdraws method t^0_- defined

[1]That is not to say, a complete proof theory for the cautious semantics is straightforward, as can be seen from this paper.

in p from being inherited in r. Since t^0_- is also defined in q, r can now inherit t^0_- from the unique source q.

A second difference with F-logic is that in F-logic a necessarily monotonic signature inheritance is *built into* the logic. There are situations where a signature defined in a superclass may have to be withdrawn from a (not necessarily immediate) subclass. F-logic's inheritable and non-inheritable method expressions help solve this problem to a limited extent. However, consider the situation where a signature is defined in o, and is inherited up to the class r, where $r : q$, $q : p$, $p : o$ and suppose it has to be withdrawn from s, where $s : r$. It is not clear how this can be accomplished in F-logic. In ORLog, this can be done rather easily by using an appropriate withdrawal atom. Finally, we remark that the concepts of locality, inheritability, and withdrawal of properties are quite useful in themselves and we are not aware of any other logic where they are given a formal status.

6.2 Comments on Implementation Issues

Despite several of its efficiency related drawbacks, the ORLog system compares very nicely with contemporary research prototypes in its class. Most of the translation based implementations have similar drawbacks, if not serious ones. It is easy to make an important observation in almost all implementations other than ORLog. Most languages take a success-failure based (deduction based) approach to overriding, hence any implementation will have to rely heavily on negation. In fact, it can be shown by taking a pathological case that to decide on the inheritability of any single method, we will have to compute the whole database. Whereas, the point of definition based approach to overriding as in ORLog makes it simple, intuitive, and efficient to the greatest extent possible. This makes ORLog a viable candidate for a serious implementation.

However, the ORLog experience offers guidelines for addressing several important issues and suggests that enhancements are possible on the present system. First, the time spent by the interface in Coral calls can be saved by managing to run Coral in a dedicated mode and make Coral talk to the interface. In that way, we need to invoke Coral only once from the interface, use shared memory space and data structures between Coral and the interface, and spend minimal amount of time in transferring control from the interface to Coral, and vice-versa. This can be improved in yet another potentially involved way. We can change the Coral interface to accept ORLog code, and then translate the code internally to Coral's native language.

Secondly, one of our goals is to eventually exploit Coral features such as modules, execution control through meta-annotations, etc. This can be achieved very easily just by adding a simple identity function in the translator and passing the declarations to Coral. Finally, in its current form, ORLog does not support persistent objects. This essential functionality needs to be added. A possible way to add this feature would require to extend Exodus storage manager used in Coral with object management capabilities.

Chapter 7

Conclusion and Future Research

The goal of this thesis was primarily to develop a logical account of behavioral inheritance in deductive object-oriented databases. To understand what behavioral inheritance means and how inheritance of behaviors work in a network of objects organized in some specialization-generalization hierarchy, we first needed to develop an abstract data model, called the OR model, for database schema design. OR model was instrumental in visualizing the semantics of inheritance at a higher level of abstraction. It thus laid the foundation of the logical query language ORLog and its underlying semantics, which was the main focus of this work.

The OR model we presented in this thesis can be viewed as an extension of SDM in the direction of OO models. It incorporates a balanced mix of features from the two paradigms and enriched the model by introducing the new concepts such as *property withdrawal* and *re-introduction*, *accessibility* of properties, *stratified constraints*, methods in relationships, inheritance conflict resolution, etc. It uses, to a large extent, widely used constructs from the well-known ER model and leading OO models and combines object and value based modeling. It facilitates dynamic schema design and incorporation of knowledge in the schema in the form of first-order rules in a way similar to DERDL [37] and DK model [38] respectively but in a much more effective and sophisticated way. The choice of stressing inter-object association based on type constructors (e.g., relationships) provides flexibility and practicality in schema design as opposed to models that stress inversion as the sole basis for defining associations as in SDM [36, 63, 39], F-logic [47], etc. Finally, it guaranteed re-use of software components by adopting a flexible notion of *is-a* hierarchy, where objects are organized at users' choice, but not based on types, or sub-superset relationships of property inclusion.

We then considered a simple and intuitive object-oriented logic called ORLog as a counterpart of the OR model and developed an elegant model theoretic and fixpoint theoretic characterization of the semantics of its definite clause fragment, accounting for multiple behavior inheritance with conflict resolution and overriding. We also provided a simple sound and complete proof theory. This was achieved by keeping the is-a hierarchy finite and static. While, compared to other logics such as F-logic, our language has limited expressive power, we have exploited the restricted setting and successfully captured the non-trivial concepts of multiple behavior inheritance within the model theory and a sound and complete proof theory, to our knowledge, for the first time. On the other hand, our simple setting still admits a large class of practically useful programs. In addition, we have proposed the notions of *locality*, *inheritability*, and *withdrawal* of methods and signatures, as first class concepts within the logic. These are useful concepts in their own right, as demonstrated earlier.

We note that even though for the sake of simplicity, the proof theory was presented based on the closure of a program, in an actual implementation, it is really not necessary to compute the entire closure of a program. More precisely, it may be possible to compute only a small and relevant subset of the closure in order to prove a given goal. Similarly, the bottom-up fixpoint computation can be made more efficient, by incorporating ideas similar to the well-known magic sets method [11]. We are currently investigating these and other optimization opportunities. We have completed a prototype implementation [43, 44] based on translation to Coral [65], and are working on a direct implementation. We are also investigating relaxations to the present restrictions on ORLog programs while still capturing behavioral inheritance within the logic. Although ORLog does not account for encapsulation, we show later in this chapter that encapsulation can be incorporated in ORLog as an orthogonal extension to the current proposal.

We also developed a technique to reduce inheritance to pure deduction, and a translation of reduced ORLog programs to first-order logic that can be exploited by several other languages for which a translational approach is critical to define their logical semantics and hence depends solely on this approach. This technique helped us implement ORLog in a first-order deductive database system, such as Coral[1]. We

[1]Note that Coral includes a few higher-order features too.

utilized this idea and implemented ORLog in Coral deductive database system as an object-oriented front-end.

7.1 Issues that are not Covered in this Thesis

Recall that the main focus of this thesis was to develop a logical semantics for behavioral inheritance, although we made provisions for other essential features of a complete object-oriented language to be included at a later time as an orthogonal extension. We will now discuss some of those missing features that we intend to carry on as our future work.

Recall that in the OR model we introduced the idea of property re-introduction, accessibility and stratified constraints. A formal definition of accessibility can be found in [40]. While we leave the issue of accessibility and stratified constraints as open issues, we can incorporate re-introduction by simply adjusting the definition of inheritability function ∇ as in Figure 7.1. Note that without developing a new syntax for re-introduction, in Figure 7.1 we use i-atoms to achieve the same effect without any loss.

The purpose of introducing the signature atoms was to utilize the signatures of properties to enforce "type safe" databases. We did not expand on this issue in this work. We can proceed to define well-typing and typed models in a way similar to F-logic [47]. We next discuss few other issues that we are currently investigating as our future work.

7.2 Future Work

Although ORLog has superior features compared to contemporary languages, it also has weaknesses, and hence there remains opportunities for further extensions. In fact the current formalization opened up a whole suit of new research issues that seem promising. In the following sections, we outline few weaknesses of ORLog and discuss several extensions that we have planned as our future work to remove some of its deficiencies.

$$\nabla'(S, m^k_{\mapsto}, o) = \begin{cases} p & \text{if } o[m^k_{\mapsto}] \notin S \text{ and } [\exists q \text{ such that } o : q \in S,\ \nabla'(S, m^k_{\mapsto}, q) = \\ & p,\ p[m^k_{\mapsto}] \in S \text{ and } (\forall r, \text{ such that } o : r \in S, \text{ one of the} \\ & \text{following holds.} \\ & \quad \bullet\ \nabla'(S, m^k_{\mapsto}, r) = r, \text{ and } r[m^k_{\mapsto}] \notin S, \text{ or} \\ & \quad \bullet\ \nabla'(S, m^k_{\mapsto}, r) = p, \text{ or } o[m^k_{\mapsto} \propto\!\lhd r] \in S, \text{ or } r[m^k_{\mapsto} \rhd\!\propto r] \in \\ & \quad\ \ S.)] \\ p' & \text{if } o[m^k_{\mapsto}] \notin S \text{ and } o[q@m^k_{\mapsto}] \in S \text{ and } [\exists q \text{ such that } o :: \\ & q \in S,\ \nabla'(S, m^k_{\mapsto}, q) = p',\ p'[m^k_{\mapsto}] \in S \text{ and } (\forall r, \text{ such that} \\ & o : r \in S, \text{ one of the following holds.} \\ & \quad \bullet\ \nabla'(S, m^k_{\mapsto}, r) = r, \text{ and } r[m^k_{\mapsto}] \notin S, \text{ or} \\ & \quad \bullet\ o[m^k_{\mapsto} \propto\!\lhd r] \in S, \text{ or } r[m^k_{\mapsto} \rhd\!\propto r] \in S, \text{ or} \\ & \quad \bullet\ \nabla'(S, m^k_{\mapsto}, r) = u, \text{ and } \exists v \text{ such that } o : v \in \\ & \quad\ \ S.\ \nabla'(S, m^k_{\mapsto}, v) = w.)] \\ o, & \text{in all other cases.} \end{cases}$$

Figure 19: New definition of ∇ to incorporate re-introduction of properties.

7.2.1 Function Symbols in ORLog

One of the technical reason for not allowing function symbols in our language was to ensure static computability of inheritability using ∇ at compile time. Although inclusion of function symbols is desirable for the modeling of much more interesting databases and dynamic creation of objects, it threatens static computation of inheritability since the object hierarchy may potentially become infinite, and termination of ∇ can not be guaranteed anymore.

However, we believe function symbols can still be included in ORLog in a restricted way. The idea is to define suitable properties for admissibility of clauses that create new objects such that the depth and breadth of the object hierarchy still remain finite. A partial solution to this effect is already at hand, while we are investigating a much more general solution such that a larger class of programs can be made admissible.

7.2.2 Dynamic Object Hierarchies

Recall that we have assumed a static object hierarchy in every ORLog program primarily by not allowing i- and p-atoms in the rule bodies of is-a clauses again to be able to manage static computing of inheritability of clauses. If we relax this restriction. static determination of inheritability would not be possible, and similar to Dobbie and Topor [29], for example, we will have to adopt a model theoretic semantics based on stratification as in negation in logic programs. In [21], we investigate the issue of dynamic is-a hierarchy and define a stable model semantics for behavioral inheritance. The language there is similar, and we believe that the idea can be extended to ORLog without much complication. One important consequence of allowing dynamic object hierarchies is that we are no longer required to compute inheritability at compile time. Hence, we may now safely allow function symbols in our language and make it more expressive, since infiniteness of the hierarchy does not matter any more. Another consequence will be that instead of a least intended model, we will have several minimal models for ORLog programs. A downside is that the high computational complexity of stable models will be incurred in this case.

7.2.3 Encapsulation in ORLog

Another most fundamental and essential concept of object-orientation is encapsulation. While this issue has been a subject of intense research in the logic programming community for quite some time, researchers in object-oriented logic programming did not address this issue seriously. Partly the reason may be issues like inheritance have given priority and not much has been achieved until now.

Encapsulation is an essential software engineering technique which help define objects with interfaces that restrict the access to the state of an object only through the set of *public* methods of the object interface. As such, it frees the users from the burden of knowing the internal structure and state of the objects by hiding all unnecessary details. This mechanism is referred to as *structural encapsulation* as opposed to the notion of *behavioral encapsulation* that refers to the concept of hiding the implementation details of the methods in objects. Objects themselves, however, can access their own properties irrespective of the state of the encapsulation status, or the interface.

Although the ideas seem to be very simple, researchers could not agree on a logical account of these simple idea until recently. Much of the research were centred around the context of modules in logic programming. Miller's theory of modules [59] was instrumental in defining a limited account of encapsulation. Dynamic visibility has been studied in [8] again in the context of modules. But all these proposals failed to give an effective interface mechanism that can be used in objects. Recently, we proposed an elegant solution to this problem in [21] which addresses the issue in an object-oriented context. Although the language considered in [21] is slightly different and more restricted, it provides a formal foundation based on which systems can be developed and tailored for a particular application. We believe that the idea proposed in [21] can be exploited to develop a formal account of encapsulation in ORLog. We plan to incorporate encapsulation in ORLog along this guideline in future.

7.2.4 Enhancing the ORLog User Interface

The current prototype interface we have developed for Coral has several limitations. We have an elaborate plan to enhance this interface so that it becomes a practical system building tool. Some of the enhancements that we are considering are as follows.

We are now working on a more user-friendly graphical user interface for ORLog. In this interface, users will be able to define database schema and objects, browse database, execute queries, navigate through the object structures and develop applications, all in a graphical way. Another graphical tool is being developed to track inheritance conflicts at compile time so that such conflicts may be resolved, if desired, by the users using the mechanisms provided in ORLog. We are also considering developing a method to ensure type safe databases in ORLog based type inference from the signatures defined in ORLog programs.

In the current state, ORLog is a main memory database. We would like to include persistence to ORLog objects. This will demand a significant amount of research and development since the issue of object persistence raises new issues that are currently being investigated by the database researchers. A particular issue that comes to ones mind is that what are the new issues that need to be addressed in a framework like ours where objects are viewed as relations since we take a translational approach to implementation. This almost certainly has performance related implications. Hence, a natural next step would be to implement ORLog using a direct approach and use persistent objects instead of relations, as our experience grows.

7.2.5 Updates and Query Optimization

So far we have only considered retrieval queries and methods that are side-effect free. To consider methods with side-effects would require a firm logical basis for updates. A possible course of action would be to investigate adapting ideas similar to transaction logic by Bonner and Kifer [15]. In [46], Kifer shows that such integration is possible in a logical way in languages similar to F-logic.

A final open issue is query optimization in ORLog. We believe that query optimization can now be investigated since we have a clear semantics of inheritance which most of the previous proposals lacked and rendered such study extremely hard.

7.2.6 Schema Integration and Evolution

We believe that OR model and ORLog can be used for schema integration and evolution applications. The notion of withdrawal proposed in OR model may become handy in modeling such applications. Although we leave it as our future work, we can cite works that are already using similar languages and models for such applications. In [54], F-logic [47] has been used to query heterogeneous databases. Since ORLog has strong similarity with F-logic, ORLog may also be used for such applications. In fact, the language SchemaLog [51, 52] has been developed mainly for modeling schema integration and evolution applications. Again, it turns out that ORLog is able to simulate SchemaLog to a limited extent. Unlike F-logic or ORLog, in which functionalities needed for interoperability need to be *simulated*, SchemaLog allows for constructs for direct and efficient interoperability. We intend to explore these issues further in our future works.

Bibliography

[1] S. Abiteboul. Towards a deductive object-oriented language. *Data and Knowledge Engineering*, (5):263-287, 1990.

[2] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, 1987.

[3] H. Aït-Kaci and R. Nasr. LOGIN: a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:182-215, 1986.

[4] H. Aït-Kaci and A. Podelski. Towards a Meaning of LIFE. Technical Report 11, Digital Paris Research Labs, 1991.

[5] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object oriented database programming language. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 565–575, Barcelona, 1991.

[6] K. R. Apt and R. Blair. Arithmetic classification of perfect models of stratified programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 765-779. The MIT Press, 1988.

[7] K. R. Apt, R. Blair, and A. Walker. *Towards a Theory of Declarative Knowledge*. Morgan-Kaufmann, 1988.

[8] M. Baldoni, L. Giordano, and A. Martelli. A Multimodal Logic to Define Modules in Logic Programming. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium ILPS'93*, pages 473–487. The MIT Press, 1993.

[9] F. Banchilhon, C. Delobel, and P Kanellakis. *Building an Object-Oriented Database System: The story of O_2*. Morgan Kaufmann, 1992.

[10] M. L. Barja, N. W. Paton, A. A. A. Fernandes, M. H. Williams, and A. Dinn. Am effective deductive object-oriented database through language integration. In *Proc. of the 20th VLDB Conference*, Satiago, Chile, 1994.

[11] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 269–283, 1987.

[12] E. Bertino. Data hiding and security in object-oriented databases. In *8th IEEE International Conference on Data Engineering*, pages 338-347, 1992.

[13] E. Bertino and D. Montesi. Towards a logical object-oriented programming language for databases. In A Pirotte, C. Delobel, and G. Gottlob, editors, *Proc. of the 3rd Intl. Conf. on EDBT*, pages 168–183. Springer-Verlag, 1992. LNCS 580.

[14] G. M. Birtwistle. O.-J. Dahl, B. Myhrhaug. and K. Nygaard. Simula begin. Technical report. Box 1717. S-222 01 Lund, Sweden, Auerbach, Philadelphia, 1973.

[15] A. Bonner and M. Kifer. Transaction logic programming. In *International Conference on Logic Programming*, pages 257-279. MIT Press, 1993.

[16] A. H. Borning and D. H. Ingalls. A type declaration and inference system for smalltalk. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 133–141, 1982.

[17] M. Bouzeghoub and E. Metais. Semantic modeling of object oriented databases. In *Proceedings of the 17th Intl. Conf. on Very Large Data Bases*, pages 3–14, 1991.

[18] S. Brass and U. W. Lipeck. Semantics of inheritance in logical object specifications. In *Proceedings of the Intl. Conf. on Deductive and Object-Oriented Databases*, pages 411-430, 1991.

[19] M. L. Brodie. On the development of data models. *On Conceptual Modeling, Perspectives from Artificial Intelligence, Databases and Programming Languages*. pages 19–48, 1984.

[20] M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.

[21] M. Bugliesi and H. M. Jamil. A logic for encapsulation in object oriented languages. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, pages 213–229, Madrid, Spain, 1994. Springer-Verlag. LNCS 844.

[22] M. Bugliesi and H. M. Jamil. A stable model semantics for behavioral inheritance in deductive object oriented languages. In G. Gottlob and M. Y. Vardi, editors, *Proceedings of the 5th International Conference on Database Theory (ICDT)*, pages 222–237, Prague, Czech Republic, 1995. Springer-Verlag. LNCS 893.

[23] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanaca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *ACM SIGMOD*, pages 225-236, 1990.

[24] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[25] M. Dalal and D. Gangopadhyay. OOLP: A translation approach to object-oriented logic programming. In *Proceedings of the First DOOD Conference*, pages 593-606, 1990.

[26] P. Dechamboux and C. Roncancio. Peplom^d: An object-oriented database programming language extended with deductive capabilities. In D. Karagiannis, editor, *Proc. of the 5th Intl. DEXA Conf.*, pages 2-14, Athens, Greece, 1994. Springer-Verlag. LNCS 856.

[27] G. Decorte, A. Eiger, D. Kroenke, and T. Kyte. An object-oriented model for capturing data semantics. In *Proceedings of the 8th IEEE Intl. Conf. on Data Engineering*, pages 126–135, 1992.

[28] Y. Dimopoulos and Antonis Kakas. Logic programming without negation as failure. In John Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 369–383, Portland, Oregon, December 1995. MIT Press.

[29] G. Dobbie and R. Topor. A Model for Inheritance and Overriding in Deductive Object-Oriented Systems. In *Sixteen Australian Computer Science Conference*. January 1988.

[30] G. Dobbie and R. Topor. A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems. In *Proc. 3rd Intl. DOOD Conf.*, pages 473–488, December 1993.

[31] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press. London. 1972.

[32] D. Chimenti et. al. The $\mathcal{LDL}$ system prototype. *IEEE Journal on Data and Knowledge Engineering*, 2(1):76–90, 1990.

[33] M. Fitting. The family of stable models. *Journal of Logic Programming*. 17(2/4):197, November 1993.

[34] A. Formica and M. Missikoff. Integrity constraints representation in object-oriented databases. In *Proc. 1st Int. Conference on Knowledge and Information Management*, 1992.

[35] K. Gerti and M. Schrefl. Object/behavior diagrams. In *Proceedings of the 7th IEEE Intl. Conf. on Data Engineering*, pages 530–539, 1991.

[36] M. Hammer and D. McLeod. Database description with sdm: A semantic database model. *ACM Transactions on Database Systems*, 6(3), September 1981.

[37] J. Han and Z. Li. Derdl - a knowledge-based data language for a deductive entity-relationship model. pages 317–325, 1989.

[38] M. A. W. Houtsma and P. M. G. Apers. *Data and Knowledge Model: A Proposal.* ACM Press, 1987.

[39] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. 19(3):201–260, September 1987.

[40] H. M. Jamil and L. V. S. Lakshmanan. ORLog: A Logic for Semantic Object-Oriented Models. In *Proc. 1st Int. Conference on Knowledge and Information Management*, pages 584–592, 1992.

[41] H. M. Jamil and L. V. S. Lakshmanan. Realizing ORLog in $\mathcal{LDL}$. In *Proceedings of the ACM SIGMOD Workshop on Combining Declarative and Object-Oriented Databases*, pages 45–59, Washington DC, May 1993. Held in conjunction with the ACM PODS and SIGMOD Conference.

[42] H. M. Jamil and L. V. S. Lakshmanan. A declarative semantics for behavioral inheritance and conflict resolution. In John Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 130–144, Portland, Oregon, December 1995. MIT Press.

[43] H. M. Jamil and L. V. S. Lakshmanan. An object-oriented front-end for deductive databases. In V. S. Alagar and M. Nivat, editors, *Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 581–584, Montreal, Canada, July 1995. Springer-Verlag. LNCS 936.

[44] H. M. Jamil, L. V. S. Lakshmanan, and I. Orenman. Reducing inheritance to deduction by completion. Technical report, Department of Computer Science, Concordia University, Montreal, Canada, March 1995.

[45] M. Jarke, M. Jeusfeld, and T. Rose. Software process modeling as a strategy for kbms implementation. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. of the 1st DOOD Confe nce*, pages 531–550, Kyoto, Japan, 1990. North-Holland.

[46] M. Kifer. Deductive and object data languages: A quest for integration. In *International Conference on Deductive and Object-Oriented Databases*, Singapore, December 1995.

[47] M. Kifer, G. Lausen, and J. Wu. Logical Foundations for Object-Oriented and Frame-Based Languages. *Journal of the Association of Co[illegible]ting Machinery*, July 1995.

[48] W. Kim, J. Banerjee, H-T Chou, J. F. Garza, and D Woelk. Composite object support in an object-oriented database system. In *Proceedings of the OOPSLA Conference*, 1987.

[49] W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. In *SIGMOD Record 18(2)*, 1989.

[50] E. Laesen and D. Vermeir. A Fixpoint Semantics for Ordered Logic. *Journal of Logic and Computation*, 1(2):159–185, 1990

[51] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proc. of the 3rd International Conference on Deductive and Object-Oriented Databases*, pages 81–100, Phoenix, Arizona, 1993.

[52] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. Logic & algebraic lnaguages for interoperability in multi-database systems. *Journal of Logic Programming*, February 1996. To appear. Preliminary verison appears in [51].

[53] C. Lecluse, P. Richard, and F. Velez. *O_2, An Object-Oriented Data Model.* ACM Press, 1987.

[54] A. Lefebvre, P. Bernus, and R. Topor. Querying heterogeneous databases: A case study. In M. Orlowska and M. Papazoglou, editors, *Proc. of the 4th Australian Database Conference*, pages 186–197, Brisbane, February 1993. World Scientific.

[55] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[56] Y. Lou and Z. M. Ozsoyoglu. An object-oriented deductive language with methods and method inheritance. In J. Clifford and R. King, editors, *Proceedings of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 198–207, Denver, Colorado, 1991.

[57] S. Manchanda. Higher-order" logic as a data model. In *Proc. of 2nd International. Workshop on Data Base Programming Languages*, pages 330–341, June 1989.

[58] F.G. McCabe. *Logic and Objects.* Prentice Hall International, London, 1992.

[59] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, (6):79–108, 1989.

[60] L. Monteiro and A. Porto. Contextual Logic Programming. In *6th ALP Intl. Conf. on Logic Programming*, 1989.

[61] L. Monteiro and A. Porto. A transformational view of inheritance in Logic Programming. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conference on Logic Programming*, pages 481–494. The MIT Press, 1990.

[62] L. Monteiro and A. Porto. Syntactic and Semantic Inheritance in Logic Programming. In J. Darlington and R. Dietrich, editors, *Workshop on Declarative Programming*. Workshops in Computing. Springer-Verlag, 1991.

[63] J. Peckham and F. Maryanski. Semantic data models. 20(3):153–189, September 1988.

[64] Teodor Przymusinski. Perfect Model Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1081–1096. The MIT Press, 1988.

[65] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL : Control, Relations and Logic. In *Proc. of 18th VLDB Conference*, 1992.

[66] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1993.

[67] K. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[68] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA*, pages 466–481, 1987.

[69] S. Shaw, L. Foggiato-Bish, I. Garcia, G. Tillman, D. Tryon, W. Wood, and C. Zaniolo. Improving data quality via $\mathcal{LDL}++$. In *Workshop on Deductive Databases, JICSLP*, pages 60–73, 1993.

[70] O. R. L. Sheng and C. P. Wei. Object-oriented modeling and design of coupled knowledge-base/database systems. In *Proceedings of the IEEE International Conference on Data and Knowledge Engineering*, pages 98–105, 1992.

[71] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. of the 19th VLDB Conference*, pages 158–170, 1993.

[72] B. Stroustrop. *The C^{++} Programming Language*. Addison-Wesley, 1986.

[73] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[74] D. S. Touretzky. *The Mathematics of Inheriance Systems.* Morgan Kaufmann, Los Altos, CA, 1986.

[75] J. H. You, S. Ghosh, L. Y. Yuan, and R. Goebel. An introspective framework for paraconsistent logic programs. In John Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 384–398, Portland, Oregon, December 1995. MIT Press.

[76] C. Zaniolo. Object identity and inheritance in deductive databases – an evolutionary approach. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. of the 1st DOOD Conference*, pages 7–24, Kyoto, Japan, 1990. North-Holland.

[77] C. Zaniolo, H. Ait-Kaci, D. Beech, S. Cammarata, L. Kerchberg, and D. Maier. Object-oriented database and knowledge systems. Technical Report DB-038-85, MCC, 1985.