An Axiomatic Approach to Deductive Object-Oriented Databases

Alvaro Adolfo Antunes Fernandes

Degree of Doctor of Philosophy
Heriot-Watt University
Department of Computing and Electrical Engineering
1995

Contents

List of Figures

List of Tables

Acknowledgments

I acknowledge with the deepest and sincerest gratitude the role played by my supervisors, Prof. M. Howard Williams and Dr Norman W. Paton, in the research reported in this dissertation. Their guidance and support were truly invaluable. In this dissertation and in the work that it describes, I have striven to do justice to their commitment and scientific excellence, and to reciprocate their valued friendship.

It was a privilege to have collaborated with Dr. Alia I. Abdelmoty, Dr. Maria L. Barja and Mr. Andrew Dinn in the design and development of the ROCK & ROLL deductive object-oriented database system. Without their competence and dedication, the contributions of the research reported here would only have been realized at the level of small scale prototypes. In many aspects of the research reported here, their own research served as a source of both information and inspiration. I am full of admiration for the fruits of their research, and deeply grateful for all the help and support they gave me.

For a brief, but crucial, initial period in the ROCK & ROLL project, Dr. Andrew Bowles was instrumental in helping me to characterize a number of issues that, in turn, proved of great relevance in shaping the research reported in this dissertation. I thank him sincerely for his part in that.

Prof. Keith G. Jeffery, Mr. Neil Smith, Dr J. Michael P. Quinn, in their collaborative role in the ROCK & ROLL project, provided many insights and raised many questions about the research reported here. Their expertise and support is gratefully acknowledged.

I am grateful to Dr. Subrata K. Das for allowing me to use the code for his prototype deductive database in testing many research results contained here.

The Department of Computing and Electrical Engineering of Heriot-Watt University is an excellent environment to do research. Most of the merit for this goes to its past and present members. I acknowledge with gratitude their help, and also their dedication and competence which are truly exemplary.

The whole of the research reported in this dissertation was done as part of the Deductive Object-Oriented Database project — funded by the Department of Trade and Industry (DTI) and the Engineering and Physical Sciences Research Council (EPSRC) within the Joint Framework for Information Technology (JFIT) programme, with industrial collaboration from Ordnance Survey and ICL; and the Active Rules project — funded by the EPSRC. I am grateful for having been given the opportunity to take part in these projects.

Thanks to all my friends and relatives for keeping my heart warm all along the way.

Declaration

I hereby declare that this is my own account of the work presented in this thesis and that this work was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgment is made. This thesis has not been submitted for any other degree to this or any other University. However, some material presented in this thesis has been or will be published, in different form, as follows:

1. External Publications

    (a) Abdelmoty, A.I., Paton, N.W., Williams, M.H., Fernandes, A.A.A., Barja, M.L., and Dinn, A. **Geographic Data Handling in a Deductive Object-Oriented Database**. In D. Karagiannis (ed.) *Database and Expert System Applications – 5th International Conference, DEXA'94, Proceedings*, LNCS 856, pp. 445–454. Athens, Greece, September 1994. Springer-Verlag. ISBN 3-540-58435-8.

    (b) Barja, M.L., Paton, N.W., Fernandes, A.A.A., Williams, M.H., and Dinn, A. **An Effective Deductive Object-Oriented Database Through Language Integration**. In J. Bocca, M. Jarke, and C. Zaniolo (eds.) *Proceedings of the Twentieth International Conference on Very Large Databases*, pp. 463–474. Santiago, Chile, September 1994. Morgan Kaufmann Publishers

    (c) Barja, M.L., Paton, N.W., Williams, M.H., and Fernandes, A.A.A. **A Database Programming Language for a Deductive Object-Oriented Database**. 1995. To appear in *Journal of Information Science and Technology.*

    (d) Barja, M.L, Fernandes, A.A.A., Paton, N.W., Williams, M.H., Dinn, A. and Abdelmoty, A.I. **Design and Implementation of ROCK & ROLL: A Deductive Object-Oriented Database System**. *Information Systems 20(3)*, pp. 185–211, 1995.

    (e) Dinn, A., Paton, N.W., Williams, M.H., Fernandes, A.A.A., and Barja, M.L. **The Implementation of a Deductive Query Language Over an Object-Oriented Database**. To appear in *Proceedings of the 4th International Conference on Deductive Object-Oriented Databases – DOOD'95*, LNCS, Springer-Verlag, Singapore, December 1995.

    (f) Fernandes, A.A.A., Paton, N.W., Williams, M.H., and Bowles, A. **Approaches to Deductive Object-Oriented Databases**. *Information and Software Technology 34(12)*, pp. 787–803 , December 1992.

    (g) Fernandes, A.A.A., Barja, M.L., Paton, N.W., and Williams, M.H. **A Deductive Object-Oriented Database for Data Intensive Application Development**. In M. Worboys and A.F. Grundy (eds.) *Advances in Databases: Proceedings of the 11th British National Conference on Databases (BNCOD 11)*, LNCS 696, pp. 176–198, Keele, England, UK, July 1993. Springer-Verlag. ISBN 3-540-56921, 0-387-56921-9.

    (h) Fernandes, A.A.A., Barja, M.L., Paton, N.W., Williams, M.H., Dinn, A., Diaz, I.P., and Abdelmoty, A.I. **ROCK & ROLL: A Deductive Object-Oriented Database System**. *Fifth International Workshop on the Deductive Approach to Information Systems and Databases*, pp. 25–32. Aiguablava, Spain, September 1994. Report de Recerca LSI/94-28-R, Universitat Politècnica de Catalunya.

    (i) Fernandes, A.A.A., Paton, N.W. and Williams, M.H. **A Logical Query Language for an Object-Oriented Data Model**. In N.W. Paton and M.H. Williams (eds.) *Rules in Database Systems: Proceedings of the 1st International Workshop*, Edinburgh, Scotland (30 August–1 September 1993), **Workshops in Computing** Series, pp. 234–250. Springer-Verlag, London, England, UK, 1994. ISBN 3-540-19846-6, 0-387-19846-6.

(j) Fernandes A.A.A., Williams, M.H., Paton, N.W., Barja, M.L., and Dinn, A. **Object-Oriented Database Programming Languages Founded on an Axiomatic Theory of Objects**. *Workshop on Logical Foundations of Object-Oriented Programming (8th European Conference on Object-Oriented Programming – ECOOP 94)*. Bologna, Italy, July 1994.

(k) Paton, N.W., Campin, Jack, Fernandes, A.A.A., and Williams, M.H. **Formal Specification of Active Database Functionality: A Survey**. 1995. To appear in *Proceedings of the 2nd International Workshop on Rules in Database Systems – RIDS'95*, LNCS, Springer-Verlag, Athens, September 1995.

(l) Williams, M.H, Paton, N.W., Abdelmoty, A., Fernandes, A.A.A. and Barja, M.L. **Applying Advanced Database Techniques to Geographic Information Systems**. In M.J. Cardwell et al. (eds.) JFIT Technical Conference Digest, pp. 221–228, Keele, England, UK, March 1993. DTI/SERC.

2. Submitted For Publication

(a) Fernandes, A.A.A., Barja, M.L, Paton, N.W. and Williams, M.H. **The Formal Specification of ROCK & ROLL: A Deductive Object-Oriented Database System**. 1995.

(b) Fernandes, A.A.A., Williams, M.H. and Paton, N.W. **A Comparative Formalization of Deductive Object-Oriented Databases**. 1994.

(c) Fernandes, A.A.A., Williams, M.H. and Paton, N.W. **Deductive Object-Oriented Databases as Horn Logic-Programs**. 1994.

(d) Fernandes, A.A.A., Williams, M.H. and Paton, N.W. **A Logic-Based Integration of Active and Deductive Databases**. 1994.

Para Regina.

Abstract

Deductive object-oriented databases (DOODs) have the potential to take the applicability and usability of database technology well beyond the current state-of-the-art, insofar as they unify two of the best known paradigms in the cutting edge of research on data management, viz., deductive databases (DDBs) and object-oriented databases (OODBs). If this potential is to be fully realized, the problem of providing DOODs with formal foundations cannot be avoided. In particular, a solution to the characterization problem of DOODs must allow the principled development not just of theoretical research but, as importantly, of robust, fully functional, industrial strength implemented systems as well. The great interest in this characterization problem demonstrates that there is widespread awareness of such needs.

This dissertation contributes a novel, *axiomatic* approach to the formal characterization of DOODs. The approach used is unique in employing a very rich axiom set to provide a logical account of most aspects of object-orientation, including those essential for behaviour-based modelling. This strategy allows a DOOD to be defined as an axiomatically presented OODB whose metalogic is predominantly shared with deductive relational databases (DRDBs).

This result is relevant in removing most of the need to develop a metalogic for DOODs from scratch, since that of DRDBs can be reused. This, in turn, has the desirable practical consequence that the techniques and methods used to implement DRDBs can be similarly reused to implement DOODs. Finally, DOODs characterized by the axiomatic approach are demonstrably flexible, and easily allow extensions.

The main contributions of this dissertation are: a formalization of an object-oriented data model as a species of mathematical structures, its reconstruction as an applied first-order predicate calculus, the conversion of this applied calculus into a class of logic programs without function symbols, and a logical analysis of the dynamic aspects of object-orientation based on a simple and intuitive extension to the standard inference engine of DDBs.

The dissertation also contains a brief overview of ROCK & ROLL, a fully operational DOOD system based on the axiomatic approach, thereby substantiating the claim that the approach allows the reuse for DOODs of implementation strategies developed for DRDBs. Finally, the flexibility conferred by the approach is exemplified by the strengthening of the data model with additional domain independent constraints, by the incorporation of a calculus of state change into the framework, and by the contribution of a logic-based model of active databases in which an independent inference engine for active rules is reconciled with the one used to drive deduction.

Introduction

This introduction constitutes a broadly sketched overview of all the research results reported in this dissertation. The introduction discusses the background, motivation and challenges for that research, and provides a preliminary overview of related work. The central problem which the research endeavoured to solve is then stated and the methods through which a solution was achieved are presented. The introduction closes with a brief summary of the contributions reported.

Relational databases (RDBs)[1] (Date, 1990; Kanellakis, 1990; Ullman, 1988) are the state-of-the-art in database management systems (DBMSs) in most business environments. However, a growing awareness of weaknesses in RDBs has led to research into, among others, two new classes of systems that specifically address these weaknesses: deductive relational databases (DRDBs) (Ceri et al., 1990b; Das, 1992; Thayse, 1989), which are deductive databases (DDBs) (Gallaire et al., 1984; Minker, 1988) over a relational model of data (RDM) (Codd, 1970), and object-oriented databases (OODBs) (Cattell, 1994; Gray et al., 1992; Hughes, 1991; Kim, 1995). As they have emerged from programmes initiated in the late 1970s, DRDBs and OODBs are prominent members of the cutting edge of research-oriented prototype databases.

However, sights are already set beyond DRDBs and OODBs. It has been suggested (Cattell, 1994; Ullman, 1988) that database systems must manage data, objects and knowledge. Taking a loose view of these categories, Ullman put forth the claim that RDBs only manage data, DRDBs do not manage objects, and OODBs do not manage knowledge. This makes it clear that a comprehensive solution is yet to be found.

This dissertation describes an approach to the characterization of a newer class of database systems, called deductive object-oriented databases (DOODs), which endeavour to extend OODBs with deductive capabilities. DOODs, therefore, are DDBs over an object-oriented model of data (OODM) and aim at a more comprehensive solution to perceived weaknesses of RDBs than can be achieved by DRDBs or OODBs alone. DOODs are designed to be tools for the management of data, objects, and knowledge.

The crucial motivation for research on DOODs is the perception that DDBs and OODBs complement one another's qualities by remedying one another's shortcomings. A DOOD system that merges DDBs and OODBs without sacrificing the separate qualities of each, and without introducing weaknesses not already found in either, would, on the one hand, take the applicability and usability of databases well beyond the current state-of-the-art and, on the other, unify two influential research programmes on data management since the 1970s.

The merging of DDBs and OODBs is widely perceived as a challenging research problem. For example, some researchers have suggested that the formalization of object-oriented concepts might require a higher-order logic (Beeri, 1990), while others (Bonner and Kifer, 1992; Kemper et al., 1992) have conjectured that a modal (e.g., dynamic) logic might be necessary. The research described in this dissertation demonstrates that both these conjectures are too pessimistic in the particular case of DOODs.

To a certain extent, the ongoing nature of the argument is a consequence of the lack of consensus on matters of definition, and, specifically, on what is considered to be the nature and scope of the concepts to be merged. While the property of being deductive is verifiable according to a

[1] Appendix E consists of a glossary of acronyms and symbols which the reader might find useful to refer to while reading this dissertation.

widespread consensus, the property of being object-oriented is still being defined in an ad-hoc fashion. Correspondingly, while DDBs are well understood formal constructs under a generally accepted characterization (Ceri et al., 1990b; Das, 1992; Gallaire et al., 1984), proposals to define OODBs formally (Beeri, 1990; Geppert et al., 1993; Scholl et al., 1992; Lécluse et al., 1988; Meseguer and Qian, 1993; Straube, 1990; Su et al., 1993) or informally (Atkinson et al., 1990; Cattell, 1994; Fong et al., 1991; Soley and Kent, 1995; Zdonik and Maier, 1990a) are still mounting. This lack of a widely agreed upon characterization of OODBs is a major hurdle for DOOD research, since without it the bounds to the task of formalizing DOODs cannot be incontrovertibly fixed, and, as a consequence, much argument over adequacy and scope is fostered.

The large number of possible characterizations of DOODs that results from this state of affairs is also partly explained by the multiple viewpoints from which the problem of providing logical foundations for object-oriented features, concepts and functionalities can be approached. It is possible to classify proposals by their stand on the trade-off between expressiveness and tractability that every logical system carries within it, as well as by where and how a particular proposal intervenes in the formal system of departure to introduce its logical account of object-orientedness.

Several formal systems have been deployed in attempts to provide formal foundations for object-orientedness. With respect to the expressiveness vs. tractability trade-off, the research scenario ranges from proposals that deploy very expressive formal systems (typically, versions of process calculi, which emphasize concurrency, or versions of λ-calculi, which emphasize type related aspects) to those that extend existing, and often very constrained, calculi. Some of the formal systems deployed (e.g., dynamic logic (Kozen and Tiuryn, 1990)) place themselves (explicitly or not) in relation to classical first-order logic (FOL), while some do not. With very expressive systems, a computational version of the logical system is typically not of concern: these tend to be purely paper proposals. Proposals (e.g., ADAM (Paton, 1989), L&O (McCabe, 1992)) also exist for restricted logical systems (of a clausal kind, in most cases), principally because a computational version of the logic is typically the ultimate target: these tend to adopt a programming language stance, with expressiveness being more valued than the decidability of evaluation procedures. Finally, proposals (e.g., F-Logic (Kifer et al., 1993), IQL+ (Abiteboul, 1990), Logres (Cacace et al., 1990; Cacace et al., 1993)) exist of more constrained logical systems (again, clausal in kind in all cases): these tend to adopt a database language stance, since the question of decidability takes priority over expressiveness.

The proposal described in this dissertation belongs to the latter class. Within the bounds of this class, one can define a DOOD as an OODB presented as a deductive database theory; or, equivalently, as a DDB over an OODM. This deliberately loose definition reflects the numerous possibilities (e.g., (Abiteboul, 1990; Bertino and Montesi, 1992; Cacace et al., 1990; Jarke et al., 1993; Kifer et al., 1993), among others) that have been explored in presenting (some notion of) OODBs as formal axiomatic systems with metalogical properties comparable to those exhibited by the classical presentation of RDBs as formal axiomatic systems of Horn clausal logic (HCL) without function symbols,[2] i.e., as a subclass of DRDBs.

With respect to the intervention to introduce a logic-based account of object-orientation, recall that a formal axiomatic system has four component parts: an alphabet, a language, a set of axioms, and a set of inferences rules. Four questions have to be addressed:

1. In which component(s) of the formal axiomatic system does one intervene in order to capture the features, concepts and functionalities provided by OODBs?

2. How, in detail, are these interventions to be carried out?

[2]The qualifier phrases *without function symbols* and *function-free* are equivalently used in this dissertation in place of the more precise, but more cumbersome, phrase *without non-nullary function symbols*. The former usage requires nullary function symbols to be referred to as *constants*, and the phrase *function symbols*, or *function*, to imply arity greater than zero. The remainder of this dissertation abides by these requirements.

3. What is the extent and depth (with respect to their proof-theoretical consequences) of these interventions?

4. What is the import of these interventions for both theoreticians and practitioners in terms of the resulting DOODs?

Several degrees and forms of intervention in one or more components have been proposed. Typically, the points of intervention have tended to be the alphabet and the language, with consequent repercussions in the set of inference rules. These repercussions have tended to be negative in several respects, insofar as they are prone to jeopardize decidability or increase the complexity of proof procedures, and, more generally, to make the metalogic significantly more involved than that of DRDBs.

The approach to the problem of characterizing DOODs used in this dissertation is unique in employing a very rich axiom set to provide a logical account of most aspects of object-orientation, thereby allowing the alphabet, language, and inference rules to be kept very close to those used in DRDBs. In this approach, a DOOD is an axiomatically presented OODB whose metalogic is predominantly shared with DRDBs.

The central question which this dissertation seeks to answer is whether this class of DOODs is nonempty. In more detail, whether there exists a nonempty class of DOODs, identified by the set of proper axioms that characterize the modelling relationships expressible within the system, such that each element in that class is a function-free theory of HCL. In short, whether DOODs can be seen as deductive database theories (DDTs).

The relevance of a positive answer to this question can be argued as follows. Since, in this approach, the axiom set of the formal system bears the brunt of intervention, and since the axiom system is largely neutral in determining the decidability and tractability of proof procedures, the most important corollary of the above result is that the metalogic of DRDBs (Ceri et al., 1990b; Das, 1992) is reusable for DOODs with minimal modifications. In other words, the need to develop a metalogic for DOODs from scratch is, to a large extent, removed.

A corollary, in turn, of this result is that the corpus of methods and techniques (Ceri et al., 1990b; Das, 1992) developed for optimizing and evaluating queries to DRDBs is reusable for DOODs, with minimal modifications. In other words, the need to develop from scratch the practice of implementing DOODs is, to a large extent, removed.

Finally, different sets of proper axioms characterize different classes of DOODs. This flexibility in characterization is a major advantage of this approach: if increasing the semantic content of databases only requires enriching the set of proper axioms, then no modification of the inference system (or engine) is implied. This approach to DOODs makes them easily extensible in important ways.

The definition of DOODs as axiomatically presented OODBs whose logical system inherit results from that of DRDBs highlights the most important contributions of this dissertation: the axiomatic definition of an OODM that ultimately allows the OODBs it generates to be presented as axiom sets within function-free HCL, and makes possible the above mentioned sharing of metalogical results. Another contribution is a logical account of the dynamic features of object-orientation such as encapsulation, overloading, overriding, and late binding. In the process of demonstrating the extensibility of DOODs characterized by the axiomatic approach, this dissertation contains two other contributions: the first is to show how axiomatically presented DOODs can be embedded into a logical calculus of state change; the second is to show how a DDB (and hence a DOOD) inference engine can be reconciled with a distinct inference engine that implements an operational semantics for active rules operating over the same database over which deductive rules operate.

The class of DOODs characterized in this dissertation is of particular interest because not only is integrated management provided for data, objects and knowledge, but this is done in specialist fashion, i.e., in this class of DOODs, data is managed as in RDBs, objects are managed as in OODBs, and knowledge is managed as in DDBs.

The remainder of this introduction comprises three parts.

Research Context: In the first part, the context in which the research reported in this dissertation was carried out is presented through a detailed discussion of the background, the motivation, and the challenges faced in carrying out that research. A preliminary overview of related work is also given. Note, however, that a detailed analysis, discussion and comparison of alternative approaches is to follow, rather than precede, the central chapters of the dissertation.

Technical Development: In the second part, focus falls on the technical development of the research. The central question addressed in this dissertation is stated in greater detail, and the methods used to solve it are outlined.

Contributions: Finally, in the third part, the contributions of the dissertation are highlighted.

Taken together, the three parts of the introduction are meant to convey a sense of where the main contributions of the dissertation lie, and of how they articulate and cohere.

Research Context

Background

To place this dissertation in the proper context, an overview is given of the main lines along which research on data management has developed since the 1970s. This, in turn, helps to highlight this dissertation's main motivation, the challenges it addresses, as well as work that is, in varying degrees, related to it.

The original proposal of the RDM in 1970 (Codd, 1970) is the most prominent landmark in the field of database research. Throughout the 1970s and 1980s, the development of the theory of RDBs produced a large corpus of theoretical results (Kanellakis, 1990), without which the principled approach to the design, development and management of database applications (Date, 1990; Ullman, 1988) would not have been possible. Helped by their theoretical underpinning, RDBs also introduced declarative languages for querying databases, thereby empowering casual users with ad-hoc querying capabilities that use tractable algorithms and do not compromise correctness or efficiency.

By the mid-1980s, RDBs had successfully completed their transformation into marketable products. From the point of view of database research, an enduring legacy of the RDM is the notion that databases can be characterized as mathematical constructs, and hence, formally studied. Unfortunately, the conciseness and elegance of the RDM detract from its adequacy for many novel application areas (Cattell, 1994; Graham, 1994).

By the late 1980s, there was general awareness that RDBs are not fully adequate for many new classes of applications. The majority of these had emerged as a result of changes in the business environment that was prevalent at the time of the original RDM proposal. As a consequence, the 1980s saw the emergence of several new approaches addressing perceived weaknesses of RDBs.[3] The qualities and shortcomings (marked (+) and (−), respectively) of RDBs are listed in Table 0.1.

In the context of this dissertation, these weaknesses are of two kinds.

1. The means for expressing implicit information, recursion, and integrity constraints in the sub-languages of the RDM are poor or nonexistent.

[3] In the ensuing discussion, what is meant by RDB is an ideal instance of the RDM originally defined in (Codd, 1970). Existing relational DBMSs fall short in some respects of that ideal, but also improve on it in others.

1	**formal basis**	(+) RDBs are well defined mathematical constructs within a well understood logical system, viz., FOL.
2	**user empowering**	(+) RDBs can be declared, manipulated and queried using declarative languages whose expressions are optimizable and for which there exist tractable query evaluation algorithms.
3	**implicit information**	(−) RDBs have limited means of expressing more complex kinds of implicit information (e.g., derived data) about application domains.
4	**integrity constraints**	(−) RDBs have limited means of expressing more complex kinds of general information (e.g., business rules) about application domains.
5	**recursion**	(−) RDBs do not model structure or behaviour recursively.
6	**identity**	(−) RDBs do not model the detachment of existence from state, i.e., RDBs are inherently value-based, as opposed to identity-based.
7	**complex structures**	(−) RDBs do not model properties that have nonatomic values.
8	**semantic orientation**	(−) RDBs do not, in the typical case, model relationships between types such as inclusion, composition, etc.
9	**behaviour orientation**	(−) RDBs have no means of assigning specific behaviour to particular types.

Table 0.1: Qualities and Shortcomings of Relational Databases

2. The modelling constructs of the RDM are either not powerful or not natural enough to capture important kinds of semantic information about certain application domains.

As a consequence, it falls on application programs to remedy these shortcomings. Since databases emerged as a powerful tool on the strength of their ability to represent and serve information efficiently, reliably and consistently with maximal independence from application programs, this transfer of responsibility to application programs is clearly unsatisfactory.

The two kinds of shortcomings of RDBs identified above have motivated two of the most prominent developments in recent data modelling research, viz., DRDBs and OODBs. Among the most important weaknesses of RDBs, DRDBs specifically address the difficulty of expressing logical consequences of the information asserted in the database, while OODBs more frontally address the conceptual dearth of the RDM and the lack of means in the latter to integrate knowledge about behaviour with knowledge about structure.

Therefore, as numbered in Table 0.1, DRDBs remedy shortcomings 3, 4 and 5 of RDBs by adding to the relational sub-languages a deductive rule language that is capable of uniformly expressing derived data, integrity constraints and recursive queries. By leaving the underlying data model of RDBs untouched, DRDBs inherit qualities 1 and 2 of RDBs, but also their shortcomings 6, 7, and 8. Some nested relational approaches (Chen and Chu, 1990; Chen and Gardarin, 1988) do address, one or more of shortcomings 6, 7 and 8, but the resulting databases are often more involved than one would expect and quality 2 of RDBs, which DRDBs preserve well, is often compromised. In any case, both nested and deductive extensions to RDBs are exclusively structural, i.e., shortcoming 9 is yet to be addressed from a relational viewpoint in adequate depth.

OODBs, on the other hand, are distinguished for having shifted the modelling viewpoint from structure to behaviour. Therefore, OODBs have 9 as their characteristic quality. Via their behavioural orientation, OODBs can also address shortcoming 3 to a certain extent, albeit not in a declarative or in as uniform a way as can DDBs. Through their roots in programming languages, OODBs typically also have 6, 7, and 8 as qualities. However, in comparison with RDBs, OODBs not only come short of a declarative, integrated solution to 3, 4 and 5, but, more importantly, their lack of 1 and 2 is a serious shortcoming. Broadly speaking, the extent to which 1 to 5 come out as qualities or shortcomings depends on the logical system, if any, that underpins the database. Both RDBs and DRDBs are underpinned by a logical system in which 1 to 5 typically come out as qualities. Since existing OODBs do not have foundations in FOL, they typically lack any means for declarative definition and manipulation of data, and, hence, 1 to 5 typically come out as shortcomings. It should be pointed out that some OODBs do provide support for declarative querying. However, they tend to be founded on object algebras, as opposed to calculi, thereby making more complex any attempt to integrate querying, integrity maintenance and data derivation in a single homogeneous framework.

In conclusion, neither DRDBs alone nor OODBs alone provide a solution for all the perceived weaknesses of RDBs. Against this background, it is possible to motivate the idea of DOODs as a class of databases that attempts to combine the complementary qualities of DRDBs and OODBs.

Motivation

Research on DRDBs and OODBs can be traced back to the late 1970s. In their own terms, both programmes have been very successful. DBMSs for OODBs are now penetrating the marketplace (Cattell, 1994; Gray et al., 1992; Hughes, 1991; Kim, 1995), with advanced prototype DBMS for DRDBs (Ceri et al., 1990a; Derr et al., 1994; Naqvi and Tsur, 1989; Ramakrishnan and Ullman, 1993; Ramakrishnan et al., 1994; Vaghani et al., 1994) following closely. However, as it became increasingly clear that moving from RDBs to either DRDBs or OODBs still left important unresolved problems, several researchers were motivated to pursue an integration of the deductive and object-oriented approaches.

With reference to the qualities and shortcomings of RDBs in Table 0.1, notice that, in general, DRDBs excel where OODBs are weak, viz., 1, 2, 3, 4 and 5. Conversely, OODBs excel where DRDBs are weak, viz., 6, 7, 8, and 9. This circumstance of DRDBs and OODBs being complementary to one another is what makes the idea of developing a DDB over an OODM, to give rise to DOODs, an appealing one. Integrating the deductive and object-oriented paradigms is now widely acknowledged to be one way of approximating a comprehensive solution to the perceived shortcomings of current generation prototype database systems. DOODs are prominent members of the class of next generation databases. The challenges involved in providing a formal characterization for them are now discussed.

Challenges

This dissertation is concerned with the characterization problem for DOODs. The targeted solution is a formalization of a deductive object-oriented approach to the representation of structural and behavioural aspects of application domains in database systems. Characterizing DOODs has been seen to be a useful endeavour. That it is also a challenging one can be argued as follows.

To begin a more detailed inspection of the problem, it is instructive to look back to the corresponding problem in a relational setting. Consider Figure 0.1.

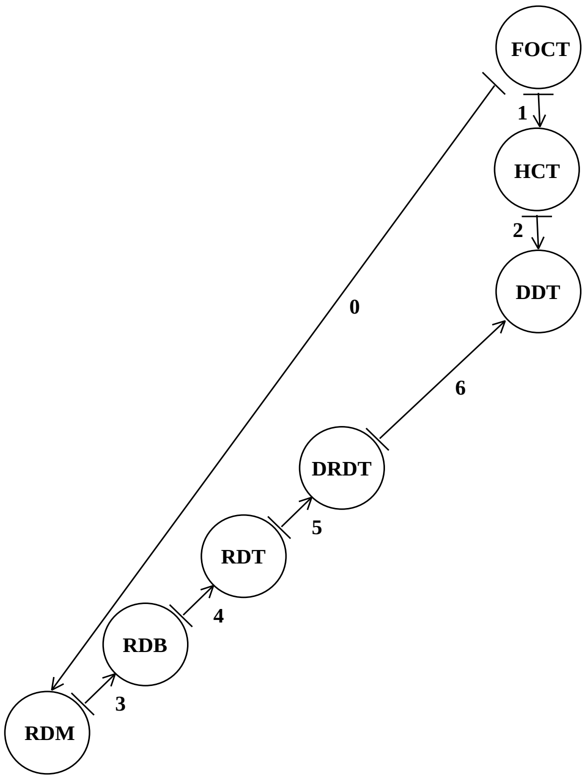


Figure 0.1: A Space of Theories: The Relational Case

Figure 0.1 depicts a space of theories. A directed edge $\Theta \mapsto_n \Theta'$ denotes a mapping n from the class of theories Θ into the class of theories Θ'. Note that the class of DDTs is the only sink, and the class of first-order clausal theories (FOCTs) the only source.[4] This choice of sink is somewhat arbitrary (e.g, from the point of view of implemented systems, DRDTs (deductive relational database theories) might be considered a more obvious choice for a sink node). The

[4]Insofar as any formula in the standard language of FOL is logically equivalent to a set of clauses, in this dissertation this presentational choice is glossed over.

reason why DDTs are chosen as the sink of the space is because they are the most constrained logical system in the space not embodying any assumptions about data modelling paradigms.

There are two paths from FOCTs to DDTs in the space. The first has Horn clausal theories (HCTs) as the only intermediate node and characterizes DDTs as the most restricted class of logic programs in the space; the second has as intermediate nodes the classes of theories: RDM, RDB, RDT (relational database theories), and DRDTs,[5] thereby characterizing DRDTs as extensions of RDTs and as a subclass of DDTs. The second path, therefore, shows how modelling assumptions can be successively mapped onto ever more general logical systems, each of which is nonetheless sufficiently constrained as not to jeopardize the decidability and tractability of an automated theorem proving approach.

Recall that any class of logical theories is determined by its language, its set of (proper) axioms and its set of rules of inference. Table 0.2 informally describes the salient features of the above components for all classes of theories depicted in Figure 0.1. The following shorthands are used in this and similar tables: ff for 'function-free', g for 'ground' and u for 'unit'. These can also appear negated: $\neg ff$ for 'non-function-free', $\neg g$ for 'nonground' and $\neg u$ for 'nonunit'.

The denotations of the symbols used in Table 0.2 are the following. RES denotes the resolution principle (see, e.g., (Chang and Lee, 1973)). RAx denotes the axioms that formalize a set of representational assumptions, denoted by RA, that every database conforms to, irrespective of its underlying conceptual basis. As defined in (Gallaire et al., 1984), RAx contains axioms for the equality predicate, as well as the closed world, domain closure and unique name axioms. $RMAx$ denotes the set of axioms that formalize relational modelling assumptions (RMA), such as uniqueness of keys and referential integrity (Date, 1990). $DedAx$ denotes the set of axioms, in the form of deductive rules, used to express domain dependent integrity constraints, derived data, etc. SAx denotes the set of ground axioms, in the form of unit clauses, i.e., facts, used to express a database state S.

		Axiom System		*Inference System*	
Class	*Language*	*Explicit*	*Implicit*	*Explicit*	*Implicit*
FOCT	clauses	$\emptyset$	$\emptyset$	{RES}	$\emptyset$
HCT	*Horn* clauses	$\emptyset$	$\emptyset$	{RES}	$\emptyset$
DDT	ff-Horn clauses	$\emptyset$	RAx	{RES}	$\emptyset$
DRDT	ff-Horn clauses	$DedAx \cup SAx$	$RMAx \cup RAx$	{RES}	$\emptyset$
RDT	ff-g-u clauses	SAx	$RMAx \cup RAx$	{RES}	$\emptyset$
RDB	ff-g-u clauses	$\emptyset$	$\emptyset$	{RES}	S,RMA,RA
RDM	ff-g-u clauses	$\emptyset$	$\emptyset$	{RES}	RMA,RA

Table 0.2: Features of Theories in the Relational Space

In Table 0.2 the distinctions made between the explicit and implicit parts of both the axiom and inference systems can be informally explained as follows. The explicit part of the axiom system is the set of axioms that are explicitly used in deductions, while the implicit part spells out what is presumed of the presentation of the theory (e.g., that it asserts all truths that there are to be asserted, thus satisfying the implicit closed world axioms). Analogously, the explicit part of the inference system is the set of inference rules that are explicitly used in deductions, while the implicit part spells out what is presumed in the use of the theory (e.g., that an application program does not set the same value for the key in two different tuples in the same relation). Note that it is the formalization of assumptions as axioms that marks the change from a nonlogical to a logical (and, more specifically, proof-theoretic) view of databases.

[5] DDTs and DRDTs are equivalent to DDBs and DRDBs, in the following sense. The former are proof-theoretical constructs; the latter, model-theoretical ones. However, given the equivalence of the proof-theoretical and the model-theoretical view of databases (Gallaire et al., 1984), any difference between them is purely one of presentation.

Table 0.2 can be used to understand what each of the mappings in Figure 0.1 involves. Table 0.3 informally describes each of those mappings in terms of what the target of the mapping includes and excludes from each component part of the source. If inclusion is in the implicit part it is prefixed by $\triangleright$; if in the explicit part, by $\triangleright\triangleright$.

Mapping			*Includes*	*Excludes*
FOCT	$\mapsto_1$	HCT	—	non-Horn clauses
HCT	$\mapsto_2$	DDT	$\triangleright RAx$	$\neg ff$ clauses
FOCT	$\mapsto_0$	RDM	$\triangleright$ RA, $\triangleright$ RMA	$\neg g$-$\neg u$ clauses
RDM	$\mapsto_3$	RDB	$\triangleright$ S	—
RDB	$\mapsto_4$	RDT	$\triangleright\triangleright\, SAx$, $\triangleright\, RAx$, $\triangleright\, RMAx$	S, RA and RMA
RDT	$\mapsto_5$	DRDT	$\neg g$-$\neg u$ clauses, $\triangleright\triangleright\, DedAx$	—
DRDT	$\mapsto_6$	DDT	—	SAx, $DedAx$, $RMAx$

Table 0.3: Mappings between Theories in the Relational Space

Together, Tables 0.2 and 0.3 constitute a synopsis of the essential aspects of the deductive database research programme in the relational case. With the benefit of hindsight, one might feel tempted to consider it a straightforward task. That this is not the case is most vividly shown by the fact that the mappings in Table 0.3 bridge more than fifteen years of research activity. For each mapping in Table 0.3, a reference to literature where the mapping and (at least) its target class are studied in detail can be given as follows: FOCT $\mapsto_1$ HCT (Apt, 1990; Das, 1992; Kowalski, 1979b; Lloyd, 1987), HCT $\mapsto_2$ DDT (Gallaire and Minker, 1978; Gallaire et al., 1984; Grant and Minker, 1989; Minker, 1988), FOCT $\mapsto_0$ RDM (Codd, 1970; Hulin et al., 1989; Ullman, 1988), RDM $\mapsto_3$ RDB (Codd, 1970; Hulin et al., 1989; Ullman, 1988), RDB $\mapsto_4$ RDT (Ceri et al., 1990b; Gallaire et al., 1984; Reiter, 1984), RDT $\mapsto_5$ DRDT (Ceri et al., 1990b; Das, 1992), DRDT $\mapsto_6$ DDT (Gallaire et al., 1984; Grant and Minker, 1989).

Most approaches to the characterization of DOODs assume the space in Figure 0.1 as a background. A detailed discussion of how influential lines of research position themselves within this space can be found in (Fernandes et al., 1992b). However, the approach contributed in this dissertation is unique in constructing a homeomorphic space[6] for the object-oriented case. All other approaches extend the space of Figure 0.1 into one containing (in varying degrees) object-oriented nodes. No other approach contributes the comprehensive solution of ensuring that the corresponding object-oriented space is homeomorphic to that of Figure 0.1.

The problem of characterizing DOODs can, therefore, be understood as the problem of defining a space of theories similar (and, in the limit, homeomorphic) to that of Figure 0.1, in which an OODB node lies in at least one path to the DDT sink. Under this formulation, the challenges involved in characterizing DOODs can be described, as follows:

1. While one can speak of *the* RDM, meaning the formulation whose root is (Codd, 1970), one can do no better than speak of *an* OODM, since no single proposal (Cattell, 1994; Fong et al., 1991; Geppert et al., 1993; Scholl et al., 1992; Soley and Kent, 1995) has gained widespread acceptance yet. Some have only been defined informally, others (e.g., ODE (Agrawal and Gehani, 1989), O_2 (O. Deux et al., 1991), GemStone (Maier and Stein, 1987), among others) are, in addition, too intimately dependent on their underlying programming language or on their realization within actual DBMSs, or both. The first task, then, is to identify a suitable OODM whose induced OODBs can be taken as the required node in the path to the sink. This turns out to be quite challenging because there is no obvious suitable candidate.

[6] Two spaces are said to be *homeomorphic* if they are topologically equivalent. Informally, two spaces are topologically equivalent if (assuming, as in the present case, the spaces to be graphs) either one can be reshaped into the other without breaking up any edge and without collapsing any two distinct nodes into one. Notice that if one graph is the mirror image of another, then they are trivially homeomorphic. This is the case in this dissertation.

2. The difficulty lies in the fact that the choice of OODM is not independent of a choice for the source, which, in the relational case, is also the most general class of theories in the space, viz., FOCTs. The second task, then, is to identify a suitable system of logic to be the source of the space. This also turns out to be less than straightforward, since there is no obvious suitable candidate for that either. Although the suitability of FOL has been called into question by some (arguing either that a second-order logic is required (Beeri, 1990), or that a modal logic is (Bonner and Kifer, 1992; Kemper et al., 1992)), most work on DOODs does take FOL for granted, more or less explicitly and more or less directly.

   In the relational case, the relationship between the RDM and FOL was implicit from the start (Codd, 1970) and was, in time, made clearer and more explicit (Reiter, 1984). The adequacy of deploying them in the characterization problem of DRDBs was never the subject of any well founded doubt. In the object-oriented case, choices are much more open to questioning. However, one can immediately conclude that, since a formal relationship must be described between the chosen OODM and the source of the space, informally defined OODMs are ruled out, as are OODMs that have no proof-theoretical characterization, or that, seen as systems of logic, are intractable or undecidable.

3. Finally, the third task is to make explicit all the mappings in the space, given the source, the sink and the required node in the path from the former to the latter. Whether this is a challenging task clearly depends both on the choice of source and on the approach used to formalize object-oriented concepts.

The preliminary discussion of related work that now follows attempts to show the choices made in alternative proposals to characterize DOODs, thereby motivating the ensuing discussion of the technical development and of the contributions of this dissertation.

Related Work

Research on the interface between logic and object-orientedness has been both broad and deep. Seen in its entirety, this large body of work is somewhat amorphous and difficult to systematize. Very different strategies have been applied, reflecting many subtle differences in what one considers to be the nature and scope of the problem. As a consequence, it is very hard to identify precisely which proposals can be fruitfully compared with one another. In order to delineate boundaries within which comparisons with the research reported in this dissertation become possible, consider how the merging of the deductive and the object-oriented paradigms can be attempted.

First, one can enrich an object-oriented system with a notion of formal proof, or, conversely, one can enrich a logical system with object-oriented concepts. The research reported in this dissertation follows the latter approach, i.e., the target formalism is a logical system that has been enriched with a logic-based account of object-oriented concepts.

To delimit further the scope of comparison, consider how one can informally categorize logical systems by their leanings on the trade-off between expressive power and the decidability and tractability of their proof procedures.

A logical system can be said to be *intractable* if, due to its very high expressiveness, its proof procedures have nonpolynomial time complexity. FOL is an intractable logical system.

A logical system can be said to be *acceptably efficient* if some expressiveness is relinquished in order to ensure the existence of a proof procedure that is efficient in comparison with those operating over unrestricted clauses, even if strictly intractable. HCL is an acceptably efficient logical system (as its embodiment in many Prolog implementations shows). HCL becomes acceptably efficient when FOL is not because, by relinquishing expressiveness, it opens the way for refinements that increase the efficiency of the proof procedure. HCL profits from more efficient resolution rules (e.g., linear resolution with selection function) and more efficient search procedures (e.g., based on depth-first construction of the proof tree). For an additional discussion of why theorem proving in HCL (as embodied in Prolog) is efficient when compared with the efficience of typical reasoning programs operating over unrestricted clauses the reader is referred to [(Wos et al., 1984), Ch. 14]. One particular feature that makes HCL not as expressive as unrestricted clauses is that Horn clauses have at most one positive literal, and hence HCL cannot express disjunctive conclusions. This makes HCL strictly less expressive than FOL. Interested readers may wish to refer to several promising attempts to extend Prolog (the best known embodiment of an HCL theorem prover) to operate over unrestricted clauses (Loveland, 1991; Plaisted, 1988; Stickel, 1988).

A logical system can be said to be *tractable* if enough expressiveness is relinquished to guarantee that it has a proof procedure with polynomial time complexity. Datalog is a tractable logical system. (Datalog queries are strictly contained in the class of fixpoint queries, and these can be evaluated in polynomial time. See [(Abiteboul et al., 1995), pp. 415–484] for a comprehensive discussion of the complexity of query evaluation algorithms for several classes of queries, including Datalog queries.)

Table 0.4 shows these categories, their stance on the trade-off used to generate the categories, and prototypical examples for each category.

Category	*Expressiveness*	*Decidability*	*Tractability*	*Example*
intractable	very high	semidecidable	intractable	FOL
acceptably efficient	high	semidecidable	intractable	HCL
tractable	low	decidable	tractable	Datalog

Table 0.4: Logical Systems: Expressiveness, Decidability and Tractability

The categorization above allows the boundaries of related work to be made more precise, as follows. Lines of research that model objects using intractable logical systems typically deploy process calculi or λ-calculi (sometimes both) and are far removed from mainstream research on DOODs. This is due to a concentration on issues of concurrency (in the former case) and on type systems (in the latter). Although such approaches can be effectively used to provide formal foundations for concrete systems, their stance in favour of expressiveness, against decidability and tractability, makes it very hard to develop them towards the kind of automated theorem proving approach to computation that is the essence of DDBs. The research reported in this dissertation does not compare with these proposals, because its target formalism is a logical system equipped with an automated proof procedure.

Proposals, and in this case even implementations, also exist that model objects using acceptably efficient logical systems, typically Prolog. ADAM (Paton, 1989) and L&O (McCabe, 1992) are well known examples. Such proposals tend to tackle the problem of merging paradigms exclusively from a programming language viewpoint (though ADAM is a notable exception in having a database slant). The targeted users are presumed to be highly skilled, thus allowing expressiveness to have priority over decidability and tractability. Even more so if, as is often the case, it is assumed that ad-hoc use is rare, and, hence, that the need for effective, generally applicable optimization and evaluation methods is low. The research reported in this dissertation does not compare with these proposals, because it targets a database system, and therefore priority is not given to expressiveness at the expense of decidability and tractability.

Proposals that model objects using tractable logics naturally tend to adopt a database language viewpoint. The presumptions in this case are that the targeted users may often be comparatively naive and ad-hoc use may be frequent, and, hence, that the need for effective, generally applicable optimization and evaluation methods is high. In these proposals, decidability and tractability have priority over expressiveness. Methods for automated theorem proving over a database, seen as a logical theory, lie at the heart of these proposals. In the context of this dissertation, only these can be considered to be proposals to characterize DOODs and, therefore, to be comparable with the work reported here.

DOODs, in the tighter definition that emerges from the preceding paragraphs, have been the focus of an intense interest both from a theoretical and a practical viewpoint. Three editions of an international conference (Ceri et al., 1993; Delobel et al., 1991; Kim et al., 1990) and a major workshop (Mumick, 1993) dedicated to DOODs have taken place in recent years, although a large part of the research presented in them is either deductive *or* object-oriented, rather than truly deductive *and* object-oriented. Nevertheless, the body of work on DOODs is quickly becoming voluminous.

One way to systematize this body of work is to note that, in one way or another, every proposal must make choices on where and how to model object-oriented concepts logically. Recall that DOODs, as defined above, are logical theories, and these are determined by their language, axioms and rules of inference. If it is possible to elicit these component parts in two alternative characterizations of DOODs, then it becomes possible to compare them with one another, as well as to any arbitrarily chosen background logical system, provided that its component parts are known. Such a background logical system could be a paradigmatic one, e.g., Datalog. This style of comparison can identify whether, and to what extent, one particular formalization of DOODs differs from the paradigmatic logical system as a consequence of modelling objects.

This not only provides terms of comparison, it also offers an insight into the relative merit of alternative proposals. Note that if one is able to elicit the component parts of two proposals for a class of logical theories, one should thereby be able to relate, and then contrast, the proof and model theories of one proposal with those of the other. Now, the proof and model theories are exactly what is needed to assess how the stated objectives of the characterization task are met. For DOODs, as previously argued, the most important objectives are decidability and tractability. In other words, proposals should aim at tractable proof procedures, and at as well understood and as easy to understand a metalogic as possible. When comparing two proposals, one is striving to gauge, as each concept is captured, what price is paid in the proof and model theories, since this determines the consequences for decidability and tractability.

Consider Datalog (Ceri et al., 1990b; Das, 1992; Ullman, 1988), the logical system underlying DRDBs, to be the paradigmatic logic. It is equipped with a tractable proof procedure, its model theory is straightforward, and its metalogic well developed and simple. All these come as a consequence of its being fully characterized by an extremely streamlined language over an unencumbered domain and a single, exceedingly lean, inference rule.

Datalog does not encompass object-oriented notions, insofar as it does not model:

1. a notion of identity that is detached from state,
2. a notion of composite structures,
3. a view of entities in the application domain as possessing structure and behaviour,
4. inheritance of structure and behaviour based on explicit taxonomies,
5. encapsulation, overloading, overriding and late binding.

Therefore, to model objects from a logical point of view taking Datalog as the logic of departure one must intervene in one or more of its language, axiom set and inference rules in order to model all of the above.

This last observation leads to a method for classifying and comparing DOOD proposals on the basis of where and how they intervene on the components of a given background logic. If one takes Datalog as the background, three influential approaches can be discerned in the literature:

restart — This approach is typical of Stony Brook, USA (Chen and Warren, 1989; Kifer and Wu, 1989b; Kifer et al., 1993). Its most salient features are a consequence of the claim that the object-oriented paradigm is distinct enough from the relation-based tradition that only the development of a new logical system will suffice to meet the objectives. It proposes to *restart* developing the characterization of DOODs from scratch. In comparison to those of Datalog, the language and rules of inference are subject to a very high level of intervention. As a consequence, the proof and model theories of the resulting DOODs are, in comparison to DRDBs, much more complex to understand and to implement efficiently. The metalogic of DOODs characterized in this approach is no more than incipient, as a result of the option to restart afresh.

The restart approach makes it very difficult to understand its solution to the characterization problem in terms of a space of theories analogous to that of Figure 0.1. This is because this approach underplays the need for an independent definition of OODBs. This means that it does not address the question of which OODBs the DOODs it characterizes are supposed to extend and how these extensions can be described.

In terms of Figure 0.1, the FOCT node is the sink for all of C-Logic (Chen and Warren, 1989), O-Logic (Kifer and Wu, 1989b), F-Logic (Kifer et al., 1993), which are sources. The lack of an independent definition of an OODM and the OODBs it induces means that no corresponding nodes exist in the theory space. Moreover, the restart approach is graphically represented by the fact that there are no intermediate nodes between the source and the sink. This bare space, made of a single edge from source to sink, means that metalogical results can only be shared with FOCTs. However, since FOCTs are intractable, many of those results are unusable in a DDB context.

mapping — This approach includes (Abiteboul et al., 1993; Bertino and Montesi, 1992; Dalal and Gangopadhyay, 1990; Jamil and Lakshmanan, 1993; Zaniolo, 1990). The unifying trait of these proposals is that they *map* one user-level, surface language incorporating object-oriented concepts into an internal, target language whose metalogic is known (e.g., Prolog for (Dalal and Gangopadhyay, 1990), $\mathcal{LDL}$ for (Bertino and Montesi, 1992; Jamil and Lakshmanan, 1993; Zaniolo, 1990)), or Datalog for (Abiteboul et al., 1993). In comparison to Datalog (when it is not itself the target), the surface language is subject to a high level of intervention, and inferencing takes place in the target language to which the surface language is mapped. In the typical case, a more complex logic than Datalog is deployed. As a consequence, the proof and model theories of the resulting DOODs are, in comparison to DRDBs, more complex to understand, though not necessarily more complex to implement efficiently, insofar as the mapping can lead to an underlying logic that is already efficiently implemented. The metalogic of DOODs characterized in this approach is that of the system being mapped into, which may be more or less close to that of DRDBs depending on the target language being mapped into. Opportunities for direct reuse of results also depend on that.

Like the restart approach, the mapping approach to the characterization problem also does not provide an independent definition of OODBs. It becomes similarly difficult to ascertain precisely how the characterized DOODs purport to extend OODBs. In terms of Figure 0.1, the source is the FOCT node for all of (Abiteboul et al., 1993; Bertino and Montesi, 1992; Dalal and Gangopadhyay, 1990; Jamil and Lakshmanan, 1993; Zaniolo, 1990), but the final targets differ. In contrast with the restart approach, there is at least one intermediate node between a source and the sink. Since this intermediate node is the target of the map that characterizes the approach, a judicious choice of target can offer opportunities for sharing useful metalogical results. The research question of interest then is how powerful a target is needed to model the expression of object-oriented concepts in the surface language.

extension — This approach is typical of INRIA, France (Abiteboul, 1990; Abiteboul and Grumbach, 1990; Abiteboul and Kanellakis, 1989a), but it has been extensively used elsewhere. It is based on the claim that it is possible to *extend* the relation-based tradition (as expressed in Datalog) to incorporate concepts of the object-oriented paradigm. This results in gradual extensions of Datalog that, taken together, provide a logical view of objects. In comparison to Datalog, the language is subject to a high level of intervention, but the single inference rule of Datalog is only affected in its unification part (e.g., to accommodate set-valued variables). As a consequence, the proof and model theories of the resulting DOODs are, in comparison to *restart* approaches, not as complex to understand and to implement efficiently. The metalogic of DOODs characterized in this approach can be based on that of DRDBs, but there is little opportunity for direct sharing of results if the language of the logical system strays too far from that of DRDBs.

Unlike restart and mapping proposals, some of the work adopting the extension approach aims at providing an independent definition of OODBs. It is therefore possible to study how the characterized DOODs purport to extend OODBs. Unfortunately, this leads to the realization, in most cases, that only the structural aspect of OODBs is being extended by the class of DOODs characterized by this approach. There is no provision in the DOODs for the behavioural viewpoint that OODBs capture distinctively.

In terms of Figure 0.1, the source is the FOCT node for all of (Abiteboul, 1990; Abiteboul and Grumbach, 1990; Abiteboul and Kanellakis, 1989a). Although the space may contain OODM and OODB nodes, there is no alternative path from source to sink. This is because the OODM is not explicitly related to FOCTs, i.e., the edge from the FOCT node to the OODM node is not explicitly constructed. The extension approach is graphically represented by the fact that there is an extension path from DRDTs to DOODs. This makes DRDTs a subclass of DOODs, which means that DRDTs can inherit metalogical results from DOODs, but the converse is not the case, as the subclass relation is antisymmetric. In other words, DOODs can only inherit results from whichever theories they are a subclass of, but these need not be desirable, and are bound to be less desirable than those of DRDTs.

The approach described in this dissertation differs in varying degrees from the above three. It is rooted on a critique of the above approaches on the grounds that a high level of intervention in the language or in the rules of inference compromises decidability and tractability properties and increases the complexity of the proof and model theories. Furthermore, it becomes difficult to share the results obtained for the metalogic of DRDTs. This, in turn, means that the theory and practice of DOODs cannot directly build upon those of DRDBs. It is this last pragmatic concern that motivated the approach described in this dissertation, viz., to reuse in DOODs as many of the methods and techniques used to study and implement DRDBs as possible. The only component of a logical system on which a high level of intervention to provide a logical view of objects does not increase the complexity in the proof and model theories is the axiom set. The approach is, for this reason, referred to as *axiomatic*.

In the approach described here, the level of intervention of the language is nil. There is a low level of intervention in the single inference rule of Datalog, since the chosen treatment of the dynamic aspects of object-orientation requires a very minor change in the way inferences are performed. In contrast, the axiom set is very rich. It bears most responsibility for capturing object-oriented concepts in a logical framework. The advantages of this approach can be gauged if one observes that the part played by the axiom set in determining decidability and tractability properties is negligible. In other words, the size and content of the axiom set is essentially neutral in shaping such metalogical properties of a formal theory as decidability and tractability. Most results obtained for the metalogic of DRDBs are directly reusable for DOODs. This has the important practical consequence that DOOD systems can be implemented along well studied lines.

Table 0.5 summarizes the relative levels of intervention on theory components of each of the four approaches sketched above.

In addition, the axiomatic approach contributed by this dissertation gives rise to a space of theories that is a near mirror of that in Figure 0.1. This is because the two spaces share their FOCT, HCT and DDT nodes, and one can construe the edges formed by them as an axis of reflection.

	Level of Intervention on		
Approach	*Language*	*Axiom Set*	*Inference Rules*
restart	very high	nil	very high
mapping	high	nil	high
extension	high	nil	low
axiomatic	nil	high	very low

Table 0.5: Approaches to DOODs: Level of Intervention on Theory Components

(Figure 0.2 shows this object-oriented space and the shade of its relational reflection.) This shows how the topology of the two spaces is very similar. In fact, if the DOOT node is left out the two spaces are homeomorphic.

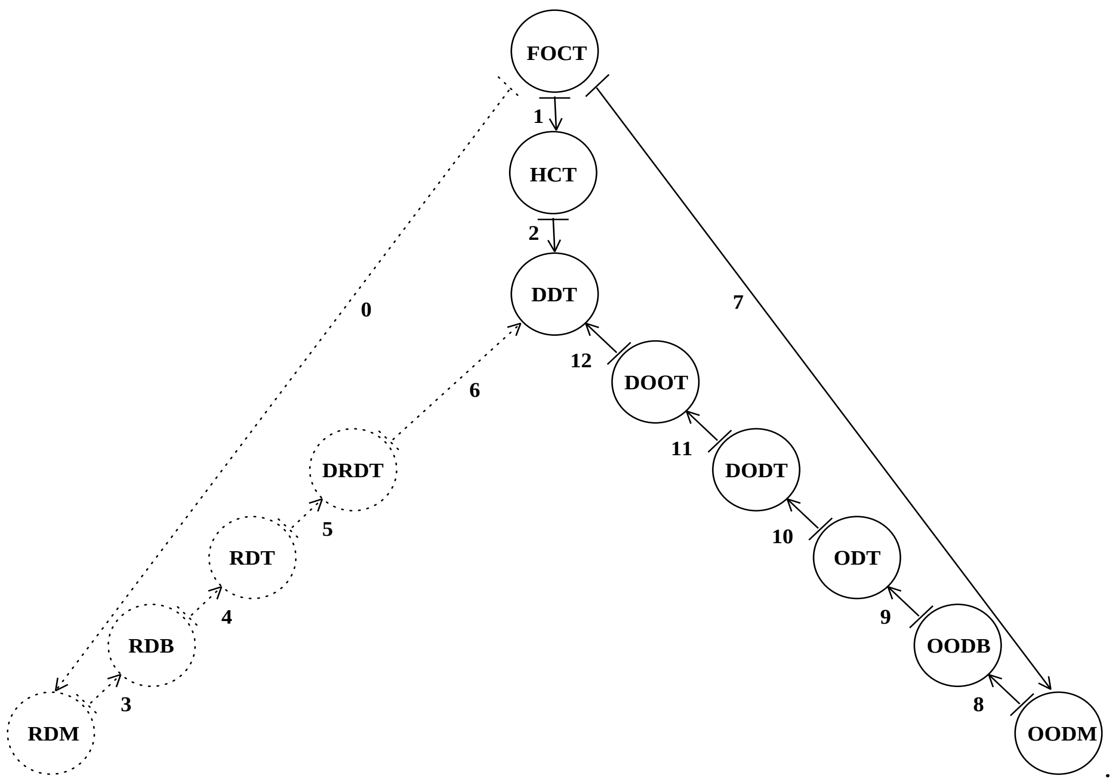


Figure 0.2: A Space of Theories: The Object-Oriented Case

Table 0.6 informally describes the salient features of the components of all classes of theories depicted in Figure 0.1, that are not already in Table 0.2. Shorthands are used exactly as defined for Table 0.2. The denotations of the new symbols used in Table 0.6 are the following. ODT, DODT, DOOT denote the classes of *object database theories*, *deductive object database theories* and *deductive object-oriented theories*, respectively. $OMAx$ denotes the set of axioms that formalize object modelling assumptions (OMA). These are given in precise detail in later chapters. Object-dynamics resolution (ODR) is a variant of classical resolution that formalizes the following object-oriented, and distinctively dynamic features (hence the OD part of the acronym), viz., encapsulation, overloading, overriding and late binding. The variant and the object-dynamics assumption (ODA) it formalizes, are also detailed in later chapters.

Table 0.6 can be used to understand what each of the new mappings in Figure 0.2 involves. Table 0.7 informally describes each of those mappings in terms of what the target of the mapping includes

		Axiom System		Inference System	
Class	*Language*	*Explicit*	*Implicit*	*Explicit*	*Implicit*
DOOT	ff-Horn clauses	$DedAx \cup SAx \cup OMAx$	RAx	$\{ODR\}$	$\emptyset$
DODT	ff-Horn clauses	$DedAx \cup SAx \cup OMAx$	RAx	{RES}	ODA
ODT	ff-Horn clauses	$SAx \cup OMAx$	RAx	{RES}	ODA
OODB	ff-g-u clauses	$\emptyset$	$\emptyset$	{RES}	S,ODA,OMA,RA
OODM	ff-g-u clauses	$\emptyset$	$\emptyset$	{RES}	ODA,OMA,RA

Table 0.6: Features of Theories in the Object-Oriented Space

and excludes from each component part of the source. Notational conventions are the same as used in Table 0.3.

Mapping			*Includes*	*Excludes*
FOCT	$\mapsto_7$	OODM	$\triangleright$ RA, $\triangleright$ OMA, $\triangleright$ ODA	$\neg g$, $\neg u$ clauses
OODM	$\mapsto_8$	OODB	$\triangleright$ S	—
OODB	$\mapsto_9$	ODT	$\triangleright\triangleright$ SAx, $\triangleright\triangleright$ $OMAx$, $\triangleright$ RAx	S, RA and OMA
ODT	$\mapsto_{10}$	DODT	$\neg g$-$\neg u$ clauses, $\triangleright\triangleright$ $DedAx$	—
DODT	$\mapsto_{11}$	DOOT	ODR	ODA, RES
DOOT	$\mapsto_{12}$	DDT	RES	SAx, $DedAx$, $OMAx$, ODR

Table 0.7: Mappings between Theories in the Object-Oriented Space

A more detailed description and comparison of alternative solutions is given in Chapter 6. A more general discussion of different approaches to DOODs can be found in (Fernandes et al., 1992b). The next part overviews the technical development that lies behind the solution depicted in Figure 0.2.

Technical Development

The technical development underlying the solution to the characterization of DOODs contributed by this dissertation is now briefly described. First, the central question addressed in this dissertation is stated, then the methods used in its solution are sketched.

The Central Question

The axiomatic approach to DOODs that is the theme of this dissertation aims at the reuse of as much of the theory and practice of DRDBs as possible. The ultimate goal is to ensure that the definition of a DOOD as an axiomatically presented OODB can be strengthened to the effect that the metalogic of a DOOD is predominantly shared with DRDBs. The central question that this dissertation seeks to answer is whether this class of DOODs is nonempty.

Notice that a DOOD in the class depicted in Figure 0.2 and described by Table 0.6 satisfies this definition, as follows:

language — Each DOOD in that class is a function-free theory of HCL. Hence, metalogical properties that are language determined are totally shared between DOODs and DRDBs.

axiom system — The axiom system of DOODs in that class differs from that of DRDBs only in the replacement of relational by object-oriented modelling assumptions. Hence, metalogical properties that are determined by the axiomatic content of the logical system are maximally shared between DOODs and DRDBs.

inference system — The inference system of DOODs in that class differs from that of DRDBs only in taking into account encapsulation, overloading, overriding and late binding in the development of proofs. This is achieved by a simple rewriting rule that requires no additional choice points in proof steps, has a determinate outcome and is computationally efficient. In fact, the most important metalogical properties that are determined by the inference system are not affected, including decidability and tractability.

This means that for this dissertation to solve the main question it addresses it is sufficient for it to provide a precise and detailed definition of the class of DOODs informally described by Table 0.6. As discussed above, this in turn can be achieved by the precise and detailed definition of the mappings informally described by Table 0.7. The methods by means of which these mappings are specified are now briefly discussed.

Methods

The solution to the main question depends on, and consists of, the precise specification of the following mappings (from Figure 0.2) and, as a consequence, of the classes of theories that play the role of source and target for such mappings:

$$FOCT \mapsto_7 OODM \tag{0.1}$$
$$OODM \mapsto_8 OODB \tag{0.2}$$
$$OODB \mapsto_9 ODT \tag{0.3}$$
$$ODT \mapsto_{10} DODT \tag{0.4}$$
$$DODT \mapsto_{11} DOOT \tag{0.5}$$
$$DOOT \mapsto_{12} DDT \tag{0.6}$$

The source and sink of the path characterized by mappings (0.1)-(0.6), viz., FOCTs and DDTs, are not specified in this dissertation. In fact, one way of stating a particular contribution of this dissertation is to say that it formally links source and sink nodes that are widely acknowledged as adequate to the objectives of the present work, with a path in which DOODs appear as an intermediate node. No particular presentation is assumed for FOCTs and DDTs. Among the extensive literature on these two classes of theories, the following are excellent book-length references on either or, more often than not, on both (Apt, 1990; Chang and Lee, 1973; Ceri et al., 1990b; Kowalski, 1979a; Das, 1992; Lloyd, 1987; Thayse, 1988).

The methods used in the solution of the main question are expected to yield the remaining required specifications as follows:

mathematical characterization — This method, pioneered in the relational case by (Codd, 1970), endeavours to yield a specification of the following arcs: FOCT $\mapsto_7$ OODM $\mapsto_8$ OODB. It requires defining with mathematical rigour a conception of an OODM, and the OODB it generates, demanding no stronger foundations than those underlying mainstream, contemporary mathematics, viz., FOL and axiomatic set theory. The mathematical characterization of an OODM and the OODBs it generates is the subject of Chapter 1.

logical reconstruction — This method, pioneered in the relational case by (Reiter, 1984), endeavours to yield a specification of the OODB $\mapsto_9$ ODT arc. It requires reconstructing the mathematical characterization of a data model and the databases induced by it as a logical system, i.e., as a system equipped with a notion of formal proof. The logical reconstruction of the OODM mathematically characterized in Chapter 1 is the subject of Chapter 2.

proof-theory automation — This method, surveyed in the relational case by (Ceri et al., 1990b; Gallaire et al., 1984), endeavours to yield a specification of the remaining arcs in the path, viz., ODT $\mapsto_{10}$ DODT $\mapsto_{11}$ DOOT $\mapsto_{12}$ DDT. It requires the definition of a tractable proof procedure for databases proof-theoretically presented, i.e., reconstructed as logical theories. Endowing the logically reconstructed OODBs characterized in Chapter 2 with an automated proof procedure is the subject of Chapter 3.

The specifications obtained by the methods sketched above comprise the solution to the characterization of DOODs contributed by this dissertation. The next part concludes this introduction with a discussion of this and other contributions.

Contributions

The major contribution of this dissertation is the comprehensive formalization of (the right half of) the space of logical theories in Figure 0.2. The particular attractiveness of that space over alternative ones lies in the fact that this space shares with the relational space of Figure 0.1 the path from FOCTs to DDTs, which means that all the theory and practice of DDTs that is orthogonal to the modelling assumptions that underlie DRDBs is shared by DOODs. No other characterization of DOODs is capable of, concomitantly, supporting as comprehensive object-oriented structural and behavioural modelling facilities and reusing as many research results.

Because FOCTs are so powerful and DDTs so brittle,[7] crafting a second path from one to the other that has OODBs and DOODs as intermediate nodes is not a trivial task. In terms of nodes in this path, the most important contributions of this dissertation are:

OODM — A mathematical characterization of an OODM as a species of mathematical structures which requires no background theory stronger than FOL and standard set theory.

This characterization has to be crafted in such a way that the logical reconstruction of OODBs as ODTs proves to be adequate in terms of admitting an automated theorem proving approach, otherwise the path to DOODs is blocked. In this respect, this dissertation demonstrates that it is desirable and possible to let the axiom system bear most of the burden involved in modelling object-oriented concepts. Heavily intervening in the language or the inference system has led alternative characterizations of DOODs to the disruption of metalogical properties of DRDBs that one would wish DOODs to possess.

Note that this mathematical characterization is an independent result and could, e.g., be developed algebraically, rather than relationally, as is the present case.

ODTs — A logical reconstruction of the OODM that transforms the OODBs induced by it into ODTs, which are formal systems of FOL.

Due to the way in which the construction of the OODM is crafted, this logical characterization enjoys the property that when the modelling axioms are translated into clausal form, all clauses obtained are function-free Horn clauses, and can, therefore, be trivially extended into DODTs.

DOOTs — A logical analysis of encapsulation, overloading, overriding and late binding that models the distinction between structure-based and behaviour-based DOODs.

The analysis models encapsulation as an alphabetic relationship between premisses and conclusion in resolution steps, and the combination of overloading, overriding and late binding as disambiguation between otherwise equivalent candidate premisses in resolution steps.

The background, motivation and challenges for the research reported in this dissertation have been discussed. The discussion was followed by a preliminary overview of related work, which is to be

[7] DDTs are said to be *brittle* in the sense that very small modifications in the alphabet, or the formation rules, or the inference system, can make query evaluation intractable and the validity problem undecidable.

detailed in a later chapter dedicated to comparison of approaches. Then, the central question with whose solution this dissertation is principally concerned was stated and the methods used to solve it were summarized. Finally, the contributions of this dissertation were briefly sketched.

The remainder of this dissertation is structured as follows. There are three main parts. The first and second parts contain the main results and contributions, while the third part is dedicated to comparison and conclusions.

Part I: The first part comprises the solution of the principal question addressed in this dissertation, viz., whether DOODs can be characterized as DDTs.

- **Chapter 1** is concerned the mathematical characterization of the OODM and the OODBs induced by it.
- **Chapter 2** is dedicated to the logical reconstruction of the OODBs characterized in Chapter 1 to yield ODTs.
- **Chapter 3** completes the solution by describing the progressive extension of ODTs into DODTs and DOOTs, culminating in an analysis of how DOOTs relate to DDTs.

Part II: The second part comprises the application and extension of the approach developed in the first part.

- **Chapter 4** substantiates the claim as to the practicality of the approach through a discussion of how it has been used in the development of ROCK & ROLL, a fully operational DOOD system developed at Heriot-Watt University.
- **Chapter 5** substantiates the claim as to the extensibility of the approach through a discussion of how the DOODs characterized in the first part, can incorporate more domain independent semantics and a logic-based notion of state, also describing in detail how DOODs can be extended to exhibit active behaviour.

Part III: The third part closes the dissertation by surveying related work and highlighting the contributions and future directions resulting from the approach described, applied and extended in Parts I and II.

- **Chapter 6** contains a survey of related work on the characterization of DOODs.
- **Chapter 7** recapitulates the main steps of the demonstration, states the main conclusions of the work, summarizes the contributions of the approach, and points to further possible developments.

Chapter 1

The Characterization of OODBs as Mathematical Structures

The purpose of this chapter is to provide a mathematical characterization of an OODM as a species of mathematical structures which requires a background theory no stronger than standard set theory.

The chapter yields a specification of the arcs

$$FOCT \mapsto_7 OODM \mapsto_8 OODB$$

in the object-oriented space of theories defined in the Introduction. Figure 1.1 reproduces that space, with the arcs that are treated in this chapter highlighted.

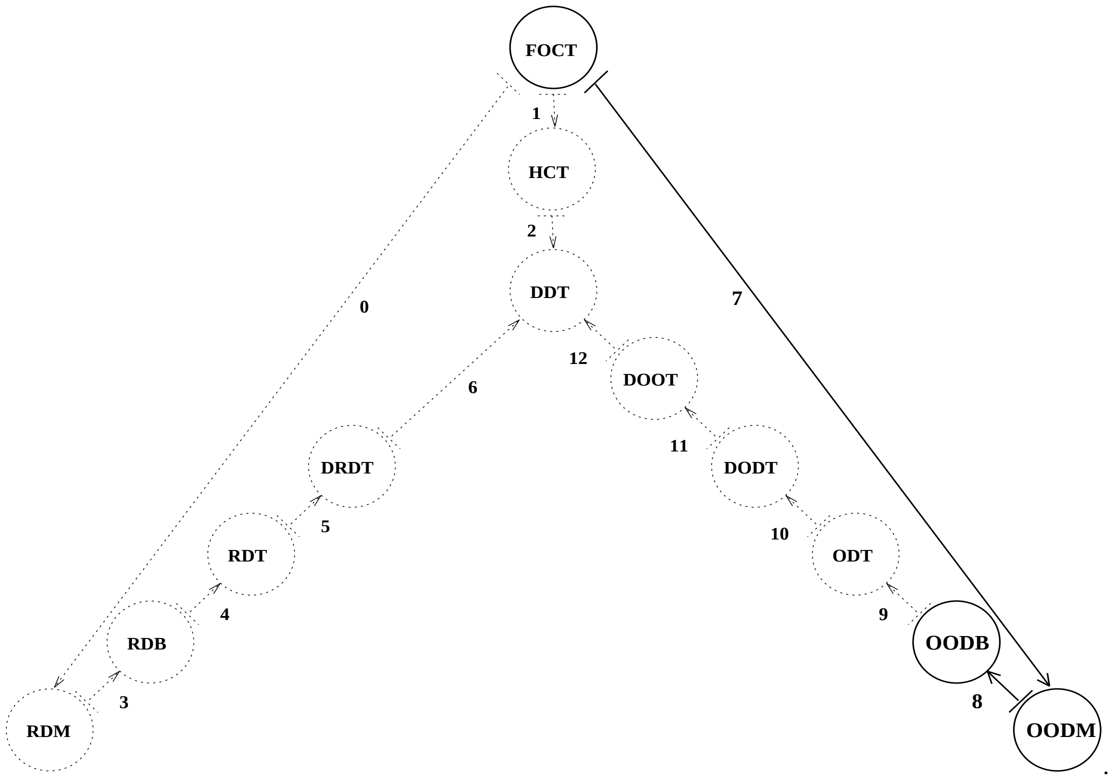


Figure 1.1: From FOCTs to OODBs in the Object-Oriented Space of Theories

The characterization of object-oriented databases as mathematical structures definable within a theory of sets is achieved by following, in its broad lines, the Bourbakian version of the axiomatic method for the characterization of a mathematical structure.

A *mathematical structure* [(Bourbaki, 1968), Ch. IV, pp. 259–264, (Itô, 1987), pp. 1546–1548] is a unified description of mathematical objects using only the concepts *set* and *relation*. Following Bourbaki in style and terminology, the steps of *the axiomatic method* are as follows:

1. The *principal and auxiliary base sets* are listed exhaustively.
2. An exhaustive list of *power sets of Cartesian products*, is exhibited, each element of which is formed from the base sets only.
3. Elements of these power sets are named exhaustively as the *basic relations* of interest, thereby giving rise to a *typical characterization*.
4. An *axiom system* (i.e., a conjunction of the typical characterization and the elements of a set of axioms) is produced to impose structure on the base sets. The crucial purpose of the axiom system is to define the *basic properties* of the basic relations, thereby characterizing their possible extensions.
5. A *proof* is given that the set of mathematical structures thus characterized is not empty, and hence, that the axiom system is consistent.

The purpose of this chapter is to construct a mathematical theory of object-oriented concepts in the context of database systems. The motivation for using the particular style of mathematical formulation briefly described above is the fact that it allows for a *first-order logical theory* to be derived from the *mathematical theory* in a small number of very simple steps. This derivation, known in the database literature as *logical reconstruction*, is the subject of Chapter 2.

At the conclusion of this chapter, the notions of *object-oriented data model* and *object-oriented database* are seen to be mathematically well defined.

This chapter consists of three sections.

Section 1.1 describes, in informal terms, the concepts whose formalization is sought. To make the concepts more concrete, numerous examples are presented.

Section 1.2 constructs, in full detail, a Bourbakian species of mathematical structures that captures in standard mathematical language the concepts informally described in Section 1.1.

Section 1.3 concludes the chapter by defining an OODM, and the OODBs it induces, in terms of the species of mathematical structures constructed in Section 1.2.

1.1 A Conceptual Scope for Object-Orientation

This section presents, in a broad and informal way, the notions that are captured in mathematical language in Section 1.2. The ontology imposed over application domains is presented and motivated by the use of examples. The distinguished relations between elements in application domains are informally defined. Finally, two sets of axioms are given whose purpose is to determine admissible extensions for the distinguished relations. The first set of axioms is seen to define how implicit information about application domains is modelled. The second set of axioms is seen to characterize legal states of application domains.

A large part of this section is devoted to the informal presentation of a set of axioms capturing object-oriented concepts. So long as the expressiveness of the OODM to which this axiom set gives rise conforms to the broad, informal standards which are beginning to emerge among OODB practitioners, the particular set of axioms described here is not in itself considered to be as important a contribution as the demonstration that an axiomatization of an OODM in FOL is indeed possible, and that this axiomatization can be reconstructed as a tractable system of logic, viz., function-free HCL.

In fact, as will be seen in Chapter 5, the approach to the formalization of an OODM adopted in this dissertation allows for the expressiveness of the OODM to be tailored and extended in a relatively straightforward manner. There is scope both to extend the range of modelling constructs and to provide different accounts of the modelling constructs described in this section. In short, the particular set of axioms presented in this dissertation is no more than an example, albeit, it is hoped, an interesting, comprehensive and useful one. Clearly, alternative axiom sets encompassing different modelling constructs, or accounting differently for some, would be equally defensible under different assumptions.

The axiom set presented in this dissertation is nevertheless distinguished by the fact that it is embodied in the ROCK & ROLL DOOD system (as described in Chapter 4). To some extent, the fact that ROCK & ROLL has been successfully used to develop nontrivial applications in challenging domains provides evidence that the axiom set presented below is a useful one — no stronger claim can be made about it.

The remarks above, however, should not obscure the fact that not all possible axiomatizations are equally adequate for the goal pursued in this dissertation. As already mentioned, only FOL axiomatizations that can be reconstructed as tractable logical systems are considered to be suitable alternatives. This criterion gives rise to the following principles, which underlay the composition of the particular set of axioms presented in this dissertation:

1. The axiomatization, and, more specifically, the properties of modelling relationships, must ensure, not compromise, the logical reconstruction of the OODM into a tractable logical system.

2. Provided that the preceding principle is strictly enforced, the axiomatization, and, more specifically, the properties of modelling relationships, should conform as closely as possible to the majority view on the meaning of object-oriented concepts (as reflected, e.g., in (Atkinson et al., 1990; Cattell, 1994; Fong et al., 1991; Soley and Kent, 1995; Zdonik and Maier, 1990a)).

The first principle above can be seen to have been applied in the following cases, among others. The use of type constructors is restricted. In particular, one is not allowed to nest type constructors, as, e.g., in type expressions such as *X is a set of sets of Y*. Instead, one is required to name all intermediate type(s), thereby breaking a nested type expression into two or more flat ones, e.g., *X is a set of Zs* and *Z is a set of Ys.*. This is necessary to ensure that the language remains function-free, since a recursively defined structure (such as, e.g., a Prolog list) is denoted in a first-order language by a recursively defined term governed by a non-nullary function symbol (such as, e.g., the Prolog functor '`./2`'). To allow the introduction of non-nullary function symbols implies allowing interpretation domains to be infinite, and this, in turn, makes any associated proof procedure undecidable.

Another example of application of the first principle above is the definition of stricter constraints on inheritance than are usually thought necessary. Thus, since multiple inheritance is supported, it is possible for an object to have a choice with respect to which version of a method it should use in response to a request. An axiom (Axiom 1.31 below) is used to enforce the fact that in every legal OODB the above choice is deterministic, i.e., it is always possible to pick up one of the versions, which, therefore, acts as the default one. This way of handling multiple inheritance is crucial to the proof-theoretical modelling of overriding and late binding that will be presented in Chapter 3.

Examples of the application of the second principle include the following stipulations, among others. Inheritance of structure and behaviour is based on a reflexive, transitive and antisymmetric inclusion relationship between type extensions (typically read as *is-a* or *a-kind-of*, e.g., a paperback is-a book). The attribution relationship between types (typically read as *has-a*) is among those affected by inheritance (e.g., if a book has-a title and a paperback is-a book, then a paperback has-a title; also, if a book is-a publication and a publication is-a book then book and publication have identical extensions). The composition relationship between types (typically read as *made-of* or *a-part-of*) is asymmetric, e.g., if an anthology is made-of articles then an article is not made-of anthologies. Such stipulations are in the spirit of (Atkinson et al., 1990; Cattell, 1994; Fong et al., 1991; Soley and Kent, 1995; Zdonik and Maier, 1990a).

1.1.1 On What There Is

Basic Sorts of Individuals

There are four basic sorts of individuals: **classes**, **objects**, **operations**, and **methods**. In a bottom-up analysis that proceeds from token to type, the most fundamental is the sort **objects**, in the sense that:

- classes have the main conceptual purpose of modelling collections of objects according to their shared structural and behavioural characteristics,
- operations have the main conceptual purpose of modelling abstract interaction with objects, and
- methods have the main conceptual purpose of modelling concrete interaction via the representation of objects.

In a top-down analysis that proceeds from type to token, the sort **classes** is the most fundamental, because knowledge of both the structural and behavioural aspects of entities in application domains is defined on entire classes rather than on the individual objects that classes collect.

In this section, examples are taken from application domains that have books in common, i.e., application domains involving knowledge about libraries, bookshops, publishers, printers, etc. Hence, examples of **classes** are those denoted by `ISBN, book, bookshop, book-title, integer`. Intuitively, an element of **classes** is an entity type in an application domain.

Examples of **objects** are those denoted by `0-387-13299-6, !1, !9, "Foundations of Logic Programming", 1`. Intuitively, an element of **objects** is an entity token in an application domain. Throughout this dissertation, a symbol whose first character is '`!`' denotes an abstract entity token in some application domain, and is not to be confused with an element of any of the traditional data types such as integers, or strings.

Examples of **operations** are those denoted by `buy, order, +`. Intuitively, an element of **operations** is an action type relating elements in **classes**.

To give examples of **methods** would require a commitment to denotations of the elements in **methods**, and this, in turn, would introduce unnecessary complexity at this stage of the exposition, since the motivation for those denotations cannot be presented yet. (Such commitment to denotations occurs in Chapter 3 after the necessary motivation). An element of **methods** is an action token relating elements in **objects**. The denotation of an element in **methods** is expected to model a real action over real objects in the application domain under consideration. Hence, it is best thought of as a program, i.e., a well formed formula in a language equipped with an operational semantics. The essential requirement in this context is that it must be possible to associate with each element in **methods** a computation carried out by some determinate abstract machine. In this sense, an element in **methods** denotes a relation between an *input sequence* and an *output sequence*, each consisting of elements in **objects**.

Dimensions over the Basic Sorts of Individuals

There are three orthogonal dimensions along which the four basic sorts of individuals can be organized:

1. In a **dependence-independence dimension** each of the four basic sorts is partitioned into two subsets, qualified by **specific** (in the sense of *application dependent*) and **generic** (in the sense of *application independent*). Hence, the following eight sorts are defined: **specific_classes** and **generic_classes**, **specific_objects** and **generic_objects**, **specific_operations** and **generic_operations**, **specific_methods** and **generic_methods**.

   For example, `ISBN`, `book`, `bookshop`, `book-title` denote elements of **specific_classes**, while `integer` denotes an element of **generic_classes**; `!1`, `!9` denote elements of **specific_objects**, while `0-387-13299-6`, `"Foundations of Logic Programming"`, `1` denote elements of **generic_objects**; `buy`, `order` denote elements of **specific_operations**, while `+` denotes an element of **generic_operations**.

   Elements of a specific sort are assumed not to be interpretable a priori, i.e., they are not assumed to be known outside the boundaries of a particular application domain. The denotation of an element of a specific sort is only a handle with which access to particular knowledge about the element can be gained. Therefore, elements of specific sorts imply detailed modelling within the context of each application domain. In contrast, elements of a generic sort are assumed to be interpretable a priori, i.e., they are assumed to be known and understood outside the boundaries of any particular application domain. The denotation of an element of a generic sort already conveys all there is to know about the element. Therefore, elements of generic sorts preclude any more detailed modelling that is particular to a specific application domain.

   For example, `ISBN`, `book`, `bookshop`, `book-title`, `!1`, `!9`, `buy`, `order` denote individuals that, in the context of the example application domains, are modelled in a way that is specific to those domains, implying by this that one is assumed not to know a priori what structure and behaviour they are perceived as having in the context of the given application domains, i.e., these individuals require detailed modelling. In contrast, `integer`, `0-387-13299-6`, `"Foundations of Logic Programming"`, `1`, `+` are modelled in a way that is not particular to the example domains, implying by this that one is assumed to know a priori what they mean, i.e., these individuals preclude any modelling that is specific to the application domains that categorize them as generic.

   In more traditional terminology, **generic_objects** are commonly called *primitive (atomic) values*, i.e., objects belonging to *primitive (atomic) data types* (which is what **generic_classes** are commonly called), while **generic_operations** are commonly called *intrinsic*, or *built-in*, *functions* (or *operators*), with which values of primitive data types can be operated upon.

   Their specific counterparts correspond closely to the notions of *classes*, *objects*, *operations* and *methods* in the object-oriented paradigm.

2. In a **schema-instance dimension**, **classes** and **operations** are used to model individuals at *schema* (or *type* or *intensional*) *level*, while **objects** and **methods** model individuals at *instance* (or *token* or *extensional*) *level*.

   For example, `ISBN`, `book`, `bookshop`, `book-title`, `integer`, `buy`, `order`, `+` denote schema-level individuals, while `0-387-13299-6`, `!1`, `!9`, `"Foundations of Logic Programming"`, `1` denote instance-level individuals, as also do all the elements in **specific_methods**. (The latter in the sense that one operation declaration can be instantiated by many different, albeit equivalent, definitions.)

3. In a **structural-behavioural dimension**, **classes** determine *structure* and **objects** determine *structural states*; while **operations** determine *behaviour* and **methods** determine *behavioural states* (as translated into the outcome of actions on structural states).

   For example, `ISBN`, `book`, `bookshop`, `book-title`, `integer` determine structure, while `0-387-13299-6`, `!1`, `!9`, `"Foundations of Logic Programming"`, `1` determine structural states. Also, `buy`, `order`, `+` determine behaviour.

Objects are organized into classes, of which they are then said to be *instances*. For example, the object `0-387-13299-6` is an instance of the class `ISBN`.

Classes are organized into inheritance hierarchies based on an inclusion relationship over their *extension*, i.e., the set of all their instances. For example, the class `paperback` is a subclass of the class `book`, or, informally, a `paperback` is a `book`. This implies that every instance of `paperback` is also an instance of `book`. Inheritance hierarchies are sometimes also referred to as *inclusion hierarchies*.

Constructs that model objects structurally are of two kinds. Firstly, an object is structurally modelled by *properties* that are *attributed* to the classes of which the object is an instance. For example, `book-title` names (the extension of) a property that is an attribute of instances of the class `book`. Secondly, a complex object (i.e., an instance of a composite class) is modelled by the *components* of its *construction*. For example, a relationship such as `published-by`, between an `author` and a `publisher`, can be represented as a composite class constructed by aggregation of the related classes. Hence, each instance of `published-by` will have in its construction an ordered pair binding an author and a publisher. For example, the instance `!101` of `published-by` might be the one that binds the instance of author whose name is `"John W. Lloyd"` is to the instance of publisher whose name is `"Springer-Verlag"`. Besides aggregations (that are record-like structures), associations (that are set-like) and orderings (that are list-like)[1] are also catered for.

Behaviourally, an object *responds to* messages invoking any *operation* that is, again, defined in a class of which the object is an instance. For example, instances of `book` respond to the operation `buy`. Upon the invocation of an operation, a method is selected and bound to the invocation so as to produce, after being executed, the required response. Inheritance is both structural and behavioural, i.e., of properties and construction, as well as of operations and methods.

Subsidiary Sorts of Individuals

It is an essential feature of an axiomatic theory that its domain be precisely defined in all the necessary detail. While the basic sorts of individuals described so far in this section are the ones that are directly employed by users to model concepts and entities in application domains, the degree of precision required in the definition of an axiomatic theory (such as the one developed later in this chapter) requires the characterization of additional sorts, referred to as *subsidiary*. The particular ones used in this dissertation can be motivated as follows.

Symbols are needed to convey the precise nature of the inclusion relationship between classes. More specifically, symbols are required to indicate whether inclusion is by *specialization*, *generalization* or *partition*. The semantics associated with each of these *inclusion modes* is different, as explained later in the chapter. Therefore, when an inclusion relationship is established between two classes, the inclusion mode must be specified. The symbols chosen for this purpose comprise the subsidiary sort **inclusion_modes**.

[1]The term *ordering* is adopted instead of the more traditional alternative, i.e., *sequence*, because the classical way to define a (finite) sequence assumes a finite, continuous segment of the natural numbers. Now, orderings only assume an index set that is a linear order. The natural numbers are an obvious choice for such an index set. If this is choice is taken, note that orderings can have gaps, in the sense that the predecessor of an index need not itself appear as an index.

Symbols are also needed to denote in which particular *inclusion hierarchy* a class is a subclass of another. This is needed because the extensions of the superclass and of its subclasses are related by certain constraints (e.g., in a partition, every instance of the superclass must be an instance of exactly one of the subclasses), but such constraints can only be enforced on a per-hierarchy basis. In other words, if a class has several inclusion hierarchies emanating from it in a given application domain, then the specific constraints associated with each inclusion mode should only be checked against those subclasses that belong to the *same* inclusion hierarchy. As a consequence, when an inclusion relationship is established between two classes, the inclusion hierarchy must be specified. The symbols chosen for this purpose comprise the subsidiary sort **inclusion_hierarchies**.

In analogy with inclusion modes, symbols are needed to convey the precise nature of the composition relationship between classes. More specifically, symbols are required to indicate whether composition is by *association*, *ordering* or *aggregation*. The semantics associated with each of these *composition modes* is different, as explained later in the chapter. Therefore, when a composition relationship is established between two classes, the composition mode must be specified. The symbols chosen for this purpose comprise the subsidiary sort **composition_modes**.

If it is the case that one class C is related to other classes C_1, C_2, etc., by composition, then an instance of the composite class C, will contain in its state references to instances of the component classes C_1, C_2, etc. Each of these references is labelled so that it is possible to ascertain that the different constraints associated with each composition mode are obeyed in the state of the constructed object. The labels chosen for this purpose comprise the subsidiary sort **indexicals**. In the case of association and aggregation, the symbols used as labels are, respectively, object and class identifiers. However, in the case of ordering, a new subsidiary sort **proper_indexes**, where **proper_indexes** $\subset$ **indexicals**, must be introduced because of the specific properties (explained later in the chapter) which that sort is required to possess.

Therefore, there are five subsidiary sorts of individuals whose existence is dictated by technicalities related to how relationships between individuals are characterized and which are, for this reason, conceptually less important than the basic sorts: **indexicals**, **proper_indexes**, **inclusion_modes**, **inclusion_hierarchies**, and **composition_modes**.

The first two of these are required to distinguish elements of constructed objects. The third and fourth are required to distinguish respectively between different possibilities of class inclusion, viz., `specialization`, `generalization`, and `partition`, and between multiple inclusion hierarchies rooted on a single superclass. The fifth is required to distinguish between different modes of object construction, viz., `association`, `ordering`, and `aggregation`.

Inclusion modes are used to determine precisely how the extension of subclasses in a given inclusion hierarchy, as determined by the inclusion criterion, are related to the extension of a common superclass. Composition modes determine precisely how references to construction elements (i.e., component objects) appear in the state of a constructed object.

indexicals and **proper_indexes** are used to impose additional structure on constructed objects. **proper_indexes** are used to model the idea of order, and for this purpose it becomes necessary to provide it with an **index_ordering** structure that makes the set isomorphic to an initial segment of the natural numbers.

1.1.2 On What Can Be Said

In any application domain, the kinds of relationships that could be used for the modelling of structural and behavioural knowledge is potentially very large. A representational formalism is best characterized by the particular set of relationships it prescribes for use in modelling application domains. Such distinguished relationships are called *modelling* (or, more often, though more vaguely, *semantic*) *relationships*.

Basic Relationships Between Individuals

The basic relationships chosen to model object-oriented concepts in this dissertation are:

1. **inclusion**, which denotes a binary relation in **classes**, gives rise to the notions of *superclass* and *subclass* of a given class. As examples, consider: a `paperback` is a kind of `book` (i.e., the class `paperback` is a subclass of the class `book`, or, equivalently, `book` is a superclass of `paperback`), an `academic_publisher` is a kind of `publisher`, a `journal_article` is a kind of `research_publication`.

   Note that inheritance by a subclass of the structure and behaviour defined for its superclasses is determined by **inclusion**.

2. **attribution**, which denotes a binary relation in **specific_classes**, gives rise to the notion of descriptive properties, or *attributes*, possessed by all objects of a class. For example, a `publisher` has a `name`, an `address`, a `catalogue`, and so on.

3. **composition**, which denotes a binary relation in **specific_classes**, gives rise to the notion of *bulk* (or *aggregate*, or *composite*) *classes*, i.e., classes whose instances are constructed (or aggregate, or composite, or complex) objects. For example, an instance of `borrowed_titles` is constructed as an unordered collection of `titles` of `books`, an instance of `authorship` is constructed as an ordered collection of `authors`, an instance of `address` is constructed as a triple consisting of a `location`, a `city` and a `postcode`.

4. **interface** denotes a k-indexed family of $(k+1)$-ary relations between **specific_classes**, k-adic **specific_operations** and corresponding *signatures*, themselves $(k-1)$-ary relations in **classes**. **interface** models the means by which one can interact with objects of a class by establishing a correspondence between elements in **specific_operations** and the classes to which each parameter passed to the operation is required to belong. As an example consider: to `buy` a `book` one specifies `price`, `bookshop`, and `edition`. In this example, the 5-ary **interface** relation relates the class `book`, the 4-adic **specific_operation** `buy`, and the 3-ary signature determined by the **classes** `price`, `bookshop` and `edition`, in that order.

   Note that there is a commitment to the class to which the interface is attached (`book`, in the example just given) but, in principle, no commitment as to whether a parameter is input (as is likely in the case of `edition`, in the example just given) or output (as is likely in the case of `price` and `bookshop`).

5. **classification**, which denotes a binary relation between **specific_objects** and **specific_classes**, gives rise to the notion of a particular object being an instance of one or more classes, by virtue of which the structure and behaviour of the object are determined. For example, the object denoted by `!9` is an instance of the class `bookshop`, thereby relating it structurally to instances of such attributes as `stock`, `address`, and guaranteeing, behaviourally, that it responds to messages invoking such operations as `start_sale` (with parameters such as `from_date`, `to_date`, and `discount_bands`) or `take_stock` (with parameters `at_date` and `stock_value`).

6. **description**, which denotes a ternary relation between **specific_objects**, **classes** and **objects**, gives rise to the notion of values for the properties of an object. For example, the instance `!1` of `book` has the value `"Foundations of Logic Programming"` for its `book_name` property.

7. **construction**, which denotes a ternary relation between **specific_objects**, **indexicals** and **objects**, gives rise to the notion of the construction elements of constructed objects. For example, an instance of `editorial_committee` refers to (and hence, is constructed as) an unordered collection of objects, each of which is an instance of `author`. Also, an instance of

`best_selling_titles` is constructed as an ordered collection of objects, each of which is an instance of `book`.

8. **implementation** denotes a k-indexed family of $(k+2)$-ary relations between **specific_classes**, k-adic **specific_operations**, corresponding $(k-1)$-ary signatures, and corresponding k-adic **specific_methods**.

   This and the remaining relations are difficult to exemplify because they involve methods, for which, as previously remarked, no denotations are being suggested. However, to try and give a flavour of what the relation **implementation** models, consider the following example. To implement an operation `decrease_price` on `book` with arguments `percentage` and `price` in ROCK (a language described later in Chapter 4) one could write:

```
class book
   decrease_price(percentage):price
   begin
      put_price((get_price@self) * (100.0 - percentage)/100)@self;
      get_price@self
   end
end-class
```

   Notice in the code above the elements related by **implementation**: the class `book`, the operation `decrease_price`, the arguments `percentage` and `price` and the method itself, i.e., the string enclosed by (and including) `begin` and `end`.

9. **execution** denotes a k-indexed family of $(k+1)$-ary relations between **specific_objects**, k-adic **specific_methods** and corresponding *responses*, themselves $(k-1)$-ary relations in **objects**. **execution** gives rise to the notion of a piece of executable code used to respond to a message sent to an object, taking into account the class of which the object is an instance.

   This implies, therefore, that *overloading* a message identifier is allowed, and so is *overriding*, in the subclass, of the code selected in the superclass. Overriding, in turn, requires delaying until run time the decision of which code to use in response to a message (an approach known as *late binding*).

   Assuming the example previously given, executing `decrease_price` on an instance of `book` whose `price` is `10.00` by a `percentage` of `30.0` yields, as a response, the `price` is `7.00`. Hence, **execution** relates input and output objects with a method. It abstracts over, in some sense, the operational semantics of whichever language is deployed in the writing of methods.

10. **typing** denotes a binary relation between **generic_objects** and **generic_classes**, and is analogous to **classification** in the case of application independent objects and classes.

11. **coercing** is analogous to **implementation** in the case of application independent classes, operations and methods. **coercing** denotes a k-indexed family of $(k+2)$-ary relations between k-adic **generic_operations**, corresponding *signatures*, themselves k-ary relations in **generic_classes**, and corresponding k-adic **generic_methods**.

    For example, in most programming language environments the generic operation `+` on the generic class `real` with signature $\langle$`real`, `real`$\rangle$ invokes (implicitly, via the compiler or interpreter) the code, native or otherwise, that computes the sum of two reals.

12. **evaluation** is analogous to **execution** in the case of application independent objects and methods. **evaluation** denotes a k-indexed family of $(k+1)$-ary relations between k-adic **generic_methods** and corresponding *responses*, themselves k-ary relations in **generic_objects**.

For example, assuming the name *two* in the name space of a programming language environment to be assigned to the generic object `2.0`, the evaluation of an expression such as *two* `+` *two* responds with the generic object `4.0`. Hence, like **execution**, **evaluation** also relates input and output objects with a method, except that in this case all of the related notions are generic, as opposed to specific. **evaluation** abstracts over, in some sense, the operational semantics of the built-in operations available in whichever language the writing of methods presupposes.

Subsidiary Relationships Between Individuals

In analogy with the earlier introduction of subsidiary sorts of individuals, subsidiary relationships are also required. The basic relationships **inclusion**, **attribution**, **composition** and **interface** comprise all the schema-level information that is specific to the domain being modelled. In particular, they blur the distinction between information that is explicitly declared and implicit information that is obtained by inference.

However, this distinction is essential in order to enforce constraints on structural and behavioural inheritance. For example, to guarantee that there is always a most specific implementation of a method which, by default, overrides all others, it must be possible to differentiate among the distinct classes where each candidate implementation has been attached by an explicit declaration. In other words, when a method invocation is ambiguous and the default case of late binding is supposed to take place, the disambiguation process relies on a distinction between declared and inherited operations.

To differentiate between declared and inferred information, four subsidiary relationships are introduced, one for each of the basic schema-level relationships. Thus, **inclusion***, **attribution***, **composition*** and **interface*** are subsidiary relationships corresponding, respectively, to the basic relationships **inclusion**, **attribution**, **composition** and **interface**.

In addition to the above mentioned need to distinguish between declared and inferred information, it is also the case that **inclusion*** is a more comprehensive specification than **inclusion**, insofar as **inclusion*** specifies the particular inclusion mode and inclusion hierarchy in which the two classes are related, whereas **inclusion** does not specify either. Similarly, **composition*** is a more comprehensive specification than **composition**, insofar as **composition*** specifies the particular composition mode in which the two classes are related, whereas **composition** does not. Again, **interface*** is a more comprehensive specification than **interface**, insofar as **interface*** makes explicit the class in which a given operation name is declared, whereas **interface** does not. In contrast to the previous three cases, **attribution*** and **attribution** are not differentiated by the fact that the former is a more comprehensive specification than the latter. The distinction between them is that the former only ranges over declared properties while the latter ranges, in addition to these, over inferred ones as well.

Therefore, there are four subsidiary relationships between individuals, viz., **inclusion***, **attribution***, **composition*** and **interface***. Their existence is dictated by technicalities related to how much information needs to be captured in order to characterize precisely which extensions of relationships are legal, and to the need to distinguish between assigned and inherited features. These subsidiary relationships are said to be *induced by* the basic relationships.

1. **inclusion*** differs from plain **inclusion** in having two additional arguments to determine, respectively, the inclusion mode (among the three available, viz., `specialization`, `generalization` and `partition`) that holds between two classes, and the inclusion hierarchy in question. In addition, **inclusion** ranges over inferred and declared information, whereas **inclusion*** only ranges over the latter.

   To illustrate the differences between the possible inclusion modes, consider the following. If `academic_book`, `fiction_book` and `reference_book` specialize the class `book` on its `category` hierarchy, this is taken to mean there may be an instance of class `book` that is not an instance of any of the above `category` subclasses. If `book` generalizes in its `format` hierarchy the classes `paperback`, `hardback`, `pocket_edition` and `book_club_edition`, this is taken to mean that each instance of class `book` is an instance of at least one of the above `format` subclasses. Finally, if `local` and `import` partition the class `book` on its `origin` hierarchy, this is taken to mean that each instance of class `book` is an instance of exactly one among the above `origin` subclasses.

2. **attribution*** differs from plain **attribution** in that the former only relates a class with its declared properties, whereas the latter additionally relates a class with its inherited properties.

   For example, suppose that a `scientific_publisher` is declared to be a `publisher` in some inclusion mode and some inclusion hierarchy. Suppose also that a `publisher` is declared to have, among others, a property `name` and that a `scientific_publisher` is declared to have, in addition (and among others), a property `specialisms` (e.g., `medicine`, `mathematics`, etc.). Then, the pair ⟨`scientific_publisher`, `name`⟩ is in the extension of **inclusion** but not in that of **inclusion***. That is to say, `name` is a property inherited by, as opposed to assigned to, `scientific_publisher`. In contrast, the pair ⟨`specialisms`, `scientific_publisher`⟩ is in the extension of **inclusion*** (because `specialisms` was assigned to `scientific_publisher`) and also in the extension of **inclusion** (because every property assigned to a class is, trivially, inherited by that class).

3. **composition*** differs from plain **composition** in having a third argument that determines the composition mode (among the three available, viz., `association`, `ordering` and `aggregation`) that holds between two classes. In addition, **composition** ranges over inferred and declared information, whereas **composition*** only ranges over the latter.

   To illustrate the differences between the possible composition modes, consider the following. `association` models unordered collections of homogeneously-typed objects (e.g., a `library` is an association of `sections`); `ordering` models ordered collections of homogeneously-typed objects (e.g., the `borrowing_history` of a `book` is an ordering of `borrowing` events); and `aggregation` models collections of tuples (e.g., a `borrowing` event is an aggregation of `date_out`, `reader`, `date_returned`).

4. **interface*** differs from **interface** in having an extra argument that determines in which superclasses, if any, the interface is also defined. For example, `lend` may be defined for `book`s in general (e.g., allowing all copies to be lent) but have a different definition in the `recommended_textbook` subclass of `book` (e.g., requiring that at least one copy is available from the shelves); in this case one can say that an instance of class `recommended_textbook` might respond to `lend` both as a `book` and as a `recommended_textbook`.

   Note that, in the presence of inheritance, the extra argument of **interface*** annotates, so to speak, the corresponding element of (plain) **interface**, thereby making it possible to distinguish whether an operation was directly assigned to a class or merely inherited.

1.1.3 On What Can Be Inferred From What Is Said

It is helpful in understanding the thrust of the formalization process to assume that the modelling of application domains results in the explicit assertion that certain relationships hold between individuals in the base sets, in a process that is detailed in Chapter 2. This can then be understood as the explicit information content, on the basis of which a set of axioms can be used to elicit implicit information, i.e., to close under logical consequence the explicit information content. Another set of axioms is used to constrain this closure, thereby determining which states are legal and which are not.

The way in which the axioms are presented in this and the next section is meant to pave the way for a characterization of DOODs (in Chapters 2 and 3) that parallels very closely the characterization of DRDBs as comprising a set of facts (the extensional database), a set of rules (the intensional database) and a set of constraints.

The minor drawback associated with this style of presentation is that some axioms that would normally be presented as one single group are split into two groups, viz., one consisting of axioms that legitimize the derivation of implicit information and another consisting of axioms that constrain this derivation process.

To illustrate the above mentioned split, consider the classical definition of a total ordering as a partial ordering in which all elements are pairwise comparable, where a partial ordering is a reflexive, transitive and antisymmetric relation on a set. Seen from the viewpoint of DDBs, these four properties, which collectively characterize a total ordering, are typically split into two groups as follows: two rules for the derivation of implicit information, viz., reflexivity and transitivity, and two constraints, viz., antisymmetry and pairwise comparability.

This is exactly the case with the **index_ordering** relationship over **proper_indexes**, which is a total ordering. It will be seen that the reflexivity and transitivity of **index_ordering** are defined in this section (by **Axiom 1.9**) whereas the antisymmetry and pairwise comparability of **index_ordering** are only defined in the next section (by **Axioms 1.35** and **1.36**).

Note that, both mathematically and logically, the formalization is neutral with respect to which relationships, and which instances thereof, are explicitly asserted, except in the cases of relationships that are induced from others (for which, see above).

Axioms 1.1-1.9 are used to elicit implicit information. They are presented below in their informal version. In Appendix A, they are given another formulation, at the level of rigour that is required to characterize OODBs as mathematical structures. The axioms used to constrain configurations are given later in this section.

In the axioms below, some infix relation symbols are introduced. They correspond to the relationships introduced above as summarized in Table 1.1.

*is_a**	**inclusion***
is_a	**inclusion**
*has**	**attribution***
has	**attribution**
*made_of**	**composition***
made_of	**composition**
responds_as_to	**interface***
responds_to	**interface**

Table 1.1: Correspondence of Informal Infix Relation Symbols to Relationships

Axiom 1.1 If a class has a superclass by some inclusion mode on some inheritance hierarchy, then it is a subclass of the latter. For example, if a `fiction_book` *is_a** `book`, then it *is_a* `book`, irrespective of the inclusion mode (e.g., `specialization`) and hierarchy (e.g., `category`). More precisely, **inclusion** is induced from **inclusion*** by disregarding the inclusion mode and inheritance hierarchy.

Axiom 1.2 Every class is a subclass of itself. Also, a class has as superclasses all the superclasses of its own superclasses. For example, a `remaindered_paperback` *is_a* `remaindered_paperback`, and if a `remaindered_paperback` *is_a* `paperback` and a `paperback` *is_a* `book` then a `remaindered_paperback` *is_a* `book`. More precisely, **inclusion** is reflexive over the set **classes** and transitive.

Axiom 1.3 Every assigned property of a class is a property of that class. For example, if `book` *has** a `book_name` then `book` *has* a `book_name`. More precisely, **attribution*** is subsumed by **attribution**, thus distinguishing assigned properties from inherited ones.

Axiom 1.4 Every property of a class is inherited by its subclasses. For example, if `book` *has* a `book_name` and a `paperback` *is_a* `book` then a `paperback` *has* a `book_name`. More precisely, **attribution** propagates through **inclusion**.

Axiom 1.5 If a class is composed of some component class by some composition mode, then it is composed of that class. For example, if `best_selling_titles` is *made_of** `books` then it is *made_of* `books`, irrespective of the composition mode (e.g., `ordering`). More precisely, **composition** is induced from **composition*** by disregarding the composition mode.

Axiom 1.6 The composition of a class is inherited by its subclasses. For example, if `editorial_committee` is *made_of* `authors` and a `programme_committee` *is_a* `editorial_committee` then `programme_committee` is *made_of* `authors`. More precisely, **composition** propagates through **inclusion**.

Axiom 1.7 The behaviour of a class is inherited by its subclasses. For example, if a `book` *responds_to* `lend` and a `fiction_book` *is_a* `book` then a `fiction_book` *responds_to* `lend` *as* a `book`. More precisely, **interface** propagates through **inclusion** by being annotated with the source class, thereby inducing **interface***.

Axiom 1.8 Every instance of a class is also an instance of the superclasses of its class. For example, if the object denoted by `!3` is an *instance_of* `remaindered_paperback` and a `remaindered_paperback` *is_a* `book` then `!3` is an *instance_of* `book`. More precisely, **classification** propagates through **inclusion**.

Axiom 1.9 Taken in conjunction with Axioms 1.35 and 1.36, this axiom characterizes **index_ordering** over **proper_indexes** as a total order. Here, **index_ordering** is defined to be reflexive over **proper_indexes** and transitive.

These, informally, are all the axioms used to elicit implicit information.

1.1.4 On What Cannot Be

In addition to the above axioms used to elicit implicit information, another set of axioms is used to constrain the extension of the basic modelling relationships, including, and with particular regard towards, those that are explicitly asserted. This can be viewed as a way of defining the set of legal states of the database, by laying down constraints that determine which extensions of the modelling relationships are valid and which not.

Axioms 1.10-1.36 are used to constrain configurations. Hence, they express invariants over application domains. They are presented below in their informal version. In Appendix A, they are given another formulation, at the level of rigour that is required to characterize OODBs as mathematical structures.

Axiom 1.10 An individual in the application domain must belong to exactly one sort. For example, a class is not an object nor is it an operation, and so on. This means that the object-oriented model described here is not uniform — as it would be if every individual in the domain were capable of being construed as an object (or a class, or an operation, etc.). More precisely, all sorts are pairwise disjoint.

Axiom 1.11 The specific and generic sorts of classes and object must have a union set. Also, **specific_classes** and **objects** must be usable as **indexicals**, along with **proper_indexes**. More precisely, there exists a set **classes** (respectively, **objects**; **indexicals**) that is the union of **specific_classes** and **generic_classes** (respectively, **specific_objects** and **generic_objects**; **specific_classes**, **objects** and **proper_indexes**).

Axiom 1.12 The possible **inclusion_modes** must be known exhaustively. More precisely, there are no **inclusion_modes** other than `specialization`, `generalization` and `partition`.

Axiom 1.13 The possible **composition_modes** must be known exhaustively. More precisely, there are no **composition_modes** other than `association`, `ordering` and `aggregation`.

Axiom 1.14 If two classes are related by **inclusion***, then, for the same hierarchy, they must be so related by exactly one inclusion mode. For example, if a `paperback` *is_a** `book` by `generalization` on `format`, then it is not, for the same hierarchy, either a `specialization` or a `partition` of `book`. More precisely, **inclusion*** determines a functional relationship from the first, third and fourth arguments to the second.

Note that the need to name inheritance hierarchies arises because there can be multiple inheritance hierarchies emanating from the same class, and, in view of this fact, inclusion constraints must be applied on a per-hierarchy basis.

For example, publishers could be partitioned by market into technical and leisure. They could concomitantly be partitioned by economic sector into private and public. The constraint on partitions that the extension of a partitioned class (`publisher`, in the example) equals the union of the extension of its pairwise disjoint subclasses can only be applied on a per-hierarchy basis (i.e., in the `by-market` hierarchy it is applied over the `technical` and `leisure` subclasses, while in the `by-economic-sector` it is applied over the `private` and `public` subclasses).

Axiom 1.15 If a `publication` *is_a* `book` and a `book` *is_a* `publication`, the set of all `book`s is identical to the set of all `publications`. More precisely, **inclusion** is antisymmetric.

Axiom 1.16 In an **inclusion** relationship only a specific class can be included in a specific class and only a generic class can be included in a generic class. For example, if a `paperback` *is_a* `book` and `book` is an element in **specific_classes**, then so must be `paperback`. More precisely, **inclusion** relates elements in the same subset of its domain.

Axiom 1.17 In multiple inheritance configurations, distinct classes with a common subclass must themselves have a common superclass. For example, if a `manual` *is_a* `reference_book` and a `manual` *is_a* `product_specific_document`, then there must exist a class, e.g., `publication`, such that `reference_book` *is_a* `publication` and `product_specific_document` *is_a* `publication`. More precisely, **inclusion** is weakly directed.

Axiom 1.18 Every object must belong to a class. For example, if `publication`, `company`, `person` and `string` were the only classes in an application domain, then any object in that application domain would have to be either a publication, a company, a person, or a string. More precisely, every element of **specific_objects** is **classification** related to some element of **specific_classes** and every element of **generic_objects** is **typing** related to some element of **generic_classes**.

Axiom 1.19 In a `generalization`, each instance of the generalized class must be an instance of at least one of its subclasses. For example, if `book` generalizes, on `format`, the subclasses `paperback`, `hardback`, `pocket_edition` and `book_club_edition`, then every `book` is either a `paperback`, a `hardback`, a `pocket_edition` or a `book_club_edition` in terms of its `format`. More precisely, in `generalization` **inclusion***, on a given hierarchy, the extension of the generalized class is the iterated union of the extension of its subclasses.

Axiom 1.20 In a `partition`, each instance of the partitioned class must be an instance of exactly one of its subclasses. For example, if the subclasses `local` and `import` partition the class `book` by `origin`, then every `book` has `origin` either `local` or `import`, but not both. More precisely, in `partition` **inclusion*** on a given hierarchy, the extension of the partitioned class is the iterated union of the extension of its subclasses and the iterated intersection of the latter is empty.

Axiom 1.21 In multiple inheritance configurations, if two classes have a common property, then they must not have a common subclass, since this would lead to ambiguity of reference in the latter. For example, if a `manual` *is_a* `reference_book` and a `manual` *is_a* `product_specific_document` then if `reference_book` and `product_specific_document` were to have a `title` it would be impossible to distinguish between the property named `title` inherited by `manual` from `reference_book` and the identically named property inherited by `manual` from `product_specific_document`.

This state of affairs stands in contrast with the legitimate multiple inheritance of identically named operations if they can be distinguished by their respective signatures. The problem with properties only arises because one can only distinguish between them by their name. More precisely, **attribution*** is not weakly directed through **inclusion**.

Axiom 1.22 If two classes are related by **composition***, then they must be so related by exactly one composition mode. For example, if `best_selling_titles` is *made_of** `books` by `ordering`, it must not also be either an `association` or an `aggregation`. More precisely, **composition*** determines a functional relationship from the first and third arguments to the second.

Axiom 1.23 Instances of a class that is a **composition*** by `association` or by `ordering` must be homogeneous with respect to the types of their components. For example, if `best_selling_titles` is an `ordering` of `books`, then all the elements in the construction of an instance of `best_selling_titles` must be instances of `book`. More precisely, `association` and `ordering` **composition*** determine a functional relationship from the first and second arguments to the third.

Axiom 1.24 A **composition** relationship between two classes must be one-way only and no class can be **composition** related to itself. For example, if an `anthology` is made of `articles`, then an `article` is not made of `anthologies`. More precisely, **composition** is asymmetric.

Axiom 1.25 In a multiple inheritance configuration, a common subclass must not inherit distinct component classes nor have conflicting composition modes. For example, if `thematic_collection` and `anthology` have `specialized_publication` as a common subclass then, if they are compositions, they must be made of the same components in the same way, e.g., an `association` of `articles`. More precisely, elements of **composition*** with distinct first arguments that are directed through the **inclusion** relation to a common sink must have pairwise equal second and third arguments.

Axiom 1.26 Each instance of a class that has a property must refer to an instance of the class that names the property. For example, if a `book` *has* a `book_name`, then the instance of `book` denoted by `!100323` must refer to an instance of `book_name`, e.g., `"Finnegans Wake"`. More concisely, proper **classification** related **description** references exist in all instances of a class for all **attribution** defined properties of the latter.

Axiom 1.27 Each instance of a class that is composed of another by `association` only refers in its construction to instances of the component class with a neutral (i.e., not order inducing) index (by default, the denotation of the referenced object). For instance, if a `library` is an `association` of `library_sections`, then if the `library` denoted by `!451` makes references to objects, e.g., `!562` and `!673`, as its construction elements, then `!562` and `!673` must denote `library_sections` and the references to them must have indexes `!562` and `!673` for the objects of identical denotation. More concisely, `association` **construction** references are properly identity indexed and **classification** related.

Axiom 1.28 Each instance of a class that is composed of another by `ordering`, refers in its construction to instances of the component class with a proper (i.e., order inducing) index. For instance, if `best_selling_titles` is an `ordering` of `books`, then if the `best_selling_titles` denoted by `!154` makes references to objects, e.g., `!265` and `!376`, as its construction elements, then `!265` and `!376` must denote `books` and the references to them must have proper indexes, say `'1` and `'2`. More concisely, `ordering` **construction** references are indexed by proper indexes and **classification** related.

Axiom 1.29 Each instance of a class that is composed of another by `aggregation`, must refer to an instance of each of its component classes. Each such reference must be indexed by the component class name (thereby inducing a notion of coordinate). For instance, if a `borrowing` is an aggregation of `borrowing_event`, `event_date` and `reader`, then if `!514` denotes a `borrowing` it makes references to objects, e.g., `!625`, `940404` and `!847`, as its construction elements, where `!625`, `940404` and `!847` denote, and are indexed by, a `borrowing_event`, an `event_date` and a `reader`, respectively. More concisely, `aggregation` **construction** references exist and are properly class indexed and **classification** related.

Axiom 1.30 Every specific class must have a behaviour, i.e., it must be possible to interact with its instances. For example, it must be possible to interact with a `book`, e.g., `buy` it. More precisely, an element in **specific_classes** responds to at least one element in **interface***.

Axiom 1.31 In a multiple inheritance configuration, if a class inherits an interface from two or more classes, then each of the latter must be comparable to the former with respect to **inclusion**. For instance, if a `book_club_edition` responds to `buy` both as a `book` and as `book_club_edition` then either `book` *is_a* `book_club_edition`, or the converse (which is, intuitively, the case). More precisely, the class annotations in **interface*** must always be **inclusion** comparable.

Axiom 1.32 Overloading must be of operation names only, i.e., taken with its signature and the class it is assigned to, an operation name has exactly one implementation. For example, `buy` might use the same method for different signatures (e.g., ⟨`first_edition`⟩ and ⟨`edition`⟩) within the same class `book`, or it might use the same method for the same signature on different classes (e.g., ⟨`price`⟩ on both `book` and `manuscript`), but it may not use more than one method for the same signature in the same class. Thus, every interface, with a given signature in a given class, has exactly one implementation. More precisely, there is no element that is **interface** related and not **implementation** related, and **implementation** determines a functional relationship from its last argument to all others.

Axiom 1.33 When an implementation is executed it responds with properly typed objects. More precisely, the execution of an implementation relates objects that conform with the signature of the interface the implementation is associated with.

Axiom 1.34 Every built-in operation can be evaluated. More precisely, every coercion pattern for a built-in is evaluable.

Axiom 1.35 Taken in conjunction with Axioms 1.9 and 1.36, this axiom characterizes **index_ordering** over **proper_indexes** as a total order. Here, **index_ordering** is defined to be antisymmetric.

Axiom 1.36 Any two distinct proper indexes must be comparable. Taken in conjunction with Axioms 1.9 and 1.35, this axiom characterizes **index_ordering** over **proper_indexes** as a total order. Here, the elements of **proper_indexes** are defined to be pairwise comparable under **index_ordering**.

These, informally, are the axioms used to constrain configurations.

1.2 A Species of Mathematical Structures over Object-Oriented Concepts

This section essentially rewrites Section 1.1 in more precise and rigorous language to dispel ambiguities and to lay the foundations for the logical treatment of object-oriented databases that starts in Chapter 2. In particular, a mathematical theory is developed whose individuals and relations are those exemplified and informally described in Section 1.1. Essential to this development is the recasting of the two groups of axioms given in the preceding section in more formal language, though their full formalization is only taken up in Chapters 2 and 3, where the notion of a formal proof makes this stricter level of formalization mandatory. The technical development of this section is closely modelled on that of [(Bourbaki, 1968), Ch. IV, pp. 259–264]. A more compact, and less rigorous, exposition of the same ideas is now given as technical background for the rest of the chapter.

1.2.1 A Brief Overview of Mathematical Structures

The notion of a mathematical structure is primarily used to characterize collections of mathematical objects purely by the relations that hold between them. From a purely mathematical point of view, its usefulness depends on its systematic application to multiple branches of mathematics. For this purpose the ontological parsimony of having sets as the only primitive notion is absolutely crucial. The technique has been used in modern mathematics to bring out essential, but hardly ever evident, similarities that span several fields of mathematical enquiry. For instance, it explains precisely why any problem in Euclidean geometry can be translated into an equivalent problem in algebra, by defining how one theory (in the example, Euclidean geometry, e.g., as axiomatized by Hilbert [(Itô, 1987), pp. 609–613]) has a model in the other (the theory of real numbers [(Stoll, 1979), Ch. 8, pp. 367–372]), in the example). In the context of this dissertation, however, the notion is used only as a means of giving mathematical expression to the notions of an OODM and of OODBs as the instances the OODM is meant to possess.

The example that follows comes from [(Itô, 1987), pp. 1546–1548]. It introduces the notion of a mathematical structure in a less rigorous style of exposition than that used in [(Bourbaki, 1968), Ch. IV, pp. 259–264]. The latter, more rigorous style is the one used later in this chapter for the actual technical development of a species of mathematical structures over object-oriented concepts.

Consider the following example. A **vector space** has:

1. two base sets:
   (a) a set K whose elements are called **scalars**, and
   (b) a set V whose elements are called **vectors**;
2. three power sets:

(a) $\mathcal{P}_1 = \mathcal{P}(K \times K \times K)$,

(b) $\mathcal{P}_2 = \mathcal{P}(V \times V \times V)$, and

(c) $\mathcal{P}_3 = \mathcal{P}(K \times V \times V)$;

3. three laws of composition and one operation:

   (a) $w_1, w_2 \in \mathcal{P}_1$,

   (b) $w_3 \in \mathcal{P}_2$, and

   (c) $w_4 \in \mathcal{P}_3$;

   (These are two laws of composition in K called **addition** and **multiplication**, one law of composition in V called **addition**, and the operation of K on V called **scalar multiplication**.)

4. basic properties described as n propositions on $K, V, w_1, \ldots, w_4$ denoted by $P_1|K, V, w_1, \ldots, w_4|, \ldots, P_n|K, V, w_1, \ldots, w_4|$. These will include:

$$\begin{aligned} \lambda(a+b) &= \lambda a + \lambda b \\ (\lambda+\mu)a &= \lambda a + \mu a \\ \lambda(\mu a) &= (\lambda\mu)a \end{aligned}$$

   for $\lambda, \mu \in K$ and $a, b \in V$, as well as those characterizing the scalars in K as the elements of a field, and the vectors in V as the elements of an Abelian group. (See [(Stoll, 1979), Ch. 8, pp. 329–331, 346–348], for a characterization of Abelian groups and fields, respectively).

The preceding paragraph describes an instance of a vector space given by K, V, and $w_1, \ldots, w_4$. To describe vector spaces in general, one replaces K, V, $w_1, \ldots, w_4$ in the above by Ξ_1, Ξ_2, $\xi_1, \ldots, \xi_4$. Then, $w_1 \in \mathcal{P}(K \times K \times K), \ldots, w_4 \in \mathcal{P}(K \times V \times V)$ become $\xi_1 \in \mathcal{P}(\Xi_1 \times \Xi_1 \times \Xi_1), \ldots, \xi_4 \in \mathcal{P}(\Xi_1 \times \Xi_2 \times \Xi_2)$; also, $P_1|K, V, w_1, \ldots, w_4|, \ldots, P_n|K, V, w_1, \ldots, w_4|$ become $P_1|\Xi_1, \Xi_2, \xi_1, \ldots, \xi_4|, \ldots, P_n|\Xi_1, \Xi_2, \xi_1, \ldots, \xi_4|$.

The set of symbols $\Sigma = \{\Xi_1, \Xi_2, \xi_1, \ldots, \xi_4\}$ is called the **signature** of vector spaces. The conjunction of the formulas $\xi_1 \in \mathcal{P}(\Xi_1 \times \Xi_1 \times \Xi_1), \ldots, \xi_4 \in \mathcal{P}(\Xi_1 \times \Xi_2 \times \Xi_2)$ is called the **typical characterization** of vector spaces. The set of formulas $\{P_1|\Xi_1, \Xi_2, \xi_1, \ldots, \xi_4|, \ldots, P_n|\Xi_1, \Xi_2, \xi_1, \ldots, \xi_4|$ is called the **axiom system** of vector spaces and the conjunction of the typical characterization of vector spaces with the conjoined elements of the axiom system of vector spaces is the formula Γ called the **axiom** of vector spaces.

In general, to define a mathematical system one lets $A_1, \ldots, A_m$ be the **base sets** (K and V in the preceding paragraphs). The **basic relations** (or **basic concepts**) $\alpha_1, \ldots, \alpha_n$ ($w_1, \ldots, w_4$ in the preceding paragraphs) are given as elements of sets that are finitely generated from $A_1, \ldots, A_m$, i.e., elements of sets obtained by a finite number of applications of the set operations for forming a Cartesian product and a power set over $A_1, \ldots, A_m$. The **basic properties** ($P_1|K, V, w_1, \ldots, w_4|, \ldots, P_n|K, V, w_1, \ldots, w_4|$ in the preceding paragraphs) are given as propositions $P_i|A_1, \ldots, A_m, \alpha_1, \ldots, \alpha_n|$. Taken together, the base sets, basic relations, and basic properties determine a **mathematical system**.

A **mathematical structure** is a pair $\langle \Sigma, \Gamma \rangle$ consisting of a *signature* $\Sigma = \{\Xi_1, \ldots, \Xi_m, \xi_1, \ldots, \xi_n\}$ comprising *symbols of base sets* and *symbols of basic relations*, plus an *axiom* Γ over Σ. When one replaces the symbols $\Xi_1, \ldots, \Xi_m, \xi_1, \ldots, \xi_n$ with sets $A_1, \ldots, A_m, \alpha_1, \ldots, \alpha_n$ and Γ evaluates to true, $(A_1, \ldots, A_m, \alpha_1, \ldots, \alpha_n)$ is called a **mathematical system with the mathematical structure** $\langle \Sigma, \Gamma \rangle$, or, equivalently, a **model of the structure** $\langle \Sigma, \Gamma \rangle$.

Two mathematical systems are said to be **similar** or **of the same kind** if they have the same, i.e., are models of the same, mathematical structure.

It should be noted that the construction of a mathematical system requires explicit enumeration of the base sets, the basic relations on the base sets and the properties of those relations, but it does not require the binding of any system of formal logic to the mathematical system. Proofs in a mathematical system are, in the strictest possible sense, not formal (i.e., they are *not* mere manipulation of strings of symbols by the mechanical application of rules that are based on the form alone of the strings and that only yield other strings of symbols as the result of rule application). Among other things, this makes mathematical systems not susceptible to mechanization. A logical system differs from a mathematical one exactly in taking the formalization requirement up to the notion of proof, thus allowing, in restricted cases, proof procedures to be mechanized.

1.2.2 The Species over Object-Oriented Concepts

The goal of this section is to characterize a *species of mathematical structures* that determines a class of *mathematical structures* such that, if one accepts that each such mathematical structure satisfies one's notion of an object-oriented database, then one is compelled to identify their species with an object-oriented data model.

Assume Σ^* to be a theory containing a **theory of sets** Σ, e.g., [(Bourbaki, 1968), Ch. II], or (Stoll, 1979). The **species of mathematical structures** $S(\Sigma^*)$ is the collection of the following expressions in the language of Σ^*:

1. Four sets of symbols enriching the language of Σ^* into a language for $S(\Sigma^*)$:

   (a) The symbols

      i. **C** for **specific_classes**,
      ii. **Gc** for **generic_classes**,
      iii. **O** for **specific_objects**,
      iv. **Go** for **generic_objects**,
      v. **Op** for **specific_operations**,
      vi. **Gop** for **generic_operations**,
      vii. **M** for **specific_methods**,
      viii. **Gm** for **generic_methods**, and
      ix. **I** for **proper_indexes**

      denoting the sets referred to as the **principal base sets of** $S(\Sigma^*)$.

   (b) The symbols

      i. **IM** for **inclusion_modes**, and
      ii. **IH** for **inclusion_hierarchies**
      iii. **CM** for **composition_modes**

      denoting the sets referred to as the **auxiliary base sets of** $S(\Sigma^*)$.

   (c) The symbols

      i. `specialization`,
      ii. `generalization`, and
      iii. `partition`

      denoting all the elements in **IM**.

   (d) The symbols

      i. `association`,
      ii. `ordering`, and

iii. `aggregation`

denoting all the elements in **CM**.

Let $\mathbf{C}^*$, for **classes**, denote $\mathbf{C} \cup \mathbf{Gc}$. Let $\mathbf{O}^*$, for **objects**, denote $\mathbf{O} \cup \mathbf{Go}$. Let $\mathbf{I}^*$, for **indexicals**, denote $\mathbf{C} \cup \mathbf{O}^* \cup \mathbf{I}$. Class names will distinguish (and hence, index) aggregation components, proper indexes will distinguish ordering components, while object names will appear as vacuous indexes in associations simply for the purposes of structural homogeneity.

2. The **typical characterization** $T(S(\Sigma^*))$, defined as the conjunction of expressions (1.1)-(1.17) in the language of $S(\Sigma^*)$:

$$\mathbf{inclusion}^* \in \mathcal{P}_1 \tag{1.1}$$

$$\mathbf{inclusion} \in \mathcal{P}_2 \tag{1.2}$$

$$\mathbf{attribution}^* \in \mathcal{P}_3 \tag{1.3}$$

$$\mathbf{attribution} \in \mathcal{P}_3 \tag{1.4}$$

$$\mathbf{composition} \in \mathcal{P}_3 \tag{1.5}$$

$$\mathbf{composition}^* \in \mathcal{P}_4 \tag{1.6}$$

$$\mathbf{interface} \in \mathcal{P}_5 \tag{1.7}$$

$$\mathbf{classification} \in \mathcal{P}_6 \tag{1.8}$$

$$\mathbf{description} \in \mathcal{P}_7 \tag{1.9}$$

$$\mathbf{construction} \in \mathcal{P}_8 \tag{1.10}$$

$$\mathbf{implementation} \in \mathcal{P}_9 \tag{1.11}$$

$$\mathbf{typing} \in \mathcal{P}_{10} \tag{1.12}$$

$$\mathbf{coercing} \in \mathcal{P}_{11} \tag{1.13}$$

$$\mathbf{interface}^* \in \mathcal{P}_{12} \tag{1.14}$$

$$\mathbf{index\_ordering} \in \mathcal{P}_{13} \tag{1.15}$$

$$\mathbf{execution} \in \mathcal{P}_{14} \tag{1.16}$$

$$\mathbf{evaluation} \in \mathcal{P}_{15} \tag{1.17}$$

where the $\mathcal{P}_i, 1 \leq i \leq 15$, are power sets defined as follows:

(a) Assume the existence of the following total *arity functions* $A_{\mathbf{Op}} : \mathbf{Op} \rightarrow \mathcal{N}$ and $A_{\mathbf{Gop}} : \mathbf{Gop} \rightarrow \mathcal{N}$, $A_{\mathbf{M}} : \mathbf{M} \rightarrow \mathcal{N}$, $A_{\mathbf{Gm}} : \mathbf{Gm} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is a finite segment of the set of natural numbers excluding zero. Then, $\mathbf{Op}_k \subseteq \mathbf{Op}$ (respectively, $\mathbf{Gop}_k \subseteq \mathbf{Gop}$; $\mathbf{M}_k \subseteq \mathbf{M}$; $\mathbf{Gm}_k \subseteq \mathbf{Gm}$) is such that for any $x \in \mathbf{Op}_k$ (respectively, $x \in \mathbf{Gop}_k$; $x \in \mathbf{M}_k$; $x \in \mathbf{Gm}_k$), $A_{\mathbf{Op}}(x) = k$ (respectively, $A_{\mathbf{Gop}}(x) = k$); $A_{\mathbf{M}}(x) = k$); $A_{\mathbf{Gm}}(x) = k$). Let K denote the union of the ranges of $A_{\mathbf{Op}}$ and $A_{\mathbf{M}}$ and K' the union of the ranges of $A_{\mathbf{Gop}}$ and $A_{\mathbf{Gm}}$.

(b) Let $\equiv$ denote the *abbreviates* (or *is-replaceable-by*) relation between expressions in the language of $S(\Sigma^*)$.

(c) Then, the $\mathcal{P}_i, 1 \leq i \leq 15$, abbreviate the following power sets of Cartesian products defined from the principal and auxiliary base sets of $S(\Sigma^*)$:

i. $\mathcal{P}_1 \equiv \mathcal{P}(\mathbf{C}^* \times \mathbf{IM} \times \mathbf{C}^* \times \mathbf{IH})$

ii. $\mathcal{P}_2 \equiv \mathcal{P}(\mathbf{C}^* \times \mathbf{C}^*)$

iii. $\mathcal{P}_3 \equiv \mathcal{P}(\mathbf{C} \times \mathbf{C})$

iv. $\mathcal{P}_4 \equiv \mathcal{P}(\mathbf{C} \times \mathbf{CM} \times \mathbf{C})$
v. $\mathcal{P}_5 \equiv \mathcal{P}(\mathbf{C} \times \mathbf{Op}_k \times \mathbf{C^*}_1 \times \ldots \times \mathbf{C^*}_{k-1})$
vi. $\mathcal{P}_6 \equiv \mathcal{P}(\mathbf{O} \times \mathbf{C})$
vii. $\mathcal{P}_7 \equiv \mathcal{P}(\mathbf{O} \times \mathbf{C} \times \mathbf{O}^*)$
viii. $\mathcal{P}_8 \equiv \mathcal{P}(\mathbf{O} \times \mathbf{I}^* \times \mathbf{O}^*)$
ix. $\mathcal{P}_9 \equiv \mathcal{P}(\mathbf{C} \times \mathbf{Op}_k \times \mathbf{C^*}_1 \times \ldots \times \mathbf{C^*}_{k-1} \times \mathbf{M}_k)$
x. $\mathcal{P}_{10} \equiv \mathcal{P}(\mathbf{Go} \times \mathbf{Gc})$
xi. $\mathcal{P}_{11} \equiv \mathcal{P}(\mathbf{Gop}_k \times \mathbf{Gc}_1 \times \ldots \times \mathbf{Gc}_k \times \mathbf{Gm}_k)$
xii. $\mathcal{P}_{12} \equiv \mathcal{P}(\mathbf{C} \times \mathbf{C} \times \mathbf{Op}_k \times \mathbf{C^*}_1 \times \ldots \times \mathbf{C^*}_{k-1})$
xiii. $\mathcal{P}_{13} \equiv \mathcal{P}(\mathbf{I} \times \mathbf{I})$
xiv. $\mathcal{P}_{14} \equiv \mathcal{P}(\mathbf{O} \times \mathbf{M}_k \times \mathbf{O^*}_1 \times \ldots \times \mathbf{O^*}_{k-1})$
xv. $\mathcal{P}_{15} \equiv \mathcal{P}(\mathbf{Gm}_k \times \mathbf{Go}_1 \times \ldots \times \mathbf{Go}_k)$

For the sake of readability an abuse of notation takes place in the definitions above, and throughout this dissertation, as follows. Expressions in which symbols with associated arity functions are involved, viz., **Op**, **Gop**, **M**, **Gm**, are actually template expressions. Each such template expression has one instantiation, which is an expression in the language of $S(\Sigma^*)$, per element of K and K'. Hence, $\mathcal{P}_5$, $\mathcal{P}_9$, $\mathcal{P}_{11}$, $\mathcal{P}_{12}$, $\mathcal{P}_{14}$, $\mathcal{P}_{15}$ should be implicitly understood to have multiple instances, and correspondingly so the relations defined in $T(S(\Sigma^*))$ as their elements. This abuse of notation is most noticeable when writing $\mathcal{P}_i, 1 \leq i \leq 15$ instead of $\mathcal{P}_i, 1 \leq i \leq (8 + |K| + |K'|)$.

3. The **mathematical axiom** $M(S(\Sigma^*))$ is defined as the conjunction of **Axioms A.1-A.36** given in full in Appendix A. These were presented less rigorously in Section 1.1 and are fully formalized as sentences of a first-order language in Appendix C. The flavour of the formulation in Appendix A can be gauged by the following samples:

   Axiom 1.4 above, i.e.,

   > Every property of a class is inherited by its subclasses.

   is formulated in Appendix A as follows:

   > **attribution** propagates through **inclusion**. In more detail, if x *has* a y and z *is_a* x, then z *has* a y.

   Axiom 1.18 above, i.e.,

   > Every object must belong to a class.

   is formulated in Appendix A as follows:

   > Every element of **specific_objects** is **classification** related to some element of **specific_classes** and every element of **generic_objects** is **typing** related to some element of **generic_classes**. In more detail, if $x \in$ **specific_objects**, then, for some $y \in$ **specific_classes**, x is an *instance_of* y, and if $x \in$ **generic_objects**, then, for some $y \in$ **generic_classes**, x is *of_type* y.

□

This completes the definition of the species of mathematical structures $S(\Sigma^*)$.

1.2.3 The Axiom of the Species

Let the **axiom of the species** $A(S(\Sigma^*))$ be defined as the following expression in the language of $S(\Sigma^*)$:

$$A(S(\Sigma^*)) \equiv T(S(\Sigma^*)) \wedge M(S(\Sigma^*))$$

By conjoining the typical characterization (which imparts structure to the base sets) and the mathematical axiom (which stipulates what essential properties the base sets possess in the system), the axiom of the species lays down the conditions under which a domain of mathematical objects can be said to belong to the species or not. If the axiom of the species holds over a given collection of mathematical objects, then the collection belongs to (i.e., has the mathematical structure of) the species, otherwise not.

In terms of the current goal of defining a model of data and the databases it generates, the axiom of the species will shortly be seen to play the role of discriminating that generation, i.e., it separates out candidate collections that conform to the OODM (and hence are called OODBs) from those that do not.

1.2.4 The Domain Tuple, Relation Tuple and Power Set Tuple

Let D (for *domain*) denote *the domain tuple* $\langle$ **C**, **Gc**, **C***, **O**, **Go**, **O***, **Op**, **Gop**, **M**, **Gm**, **I**, **I*** $\rangle$. Similarly, let R (for *relations*) denote *the relation tuple* $\langle$ **inclusion***, **inclusion**, **attribution***, **attribution**, **composition**, **composition***, **interface**, **classification**, **description**, **construction**, **implementation**, **typing**, **coercing**, **interface***, **index_ordering**, **execution**, **evaluation** $\rangle$, and let $\mathcal{P}$ (for *power sets*) denote *the power set tuple* $\langle \mathcal{P}_1, \ldots, \mathcal{P}_{15} \rangle$.

When the need arises to denote a determinate D (or R or $\mathcal{P}$) with an assigned extension to all its component sets, the superscripts $^\bullet$ and $\circ$ are used. Note that the typical characterization and the mathematical axioms may also be superscripted in the way described, in which case rather than assigned extensions, the connotation is of an assigned alphabet.

The domain and relation tuples constitute a presentation of the mathematical objects that make up a domain. Ultimately, a database (or, more precisely, a database state) will be presented as a pair $O = \langle D^\bullet, R^\bullet \rangle$. The power set tuple underlies the process through which the axiom of the species is verified to hold or not for a given $\langle D^\bullet, R^\bullet \rangle$. Hence, the power set tuple will underlie the process through which a database O is verified to be in a state that conforms to the data model or not.

1.2.5 The Theory of the Species

The theory whose proper axioms and axiom schemes are those of Σ^* together with $A(S(\Sigma^*))$ and whose proper symbols are those in the language of Σ^* together with the symbols appearing in $A(S(\Sigma^*))$ is called the **theory $\mho$ of the species of mathematical structures** $S(\Sigma^*)$.

The theory $\mho$ comprises all the theorems inherited from the background theory plus all the theorems that become derivable from the union of the latter with the axiom of the species. In terms of the mathematical characterization of a model of data, $\mho$ can be thought of as defining the application independent information content of the model, including explicitly the assumed mathematical foundations (in the present case, standard axiomatic set theory).

Therefore, the characterization of $\mho$ defines the class of all database states that conform to the stipulations of the data model as laid down in the axiom of the species.

1.2.6 Mathematical Structures and Sets of Mathematical Structures of the Species

Let Ω be a theory containing Σ^*, and let $D^\bullet$ and $R^\bullet$ be tuples of terms in Ω. In Ω, the tuple $\langle D^\bullet, R^\bullet \rangle$ is called a **mathematical structure of species** $S(\Sigma^*)$ **on the principal base sets** $D^\bullet$**, with auxiliary base sets IM**, **IH** and **CM** if $A(S(\Sigma^*))^\bullet$ is a theorem in Ω.

If this is the case, then for each theorem in $\mho$ there exists a corresponding theorem in Ω constructed by replacing every occurrence of a symbol of $\mho$ by its corresponding term in Ω.

Ω is specific to the domain that assigns particular extensions to the domain, relation and power set tuples. Provided the axiom of the species is satisfied in it, it can be thought of as an instantiation of $\mho$, or as the selection of a particular database among those that comply with the model of data arising from the axiomatization.

In $\mho$, R is called the **generic mathematical structure of the species** $S(\Sigma^*)$. In Ω, the principal base sets $D^\bullet$ are said to be **endowed with the mathematical structure** $R^\bullet$. The set $\mathbf{M}^\circ$ of tuples R° such that $M(S(\Sigma^*))^\circ$ is satisfied is called the **set of mathematical structures of the species** $S(\Sigma^*)$ **on** D°.

$\mathbf{M}^\circ$ is the set of all database states that conform to the model of data arising from the axiomatization. If the theory were to be inconsistent, no database state would be generated, i.e., $\mathbf{M}^\circ$ would be empty. That this is not the case is expressed by the following theorem.

THEOREM: **Existence of a Mathematical Structure of the Species** $S(\Sigma^*)$:

$\mathbf{M}^\circ$ is nonempty.

PROOF: The proof is by construction of a witness $W = \langle D^W, R^W \rangle$, exhaustively given in Appendix B.

The verification by inspection that $A(S(\Sigma^*))^W$ is a theorem in Ω proves that, in Ω, W is a mathematical structure of species $S(\Sigma^*)$ on the principal base sets D^W, with auxiliary base sets **IM**, **IH** and **CM**, and, hence that $\mathbf{M}^\circ$ is nonempty.

□

COROLLARY: **Consistency of** $\mho$:

$\mho$ is consistent.

□

The witness mathematical structure is wholly abstract. There is nothing unique about the particular witness presented; constructing a witness is a matter of avoiding the violation of any axiom. This task is trivial, for abstract structures at least. The witness given in Appendix B is annotated so as to help bring out the role the axioms play in characterizing the mathematical structure. Verifying that every axiom holds in the candidate domain and relations is a trivial, though exceedingly tedious, task.

1.3 The Mathematical Characterization of an OODM and its OODBs

This section characterizes an object-oriented data model, and the object-oriented databases that are induced by it, in terms of the species of mathematical structures defined in Section 1.2.

To motivate the characterization, consider how a relational model of data and the set of relational databases that are instances of that model can be characterized in terms of the technique applied in Section 1.2. The atomic domains with which attributes are associated correspond to principal base sets, relation schemas correspond to the typical characterization, and constraints such as those on key uniqueness and referential integrity correspond to axioms constraining configurations. Auxiliary base sets are not needed, unless special denotations are required, e.g., for null values. Since the relational model is conspicuously lacking in means for modelling application independent implicit information, there is also no need for axioms that elicit it.

The example above indicates that a model of data corresponds to a species of mathematical structures, and a database instantiating that model corresponds to an element in the set of mathematical structures of the species. Therefore, the following definitions are obtained.

The species of mathematical structures $S(\Sigma^*) = \langle D, R \rangle$ is an **object-oriented model of structural and behavioural data.**

Given $\langle D^\circ, R^\circ \rangle$, an **object-oriented database** is an element in the set $\mathbf{M}^\circ$ of tuples R°. Hence, in an object-oriented database $A(S(\Sigma^*))^\circ$ is satisfied.

The **set of mathematical structures of the species** $S(\Sigma^*)$ **on** D° is the set of all possible OODBs under the OODM $S(\Sigma^*) = \langle D, R \rangle$.

This shows that the notions of an OODM and of an OODB are mathematically well defined.

Now, consider again the space of theories in Figure 1.1. Insofar as the theory Σ^* is definable within FOL (and, hence, within FOCTs), the definition of the OODM $S(\Sigma^*)$, as detailed above, constitutes a specification of the arc

$$FOCT \mapsto_7 OODM.$$

Insofar as OODBs are defined as elements in the set of mathematical structures of the OODM species, the definition of the OODM $S(\Sigma^*)$, as detailed above, also constitutes a specification of the arc

$$OODM \mapsto_8 OODB.$$

Summary of Chapter 1

The lack of a widely accepted mathematical characterization of an OODM has been an obstacle to researchers in the database field. The main purpose of this chapter has been to define such an OODM in terms that facilitate the characterization of DOODs as DDTs. The chapter, therefore, has laid down the mathematical foundations underlying the logical treatment of object-oriented concepts in subsequent chapters.

This chapter had three sections.

Section 1.1 began by informally presenting a choice of concepts as being essential to characterize a database as object-oriented.

Section 1.2 initiated the formal treatment of OODBs. It used a Bourbakian style of presentation, since this simplifies the construction of the corresponding logical theory — a task to be carried out in Chapter 2. Notably, the resulting characterization is first-order and no theory stronger than a theory of sets is strictly required. The axiom system associated with the species of mathematical structures was proved consistent as a corollary to the proof that the set of mathematical structures of the species is not empty through the construction of a witness.

Section 1.3 showed how the notions of an *object-oriented model of structural and behavioural data* and of an *object-oriented database* can be given a definition in mathematically precise terms. This result constitutes the first contribution of this dissertation.

The mathematical theory developed in this chapter simplifies the task of constructing a logical theory of objects in databases. In the chapters that follow, the results obtained in this chapter are revisited from the point of view of formal, logical theories. In particular, it is shown that logically reconstructing the object-oriented mathematical structures defined in this chapter provides a path to the characterization of DOODs under an OODM that mirrors very closely the one used to define DRDBs under an RDM.

This chapter provided a mathematical characterization of an OODM as a species of mathematical structures, thereby yielding a specification of the highlighted arcs in Figure 1.1, viz.,

$$FOCT \mapsto_7 OODM \mapsto_8 OODB.$$

These arcs constitute the first step in the solution of the principal question which this dissertation seeks to provide an answer to. The second step, viz., the logical reconstruction of OODBs to yield ODTs, is the subject of Chapter 2. The solution is concluded in Chapter 3.

Chapter 2

The Logical Reconstruction of OODBs as Object Database Theories

The purpose of this chapter is to provide a logical reconstruction of the OODM that transforms the OODBs induced by it into ODTs, which are formal systems of FOL.

The chapter yields a specification of the arc

$$OODB \mapsto_9 ODT$$

in the object-oriented space of theories defined in the Introduction. Figure 2.1 reproduces that space, with the arc that is treated in this chapter highlighted.

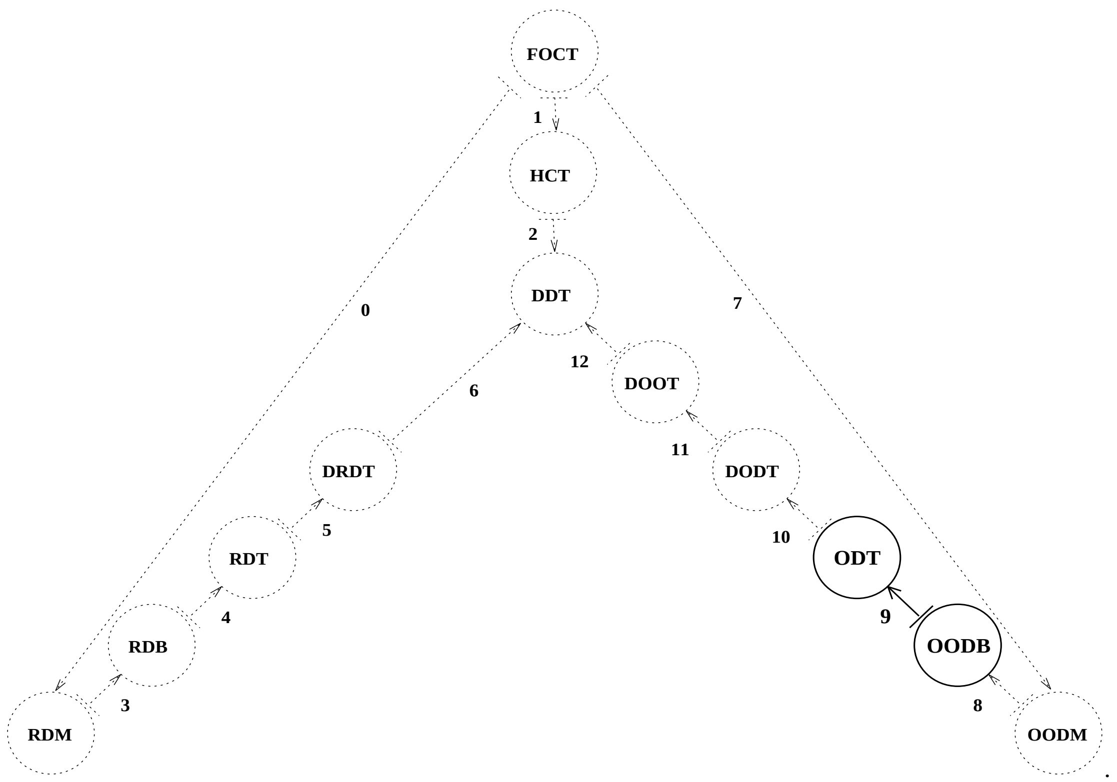


Figure 2.1: From OODBs to ODTs in the Object-Oriented Space of Theories

In Chapter 1, the Bourbakian method for the definition of mathematical structures was applied to capture, in a database context, a comprehensive set of object-oriented concepts, thereby providing a mathematically precise characterization of an OODM and the OODBs generated by it. In the context of this dissertation, the most important property of this characterization is the short and straightforward route it provides to the development of a logical system over a domain of objects, classes, operations and methods.

The OODM and the OODBs generated by it were defined in Chapter 1 as follows:

1. An *object-oriented model of structural and behavioural data* was defined as the species of mathematical structures $S(\Sigma^*) = \langle D, R \rangle$, where D, the domain tuple, is a tuple of base sets and R is a relation tuple over D, with each component relation in R having the properties defined by the axiom system $A(S(\Sigma^*))$.

2. An *object-oriented database* is an instantiation of the relations R over the domain D such that $A(S(\Sigma^*))$ holds in that instantiation.

The OODM and the OODBs generated by it are, therefore, *mathematical systems*. In this chapter they are reconstructed as *logical systems*, i.e., mathematical systems equipped with a notion of formal proof.

It is worth remarking that the technical content of Chapter 1 suffices for the development of a theory of OODBs with full mathematical rigour. That technical content is comparable in intention, albeit not in scope, to Codd's original work on the RDM (Codd, 1970), which pioneered databases as subjects for theoretical study. However, neither Codd's RDM nor the OODM defined in Chapter 1 support any notion of formal deduction. Both are model-theoretic constructions, neither is equipped with a proof theory.

However, one could envisage developing the metatheory of the $S(\Sigma^*)$ OODM example, to yield, e.g., a theory of what constitutes good design for the OODBs generated by it. This development would bear resemblance with the development of dependency theory for RDBs. Alternatively, one could set out to investigate problems in the fields of query optimization and evaluation, such as complexity and decidability.

Strictly speaking, it would be legitimate to view an OODB, as defined in Chapter 1, even as a set of premisses. In which case, by applying classical methods of mathematical reasoning, one could set out to realize a notion of query answering as deduction by proving, rigorously but informally, that facts hold which may or may not be explicitly asserted in the OODB.

However, to formalize these methods of proof, and thus pave the way to a true DDB, two steps are required:

1. It is necessary to make the deductive apparatus itself an integral and explicit part of the OODM theory. In precise terms, what is required is to transform the mathematical theory that arose from Chapter 1 into a *logical theory*. Since Reiter's seminal work (Reiter, 1984), this task has become known in the database literature as one of *logical reconstruction*.

2. It is necessary to automate the proof theory so that query answering, integrity constraint checking, etc., can be construed as theorem proving.

This chapter shows how the first of these requirements can be fulfilled, while Chapter 3 does the same for the second.

The task of logically reconstructing the RDM into an RDT is, in retrospect, not very complex. This is because, seen from a Bourbakian perspective, relations are exceedingly simple mathematical constructs. Although (Reiter, 1984) does not employ the Bourbakian technique used in Chapter 1, one could say that the axiom of the RDM species is a very undemanding one. More precisely, the typical characterization is almost the sole constraining component in the axiom of the species. The mathematical content of the RDM species, properly said, is simple to the extent that it is usually left out of most formulations, including Reiter's. Specifically, in its most widely used formulations, the mathematical axiom of the RDM consists of two conjuncts: one defining the uniqueness of keys and the other defining correctness of foreign key references. Compared with the OODM of Chapter 1, the resulting RDM species has much less mathematical structure. It is precisely in this sense that one can say that, in comparison to the RDM, an OODM is capable of capturing more of the semantic content of application domains. With the benefit of hindsight, this is what makes reconstructing the RDM mathematical theory as a logical theory a relatively simple task, and the credit for this insight is Reiter's, building upon previous work in the integration of logic and databases.

In the light of the remarks above, the main purpose of characterizing an OODM as done in Chapter 1 is precisely to minimize the complexity of the logical reconstruction that is required to describe how conventional OODBs induce deductive OODBs. Having shown how a comprehensive set of object-oriented concepts can be captured as a first-order mathematical theory, the task of logical reconstruction offers less technical barriers than would otherwise be the case (e.g., if a second-order theory had proved necessary, as some, e.g., (Beeri, 1990), have suggested in the recent past).

At the conclusion of this chapter, the mathematical theory of Chapter 1 is seen to facilitate a reconstruction as a logical theory of objects. In this scenario, each (state of an) OODB[1] generated by the OODM is a logical theory, called an **object database theory** (ODT). In Chapter 3, ODTs are shown to be expressible as function-free Horn clausal theories. This is an important property of ODTs, insofar as it means that the OODBs generated by the OODM are logic programs, and inherit the latter's procedural, fixpoint and model-theoretic semantics. Furthermore, the absence of function symbols in the language of ODTs means that their metalogic has most of the desirable properties of Datalog (Ceri et al., 1990b). This, in turn, opens the way for the expression of application specific integrity constraints and intensionally defined data by means of deductive rules.

This chapter consists of three sections.

Section 2.1 contains all the essential results of the chapter. It characterizes the language and the axiomatic structure of ODTs in terms of the results obtained in Chapter 1. ODTs are OODBs represented as logical systems of FOL.

Section 2.2 proposes and exemplifies a diagrammatic notation that can be used to represent ODTs in a concise way.

Section 2.3 exemplifies the expressive possibilities of ODTs through the selection of four example applications described in the deductive, object-oriented and semantic data modelling literature, each one of which is then cast as an ODT.

2.1 Characterizing Object Database Theories from OODBs

This section describes how an OODB can be presented as an axiomatic system of first-order logic, culminating in the characterization of the OODB as a domain specific ODT, i.e., a set of axioms in an applied first-order language, or, equivalently, as an applied first-order calculus.

2.1.1 Defining a Language for Object Database Theories

The language of ODTs is called OLL (for Object-Logic Language). OLL is an applied language of first-order logic whose interpretation structures are OODBs $\langle D^\circ, R^\circ \rangle$ generated by the OODM

[1] From now until Chapter 6, unless otherwise noted, the acronyms 'OODM' and 'OODB' denote, respectively, the species $S(\Sigma^*)$ of mathematical structures and an element in the set of the mathematical structures of the species $S(\Sigma^*)$, as defined in Chapter 1.

$S(\Sigma^*) = \langle D, R \rangle$ defined in Chapter 1. The modelling of an application domain takes the form of a set of OLL axioms, some of which are ground and denote facts. In this context, the proper symbols of OLL with respect to an application domain are determined as follows.

The axioms of the application domain determine the set $\mathcal{C}(D)$ of *individual constants* (0-ary function symbols) of OLL with respect to the application domain D: the set of individual constants contains those that actually occur in the axioms which describe D and only those. The set of OLL function symbols with arity greater than zero is empty for any application domain D. The set $\mathcal{P}$ of predicate symbols of OLL is fixed for all application domains, and is given below by enumeration. Let $\mathcal{V}$ be a countably infinite set of individual variables. As usual, $\mathcal{V}$ is invariant with respect to application domains. Given an application domain D, the set of *individual variables* actually occurring in the axioms that describe D is finite, and denoted by $\mathcal{V}(D)$. Then the **(nonlogical) alphabet** of OLL over an application domain D is given by the triple

$$\mathcal{A}(D) = \langle \mathcal{C}(D), \mathcal{P}, \mathcal{V}(D) \rangle$$

The set of **terms** of OLL over $\mathcal{A}(D)$ is given by the union

$$\mathcal{T}(D) = \mathcal{C}(D) \cup \mathcal{V}(D)$$

Let $\bar{u}, \bar{v}, \bar{w}_1, \ldots, \bar{w}_n, \bar{x}, \bar{y}, \bar{z}$ be syntactic variables ranging over $\mathcal{T}(D)$. Let $\hat{u}$, $\hat{v}$, $\hat{w}_1$, ..., $\hat{w}_n$, $\hat{x}$, $\hat{y}$, $\hat{z}$ be syntactic variables ranging over the base sets of $S(\Sigma^*)$. As in the previous chapter, let $\equiv$ denote the *abbreviates* (or *is-replaceable-by*) relation between expressions in the language of $S(\Sigma^*)$. The set $\mathcal{F}(D)$ of all **atomic formulas** of OLL over $\mathcal{T}(D)$ is defined as the set of all instances generated by a fixed set of templates, each of which is an abbreviation of an expression in the language of $S(\Sigma^*)$, as follows:

1. Expressions (i)-(xv) are templates generating atomic formulas of OLL that abbreviate formulas in the language of $S(\Sigma^*)$ expressing *membership in base sets*:

$$
\begin{aligned}
specific\_class(\bar{x}) &\equiv \hat{x} \in \mathbf{C} && \text{(i)}\\
generic\_class(\bar{x}) &\equiv \hat{x} \in \mathbf{Gc} && \text{(ii)}\\
class(\bar{x}) &\equiv \hat{x} \in \mathbf{C}^* && \text{(iii)}\\
inclusion\_mode(\bar{x}) &\equiv \hat{x} \in \mathbf{IM} && \text{(iv)}\\
composition\_mode(\bar{x}) &\equiv \hat{x} \in \mathbf{CM} && \text{(v)}\\
inclusion\_hierarchy(\bar{x}) &\equiv \hat{x} \in \mathbf{IH} && \text{(vi)}
\end{aligned}
$$

$$
\begin{aligned}
specific\_operation(\bar{x}) &\equiv \hat{x} \in \mathbf{Op} && \text{(vii)}\\
generic\_operation(\bar{x}) &\equiv \hat{x} \in \mathbf{Gop} && \text{(viii)}
\end{aligned}
$$

$$
\begin{aligned}
specific\_object(\bar{x}) &\equiv \hat{x} \in \mathbf{O} && \text{(ix)}\\
generic\_object(\bar{x}) &\equiv \hat{x} \in \mathbf{Go} && \text{(x)}\\
object(\bar{x}) &\equiv \hat{x} \in \mathbf{O}^* && \text{(xi)}\\
proper\_index(\bar{x}) &\equiv \hat{x} \in \mathbf{I} && \text{(xii)}\\
indexical(\bar{x}) &\equiv \hat{x} \in \mathbf{I}^* && \text{(xiii)}
\end{aligned}
$$

$$
\begin{aligned}
specific\_method(\bar{x}) &\equiv \hat{x} \in \mathbf{M} && \text{(xiv)}\\
generic\_method(\bar{x}) &\equiv \hat{x} \in \mathbf{Gm} && \text{(xv)}
\end{aligned}
$$

2. Expressions (xvi)-(xxxii) are templates generating atomic formulas of OLL that abbreviate formulas in the language of $S(\Sigma^*)$ expressing *membership in power sets of the typical characterization*:

$$
\begin{aligned}
is\_a^*(\bar{x}, \bar{y}, \bar{u}, \bar{v}) &\equiv \langle \hat{x}, \hat{y}, \hat{u}, \hat{v} \rangle \in \mathbf{inclusion}^* && \text{(xvi)}\\
has^*(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{attribution}^* && \text{(xvii)}\\
made\_of^*(\bar{x}, \bar{y}, \bar{z}) &\equiv \langle \hat{x}, \hat{y}, \hat{z} \rangle \in \mathbf{composition}^* && \text{(xviii)}\\
responds\_to(\bar{x}, \bar{y}, \bar{w}_1, \ldots, \bar{w}_n) &\equiv \langle \hat{x}, \hat{y}, \hat{w}_1, \ldots, \hat{w}_n \rangle \in \mathbf{interface} && \text{(xix)}
\end{aligned}
$$

$$
\begin{aligned}
instance\_of(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{classification} && \text{(xx)}\\
property(\bar{x}, \bar{y}, \bar{z}) &\equiv \langle \hat{x}, \hat{y}, \hat{z} \rangle \in \mathbf{description} && \text{(xxi)}\\
component(\bar{x}, \bar{y}, \bar{z}) &\equiv \langle \hat{x}, \hat{y}, \hat{z} \rangle \in \mathbf{construction} && \text{(xxii)}\\
implemented\_in\_by(\bar{x}, \bar{y}, \bar{w}_1, \ldots, \bar{w}_n, \bar{u}) &\equiv \langle \hat{x}, \hat{y}, \hat{w}_1, \ldots, \hat{w}_n \rangle \in \mathbf{implementation} && \text{(xxiii)}
\end{aligned}
$$

$$
\begin{aligned}
is\_a(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{inclusion} && \text{(xxiv)}\\
has(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{attribution} && \text{(xxv)}\\
made\_of(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{composition} && \text{(xxvi)}\\
responds\_as\_to(\bar{x}, \bar{y}, \bar{z}, \bar{w}_1, \ldots, \bar{w}_n) &\equiv \langle \hat{x}, \hat{y}, \hat{z}, \hat{w}_1, \ldots, \hat{w}_n \rangle \in \mathbf{interface}^* && \text{(xxvii)}
\end{aligned}
$$

$$
responds\_with(\bar{x}, \bar{y}, \bar{w}_1, \ldots, \bar{w}_n) \equiv \langle \hat{x}, \hat{y}, \hat{w}_1, \ldots, \hat{w}_n \rangle \in \mathbf{execution} \qquad \text{(xxviii)}
$$

$$
\begin{aligned}
precedes(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{index\_ordering} && \text{(xxix)}\\
of\_type(\bar{x}, \bar{y}) &\equiv \langle \hat{x}, \hat{y} \rangle \in \mathbf{typing} && \text{(xxx)}\\
coerces\_to(\bar{x}, \bar{y}, \bar{w}_1, \ldots, \bar{w}_n) &\equiv \langle \hat{x}, \hat{y}, \hat{w}_1, \ldots, \hat{w}_n \rangle \in \mathbf{coercing} && \text{(xxxi)}\\
evaluates\_to(\bar{x}, \bar{w}_1, \ldots, \bar{w}_n) &\equiv \langle \hat{x}, \hat{w}_1, \ldots, \hat{w}_n \rangle \in \mathbf{evaluation} && \text{(xxxii)}
\end{aligned}
$$

Every atomic formula of OLL is an instance of a template appearing in the left hand side of one among the abbreviations (i)-(xxxii) above. The set $\mathcal{P}$ of *predicate symbols* of OLL is the set of symbols occurring as predicate symbols in templates (i)-(xxxii).

The set $\mathcal{L}(D)$ of all **sentences** of OLL over an application domain D is defined in terms of the set $\mathcal{F}(D)$ of atomic formulas over D using the connectives and quantifiers of FOL in the standard way (Mendelson, 1987; Stoll, 1979). When a particular domain D is not relevant or is fixed by the context, $\mathcal{C}(D)$, $\mathcal{V}(D)$, $\mathcal{A}(D)$, $\mathcal{T}(D)$, $\mathcal{F}(D)$ and $\mathcal{L}(D)$ are written as $\mathcal{C}$, $\mathcal{V}$, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{L}$, respectively.

This completes the definition of the proper symbols of ODTs, and, hence, of the language OLL. Note that it is the reification of all categories of entities, viz., objects, classes, operations and methods, that makes it possible for the set of predicate symbols of OLL to be defined as closed, i.e., generic, or application independent. Although a theme not pursued in this dissertation, this makes the metalogic of OLL much simpler to develop, in this particular respect, than if predicate symbols were specific to application domains, as is the case with RDBs.

2.1.2 Structuring the Axiomatization of Object Database Theories

In logical terms, databases can be viewed both model and proof-theoretically (Gallaire et al., 1984). Thus, in model-theoretic terms, an OODB can be viewed as an interpretation structure for the application of OLL to a certain domain, with query evaluation amounting to semantic interpretation of an OLL formula against the OODB. In this dissertation, however, the proof-theoretic view of databases is adopted throughout. The crucial import of this view is that (the state of) a database is seen as a set of axioms, and query evaluation is seen as deduction from that axiom set.

The set of *database axioms* that characterizes an OODB contains four axioms: the *state* axiom, the *sort* axiom, the *completion* (or *positive*) axiom, and the *exception* (or *negative*) axiom. Each of these is a conjunction. In this dissertation, reference is sometimes made to the *set of* state (or sort, or completion, or exception) *axioms*. This usage is legitimate insofar as any of the four axioms can be represented as a set, the elements of which are the conjuncts, under the assumption that elements are implicitly conjoined.

Each of the above axioms is quite long. This is due to the fact that the OODM captures, in comparison with an RDM, a greater amount of knowledge about application domains. In itself, the size of the axioms has little impact on the design or use of OODBs, and is, therefore, largely transparent to users. However, an analysis showing how an ODT, i.e., an OODB seen as an axiom set, can be partitioned helps in understanding the informational content of OODBs.

The proper axioms of an ODT include:

1. A set $RAx = EqAx \cup PAx$ of *representation axioms* (see Table 0.6), where:

   (a) $EqAx$ is the set of *equality* axioms that define the equality predicate, and

   (b) $PAx = CWAx \cup DCAx \cup UNAx$ is the set of *particularization axioms*, where:

      i. $CWAx$ is the set of *closed world axioms* that formalize CWA, the closed world assumption,

      ii. $DCAx$ is the singleton containing the *domain closure axiom* that formalizes DCA, the domain closure assumption,

      iii. $UNAx$ is the set of *unique name axioms*, that formalize UNA, the unique name assumption.

   The representation axioms are required in the proof-theoretic view of databases irrespective of their underlying model of data. However, they need not be made explicit.[2]

2. A set $DBAx = SAx \cup OMAx$ of **database axioms** that determines a particular ODT (i.e., a state of an OODB), where:

   (a) $SAx = SCH \cup INS$ (see Table 0.6) is the set of **state axioms** that capture the schema of an application domain and one of its legal instances (i.e., the particular state of the domain). SAx is referred to as a **declared ODT**. All axioms in SAx are ground atomic formulas of OLL. SCH comprises instances of templates (i)-(viii) and (xvi)-(xix); while INS comprises instances of templates (ix)-(xv) and (xx)-(xxiii). The axioms in SAx bind an ODT to a particular application domain in a particular state. This means, for example, that the declared ODT determines the lexical content of the OODB.

   (b) $OMAx = T \cup C \cup E$ (see Table 0.6) is the set of **object modelling axioms**, which are invariant with respect to application domains, with the following disjoint subsets:

[2]See (Gallaire et al., 1984) for a formulation of the above axioms, a discussion of their role and of the circumstances under which they need not be made explicit, all of which apply in the context of this dissertation.

i. T is the set of **sort axioms**. They characterize $W = T \cup SAx$ as a **sorted ODT**, in the sense that, by taking the union of T and SAx, the extension of the sorts that structure the name space of application domains is derivable by logical consequence. The elements in T are implicitly conjoined, each conjunct being a universal closure. When expressed as a conjunction of universal closures, T is referred to as **the sort axiom** of ODTs. T formalizes the typical characterization $T(S(\Sigma^*))$ of the OODM.

ii. C is the set of **completion axioms**. They characterize $D = C \cup W$ as a **completed ODT**, in the sense that, by taking the union of C and W, all the positive implicit information about the application domain is derivable by logical consequence from the explicit information declared in W. The elements in C are implicitly conjoined, each conjunct being a universal closure. When expressed as a conjunction of universal closures, C is referred to as **the completion** (or **positive**) **axiom** of ODTs. Axioms in C will be seen to take the form of conditionals, most of whose antecedents involve templates (xxiv)-(xxvii). C formalizes axioms (A.1)-(A.9) of the OODM (see Appendix A).

iii. E is the set of **exception axioms**. They characterize $DBAx = E \cup D$ as a **(constrained) ODT**, in the sense that, by taking the union of E with D, all the negative implicit information about the application is derivable by logical consequence from the positive information modelled by D. Axioms in E stipulate conditions which, if satisfied, characterize exceptions, i.e., a *constraint violation*. The elements in E are implicitly conjoined, each conjunct being a universal closure. When expressed as a conjunction of universal closures, E is referred to as **the exception** (or **negative**) **axiom** of ODTs. E formalizes axioms (A.10)-(A.36) (see Appendix A).

Note that the extension of the relationships that give rise to templates (xxviii)-(xxxii) is neither explicitly asserted nor inferred from explicitly asserted ones. These extensions are determined by the computational environment in which the ODT is embedded. There is a difference, however, between those concerning generic classes, generic objects, and proper indexes (i.e., templates (xxix)-(xxxii)) and **evaluation** (i.e., template (xxviii)): the semantics of the latter is defined, for all environments, in Chapter 3, whereas that of the former group is inherited from, and, therefore, is particular to, each environment.

Figure 2.2 is a Venn diagram depicting the structure of the database axioms of ODTs. Insofar as the representation axioms remain implicit, the database axioms can be identified with the proper axioms of ODTs.

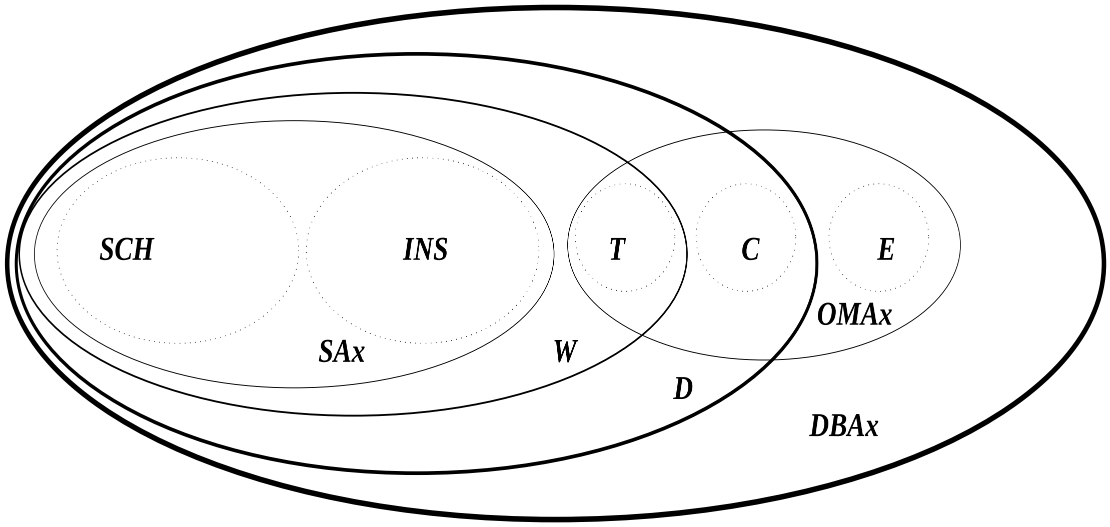
2.1.3 Formalizing the Proper Axioms of Object Database Theories

Notational Conventions

In writing down OLL axioms the following notational conventions are used. Letters taken from the beginning and the end of the lower case alphabet, e.g., a, b, u, v, w, x, y, z, possibly primed or subscripted denote OLL variables.

If $x_1, \ldots, x_n$ are all the free variables in a quantifier-free formula ϕ, the **universal closure** $(\forall x_1) \ldots (\forall x_n)(\phi)$ of ϕ is usually written $\forall(\phi)$. A sequence of quantifiers of the same kind $(\chi\ x_1) \ldots (\chi\ x_n)$ is usually written $(\chi\ x_1 \ldots x_n)$.

Note that the presence of an ellipsis '...' in atomic formulas (e.g., $responds\_to(x, y, w_1, \ldots, w_n)$) characterizes the sentence as a template, because the arity of the corresponding predicate symbol is determined in an application dependent way. This fact is glossed over in the rest of the dissertation. In all other circumstances, predicate symbols have fixed arity, as written.



SCH	*schema axioms*	*T*	*sort axioms*	*E*	*exception axioms*
INS	*instance axioms*	*W*	*sorted ODT*	*OMAx*	*object modelling axioms*
SAx	*declared ODT*	*C*	*completion axioms*	*DBAx*	*(constrained) ODT*
		D	*completed ODT*		

Figure 2.2: An OODB as a (Constrained) ODT

Formalizing the Typical Characterization of the Species

The **sort axiom** T is the conjunction of the universal closures of the formulas (1)-(20) of OLL:

$$is\_a^*(x,y,u,v) \Rightarrow (class(x) \wedge inclusion\_mode(y) \wedge class(u) \wedge inclusion\_hierarchy(v)) \tag{1}$$

$$is\_a(x,y) \Rightarrow (class(x) \wedge class(y)) \tag{2}$$

$$has^*(x,y) \Rightarrow (specific\_class(x) \wedge specific\_class(y)) \tag{3}$$

$$has(x,y) \Rightarrow (specific\_class(x) \wedge specific\_class(y)) \tag{4}$$

$$made\_of(x,y) \Rightarrow (specific\_class(x) \wedge specific\_class(y)) \tag{5}$$

$$made\_of^*(x,y,u) \Rightarrow (specific\_class(x) \wedge composition\_mode(y) \wedge specific\_class(u)) \tag{6}$$

$$responds\_to(x,y,w_1,\ldots,w_n) \Rightarrow (specific\_class(x) \wedge specific\_operation(y) \wedge class(w_1) \wedge \ldots \wedge class(w_n)) \tag{7}$$

$$instance\_of(x,y) \Rightarrow (specific\_object(x) \wedge specific\_class(y)) \tag{8}$$

$$property(x,y,u) \Rightarrow (specific\_object(x) \wedge specific\_class(y) \wedge object(u)) \tag{9}$$

$$component(x,y,u) \Rightarrow (specific\_object(x) \wedge indexical(y) \wedge object(u)) \tag{10}$$

$$implemented\_in\_by(x,y,w_1,\ldots,w_n,u) \Rightarrow (specific\_class(x) \wedge specific\_operation(y) \wedge class(w_1) \wedge \ldots \wedge class(w_n) \wedge specific\_method(u)) \tag{11}$$

$$of\_type(x,y) \Rightarrow (generic\_object(x) \wedge generic\_class(y)) \tag{12}$$

$$coerces\_to(x,w_1,\ldots,w_n,u) \Rightarrow (generic\_operation(x) \wedge generic\_class(w_1) \wedge \ldots \wedge generic\_class(w_n) \wedge generic\_method(u)) \tag{13}$$

$$responds\_as\_to(x,y,z,w_1,\ldots,w_n) \Rightarrow (specific\_class(x) \wedge specific\_class(y) \wedge specific\_operation(z) \wedge class(w_1) \wedge \ldots \wedge class(w_n)) \tag{14}$$

$$precedes(x,y) \Rightarrow (proper\_index(x) \wedge proper\_index(y)) \quad (15)$$

$$responds\_with(x,y,w_1,\ldots,w_n) \Rightarrow (specific\_object(x) \wedge specific\_method(y) \wedge object(w_1) \wedge \ldots \wedge object(w_n)) \quad (16)$$

$$evaluates\_to(x,w_1,\ldots,w_n) \Rightarrow (generic\_method(x) \wedge generic\_object(w_1) \wedge \ldots \wedge generic\_object(w_n)) \quad (17)$$

$$(specific\_class(x) \vee generic\_class(x)) \Rightarrow class(x) \quad (18)$$

$$(specific\_object(x) \vee generic\_object(x)) \Rightarrow object(x) \quad (19)$$

$$(specific\_class(x) \vee object(x) \vee proper\_index(x)) \Rightarrow indexical(x) \quad (20)$$

Let ϕ_t denote formula (t), 1$\leq$ t $\leq$20. Then, $\forall\phi_t$ is the sentence associated with ϕ_t and the formalization of the typical characterization $T(S(\Sigma^*))$ of the OODM is **the sort axiom of ODTs**, defined as the following sentence of OLL:

$$T \equiv \forall\phi_1 \wedge \ldots \wedge \forall\phi_{20}$$

Formalizing the Mathematical Axiom of the Species

Axioms (A.1)-(A.36) of Appendix A are reproduced as Axioms (C.1)-(C.36) in Appendix C. Each of the axioms is concomitantly formalized as one or more formulas of OLL. The positive and the negative axioms C and E, respectively, are formalized as follows.

1. The **completion** (or **positive**) **axiom** C is the conjunction of the universal closure of the formulas (1)-(12) of OLL given in full in Appendix C. These were first presented in Section 1.1 and then, more rigorously, in Appendix A. The flavour of the formulation in Appendix C can be gauged by the following example:

   Axiom A.4 in Appendix A, i.e.,

   > **attribution** propagates through **inclusion**. In more detail,
   > if x *has* a y and z *is_a* x, then z *has* a y.

   is formulated in Appendix C as follows:

   > If x *has* a y and z *is_a* x, then z *has* a y. Formally,
   >
   > $$(has(x,y) \wedge is\_a(z,x)) \Rightarrow has(z,y)$$

   Let ϕ_p denote formula (p), 1$\leq$ p $\leq$12. Then, $\forall\phi_p$ is the sentence associated with ϕ_p and **the positive axiom of ODTs** is defined as the following sentence of OLL:

   $$C \equiv \forall\phi_1 \wedge \ldots \wedge \forall\phi_{12}$$

2. The **exception** (or **negative**) **axiom** E is the conjunction of the universal closure of each of the formulas (13)-(55) of OLL given in full in Appendix C. These were first presented in Section 1.1 and then, more rigorously, in Appendix A. The flavour of the formulation in Appendix C can be gauged by the following example:

   Axiom A.18 in Appendix A, i.e.,

> Every element of **specific_objects** is **classification** related to some element of **specific_classes** and every element of **generic_objects** is **typing** related to some element of **generic_classes**. In more detail, if $x \in$ **specific_objects**, then, for some $y \in$ **specific_classes**, x is an *instance_of* y, and if $x \in$ **generic_objects**, then, for some $y \in$ **generic_classes**, x is *of_type* y.

is formulated in Appendix C as follows:

> If $x \in$ **specific_objects**, then, for some $y \in$ **specific_classes**, x is an *instance_of* y, and if $x \in$ **generic_objects**, then, for some $y \in$ **generic_classes**, x is *of_type* y. Formally:
>
> $$specific\_object(x) \Rightarrow \exists y(instance\_of(x, y))$$
>
> $$generic\_object(x) \Rightarrow \exists y(of\_type(x, y))$$

Let ϕ_n denote formula (n), $13 \leq n \leq 55$. Then, $\forall\phi_n$ is the sentence associated with ϕ_n and **the negative axiom of ODTs** is defined as the following sentence of OLL:

$$E \equiv \forall\phi_{13} \wedge \ldots \wedge \forall\phi_{55}$$

This completes the formalization of Axioms (A.1)-(A.36) of Appendix A. The formalization $\phi(M(S(\Sigma^*)))$ of the mathematical axiom $M(S(\Sigma^*))$ of the OODM is **the semantic axiom of ODTs**, defined as the following sentence of OLL:

$$\phi(M(S(\Sigma^*))) \equiv C \wedge E$$

Formalizing the Axiom of the Species

The formalization $\phi(A(S(\Sigma^*)))$ of the axiom $A(S(\Sigma^*))$ of the OODM is **the (object) modelling axiom of ODTs**, defined as the following sentence of OLL:

$$\phi(A(S(\Sigma^*))) \equiv T \wedge \phi(M(S(\Sigma^*)))$$

The set of object modelling axioms $OMAx$ is defined as the set of implicitly conjoined sentences of OLL that occur as conjuncts in $\phi(A(S(\Sigma^*)))$.

Formalizing the OODM and its OODBs

Let K be the class of first-order theories (e.g., [(Mendelson, 1987), Ch. 2, pp. 54–57]) and K^{OLL} the instance of K whose language is OLL. Then, the mathematical theory $S(\Sigma^*)$ (i.e., the OODM) is formalized as the logical theory K^{OLL}_{OMAx} that is obtained by taking $OMAx$ as a (set of) proper axiom(s).

Similarly, a set SAx of state axioms can be said to formalize an element in the set of mathematical structures of the species $S(\Sigma^*)$ (i.e., an OODB). Given $DBAx = SAx \cup OMAx$, the set of all ground atoms of OLL obtained by the closure under logical consequence of $DBAx$ defines, via expressions (i)-(xxxii) above, exactly one OODB.

An ODT is a logical theory K^{OLL}_{DBAx}. The proof and model theories of ODTs are those of first-order theories (Hunter, 1971; Mendelson, 1987; Stoll, 1979). As usual, an ODT is identified with its set of proper axioms, i.e., K^{OLL}_{DBAx} is identified with $DBAx$.

An ODT $DBAx$ can be converted into a set of clauses. In Chapter 3, advantage is taken of this fact to show how an ODT can be easily extended into a DODT.

2.2 Representing Object Database Theories with Diagrams

It is usually helpful to have a graphical notation to represent as much as possible of the design of an application domain. This usually concentrates on schema-level information, which deals with intensions of concepts and is, therefore, more concise. Nevertheless, such notations are typically easy to extend to instance-level information as well.

An ODT can be presented in diagrammatic form by a simple correspondence that binds symbols for sorts to nodes and symbols for relations to labels of directed edges. One such correspondence is described below for declared information at both schema and instance-level, except for instances of methods, because, as noted in Chapter 1, a commitment to their representation is being postponed until Chapter 3.

A class is drawn as a rectangle, an object as a circle, an operation as the upper half of a clipped-corner rectangle, a method as the lower half of a clipped-corner rectangle. If the border of any of the above is bold, then the sort is specific, otherwise, the sort is generic. Symbols for sorts are nodes; the content of a node is the denotation of an element of the represented sort.

Modelling relationships are drawn as directed edges between nodes. The label of the edge identifies the relationship. An **inclusion*** that is a `specialization` has label '*specializes*[ι]', one that is a `generalization` has label '*generalizes*[ι]', and one that is a `partition` has label '*partitions*[ι]', where, in all the above, ι denotes the inheritance hierarchy involved. An **attribution*** has label '*has*'. A **composition*** that is an `association` has label '*associates*', one that is an `ordering` has label '*orders*', and one that is an `aggregation` has label '*aggregates*'.

An **interface** is labelled by '*responds_to*'; an **implementation**, by '*implemented_in_by*'. If an operation has argument classes (as distinguished from the class in which the operation is declared), then a directed, broken line is drawn from the argument class until it meets the relationship edge. Each such broken line is labeled with the position of its source node in the corresponding operation signature.

A **classification** is labelled by '*instance_of*'. A **description** is labelled '*property*[δ]', where δ denotes the class that names the property involved. A **construction** is labelled '*component*[κ]', where κ denotes the class, object or proper index that names the index involved.

For generic classes, **typing** is labelled by '*of_type*'; while **coercing** is labelled by '*coerces_to*', following similar conventions to those described for **interface** above. Note that **execution** and **evaluation** are relationships described by an operational semantics. A diagrammatic technique encompassing them is beyond the scope of this dissertation. Finally, **index_ordering** does not require any diagram to be captured as it is isomorphic to the $\leq$ relation, under the standard interpretation, over the natural numbers.

This diagrammatic notation is exemplified and made use of in Section 2.3.

2.3 Applying Object Database Theories

Four example ODTs are now presented. The state axioms below only model the structural aspects of applications at schema level. The treatment of behavioural aspects is deferred to Chapter 3.

Each of the example ODTs models a database described in the literature on DDBs, OODBs, semantic data models and DOODs, as follows:

1. The **land parcel selection** example from (Abdelmoty et al., 1994; Barja et al., 1994).
2. The **antitrust control** example from [(Ceri et al., 1990b), pp. 30–33].
3. The **Media Storage Services** example from (Zdonik and Maier, 1990a).

4. The **library** example from (Peckham and Maryanski, 1988).

Each of the above examples is covered in a separate subsection. The main concern in the examples is to follow the verbal descriptions as closely as possible. This preoccupation with readability has had the consequence that some axioms that are declared below could have been left implicit, insofar as they are logical consequences of other declared axioms.

Note that no claim is made that the particular ODT that corresponds to each example is the best (or even a good) solution to the task of modelling the example problem using the OODM defined in this dissertation. Note also that OLL is not a language that end users, or even designers, would see or use. It is not difficult to devise a much friendlier surface syntax for OLL. One implemented proposal of such a user-level syntax can be glimpsed from the description of ROCK & ROLL in Chapter 4.

2.3.1 The *land parcel selection* Example

The land parcel selection (LPS) problem is the example used, among others, in (Abdelmoty et al., 1994; Barja et al., 1994). It describes (part of) a typical application of a geographical information system, viz., the selection of the most suitable land parcel, with respect to a given number of desirable properties, through the manipulation of several map layers describing the targeted areas.

The diagrammatic notation described in Section 2.2 is used in Figure 2.3 to represent the part of the LPS application that is useful in exemplifying ODTs. (Note that not all symbols suggested in Section 2.2 are used in the diagrams which appear in this dissertation.)

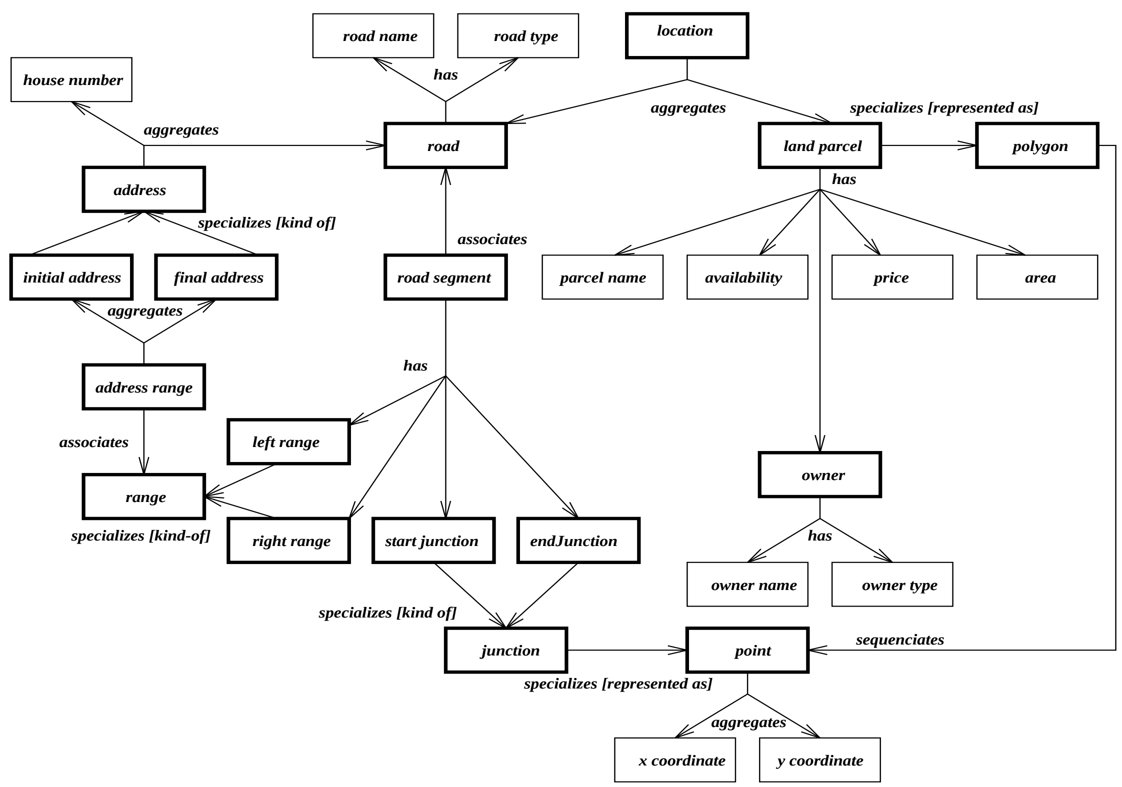


Figure 2.3: The *land parcel* Example in Diagrammatic Form

The information content of Figure 2.3 can be described as follows, with the interspersed axioms that belong to the set of state axioms SAx_{LPS}.

A road has attributes name and type, and is a collection of road segments, each of which has a start and an end junction, and a left and a right range.

```
specific_class(road)
    has*(road,road_name)
        generic_class(road_name)
            is_a*(road_name,specialization,string,represented_as)
                generic_class(string)
    has*(road,road_type)
        generic_class(road_type)
            is_a*(road_type,specialization,string,represented_as)
    made_of*(road,association,road_segment)
```

```
specific_class(road_segment)
    has*(road_segment,start_junction)
    has*(road_segment,end_junction)
    has*(road_segment,left_range)
    has*(road_segment,right_range)
```

```
specific_class(start_junction)
    is_a*(start_junction,specialization,junction,kind_of)
```

```
specific_class(end_junction)
    is_a*(end_junction,specialization,junction,kind_of)
```

```
specific_class(left_range)
    is_a*(left_range,specialization,range,kind_of)
```

```
specific_class(right_range)
    is_a*(right_range,specialization,range,kind_of)
```

A junction is a point, i.e., a pair made of an x- and a y-coordinate.

```
specific_class(junction)
    is_a*(junction,specialization,point,represented_as)
```

```
specific_class(point)
    made_of*(point,aggregation,x_coordinate)
        generic_class(x_coordinate)
            is_a*(x_coordinate,specialization,real,represented_as)
                generic_class(real)
    made_of*(point,aggregation,y_coordinate)
        generic_class(y_coordinate)
            is_a*(y_coordinate,specialization,real,represented_as)
```

A range is a collection of address ranges, each of which is a pair made of an initial and a final address.

```
specific_class(range)
    made_of*(range,association,address_range)
```

specific_class(address_range)
  made_of(address_range,*`aggregation`*,initial_address)*
  made_of(address_range,*`aggregation`*,final_address)*

specific_class(initial_address)
  is_a(initial_address,*`specialization`*,address,kind_of)*

specific_class(final_address)
  is_a(final_address,*`specialization`*,address,kind_of)*

An address is a pair made of a house number and a road.

specific_class(address)
  made_of(address,*`aggregation`*,house_number)*
    generic_class(house_number)
      is_a(house_number,*`specialization`*,integer,represented_as)*
  made_of(address,*`aggregation`*,road)*

Location is a relationship between roads and land parcels.

specific_class(location)
  made_of(location,*`aggregation`*,road)*
  made_of(location,*`aggregation`*,land_parcel)*

A land parcel is a polygon, i.e., a sequence of points, and has attributes name, availability, price, area and owner.

specific_class(land_parcel)
  is_a(land_parcel,*`specialization`*,polygon,represented_as)*
  has(land_parcel,land_parcel_name)*
    generic_class(land_parcel_name)
      is_a(land_parcel_name,*`specialization`*,string,represented_as)*
  has(land_parcel,availability)*
    generic_class(availability)
      is_a(availability,*`specialization`*,integer,represented_as)*
  has(land_parcel,price)*
    generic_class(land_price)
      is_a(land_price,*`specialization`*,integer,represented_as)*
  has(land_parcel,area)*
    generic_class(area)
      is_a(area,*`specialization`*,real,represented_as)*
  has(land_parcel,owner)*

specific_class(polygon)
  made_of(polygon,*`ordering`*,point)*

An owner has attributes name and type.

specific_class(owner)
has(owner,owner_name)*
generic_class(owner_name)
is_a(owner_name,*`specialization`*,string,represented_as)*
has(owner,owner_type)*
generic_class(owner_type)
is_a(owner_type,*`specialization`*,string,represented_as)*

The formal sentences above comprise a subset of a set of state axioms SAx_{LPS} that models the described application domain. Several operations and queries are required in this application. These are addressed in later chapters.

2.3.2 The *antitrust control* Example

(Ceri et al., 1990b) is a survey of DDB research and uses two examples to show how intrinsically recursive notions that cause problems for RDBs are elegantly solved using deductive rules. The first of their examples, viz., *the antitrust control problem* (ACP), is captured by the ODT below.

The diagrammatic notation described in Section 2.2 is used in Figure 2.4 to represent the part of the ACP application that is useful in exemplifying ODTs.

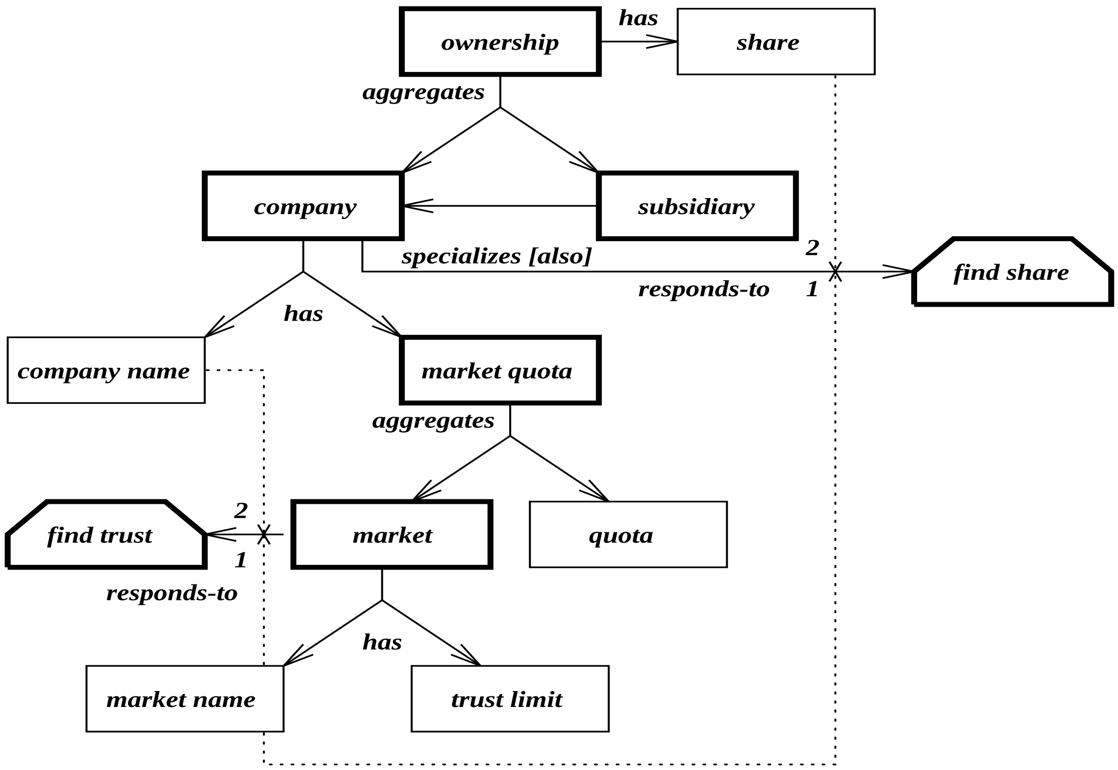


Figure 2.4: The *antitrust control* Example in Diagrammatic Form

The antitrust control problem is described in (Ceri et al., 1990b) as follows:

> "When a certain company produces goods that are sold in some geographical area of demand for these goods, or market, we say that this company *owns directly a quota* of that market. A company can also control a market quota by owning a majority share of another company, which in its turn controls a part of the market.

Some countries issue economic policies for the purpose of preventing firms from controlling, either directly or indirectly, more than a fixed market quota. Companies that escape these rules are said to form a *trust*." [in (Ceri et al., 1990b), pp. 31]

Each company has a name, and operates in one or more markets, in each of which the quota directly owned by the company is known, in percentage terms. It is also known what percentage of shares of other companies each company owns. Finally, for each market, the quota of the market that characterizes a trust is known.

"If firm A directly owns more than 50% of firm B, we say that A controls B. In this case, A has control over all the market quotas owned by B, and over the companies controlled by B. Thus, the problem of determining whether a given company violates the antitrust control is inherently recursive." [in (Ceri et al., 1990b), pp. 31]

An operation *find_share* is defined on company such that, given a company and a market, it returns the percentage of the market that is owned by the company, taking into account the share of the market that is owned, in turn, by the companies controlled by the given company. An operation *find_trust* is defined on market such that it returns a company if its share of that market is greater than the trust limit for the market.

The example above can be modelled by the following set of state axioms SAx_{ACP}. (The implementation of operations is deferred to Chapter 3.)

specific_class(ownership)
  has(ownership,share)*
    generic_class(share)
      is_a(share,*`specialization`*,integer,represented_as)*
        generic_class(integer)
  made_of(ownership,*`aggregation`*,company)*
  made_of(ownership,*`aggregation`*,subsidiary)*

specific_class(company)
  has(company,company_name)*
    generic_class(company_name)
      is_a(company_name,*`specialization`*,string,represented_as)*
        generic_class(string)
  has(company,market_quota)*
  responds_to(company,find_share,market_name,share)

specific_class(subsidiary)
  is_a(subsidiary,*`specialization`*,company,also)*

specific_class(market_quota)
  made_of(market_quota,*`aggregation`*,market)*
  made_of(market_quota,*`aggregation`*,quota)*
    generic_class(quota)
      is_a(quota,*`specialization`*,integer,represented_as)*

specific_class(market)
  has(market,market_name)*
    generic_class(market_name)
      is_a(market_name,*`specialization`*,string,represented_as)*

has(market,trust_limit)*
generic_class(trust_limit)
is_a(trust_limit,*`specialization`*,integer,represented_as)*
responds_to(market,find_trust,market_name,company_name)

The formal sentences above comprise a subset of a set of state axioms SAx_{ACP} that models the described application domain. Operations and queries that are required in this application are addressed in later chapters.

2.3.3 The *Media Storage Services* Example

Stanley Zdonik and David Maier proposed what they have called "a threshold OODM" (and, extending that, "a reference OODM") in (Zdonik and Maier, 1990a). The running example they use is an application which they describe as follows:[3]

> "Consider a company, **Media Storage Services** (MSS), that offers safe storage for computer disks and tapes. MSS has a fleet of vans that pick up and deliver magnetic media of clients throughout the city. MSS wants to build a database to help in planning the best routes for the drivers to take each day. Thus, the database must have information about the road network in the city, the locations of the clients, pick-up and delivery requests, and the planned routes. The database will be used each night by route planners to plan the next day's routes, and will also generate a route map for each driver.
>
> The main entities in the street network are roads, road segments, and intersections. Intersections divide a road into a sequence of road segments. Each road segment has a beginning and ending intersection. Each intersection represents the convergence of two or more road segments. Each road is described as an alternating sequence of road segments and intersections.
>
> Each road has a name and a road type (street, avenue, etc.). Each road segment knows to which road it belongs. Road segments have a shape, for display purposes, which may be either a straight line or a curved arc. Each road segment also has an address range, a side for odd numbered addresses and a direction (two-way or one-way). Intersections have an associated position in coordinates relative to a map. [...]
>
> Each client will have a company name, and address (including the road segment and street address), possibly a standing request for pick-ups at regular times, and a collection of individual requests for pick-ups and deliveries. [...]
>
> A route is much like a road, being represented by a sequence [...] of alternating road segments and intersections. A road has a list of stops, which can involve a pick-up, a delivery, or both. [...]
>
> There is an initial load associated with the route which is what the driver loads into the van in the morning. The route also identifies the vehicle for the route. Vehicles have varying capacities, and a constraint on a route is that the vehicle never be loaded past capacity. We assume that all pick-ups and deliveries are in terms of bins, so the capacity of a vehicle is a number of bins, a stop lists bins to pick up or deliver, and the initial load for a route is a set of bins." [in (Zdonik and Maier, 1990a), p. 4]

Certain requirements in the description above are best modelled as application specific integrity constraints, which in this dissertation require the deductive extensions presented in Chapter 3.

[3] In quotations, omission of text is denoted by the ellipsis mark '[...]'. Similarly, text that is added for readability purposes is also enclosed in square brackets.

The structural aspects of the example above can be modelled by the following set of state axioms SAx_{MSS}:

specific_class(road)
  has*(road,road_name)
    generic_class(road_name)
      is_a*(road_name,`specialization`,string,represented_as)
        generic_class(string)
  has*(road,road_type)
    generic_class(road_type)
      is_a*(road_type,`specialization`,string,represented_as)
  made_of*(road,`ordering`,segment)

specific_class(segment)
  made_of*(segment,`aggregation`,road_segment)
  made_of*(segment,`aggregation`,intersection)

specific_class(road_segment)
  has*(road_segment,in)
  has*(road_segment,segment_shape)
    generic_class(segment_shape)
      is_a*(segment_shape,`generalization`,arc,represented_by)
      is_a*(segment_shape,`generalization`,line,represented_by)
        generic_class(arc)
        generic_class(line)
  has*(road_segment,start)
  has*(road_segment,end)
  has*(road_segment,direction)
    generic_class(direction)
      is_a*(direction,`specialization`,string,represented_as)
  has*(road_segment,address_range)
  has*(road_segment,odd_side)
    generic_class(odd_side)
      is_a*(odd_side,`specialization`,string,represented_as)

specific_class(in)
  is_a*(in,`specialization`,road,kind_of)

specific_class(start)
  is_a*(start,`specialization`,intersection,occurs_as)

specific_class(end)
  is_a*(end,`specialization`,intersection,occurs_as)

specific_class(address_range)
  made_of*(address_range,`aggregation`,initial)
  made_of*(address_range,`aggregation`,final)

```
specific_class(initial)
    is_a*(initial,specialization,address,role)

specific_class(final)
    is_a*(final,specialization,address,role)

specific_class(address)
    made_of*(address,aggregation,road_segment)
    made_of*(address,aggregation,postal_address)
        generic_class(postal_address)
            is_a*(postal_address,specialization,string,represented_as)

specific_class(intersection)
    has*(intersection,incoming_segments)
    has*(intersection,position)

specific_class(incoming_segments)
    made_of*(incoming_segments,association,road_segment)

specific_class(position)
    is_a*(position,specialization,coordinate,represented_as)

specific_class(coordinate)
    made_of*(coordinate,aggregation,x_coordinate)
        generic_class(x_coordinate)
            is_a*(x_coordinate,specialization,point,represented_as)
    made_of*(coordinate,aggregation,y_coordinate)
        generic_class(y_coordinate)
            is_a*(y_coordinate,specialization,point,represented_as)
                generic_class(point)

specific_class(client)
    has*(client,company_name)
        generic_class(company_name)
            is_a*(company_name,specialization,string,represented_as)
    has*(client,address)
    has*(client,standing_requests)
    has*(client,individual_requests)

specific_class(standing_requests)
    made_of*(standing_requests,association,standing_request)

specific_class(standing_request)
    is_a*(standing_request,specialization,pickup_request,kind_of)

specific_class(individual_requests)
    made_of*(individual_requests,association,individual_request)
```

```
specific_class(individual_request)
    is_a*(individual_request,specialization,request,kind_of)

specific_class(pickup_request)
    is_a*(pickup_request,specialization,request,kind_of)

specific_class(delivery_request)
    is_a*(delivery_request,specialization,request,kind_of)

specific_class(request)
    has*(request,time)

specific_class(route)
    has*(route,stops)
    has*(route,initial_load)
    has*(route,vehicle)
    has*(route,driver)
        generic_class(driver)
            is_a*(driver,specialization,string,represented_as)
    made_of*(route,ordering,segment)

specific_class(stops)
    made_of*(stops,ordering,stop)

specific_class(stop)
has*(stop,client)
has*(stop,to_pickup)
has*(stop,to_deliver)

specific_class(to_pickup)
    is_a*(to_pickup,specialization,bins,task)

specific_class(to_deliver)
    is_a*(to_deliver,specialization,bins,task)

specific_class(bins)
    made_of*(bins,association,bin)

specific_class(bin)

specific_class(initial_load)
    made_of*(initial_load,association,bins)

specific_class(vehicle)
    has*(vehicle,capacity)
        generic_class(capacity)
            is_a*(capacity,specialization,integer,represented_as)
                generic_class(integer)
```

The formal sentences above comprise a subset of a set of state axioms SAx_{MSS} that models the described application domain. Part of the information requirements is captured by integrity constraints and intensionally defined data, as described in Chapter 3.

2.3.4 The *library* Example

In their survey of semantic data modelling research (Peckham and Maryanski, 1988), Joan Peckham and Fred Maryanski use a running example which they describe as follows:

> "We develop a conceptual model of the library [LIB] based on the following statements:
>
> 1. A publication entity has title and author attributes.
> 2. Books, journal papers, and conference papers are particular kinds of publication.
> 3. A book entity has sales, topic, author, price, publisher, interest index, and acquisition priority attributes.
> 4. A best-selling book has sales greater than 10,000.
> 5. A writer entity has income, name, and stipend attributes.
> 6. The author attribute of book is of type writer.
> 7. A pub[lisher] entity has name and gross [revenue] attributes.
> 8. A person entity has a name attribute.
> 9. A reviewer entity has a stipend attribute.
> 10. A literary figure is both a writer and a reviewer.
> 11. A borrow[ing event] entity has publication and due date attributes.
> 12. A library entity has acquisitions and ordered pub[lication] list attributes.
> 13. The set of database books contains all books with topic = "database" and also has a total cost attribute.
> 14. The set of artificial intelligence books contains all books with topic = "artificial intelligence".
> 15. The books in the set of good books are identified by the end user.
> 16. The set database artificial intelligence [...]books consists of books in both database book[s] and artificial intelligence book[s].
> 17. All books in the set research group collections are good books.
> 18. 'Wr[itten by]' is a relationship between book and author.
> 19. 'Published[by]' is a relationship between book and pub[lisher].
> 20. 'Review[ed by]' is a relationship between book and reviewer with attributes rating and [review]date.
> 21. A reviewer is a person.
> 22. 'H[eld-by]' is a relationship between book and library."
>
>   [in (Peckham and Maryanski, 1988), p. 164]

The structural aspects of the example above can be modelled by the set of state axioms SAx_{LIB} below.

1. *specific_class(publication)*
    has(publication,author)*
    has(publication,title)*
        generic_class(title)
            is_a(title,*`specialization`*,string,represented_as)*
                generic_class(string)

2. and 3. *specific_class(journal_paper)*
is_a(journal_paper,*`specialization`*,publication,kind_of)*

specific_class(conference_paper)
is_a(conference_paper,*`specialization`*,publication,kind_of) specific_class(book)*
is_a(book,*`specialization`*,publication,kind_of)*
has(book,sales)*
generic_class(sales)
is_a(sales,*`specialization`*,integer,represented_as)*
generic_class(integer)
has(book,topic)*
generic_class(topic)
is_a(topic,*`specialization`*,string,represented_as)*
has(book,author)*
has(book,publisher)*
has(book,price)*
generic_class(price)
is_a(price,*`specialization`*,integer,represented_as)*

has(book,interest_index)*
generic_class(interest_index)
is_a(interest_index,*`specialization`*,integer,represented_as)*
has(book,acquisition_priority)*
generic_class(acquisition_priority)
is_a(acquisition_priority,*`specialization`*,integer,represented_as)*

4. This item in the description is best modelled as a derivation of implicit data, which in this dissertation requires the deductive extensions presented in Chapter 3.

5. *specific_class(writer)*
is_a(writer,*`specialization`*,person,kind_of)*
has(writer,writer_stipend)*
has(writer,income)*

6. *specific_class(author)*
is_a(author,*`specialization`*,writer,kind_of)*

7. *specific_class(publisher)*
has(publisher,publisher_name)*
generic_class(publisher_name)
is_a(publisher_name,*`specialization`*,string,represented_as)*
has(publisher,gross_revenue)*
generic_class(gross_revenue)
is_a(gross_revenue,*`specialization`*,integer,represented_as)*

8. *specific_class(person)*
has(person,name)*
generic_class(name)
is_a(name,*`specialization`*,string,represented_as)*

9. *specific_class(reviewer)*
has(reviewer,reviewer_stipend)*
generic_class(reviewer_stipend)
is_a(reviewer_stipend,*`specialization`*,integer,represented_as)*

10. *specific_class(literary_figure)*
    is_a(literary_figure,*`specialization`*,reviewer,both)*
    is_a(literary_figure,*`specialization`*,writer,both)*
11. *specific_class(borrowing_event)*
    made_of(borrowing_event,*`aggregation`*,publication)*
    made_of(borrowing_event,*`aggregation`*,due_date)*
        generic_class(due_date)
            is_a(due_date,*`specialization`*,integer,represented_as)*
12. *specific_class(library)*
    has(library,acquisitions)*
    has(library,ordered_publications_list)*

    specific_class(acquisitions)
    made_of(acquisitions,*`association`*,publication)*

    specific_class(ordered_publications_list)
    made_of(ordered_publications_list,*`ordering`*,publication)*
13. *specific_class(database_books)*
    has(database_books,total_cost)*
    made_of(database_books,*`association`*,database_book)*

    specific_class(database_book)
    is_a(database_book,*`specialization`*,book,topic)*
14. *specific_class(artificial_intelligence_books)*
    has(artificial_intelligence_books,total_cost)*
    made_of(artificial_intelligence_books,*`association`*,artificial_intelligence_book)*

    specific_class(artificial_intelligence_book)
    is_a(artificial_intelligence_book,*`specialization`*,book,topic)*
15. *specific_class(good_book)*
    is_a(good_book,*`specialization`*,book,quality)*
16. *specific_class(database_artificial_intelligence_book)*
    is_a(database_artificial_intelligence_book,*`specialization`*,artificial_intelligence_book,topic)*
    is_a(database_artificial_intelligence_book,*`specialization`*,database_book,topic)*
17. *specific_class(research_group_collection)*
    made_of(artificial_intelligence_books,*`association`*,good_book)*
18. *responds_to(book,written_by,author)*
19. *responds_to(book,published_by,publisher)*
20. *specific_class(reviewed_by)*
    has(reviewed_by,rating)*
        generic_class(rating)
            is_a(rating,*`specialization`*,integer,represented_as)*
    has(reviewed_by,review_date)*
        generic_class(review_date)
            is_a(review_date,*`specialization`*,integer,represented_as)*
    made_of(reviewed_by,*`aggregation`*,book)*
    made_of(reviewed_by,*`aggregation`*,reviewer)*
21. *is_a*(reviewer,*`specialization`*,person,also)*

22. *specific_class(held_by)*
   made_of(held_by,*`aggregation`*,book)*
   made_of(held_by,*`aggregation`*,library)*

The formal sentences above comprise a subset of a set of state axioms SAx_{LIB} that models the described application domain. Part of the information requirements is captured by integrity constraints and intensionally defined data, as described in Chapter 3.

Summary of Chapter 2

This chapter had the main purpose of deriving a logical theory from the mathematical theory developed in Chapter 1. A mathematical theory allows databases to be viewed model-theoretically, i.e., as interpretation structures defining the truth value of queries and integrity constraints. The notion of a deductive database, however, is primarily a proof-theoretical one. This means that it is grounded on the notion of a formal proof, with which mathematical theories are not necessarily equipped.

This chapter had three sections.

Section 2.1 logically reconstructed the OODM as a class of applied first-order calculi, i.e., a class of first-order theories with proper axioms. The language OLL of such calculi was shown to be derivable from the language of the OODM. Then, the typical characterization and the mathematical axiom of the OODM were formalized as sentences of OLL. This allows an OODB to be viewed proof-theoretically. This result constitutes the second contribution of this dissertation.

Section 2.2 introduced a diagramming notation that allows the more important elements of ODTs to be expressed more concisely.

Section 2.3 presented as ODTs several paradigmatic example applications found in the literature on data modelling. This allowed the expressiveness of ODTs to be gauged and the more abstract ideas to be extensively exemplified.

The OODM has now been logically reconstructed. Thus, formal proofs can be carried out both *about* and *within* OODBs; in other other words, both at the level of the metatheory and at the level of the theory. However, the deductive apparatus available is that of classical FOL. Since FOL is an intractable logical system, an automated theorem-proving approach to queries, constraints and derived data is not, in principle, practical.

Automated proof procedures are extremely inefficient over unrestricted first-order clauses. Awareness of this barrier was behind the rise to prominence of the Horn case of first-order clauses, for which the inefficiency of resolution-based proof procedures is reduced to an acceptable level, witness the relative success of the Prolog language. In short, to make the adoption of an automated theorem-proving approach to queries, constraints and derived data a practical possibility, one has to work within more restricted logical systems, either acceptably efficient (as HCL) or tractable (as function-free HCL).

In Chapter 3 it is shown that every ODT can be cast as a set of function-free Horn clauses, thereby characterizing OODBs as a class of function-free logic programs.

This chapter provided a logical reconstruction of the OODM defined in Chapter 1 that transforms the OODBs induced by it into ODTs, thereby yielding a specification of the highlighted arc in Figure 2.1, viz.,

$$OODB \mapsto_9 ODT.$$

These arcs constitute the second step in the solution of the principal question to which this dissertation seeks to provide an answer. The third and final step, viz., the extension of ODTs, first into DODTs, and then into DOOTs, is the subject of Chapter 3, thus concluding the solution.

Chapter 3

The Derivation of Deductive Object-Oriented Databases from ODTs

The purpose of this chapter is to extend ODTs, first into DODTs, and then into DOOTs, while showing that DOOTs indeed map into DDTs.

The chapter yields a specification of the arcs

$$ODT \mapsto_{10} DODT \mapsto_{11} DOOT \mapsto_{12} DDT$$

in the object-oriented space of theories defined in the Introduction, thereby completing the desired path from the source to the sink of the space. Hence, the chapter concludes the solution of the central question addressed by this dissertation. Figure 3.1 reproduces that space, with the arcs that are treated in this chapter highlighted.

In Chapter 2 the logical reconstruction of the OODM defined in Chapter 1 was carried out. A mathematically rigorous definition of OODBs and a first-order logical calculus that corresponds to it are now available, thereby allowing OODBs to be viewed proof-theoretically, as ODTs. This is a reassuring result, but only to the extent that it demonstrates that recourse to second-order theories has been averted. However, the result is not enough to legitimize a view of query answering and integrity constraint checking as automated theorem proving. FOCTs, the clausal version of the logical theory that ODTs presume, are intractable logical systems.

For both theoretical and practical reasons, a more desirable result would be that ODTs admit of a more restricted clausal form. The first target in this context would be for ODTs to be expressible in *Horn* (also known as *definite*) *clauses*. For theoretical purposes, this would ensure that the metalogic of ODTs is easier to develop, and that ODTs are equipped with a simpler and more flexible semantic characterization than FOCTs. It would also imply that ODTs share, to a large extent, the invaluable corpus of results obtained in the development of the theory of logic programs (Apt, 1990; Das, 1992; Lloyd, 1987) since the mid-1970s. Finally, and more importantly, the fact that HCTs are acceptably efficient logical systems would make a view of query answering and integrity constraint checking as automated theorem proving a practical possibility for ODTs.

This chapter demonstrates that, indeed, ODTs can be cast as HCTs, and hence that their proof theory can be automated. Even more desirably, DODTs — the clausal form of ODTs — are function-free, i.e., DODTs can be viewed as Datalog programs. Therefore, DODTs are tractable logical systems, and not just acceptably efficient ones. This is an important result, since it precisely situates DODTs, and hence ODTs, within the theory of DRDBs, and by so doing, defines precisely how OODBs and RDBs are related in logical terms.

This very positive result, as one might expect, is achieved by design, in the sense that the OODM

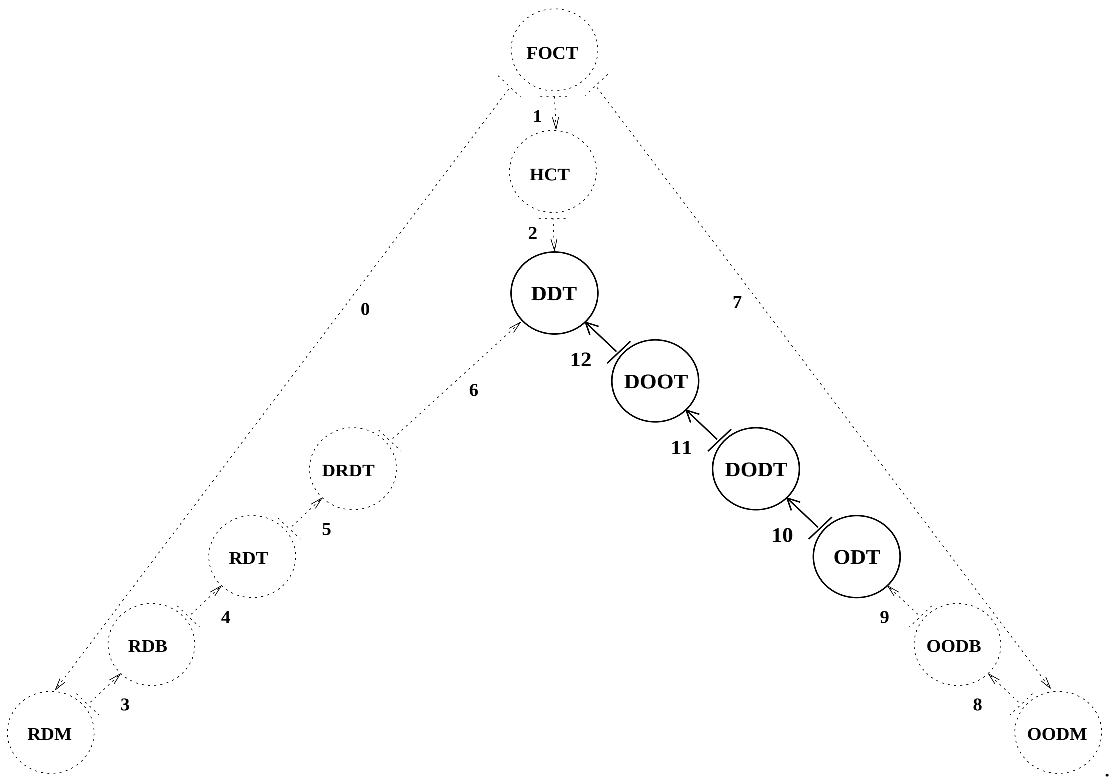


Figure 3.1: From ODTs to DODTs, DOOTs and DDTs in the Object-Oriented Space of Theories

was so defined in Chapter 1 as not to require more expressive logical formalisms — since the price for greater expressiveness is paid both in metalogical complexity and computational inefficiency, as discussed in some detail in the Introduction. Formally defining an OODM is a comparatively easier task than formally defining an OODM that can be reconstructed as a tractable logical system. Clearly, the latter task faces much stricter constraints, insofar as the resulting OODM must have the metatheoretical properties that make viable a proof-theoretical view of OODBs.

Among the characteristics of the OODM that prove crucial in achieving the result above are the reification of classes, operations and methods, and not only of objects, as one would normally expect. This leads to an explicitly intensional view of collections (i.e., collections may be distinct even when they have the same members) rather than the extensional one of standard axiomatic set theory (in which collections that have the same members are indistinguishable). The OODM also imposes that all references, e.g., to values of properties and to components of constructions, are denoted by names. For example, rather than directly assigning multiple values to a property, one assigns to the property a reference to a complex object in whose state all the multiple values are referenced, thereby imposing an intermediate level between a property and its values. The motivation for this is both to keep the language function-free and to retain its first-order semantics (i.e., there are no variables that denote sets of individuals).

Also noteworthy, and of crucial import to the comparison with related work carried out in Chapter 6, is the fact that objects (as well as classes, operations and methods) are uniformly represented in the logical system as theories (i.e., sets of sentences), rather than terms. The motivation, again, is to avoid the introduction of function symbols in the alphabet, and thereby relinquish the finiteness of the underlying Herbrand domain.

At the end of this chapter, besides possessing a mathematically rigorous definition of OODBs and the corresponding logical calculus, which allows them to be viewed proof-theoretically, it is seen that OODBs are, more specifically, a class of function-free logic programs. This means that, besides facts (i.e., ground, unconditional assertions), one can also use deductive rules (nonground, nonunit definite clauses) to describe, and constrain, application domains. This opens the way for the expression of both implicit information and application specific integrity constraints by means of deductive rules. The resulting formalism is finally extended to provide a logic-based account of *encapsulation* as well as those features of an object-oriented style of message-based interaction known as *overloading, overriding* and *late binding.*

This chapter consists of three sections.

Section 3.1 demonstrates that an OODB, cast as an ODT, can be expressed as a set of function-free Horn clauses, thus transforming ODTs into DODTs. The possibility of including deductive rules as axioms of application domains increases the expressiveness of OODBs in ways that parallel the RDB-DRDB case.

Section 3.2 shows how certain dynamic aspects of object-orientedness can be captured in a logical setting, thus extending DODTs into DOOTs, which are the constructs whose characterization is sought.

Section 3.3 shows how DOOTs and DDTs are formally related.

3.1 From ODTs to DODTs: Equipping ODTs with Efficient Proof Procedures

This section contains a demonstration that an OODB can be expressed as a set of function-free Horn clauses. The addition of rules allows the expression of application specific integrity constraints and intensionally defined data, thereby extending ODTs into DODTs.

3.1.1 Translating Application Independent Axioms into Clausal Form

The first step in the extension of an OODB with deduction is to translate the set $OMAx$ of OLL sentences, which axiomatizes the application independent part of every OODB into clausal form. The algorithm that achieves this has one step for each conjunct of $OMAx = T \wedge C \wedge E$, and is defined as follows:

Step 1: The algorithm defined in (Lloyd and Topor, 1984) that transforms a universal closure of the form

$$\forall(F \Rightarrow L),$$

where F is any FOL formula and L is an atom, into a set of Horn clauses is used as follows.

1. Given the sort axiom

$$T \equiv \forall\phi_1 \wedge \ldots \wedge \forall\phi_{20},$$

where ϕ_t denotes formula (t) in Appendix C, and $1 \leq t \leq 20$, a set $\mathcal{P}_T(OMAx)$ of nullary predicate symbols of the form

$$violation\_t,$$

is introduced.

2. For each conjunct ϕ_t of T the sentence

$$\forall(\neg\phi_t \Rightarrow violation\_t)$$

is input into the Lloyd-Topor algorithm, yielding a set of Horn clauses that, in logic programming parlance, defines $violation\_t$. This means that each conjunct in T is viewed as an application independent integrity constraint (see, e.g., [(Das, 1992), Chs. 12–13, pp. 275–313] for an exposition and comparison of constraint checking methods that could be used for clauses with the form above).

As an example, consider formula (2) of Appendix C:

$$is\_a(x,y) \Rightarrow (class(x) \wedge class(y))$$

It stipulates that inclusion hierarchies can only involve classes. The Lloyd-Topor algorithm applied to:

$$\forall(\neg(is\_a(x,y) \Rightarrow (class(x) \wedge class(y)) \Rightarrow violation\_2)$$

yields the following set of clauses:

```
violation_2 ←
    is_a(B,C),
    ~ class(B).
violation_2 ←
    is_a(C,B),
    ~ class(B).
```

Let $clausal(T)$ denote the set of Horn clauses that is the union of the set of clauses corresponding to each conjunct of T. Every clause in $clausal(T)$ is a rule, i.e., a Horn clause with exactly one negative literal, viz., $violation\_t$. (The reader may verify this by inspecting Section D.1 in Appendix D.)

Step 2: The well known Skolem algorithm (Das, 1992; Kowalski, 1979a) that translates a FOL sentence into the Skolem standard form traditionally used by refutation procedures is applied to each conjunct of the completion axiom

$$C \equiv \forall\phi_1 \wedge \ldots \wedge \forall\phi_{12}$$

where ϕ_p denotes formula (p) in Appendix C, and $1 \leq p \leq 12$.

The application of the Skolem algorithm to each conjunct in C yields a set of Horn clauses. As an example, consider formulas (1), (2), (3), and (4) of Appendix C:

$$\begin{aligned}(is\_a^*(x,y,u,v) \wedge (y = \texttt{specialization} \vee y = \texttt{partition})) &\Rightarrow is\_a(x,u)\\ (is\_a^*(x,y,u,v) \wedge y = \texttt{generalization}) &\Rightarrow is\_a(u,x)\\ class(x) &\Rightarrow is\_a(x,x)\\ (is\_a(x,y) \wedge is\_a(y,z)) &\Rightarrow is\_a(x,z)\end{aligned}$$

They define how is_a is induced by the declaration that one class specializes, or generalizes, or partitions another. These, in conjunction with a law of reflexivity and a law of transitivity, determine the full extension of is_a from the declared information in is_a^*. The Skolem algorithm applied to the universal closure of the formulas above yields the following set of clauses:

```
is_a(A,B) ←                  is_a(A,A) ←             is_a(A,B) ←
    C==specialization,           class(A).               is_a(A,C),
    is_a*(A,C,B,D).                                      is_a(C,B).
is_a(A,B) ←
    C==partition,
    is_a*(A,C,B,D).
is_a(A,B) ←
    C==generalization,
    is_a*(B,C,A,D).
```

Note that '==' denotes identity, and can be implemented by term unification, which, in turn, is much simpler in the present context of a function-free language than is the case in HCL.

Let $clausal(C)$ denote the set of Horn clauses that is the union of the set of clauses corresponding to each conjunct of C. Every clause in $clausal(C)$ is, again, a rule. (The reader may verify this by inspecting Section D.2 in Appendix D.)

Step 3: In close analogy with the treatment of the sort axiom, the Lloyd-Topor algorithm is used again as follows.

1. Given the exception axiom
$$E \equiv \forall\phi_{13} \wedge \ldots \wedge \forall\phi_{55}$$
where ϕ_n denotes formula (n), in Appendix C, and 13$\leq$ n $\leq$55, a set $\mathcal{P}_E(OMAx)$ of nullary predicate symbols of the form
$$violation\_m,$$
is introduced, where $m = n + 20$.
2. For each conjunct ϕ_n of E the sentence
$$\forall(\neg\phi_n \Rightarrow violation\_m)$$
is input into the Lloyd-Topor algorithm, yielding a set of Horn clauses that defines $violation\_m$. Hence, each conjunct in E is also viewed as an application independent integrity constraint.

In contrast with T, certain conjuncts of E contain existentially quantified variables, in which case the Lloyd-Topor algorithm introduces new predicate symbols whose arity is equal to the number of variables in the scope of the existential quantifier. In this dissertation, these predicate symbols comprise a set $\mathcal{P}_{E'}(OMAx)$ each of whose elements is assigned the form

$$also\_m$$

where $m = n{+}20$, when ϕ_n is the conjunct whose translation causes $also\_m$ to be introduced.

As an example, consider formula (30), and (33) of Appendix C:

$$(is\_a(x,y) \wedge is\_a(y,x)) \Rightarrow x = y$$

$$(is\_a(x,y) \wedge is\_a(x,z) \wedge \neg(y = z)) \Rightarrow \exists w(is\_a(y,w) \wedge is\_a(z,w))$$

They stipulate that is_a is antisymmetric and weakly connected. The Lloyd-Topor algorithm applied to:

$$\forall(\neg((is\_a(x,y) \wedge is\_a(y,x)) \Rightarrow x = y) \Rightarrow violation\_50)$$

$$\forall(\neg((is\_a(x,y) \wedge is\_a(x,z) \wedge \neg(y = z)) \Rightarrow \exists w(is\_a(y,w) \wedge is\_a(z,w))) \Rightarrow violation\_53)$$

yields the following set of clauses:

```
violation_50 ←            violation_53 ←            also_53(X,Y,Z) ←
    (is_a(B,C),               (is_a(D,E),               is_a(X,Y),
    is_a(C,B)),               is_a(D,X),                is_a(Z,Y).
    ~ B==C.                   ~ E==X),
                              ~ also_53(E,Y,X).
```

in which the existential quantification in (33) caused the introduction of a ternary predicate of the form *also_m*, viz., *also_53*

Note that, if the input formula contains existential quantifiers, the Lloyd-Topor algorithm yields a clause containing in its body a negated subgoal in which a variable occurs that does not also occur in a positive goal elsewhere in the body of the clause. This means that the Lloyd-Topor algorithm yields non-range-restricted clauses.

If an inflationary semantics [(Abiteboul et al., 1995), Ch. 14, p. 355 ff.] is adopted, the existence of non-range-restricted clauses has no impact on the semantics of DOODs, since the inflationary semantics of Datalog with negation is identical whether clauses are range-restricted or not (Abiteboul et al., 1995). Intuitively, this is because the domain closure assumption and the guaranteed finiteness of the domain ensure that all possible bindings can be computed for every variable in every clause.

However, even for a top-down approach to query evaluation such as SLDNF-resolution no harm is caused in the specific case of the conjuncts of the exception axiom, insofar as the output of the algorithm is such that the soundness of the SLDNF-resolution is not compromised. This is so because the standard safeness condition to ensure the soundness of SLDNF-resolution that only ground negative subgoals are allowed to proceed can be weakened as follows.

Let a non-ground negative subgoal be allowed to proceed (i.e., to be selected for resolution). If it succeeds then the proof procedure can continue normally (i.e., soundness is not compromised), but if it fails then one must ensure that no bindings were made to variables in the top-level goal of the corresponding refutation. If no such binding was made, then the negative subgoal is allowed to fail and the proof procedure can continue normally; only if any such binding *was* made is the proof procedure abnormally terminated because soundness is compromised.

Now, in the particular case of the conjuncts of the exception axiom, whenever the Lloyd-Topor algorithm yields a rule containing a non-range-restricted subgoal the following proposition holds: the case in which the adoption of the weakened safeness condition would lead to unsoundness (and hence cause an abnormal termination) of the SLDNF-resolution proof procedure never occurs.

This is because the success or failure of the non-ground negative subgoal never binds variables in the top-level goal of the corresponding refutation. Note that, using the standard leftmost-literal selection rule of SLDNF-resolution, whenever the use of the weakened condition allows a non-ground negative subgoal to proceed, the only variables occurring in the clause that are still unbound are exactly the variables that only occur in the said subgoal. In other words, the success or failure of the subgoal cannot bind any other variable in the clause, and hence the weakened safeness condition suffices to ensure the soundness of SLDNF-resolution over the clauses yielded by the Lloyd-Topor algorithm for the conjuncts of the exception axiom. [(Lloyd, 1987), Ch. 3, pp. 94–95] contains a fuller discussion of the fact that the soundness of SLDNF-resolution is not affected by the weakened safeness condition described above.

Let $clausal(E)$ denote the set of Horn clauses that is the union of the set of clauses corresponding to each conjunct of E. Every clause in $clausal(E)$ is a rule, i.e., a Horn clause with exactly one negative literal, viz., *violation_m* or *also_m*. (The reader may verify this by inspecting Section D.3 in Appendix D.)

Since both the Lloyd-Topor and Skolem algorithms terminate in a finite number of steps, the three steps above also define an algorithm whose input is the object modelling axiom

$$OMAx = T \wedge C \wedge E$$

and whose output is the set of function-free Horn clauses

$$clausal(OMAx) = clausal(T) \cup clausal(C) \cup clausal(E).$$

given in full in Appendix D.

Note that $clausal(OMAx)$ is fixed for every OODB. In practical implementations, it is probably worthwhile to try to devise ways of taking its clauses into account which are more efficient than their explicit addition to the set of clauses that is input to the proof procedure. In other words, like the representation axioms RAx, the $OMAx$ conjunct of the database axiom $DBAx$ could conceivably be handled in the implicit part of the axiom system. In this dissertation, however, $clausal(OMAx)$ is assumed to be explicit. This is done because it makes it easier to understand the part played by $OMAx$ in reaching a solution to the characterization problem addressed in this dissertation.

It has been seen that $clausal(OMAx)$ is a set of function-free Horn clauses. Note that, because the alphabet of OLL is function-free, so is SAx. Since for any OODB, the state axioms SAx, by definition, contain only ground atomic sentences, any axiom in any SAx is a (function-free) ground fact. In other words, the state axioms are already in clausal form. More formally, for every application domain declared as a set of state axioms SAx,

$$SAx = clausal(SAx),$$

and every clause in $clausal(SAx)$ is a function-free fact.

Since

$$DBAx = SAx \cup OMAx,$$

and since, for any application domain, $clausal(SAx)$ and $clausal(OMAx)$ only contains function-free Horn clauses so does

$$clausal(DBAx) = clausal(SAx) \cup clausal(OMAx).$$

Therefore, to every ODT $DBAx$, there corresponds a set of function-free Horn clauses $clausal(DBAx)$.

Let $\mathcal{P}'(OMAx) = \mathcal{P} \cup \mathcal{P}_T(OMAx) \cup \mathcal{P}_E(OMAx) \cup \mathcal{P}_{E'}(OMAx)$, where $\mathcal{P}$ is as defined in Section 2.1.1, and $\mathcal{P}_T(OMAx)$, $\mathcal{P}_E(OMAx)$ and $\mathcal{P}_{E'}(OMAx)$ are as defined in the preceding paragraphs. Then, the language $\mathcal{CL}(DBAx)$ of $clausal(DBAx)$ is wholly determined by the alphabet

$$\mathcal{CA}(DBAx) = \langle \mathcal{C}(DBAx), \mathcal{P}'(OMAx), \mathcal{V}(DBAx) \rangle$$

where $\mathcal{C}(DBAx)$ and $\mathcal{V}(DBAx)$ are, respectively, the set of constant symbols and individual variables occurring in $DBAx$. Since $\mathcal{P}'(OMAx)$ is fixed for all application domains, $\mathcal{CA}$ is determined by $\mathcal{C}(DBAx)$, which, in turn, is determined by SAx.

The significance of this result can better be gauged by recalling that an ODT is the logical theory of an OODB, and hence, the result above can also be expressed as follows: to every OODB there corresponds one, and only one, set $clausal(DBAx)$ of function-free Horn clauses. This means that OODBs, as defined in Chapters 1 and 2, can be extended with deduction in a way that closely parallels the way in which RDBs are extended with deduction. The details of this extension are the subject of the remaining sections of this chapter.

$clausal(OMAx)$ models all the application independent semantics captured by the OODM that can be understood statically. Specifically, $clausal(OMAx)$ can be seen as defining the implicit information content and those integrity constraints that hold *for all application domains*. This fact is best perceived in the light of the following facts about $clausal(DBAx)$:

1. $\mathcal{CL}$, the language of $clausal(DBAx)$, is Datalog with negation and an identity predicate.

2. All elements in $clausal(SAx)$ are facts.

3. $clausal(SAx)$ is an *extensional database* (EDB), i.e., the extensional part of a DDB (Ceri et al., 1990b; Das, 1992).

4. All the elements in $clausal(OMAx)$ are rules.

5. All rules are *safe*, i.e., all variables appearing in the head also appear in the body and all non-ground negative literals can be evaluated by SLDNF-resolution.

6. $clausal(OMAx)$ is *stratified* (Ceri et al., 1990b).

7. $clausal(OMAx)$ is an *intensional database* (IDB), i.e., the intensional part of a DDB (Ceri et al., 1990b; Das, 1992), and hence, the rules in $clausal(OMAx)$ that declaratively formalize the axiom of the OODM can be seen to express intensionally defined data and integrity constraints that are valid over any application domain declared by SAx.

The next step is to use $\mathcal{CL}$ and express implementation of operations, application specific implicit information and integrity constraints as rules that are then incorporated into the IDB characterized by $clausal(OMAx)$ in the way described above.

3.1.2 Expressing Operation Implementations with Deductive Rules

A commitment to the representation of operation implementations has been postponed since Chapters 1 and 2. $\mathcal{CL}$ is a language endowed with an operational semantics, and hence suitable for expressing operation implementations that do not require changing the state of the database, i.e., the set of axioms $clausal(DBAx)$. This subsection uses this property of $\mathcal{CL}$ to address that postponed issue.

Note that if the operation is such that it produces side effects and, as a consequence, changes the state of the database, then it cannot be implemented in $\mathcal{CL}$ as defined below. Naturally, $\mathcal{CL}$ can be extended to handle updates in exactly the same way as Datalog can, albeit incurring the same penalties. However, in this dissertation this extension path is not pursued. In Chapter 4, a fully implemented DOOD system is described in which updates are the province of an imperative, computationally complete language that pairs up, and is tightly integrated with, a user-level version of $\mathcal{CL}$. The two languages share the same type system and the same evaluation paradigm, thereby providing comprehensive control and update facilities without incurring typical impedance mismatches.

Axioms (50) and (51) of Appendix C stipulate that to every declared operation interface there corresponds exactly one operation implementation. This stipulation is satisfied by associating with each ground fact of the form

$$responds\_as\_to(x, \bar{x}, y, z_1, \ldots, z_n)$$

(i.e., an inferred interface), exactly one ground fact of the form

$$implemented\_in\_by(\bar{x}, y, z_1, \ldots, z_n, w),$$

(i.e., a method declaration), where $\bar{x}$ is the class on which the operation y is defined with argument classes $z_1, \ldots, z_n$, and w denotes the method that implements the interface.

Now, to each ground atom of the form $implemented\_in\_by(\bar{x}, y, z_1, \ldots, z_n, w)$ one can associate a set of rules all of whose elements have as head an atom of the form

$$responds\_with(x', w, z_1', \ldots, z_n'),$$

where $x', z'_1, \ldots, z'_n$ denote objects that are instances of the classes denoted by $\bar{x}, z_1, \ldots, z_n$, respectively, as required by axiom (52) of Appendix C. Such rules constitute a *definition* of the method w whose *declaration* is $implemented\_in\_by(\bar{x}, y, z_1, \ldots, z_n, w)$. In other words, the rules are seen as a *procedure*, in the sense of (Kowalski, 1979b), and define $responds\_with(x', w, z'_1, \ldots, z'_n)$ insofar as an operational semantics is provided for the clause set.

As an example, consider the *land parcel selection* application described in Subsection 2.3.1. Let an operation *has_easy_access* be defined on *land_parcel*s, where a land parcel is said to have easy access only if it is located beside a motorway. Firstly, the interface of the operation *has_easy_access* is declared by asserting the fact

$$responds\_to(land\_parcel, has\_easy\_access).$$

Note that the operation has no arguments other than the land parcel to which requests are addressed. Secondly, the implementation of *has_easy_access* is declared by the fact

$$implemented\_in\_by(land\_parcel, has\_easy\_access, has\_easy\_access').$$

Finally, the implementation of *has_easy_access'* is defined by the following procedure

```
responds_with(Land_Parcel,has_easy_access') ←
    instance_of(Location,location),
    component(Location,land_parcel,Land_Parcel),
    component(Location,road,Road),
    property(Road,road_type,Road_Type),
    Road_Type == "motorway".
```

Let $INT\_DEC(\mathcal{CL})$ denote the set of all operation interface declarations, $IMP\_DEC(\mathcal{CL})$ the set of all operation implementation declarations, and $IMP\_DEF(\mathcal{CL})$ the set of all operation implementation definitions. When the context is clear, these will be written as INT_DEC, IMP_DEC and IMP_DEF, respectively.

Figure 3.2 depicts how each element of the set $INT\_DEC \subset SCH$ of operation interface declarations (i.e., ground facts of the form $responds\_as\_to(x, \bar{x}, y, z_1, \ldots, z_n)$) is mapped into an element of the set $IMP\_DEC \subset INS$ of operation implementation declarations (i.e., ground facts of the form $implemented\_in\_by(x, y, z_1, \ldots, z_n, w)$), and how each of these is mapped to an element of the set $IMP\_DEF \subset INS$ of operation implementation definitions, each of which is a set of rules. To each operation interface declaration there corresponds exactly one operation implementation declaration, and vice-versa. To each operation implementation declaration there corresponds exactly one operation implementation definition, and vice-versa. The way in which the overriding and late binding of methods to requests is handled is discussed in Section 3.2.

3.1.3 Expressing Application Specific Integrity Constraints with Deductive Rules

Rules expressing application specific integrity constraints have the same syntax and semantics as the rules resulting from the translation of the sort and exception axioms described in Subsection 3.1.1, except that in the latter two sets no application specific constants occur.

Consider, for instance, the *Media Storage Services* example of Subsection 2.3.3. One way to represent the constraint that the admissible values for *road type* are enumerated in the specification is to write:

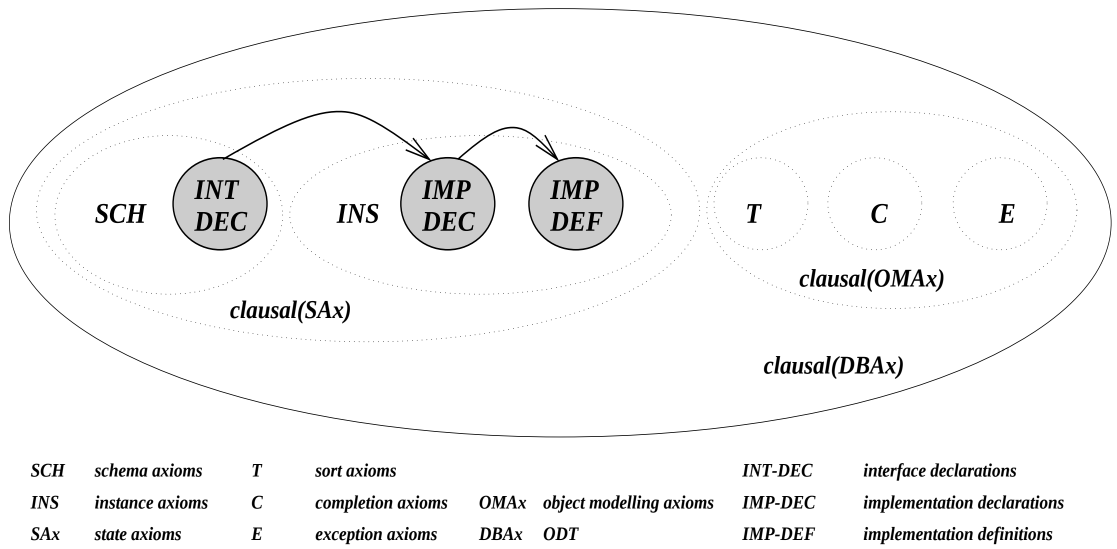


Figure 3.2: Relating Operation Interfaces to Operation Implementations

violation_ROAD_TYPE_VALUE ←
  of_type(Road_Type,road_type),
  ∼(Road_Type == "road"),
  ∼(Road_Type == "avenue"),
  ∼(Road_Type == "boulevard"),
  ∼(Road_Type == "lane"),
  . . . .

where $violation\_ROAD\_TYPE\_VALUE$ is an arbitrary predicate symbol introduced by the programmer into $\mathcal{CL}$.

Let the set of all such predicates in a given set $DBAx$ be denoted by $\mathcal{P}^\bullet(DBAx)$. Then, application specific integrity constraints redefine the alphabet

$$\mathcal{CA}(DBAx) = \langle \mathcal{C}(DBAx), \mathcal{P}'(OMAx), \mathcal{V}(DBAx) \rangle$$

of $\mathcal{CL}$ into

$$\mathcal{CA}^\bullet(DBAx) = \langle \mathcal{C}(DBAx), (\mathcal{P}(OMAx) \cup \mathcal{P}^\bullet(DBAx)), \mathcal{V}(DBAx) \rangle$$

and $\mathcal{CA}^\bullet$ induces a language $\mathcal{CL}^\bullet$ in the standard way.

Further examples of integrity constraints, for the admissible values of *direction* and of *odd side*, are:

violation_DIRECTION_VALUE ←
  of_type(Direction,direction),
  ∼(Direction == "one_way_start_to_end"),
  ∼(Direction == "one_way_end_to_start"),
  ∼(Direction == "two_way").

violation_ODD_SIDE_VALUE ←
  of_type(Odd_Side,odd_side),
  ∼(Odd_Side == "left"),
  ∼(Odd_Side == "right").

The following two integrity constraints presume that comparison predicate symbols, e.g., $<$, $>$, $=$, etc., are introduced into $\mathcal{CL}^\bullet$ under the usual safety conditions (Ceri et al., 1990b), viz., all variables appearing in a comparison predicate also appear in a noncomparison predicate. The next integrity constraint ensures that the $From$ value is not larger than the To value in an address range:

```
violation_ADDRESS_RANGE_EXTREMES ←
    instance_of(Address_Range,address_range),
    component(Address_Range,from,From),
    component(Address_Range,to,To),
    From > To.
```

The next few rules hypothesize the existence of external predicates, e.g., aggregate (such as **count**), constructor (such as **concatenate**), numerical (such as **add**), system dependent (such as **today**).

Note, however, that such external predicates are *not* envisaged in the characterization of DODTs contributed in this dissertation, i.e., $\mathcal{CL}^\bullet$ cannot express such predicates. They are only used in the rules that follow as an illustration of the possibilities that arise when $\mathcal{CL}^\bullet$ is integrated with other, more expressive languages.

The remarks above highlight the fact that $\mathcal{CL}^\bullet$ is far from self-sufficient, in the sense of being capable of expressing all queries. This is intentional. The strategic level justification not to extend $\mathcal{CL}^\bullet$ with any predicate of the above kind is to keep the metalogic of $\mathcal{CL}^\bullet$ as simple as that of Datalog with negation. The main design objective here is to make it possible for $\mathcal{CL}^\bullet$ queries to be extensively optimized and tractably evaluated using very similar techniques to those used in recent years for DRDBs.

This lack of expressiveness has to be seen in its proper context. In the overall approach, $\mathcal{CL}^\bullet$ is thought of as but one component in an architecture that also comprises a computationally complete imperative language with which it can be fully integrated. This two-language approach has been pioneered by the Glue-Nail DRDB system (Derr et al., 1994). In this approach, it is left to the imperative language to cater not only for external predicates such as the ones mentioned above, but also for state transitions.

Chapter 4 describes in detail, a fully implemented system in which this two-language architecture is realized for the DODTs described in this chapter. In the implemented system the lack of self-sufficiency in (the surface form of) $\mathcal{CL}^\bullet$ is amply compensated by the imperative language with which it is tightly integrated.

It is beyond the scope of this dissertation to discuss the details of how this integration is achieved and how query evaluation of the integrated language is implemented. Some additional information is given in Chapter 4 and full details can be found in (Dinn et al., 1995). Suffice it to say that, if an external predicate occurs in the body of a clause then query evaluation terminates if and only if the evaluation of the external predicate (which is carried out by a different interpreter) terminates. If, as is typically the case, one assumes the imperative language in which the external predicates are implemented to be computationally complete, this property is, of course, not decidable. On the other hand, it is always possible to ensure the declarativeness of queries because it is possible to proscribe from the bodies of clauses an external predicate that side-effects the global state (system data included). An algorithm is described in (Goñi, 1995) that detects at compile time whether an external predicate side-effects the global state. Finally, the type system and the integrity constraints are shared between the two interpreters, as described in more detail in Chapter 4.

Bearing in mind the cautionary remarks above, more examples are now presented. The next example of integrity constraint takes the form of a rule that characterizes a violation if the capacity of the vehicle assigned to a route does not suffice for the initial load, in number of bins, with which that route starts off:

violation_VEHICLE_CAPACITY ←
instance_of(Route,route),
property(Route,vehicle,Vehicle),
property(Vehicle,capacity,Capacity),
property(Route,initial_load,Initial_Load),
count(Initial_Load, Number_of_Bins),
Number_of_Bins > Capacity.

Finally, consider again the *library* example of Subsection 2.3.4. There it was said that item 6 of the specification, viz., *"The author attribute of book is of type writer."*, is best modelled as an integrity constraint. This now follows:

violation_BOOK_AUTHOR ←
instance_of(Book,book),
property(Book,author,Author),
∼ instance_of(Author,writer)

Let the set of all application specific integrity constraints be denoted by $IC(\mathcal{CL}^\bullet)$. When the context is clear, this will be written as IC.

3.1.4 Expressing Implicit Information with Deductive Rules

Rules expressing application specific implicit information have the same syntax and semantics as the rules resulting from the translation of the completion axiom described in Subsection 3.1.1, except that in the latter set no application specific constants occur.

Consider, for instance, the *library* example of Subsection 2.3.4. There it was said that item 4 of the specification, viz., *"A best-selling book has sales greater than 10,000."*, is best modelled as a derivation of implicit data using a deductive rule. This now follows:

instance_of(Book,best_selling_book) ←
instance_of(Book,book),
property(Book,sales,Sales),
Sales > 10000

The rule above defines the extension of the class of *best_selling_book* in terms of the value of the *sales* property of the instances of *book*, and both query evaluation and integrity checking reflect this fact.

Many examples of intensionally defined data arise in applications. Two other examples, based on Subsections 2.3.1 to 2.3.4 now follow. In the *land parcel selection* example, a rule such as:

property(Land_Parcel,land_parcel_name,Land_Parcel_Name) ←
instance_of(Location,location),
component(Location,land_parcel,Land_Parcel),
property(Land_Parcel,owner,Owner),
property(Owner,owner_name,Owner_Name),
component(Location,road,Road),
property(Road,road_name,Road_Name),
concatenate(Owner_Name,Road_Name,Land_Parcel_Name)

intensionally defines that the *land_parcel_name* property of every *land_parcel* object is the concatenation of the names of the land parcel's owner and of the road the land parcel is located on. Under similar circumstances as the aggregate function **count** discussed above, the external constructor function **concatenate** (on strings) is also assumed.

Finally, in the *library* example, one can set the date that a borrowed book is due to seven days from the current date:

property(Borrowing,due_date,Due_Date) $\leftarrow$
 instance_of(Borrowing,borrowing_event),
 today(Today),
 add(Today,7,Due_Date)

where **add** and **today** are external functions with the expected meaning.

Let the set of all application specific rules characterizing intensionally defined data be denoted by $ID(\mathcal{CL}^\bullet)$. When the context is clear, this will be written as ID.

3.1.5 The Relationship between ODTs and DODTs

The results of this section equip OODBs with expressive means that parallel those possessed by DRDBs. For this reason, given an ODT in clausal form $clausal(DBAx)$ and the application specific axiom sets ID and IC, one can characterize the DODT they give rise to as the set of deductive database axioms

$$DDBAx = clausal(DBAx) \cup IC \cup ID.$$

More precisely, the extensional, intensional and constraint parts of a deductive object database $DODB$ (denoted below by EDB, IDB and CDB, respectively) are characterized by the following sets of axioms, respectively:

$$\begin{aligned} EDB &= SCH \cup (INS \setminus IMP\_DEF) \\ IDB &= clausal(C) \cup IMP\_DEF \cup ID \\ CDB &= clausal(T) \cup clausal(E) \cup IC \\ DODB &= EDB \cup IDB \cup CDB \end{aligned}$$

This is depicted in Figure 3.3.

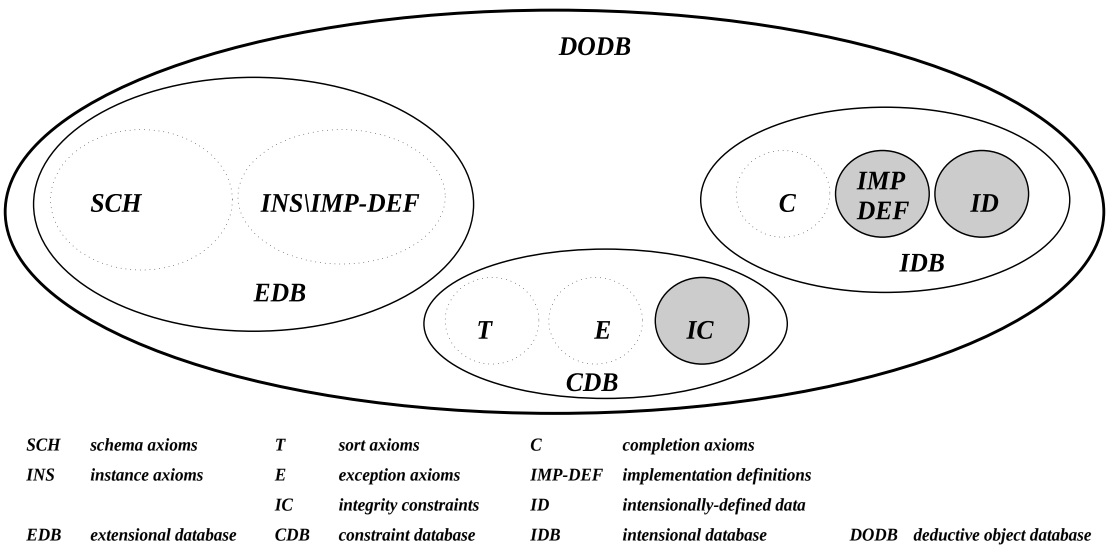


Figure 3.3: Characterizing Deductive Object Database Theories

This completes the specification of the relationship between ODTs and DODTs. Since DODTs are tractable logical systems equipped with the same well known language and proof procedure (Ceri et al., 1990b) of the logical system underlying DRDBs, this result constitutes the first part in the positive answer to the central question addressed in this dissertation as to whether DOODs can be so characterized as to predominantly share the metalogic of DRDBs. The second part requires addressing aspects of object-orientation not directly captured by DODTs. These are addressed in the next section.

3.2 From DODTs into DOOTs: Supporting Dynamic Aspects of Object-Orientation

Section 3.1 showed how OODBs, in their ODT representation, can be viewed proof-theoretically. Specifically, an ODT $DBAx$ can be expressed as set of function-free Horn clauses, thereby guaranteeing that their syntax and semantics are well defined and amenable to theorem proving approaches to query evaluation and integrity constraint checking.[1] The syntax and semantics of $\mathcal{CL}^{\bullet}$) is that of Datalog with negation. Section 3.1 also showed how the rule sets in $IMP\_DEF \subset INS$ define the operation implementations (declared in $IMP\_DEC \subset INS$) that make concrete the operation interfaces declared in $INT\_DEC \subset SCH$. Finally, extending what OODBs can normally express, Section 3.1 showed how a set IC of integrity constraints and a set ID of deductive rules, which define implicit information, can be uniformly added to OODBs, thereby enhancing their ability to capture application specific semantics. The clause set $DDBAx$ that results from these extensions is referred to as a *deductive object database theory* (DODT).

DODTs support:

- *object identity*, insofar as an object is denoted by its identifier (i.e., x in atoms of the form $instance\_of(x, y)$) and described by the set of facts, asserted or inferred, that are known about it (e.g., atoms of the form $property(x, y, z)$, $component(u, v, w)$, etc.). Hence, the state of an object does not denote it.
- *inheritance* through the is_a relation.
- *complex values* through the *component* relation.
- The expression of *method definitions*, *integrity constraints* and *derived data* by means of deductive rules.

However, a DODT lacks some essential features of the object-oriented paradigm, viz., the enforcement of encapsulation, and a discipline for overloading, overriding and late binding in a proof-theoretic setting. This fact justifies distinguishing them from a *deductive object-oriented database theory* (DOOT), which accounts for all of the latter. Without these missing features the behaviour-based view of objects that is at the heart of the object-oriented paradigm is somewhat compromised.

Supporting this behaviour-based view is one the greatest challenges for DOOD proposals, as, in the face of it, the notions involved are not easily captured in proof-theoretic terms. These missing features are addressed in this section. In so doing, this section describes in detail how DODTs differ from DOOTs and how the latter can be characterized proof-theoretically.

3.2.1 Supporting Overloading

Among the missing features, *overloading* is the least challenging to cater for whenever it does not occur along an inheritance path (since, when it does, overloading gives rise to questions as

[1]Again, it is stressed that the external predicates used for illustration purposes in the preceding section are not expressible in DODTs.

to whether to override, and of how and when to select a method to answer a request). Making operation names dependent on context is both intuitive and technically straightforward whenever, as is the case in hand, the focus of modelling and representation is on classes and objects and not on functions or processes. In this case, there need be no ambiguity involved in catering, in distinct ways, for requests invoking distinct, even if identically named, operations, e.g., to *display* a *map* and to *display* a *bank_account*.

The OODM described in this dissertation already supports overloading of operation requests by the following stipulations:

1. An operation is not identified with the operation name alone.
2. An operation is not identified with the operation signature alone.
3. An operation is identified by the operation interface (i.e., name and signature) and the class it is declared on.

Thus, every declared interface needs to be annotated with the class it is declared on. This is achieved by axiom (9) of Appendix C, which formalizes the distinction between a declared interface from the inheritable annotated interface that corresponds to it.

Distinct operations can have the same name and the same interface on distinct classes. Distinct operations can have either the same name or the same interface, but not both, on the same class. As discussed in the previous section, an implementation (both its declaration and its definition) corresponds to exactly one interface, and hence no ambiguity arises.

3.2.2 Supporting Overriding

In the presence of inheritance, overloading may require support for *overriding*. For example, if *display* is implemented differently on *bank_account* and on *interest_bearing_bank_account*, where the latter is a subclass of the former, one has to decide under which conditions, if any, the implementation of the operation in the subclass is to override the implementation inherited from the superclass.

Catering for overriding is more challenging than catering for overloading of operation requests across non-inheritance related classes because the implementation to be used in answering the request may depend on inheritance relationships between the classes involved. Overriding takes place according to a preference relation, which, typically, can only be decided dynamically, i.e., at the time the request is made.

The OODM described in this dissertation guarantees the necessary conditions for overriding to take place by making the following stipulations:

1. Every annotated interface has exactly one implementation associated with it. This state of affairs is achieved by axioms (50) and (51) of Appendix C, and has already been made use of in describing the implementation of operations and the overloading of operation names.
2. If an object is an instance of several classes, and if it inherits distinct operations with overloaded names from more than one of these classes, then it is always possible to conclude which is the most specific class providing an implementation for the overloaded operation name. This state of affairs is achieved by axiom (49) of Appendix C.

Consider the following example. If *display* has different implementations on both *bank_account*s and on *interest_bearing_bank_account*s, one might prefer that an interest bearing bank account responds to *display* less specifically, i.e., as a bank account, or more specifically, i.e., as an interest bearing account.

This dissertation uses the standard preference relation of selecting the implementation in the most specific class of the addressee in which one such implementation is available.

3.2.3 Supporting Encapsulation and Late Binding

Because the outcome of this selection process depends on the class of the object to which the request is addressed, it cannot be determined statically. This characterizes the need for the late binding of implementations to operation requests. The notion of *late* (or *dynamic*) *binding* is the least straightforward to cater for in proof-theoretic terms.

Informally, the approach taken in this dissertation is to model this selection process as an additional feature of the inferencing process. Insofar as the run time of an automated theorem prover equates to proof composition time, late binding is most intuitively captured as a step that becomes part of the proof procedure. The approach also has the advantage that it provides a simple and intuitive account of encapsulation.

Let the following definitions hold. Let @ be a family of n-ary predicate symbols, with $n \geq 1$. A *request* is a literal of the form

$$y(z_1, \ldots, z_n)@x : w$$

Intuitively, a request *invokes* the operation y, with argument objects $z_1, \ldots, z_n$ on the *addressee* object x in class w. Let r be a syntactical variable over requests.

A request $r = y(z_1, \ldots, z_n)@x : w$ is said to be *well formed* if x is an instance of at least one class w for which y is declared with argument classes $z'_i, 0 \leq i \leq n$. A request $r = y(z_1, \ldots, z_n)@x : w$ is said to be *well typed* if z_i is an instance of $z'_i, 0 \leq i \leq n$, and x is an instance of w. Note that if x has a most specific class, then a request can refer to it by default and, hence, take the form

$$y(z_1, \ldots, z_n)@x$$

This is typically the case, and, in the sequel, is assumed to be always the case in order to simplify the exposition.

The procedure *selectMethod* in Figure 3.4 describes the selection of the correct method y' with which to answer a well formed, well typed request r when the addressee has a most specific class.

<u>**proc**</u> *selectMethod*
    <u>**input**</u> $r = y(z_1, \ldots, z_n)@x$,
    <u>**output**</u> y'
<u>**body**</u>
    <u>**begin**</u>
        $W := \{w_i | instance\_of(x, w_i), responds\_as\_to(w_i, w_j, y, z'_1, \ldots, z'_n), instance\_of(z_k, z'_k),$
            $1 \leq i, j \leq m, 0 \leq k \leq n\}$;
        <u>**select**</u> $w \in W$ <u>**such_that**</u> $\forall w_i \in W, 1 \leq i \leq m, is\_a(w, w_i)$;
        <u>**if**</u> $implemented\_in\_by(w, y, z'_1, \ldots, z'_n, y')$
            <u>**then**</u> <u>**return**</u> y'
            (* *the ELSE case is guaranteed not to occur by constraining axioms* *)
        <u>**fi**</u>
    <u>**end**</u>
<u>**endproc**</u>

Figure 3.4: A Procedure to Select a Method to Answer a Request

The force of axioms (49)-(51) is to guarantee that the procedure is terminating and deterministic for every well formed and well typed operation request. That the procedure is terminating can be seen from the fact that the set W is finite (therefore the first and second steps terminate), and the fact that the **if** step tests a finite number of expressions (therefore a value is returned after a finite number of steps). That the procedure is deterministic can be seen from the fact that if more than one operation interface is suitable (i.e., $|W| > 1$) then the selection condition in the second step ensures that the least element in W is selected (such a least element is guaranteed to exist because is_a over W is a total ordering), and since it is guaranteed that there is exactly one implementation for every interface the **if** step returns exactly one value.

Note that once the addressee is known the outcome of the procedure can be computed using the state axioms SAx alone. If the addressee does not have a most specific class then the request must explicitly mention in which class the method is to be found that will adequately respond to it, in which case the request takes the form $r = y(z_1, \ldots, z_n)@x : w$ and w is preset.

Given a request $r = y(z_1, \ldots, z_n)@x$, the implementation of y used to answer r is the set of clauses of the form

$$responds\_with(x', y', z'_1, \ldots, z'_n)$$

where $y' = selectMethod(r)$.

Now, let $\mathcal{C}_1^\circ(DDBAx) \subset \mathcal{C}(DDBAx)$ be the set of operation names defined over $DDBAx$. Let $\mathcal{T}^\circ(DDBAx)$, defined as the union $\mathcal{C}_2^\circ(DDBAx) \cup \mathcal{V}(DDBAx)$, be a set of terms, where $\mathcal{C}_2^\circ(DDBAx) \subset \mathcal{C}(DDBAx)$ is the set of object names defined over $DDBAx$ and $\mathcal{V}(DDBAx)$ is the set of individual variables defined over $DDBAx$. Then, let $\mathcal{CL}^\circ(DDBAx)$ be the set of literals of the form

$$y(z_1, \ldots, z_n)@x$$

where $y \in \mathcal{C}_1^\circ(DDBAx)$, $x \in \mathcal{T}^\circ(DDBAx)$, and $z_i \in \mathcal{T}^\circ(DDBAx), 1 \le i \le n$.

Note that $\mathcal{CL}^\circ(DDBAx)$ is a special clausal language, all of whose expressions are literals. As discussed above, every request $y(z_1, \ldots, z_n)@x \in \mathcal{CL}^\circ(DDBAx)$ can be translated into a literal $responds\_with(x', y', z'_1, \ldots, z'_n) \in \mathcal{CL}^\bullet(DDBAx)$. This establishes a bijection from requests of $\mathcal{CL}^\circ(DDBAx)$ to goals of $\mathcal{CL}^\bullet(DDBAx)$ by recourse to the procedure in Figure 3.4.

Now, since every request maps to one, and only one, goal in the language of $DDBAx$, one can use the language $\mathcal{CL}^\circ(DDBAx)$ to query $DDBAx$. If one is restricted to querying $DDBAx$ using *only* $\mathcal{CL}^\circ(DDBAx)$ then strict encapsulation is enforced, since the constants exported, so to speak, from $\mathcal{CL}^\bullet$ to $\mathcal{CL}^\circ$, exclude those with which a request could be formed to gain access to the state of objects.

Thus, strict encapsulation can be supported by enforcing a strict distinction between $\mathcal{CL}^\bullet$ and $\mathcal{CL}^\circ$, where $\mathcal{CL}^\bullet \supset \mathcal{CL}^\circ$. In other words, making $\mathcal{CL}^\bullet$ the language of class designers and $\mathcal{CL}^\circ$ the language of users (including the methods defined by designers of other classes), and exporting from $\mathcal{CL}^\bullet$ to $\mathcal{CL}^\circ$ only the operation names defined by the designers in the interface of each class, suffices to enforce strict encapsulation in a way that is both faithful to the object-oriented paradigm and easy to capture in proof-theoretic terms.

Under this approach to strict encapsulation, it becomes illegal for a method in one class to contain in the body of any procedure of the form $responds\_with(x, y, z_1, \ldots, z_n)$ a goal of the form $responds\_with(x', y', z'_1, \ldots, z'_n)$ where y' is not implemented in the class of the method. In other words, if a method needs services defined in the interface of another class, it must use a request in $\mathcal{CL}^\circ$ not a goal in $\mathcal{CL}^\bullet$.

Now, it has been seen that $\mathcal{CL}^\bullet$ is a function-free Horn clausal language. It follows that three distinct, well defined evaluation styles are defined on $\mathcal{CL}^\bullet$: a model-theoretic style (based on least Herbrand models), a fixpoint-based style (based on an immediate consequence operator on Herbrand bases), and a procedural one (based on the SLDNF-resolution proof procedure). (See (Ceri et al., 1990b) for the function-free case, and (van Emden and Kowalski, 1976) for the more general case.)

Since every request $r \in \mathcal{CL}^\circ$ can be translated into the goal $g(r) \in \mathcal{CL}^\bullet$ that uniquely corresponds to it, and since evaluation is defined for $g(r)$, there is no need to define evaluation for $\mathcal{CL}^\circ$.

Instead, whenever a literal in the body of a clause of $\mathcal{CL}^{\bullet}$ is a request $r \in \mathcal{CL}^{\circ}$, one translates r into the literal $g(r) \in \mathcal{CL}^{\bullet}$ that corresponds to it, and evaluates $g(r)$. This also allows negated requests of the form $\sim r$ to be evaluated in the form of $\sim g(r)$.

Carrying out this translation at evaluation (model-theoretic, fixpoint, or resolution) time, characterizes late binding in the setting that concerns this dissertation. Intuitively, a request may be ambiguous with respect to the class of the addressee. When a well formed, well typed request is to be evaluated the class of the addressee is known from the binding patterns that hold for the clause at the time. Hence, one can disambiguate the request (using the $selectMethod$ procedure) and recast the request as a goal with a definite indication of the procedure that is to answer it.

As an example, assume that in the *land parcel selection* application, there is a subclass *urban_land_parcel*. This new class has one property $closest\_underground$ which is an integer representing the distance from the land parcel to the closest underground station. Assume also that the operation *has_easy_access* on *land_parcel* is as defined in Section 3.1.2. Finally, assume that an operation *is_available* is defined on *land_parcel* as follows:

responds_with(Land_Parcel,is_available') ←
    property(Land_Parcel,availability,Availability),
    Availability > 0.

Hence, a land parcel is considered to be available if its availability (an integer) is greater than zero.

Now, let the definition of *has_easy_access* on *land_parcel* be overridden in the subclass *urban_land_parcel* in the following way. An interface

$$responds\_as\_to(urban\_land\_parcel, urban\_land\_parcel, has\_easy\_access)$$

is inferred, and paired with the following declaration

$$implemented\_in\_by(urban\_land\_parcel, has\_easy\_access, has\_easy\_access''),$$

defined as follows:

responds_with(Urban_Land_Parcel,has_easy_access") ←
    property(Urban_Land_Parcel,closest_underground,Distance),
    Distance ≤ 100.

Hence, only those urban land parcels that are no more than 100 (say, yards) from the closest underground station are classified as having easy access.

Now, suppose one queries the database selecting land parcels that have availability greater than zero and have easy access. In a query such as

$$\leftarrow is\_available@Land\_Parcel, has\_easy\_access@Land\_Parcel$$

the first request is always mapped to a goal of the form $responds\_with(X, is\_available')$, i.e., always to a goal governed by $is\_available'$, as there is no need to override.

As a result of evaluating goals of the form $responds\_with(X, is\_available')$, the variable *Land_Parcel* may be bound to land parcels that are urban land parcels. If they are, then, since $is\_a(urban\_land\_parcel, land\_parcel)$, and not vice-versa, the mapping of $has\_easy\_access@Land\_Parcel$ is to a goal of the form

$$responds\_with(Urban\_Land\_Parcel, has\_easy\_access''),$$

i.e., the implementation associated with the more specific class, and the overriding implementation is the one used.

However, if the variable $Land\_Parcel$ is bound to land parcels that are not urban land parcels, then the most specific class of the binding of $Land\_Parcel$ is *land_parcel*. This means that $has\_easy\_access@Land\_Parcel$ maps to a goal of the form

$$responds\_with(Urban\_Land\_Parcel, has\_easy\_access'),$$

i.e., the implementation associated with *land_parcel* is not overridden by a more specific one and is the one used.

Of course, the evaluation of goals that appear in bodies of clauses is handled in an analogous manner. Note also how encapsulation is enforced by constraining queries (and, when appropriate, clause bodies) to be made entirely of requests in $\mathcal{CL}^\circ$ rather than goals in $\mathcal{CL}^\bullet$.

3.2.4 The Relationship between DODTs and DOOTs

The results of this section yield *deductive object-oriented theories*, insofar as they enrich DODTs with support for overloading, overriding and late binding. A DODT $DDBAx$ is related to a DOOT in the following way:

1. The language of DOOTs is $\mathcal{CL}^\circ$, and hence it is a language of requests, in the sense that all its expressions are queries (or goals, i.e., negative clauses).
2. DOOT requests abide by strict encapsulation, insofar as the predicate symbols $\mathcal{CL}^\bullet$ (i.e., of the corresponding DODT) used to represent the state of objects are not symbols of $\mathcal{CL}^\circ$.
3. DOOT requests are late bound whenever the request bears an overloaded operation name that configures an overriding situation. Late binding is effected by the translation of requests in $\mathcal{CL}^\circ$ into goals in $\mathcal{CL}^\bullet$ of the form

   $$responds\_with(x', y', z'_1, \ldots, z'_n)$$

   where $y' = selectMethod(r)$, i.e., the implementation of operation y that is available in the most specific class of the addressee that has one such implementation.

Informally, a DODT is a DOOT without requests that enforce encapsulation, and a discipline for overriding and late binding. A DOOT is a DODT with a language for such requests.

This interaction between DODTs and DOOTs is depicted in Figure 3.5.

This completes the specification of the relationship between DODTs and DOOTS. The object-oriented aspects known as encapsulation, overloading, overriding and late binding were accounted for using proof-theoretic concepts alone. This is a feature of the DOODs characterized in this dissertation that is possessed by no other proposed DOOD characterization. This result constitutes the second part in the positive answer to the central question addressed in this dissertation as to whether DOODs can be so characterized as to predominantly share the metalogic of DRDBs. The first part of the solution is the characterization of DODTs given in the previous section.

3.3 From DOOTs to DDTs: A Subsumption Relationship

This section shows how DOOTs extend DDTs with a richer axiomatic content and an inference engine that caters for an object-oriented style of querying the data.

It can be seen that the principle guiding the development of an axiomatic approach to DOODs as described in this and the preceding two chapters is that of reducing a complex problem to a simpler one for which a solution is known and tested.

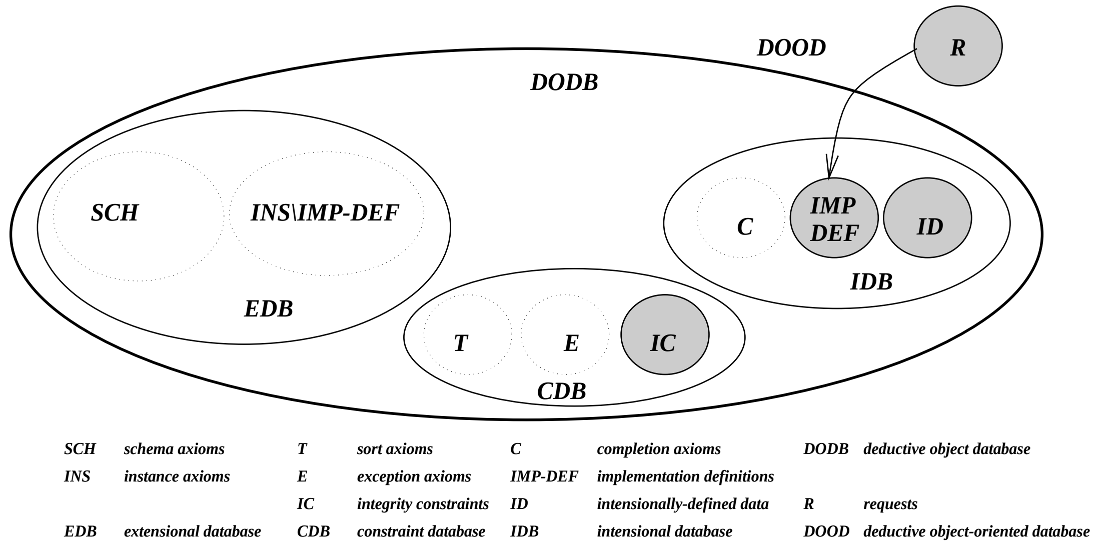


Figure 3.5: Characterizing Deductive Object-Oriented Theories

DOODs can be characterized in distinct ways, which are discussed in some detail in Chapter 6. The work reported in this dissertation was carried out as part of a broader research project whose main goal was to implement a robust DOOD system with near industrial performance. In that context, the most desirable property of a solution to the DOOD characterization problem was considered to be its capacity to facilitate the implementation effort. Correspondingly, this is the most salient characteristic of the axiomatic approach described in this dissertation.

Given that the DRDB enterprise had been successful in delivering such systems and, in the process of so doing, had developed a large corpus of theoretical results and practical techniques, an open question presented itself as to whether DOODs could, as DRDBs can, be expressed as systems of function-free HCL. The benefits of doing so are numerous, in terms of both the theoretical and practical consequences.

The solution developed in this and the preceding two chapters is a positive answer to the question above: DOODs can indeed be expressed as systems of function-free HCL. It is noticeable that as the mappings in the object-oriented space of theories depicted in Figure 0.2 are developed towards the DDT sink they become simpler to grasp and articulate. No more so than at this last mapping from DOOTs to DDTs.

DOOTs, as described in the previous section, do not exist independently of DODTs, and every expression in the language of DOOTs can be evaluated against the DODT after rewriting. Then, the mapping from DOOTs to DDTs is describable as a mapping from DODTs into DDTs. A DODT is a DDT in which:

1. There is set of axioms $clausal(OMAX)$, which formalizes all object-oriented modelling assumptions that are not dynamic in essence, and is, therefore, invariant over application domains.

2. The set of predicate symbols of the theory is also invariant, except for those predicate symbols introduced by application specific integrity constraints, which, by definition, do not add facts to those already inferable from legitimate states of a DODT.

3. The proof procedure is extended with a terminating, deterministic step that rewrites requests in the DOOT language $\mathcal{CL}^{\circ}$ into the DODT language $\mathcal{CL}^{\bullet}$.

It is crucial to note that, apart from the controlled alphabet and the inference rule incorporating the rewriting step described above, $\mathcal{CL}^\bullet$ has the same metalogic as Datalog, insofar as the axioms in $clausal(OMAX)$ bear no significant metatheoretical import and do not seriously compromise the efficiency of query evaluation with the respect to the size of derivations.

The subsumption relationship that characterizes DOOTs as applied systems of function-free HCL has been described in detail. This completes the specification of the relationship between DOOTs and DDTs.

Summary of Chapter 3

The results of Chapters 1 and 2 provided OODBs with a foundation in FOL. This chapter has shown how translating ODTs into clausal form equips them with an efficient proof procedure and places them on a simple extension path that leads to DODTs and DOOTs.

This chapter had three sections.

Section 3.1 demonstrated that an OODB, cast as an ODT, can be expressed as a set of function-free Horn clauses. This makes it possible for designers to express application specific integrity constraints and implicit information using deductive rules. It also allows for certain kinds of operations to be implemented using deductive rules. The resulting theories are called DODTs. DODTs extends OODBs in the same way that DRDBs extend RDBs.

Section 3.2 showed how the the notions of overloading, overriding and late binding can be captured in a proof-theoretic setting, thus extending DODTs into DOOTs.

Section 3.3 showed how DOOTs and DDTs are formally related.

The characterization of DOOTs completed in this chapter had as its main purpose to facilitate the task of developing a practical implementation of a DOOD system. The main goal has been to make it viable for implementors to maximize the reuse of the methods and techniques developed in the recent past to implement DRDBs. The practicality and extensibility of this characterization will be discussed in the chapters that follow.

This chapter extended the ODTs defined in Chapter 2, first into DODTs, and then into DOOTs, thereby yielding a specification of the highlighted arcs in Figure 3.1, viz.,

$$ODT \mapsto_{10} DODT \mapsto_{11} DOOT \mapsto_{12} DDT.$$

These arcs constitute the third, and final, step in the solution of the principal question to which this dissertation seeks to provide an answer.

With the completion of the desired path from the FOCT source to the DDT sink through the DOOT intermediate node, this chapter also concludes the first part of the dissertation with a positive solution to the question of whether DOODs can be characterized as DDTs.

The second part, spanning Chapters 4 and 5 discusses the application and extension of the approach developed in the first part. The third and final part closes the dissertation by surveying work that is related to the approach described and applied in the first and second parts, and by highlighting conclusions, contributions and future developments suggested by the research reported in this dissertation.

Chapter 4

Applying the Axiomatic Approach: The ROCK & ROLL DOOD System

The purpose of this chapter is to substantiate the claim concerning the practicality of the axiomatic approach to the characterization of DOODs through a presentation of how the approach has been applied to the engineering of ROCK & ROLL, a fully operational DOOD system developed at Heriot-Watt University.

The first part of this dissertation, comprising Chapters 1 to 3, described a solution to the problem of characterizing DOODs which situates them as a class of DDTs, i.e., systems of function-free HCL. The approach is referred to as the axiomatic approach, as it relies on a rich axiom set to model object-oriented concepts instead of redesigning syntactically or semantically the language component of the database theory.

The most desirable properties of the axiomatic approach are its practicality and extensibility. In the context of this dissertation, an approach to characterization will be said to be more, or less, *practical* depending on whether it simplifies, or not, the task of engineering a fully functional system. Similarly, an approach to characterization will be said to be more, or less, *extensible* depending on whether it simplifies or not, the task of enhancing the semantic content of a system or its functionality or both.

Naturally, other properties of a characterization are also significant. For example, *self-sufficiency*, where a characterization could be said to be more, or less, self-sufficient depending on whether it unifies more, or less, language subcomponents of databases among the following standard ones: definition, querying, constraining, view definition, and state transition. As defined in Chapters 1 to 3, the characterization resulting from the axiomatic approach, like pure Datalog-based systems in the DRDB case, is not totally self-sufficient in that it does not cater for updates directly. However, it could be extended to do so using strategies that parallel those used to extend Datalog with updates (Kiernan et al., 1990), or else it can be embedded in a calculus of evolving states, of which a well known example is the *event calculus* (Kowalski and Sergot, 1986; Kowalski, 1992; Sripada, 1988). In Chapter 5, it is shown how to embed DDTs (and hence DOOTs) into the Kowalski-Sergot event calculus, thereby making the DOODs characterized by the axiomatic approach more self-sufficient.

Important as self-sufficiency undeniably is, in the particular circumstances in which the work reported in this dissertation was undertaken, the properties of paramount importance were those of practicality and extensibility. The work took place in the context of a research project endeavouring to deliver a fully functional, robust, near industrial strength DOOD system using limited resources. That this goal has been achieved is, to a significant degree, evidence to the claim as to the practicality of the axiomatic approach. At the time of writing, that DOOD system is about to be extended to incorporate active functionality, and there are grounds to suppose that the claim to extensibility will be similarly substantiated.

At the close of this chapter, evidence will have been presented for the claim that the axiomatic approach offers significant advantages when the need is present for the swift, but nonetheless sound and well argued, engineering of a realistic system. The next chapter shows that the same applies when the semantic content or the functionality of the database theory has to be tailored, and possibly enhanced, in a nondisruptive way.

This chapter has four sections.

Section 4.1 provides an overview of the design and implementation strategies embodied in ROCK & ROLL, a fully implemented DOOD system based on the axiomatic approach and implemented at Heriot-Watt University between 1990 and 1994.

Sections 4.2, 4.3 and 4.4 are concerned with showing how the axiomatic approach has underpinned the design and implementation of the three major software components of the ROCK & ROLL system. Their main purpose is to provide evidence that the axiomatic approach offers advantages to implementors of DOOD systems.

4.1 ROCK & ROLL: A DOOD System

ROCK & ROLL is a DOOD system developed using the axiomatic approach described in Chapters 1 to 3. The ROCK & ROLL system adopts the strategy of separating out in different language components the need to support an imperative and a declarative style of expression (Fernandes et al., 1994c; Fernandes et al., 1994a). This strategy has been used before, e.g., in the Glue-Nail project (Derr et al., 1994; Derr et al., 1993; Phipps et al., 1991). It allows the design and implementation efforts to concentrate on bringing the two programming paradigms together into as tight a coupling as possible, but not beyond the limit where the strengths of either paradigm is sapped by attempts to subsume the other. ROCK & ROLL is described in detail elsewhere (Barja et al., 1995b; Barja et al., 1994; Barja et al., 1995a; Fernandes et al., 1993). It has been applied to the modelling applications that use spatial data (Abdelmoty et al., 1993b; Abdelmoty et al., 1993a; Abdelmoty et al., 1994; Williams et al., 1993). This section focuses on describing how the axiomatic approach is realized in the components of ROCK & ROLL. Note that besides their realization in ROCK & ROLL, most of the technical contributions described in the previous three chapters have been fully prototyped using Megalog (Bocca, 1991). A full description of the techniques, methods, functionalities and results deriving from that implementation is given in (Fernandes et al., 1994b).

ROCK (for Rule-Object Computational Kernel) is a persistent, imperative, procedural, object-oriented, computationally complete database programming language. Hence, ROCK can express:

- the structure and behaviour of persistent data,
- control of the flow of evaluation,
- state transition, and input and output (IO) requests in particular.

Being imperative and computationally complete, code written in ROCK is prohibitively complex to optimize in practical situations and cannot be guaranteed to terminate. The primary software component of the ROCK & ROLL system, and the one with which most users interact most is the ROCK interpreter [see, especially, (Goñi, 1995)]. ROCK defines a concrete syntax for data declaration and manipulation in the ROCK & ROLL database system. Had the syntaxes of ROCK and of ROLL not been smoothed to eliminate rough distinctions with respect to one another, ROCK would be perceived by users as the data definition (DDL) and data manipulation language (DML) of the ROCK & ROLL DOOD system.

ROLL (for Rule-Object Logic Language) is a declarative, optimizable, terminating,[1] deductive, database querying language. Hence, ROLL can express:

- intensionally defined data, and views in particular,
- ad-hoc and embedded queries, and
- integrity constraints.

Being declarative and terminating, code that is exclusively written in ROLL cannot express state transitions. In particular, ROLL evaluation cannot create new objects nor effect IO operations. The secondary software component of the ROCK & ROLL system is the ROLL interpreter [see, especially,(Dinn et al., 1995)]. ROLL defines a concrete syntax for data querying in the ROCK & ROLL database system. Had the syntaxes of ROCK and of ROLL not been smoothed to eliminate rough distinctions with respect to one another, ROLL would be perceived by users as the data query language (DQL) of the ROCK & ROLL DOOD system.

The design and implementation of the ROCK & ROLL system was geared towards maximizing the advantages, and minimizing the disadvantages, of this dual-component approach. It has been known for some time that dual-component approaches often incur impedance mismatches. Two kinds of impedance mismatches are often cited:

1. *type-system mismatches*, which arise when the type system of the calling language is sufficiently distinct from that of the language being called as to impose on programmers of the former the burden of writing code that is specifically designed to even out type-related incongruences with the latter. The most often cited example is the embedding of relation-based SQL queries in character-based languages, such as C.
2. *evaluation-strategy mismatches*, which arise when the evaluation strategy of the calling language is sufficiently distinct from that of the language being called as to impose on programmers of the former the burden of writing code that is specifically designed to even out evaluation-related incongruences with the latter. Again, the most often cited example is the embedding of set-at-a-time SQL queries in record-at-a-time languages, such as C.

In both the above cases, programmers must be aware of the difference in the types of the values being manipulated and of the precise way in which values are made available for further manipulation. In some cases, such as that between C and SQL, this demand for attention becomes a hindrance to productivity and a source of mistakes.

ROCK & ROLL incurs neither of the above impedance mismatches. The main factor in avoiding such mismatches is the role played by the OODM, as implemented in the third software component of the ROCK & ROLL system, the persistent object store known as OM (for Object Manager).

The crucial role played by the OODM in the avoidance of the above impedance mismatches is twofold:

1. The OODM fixes *an object-oriented type system* and imposes it on *both* ROCK and ROLL, thereby justifying, from the point of view of the type system, thinking about the two languages as one, viz., ROCK & ROLL. This type system is determined by the sets, relations and axioms described in Chapters 1 and 2. Note that propagating the type system from the OODM to ROCK *and* ROLL makes ROCK & ROLL a *structurally object-oriented* DDB system.

[1] It will be seen that insofar as ROCK-defined predicates can appear as subgoals in the body of ROLL clauses and the termination of such predicates is not guaranteed, when mention is made of ROLL as terminating it is implicitly meant that the subset of ROLL that does not contain ROCK-defined predicates as subgoals is terminating.

2. The OODM fixes *an object-oriented evaluation strategy* and imposes it on *both* ROCK and ROLL, thereby justifying, from the point of view of the evaluation strategy, thinking about the two languages as one. This evaluation strategy centres around the message-based discipline for encapsulation, overloading, overriding and late binding that emerges from the sets, relations and axioms described in Chapters 1 and 2. Note that propagating the evaluation strategy from the OODM to ROCK *and* ROLL makes ROCK & ROLL a *behaviourally object-oriented* DDB system.

The object store OM serves the ROCK and ROLL interpreters with fully compatible values and types and these, in turn, utilize fully compatible evaluation strategies. The values manipulated by the ROCK and the ROLL interpreters are fully compatible in the sense that for any call to the ROLL interpreter in the body of a ROCK method, the interpreter returns a value that need undergo no structural transformation for it to be usable. The same is true of a call to ROCK in the body of a ROLL method. The evaluation strategies utilized by the ROCK and the ROLL interpreters are fully compatible in the sense that when programmers write ROCK code that requires handing over the flow of control to the ROLL interpreter, they need only write the call, i.e., no interfacing syntax is required, let alone external linking of libraries. The same is true of when, from within ROLL, control is handed over to the ROCK interpreter. In both cases, the callee interpreter hands control back to the caller as transparently to programmers as it was originally handed over.

It is possible for invocations of ROLL-coded methods to appear embedded in ROCK-coded methods without any type- or evaluation-related penalties on the ROCK & ROLL programmer. Conversely, ROLL-coded methods (including queries, as discussed in Chapter 3) may invoke ROCK-coded methods without any type- or evaluation-related penalties on the ROCK & ROLL programmer.

No restrictions exist on ROCK invoking ROLL. There exists a restriction that ROLL can only invoke ROCK methods if the method invoked does not produce side effects on state elements that are not local to it. The property that a ROCK method has no global side effects is statically checked. This enables ROLL to remain order-independent and allows ROLL queries to be comprehensively optimized. Termination of a ROLL query containing a ROCK method invocation as a subgoal is dependent on the termination of the ROCK method invoked. It is undecidable whether a ROCK method halts. This means that although ROLL queries which contain no invocations to ROCK can be guaranteed to terminate, the same cannot be said of ROLL queries that contain ROCK-defined predicates as subgoals. Static type checking and type inferencing is implemented for ROCK & ROLL, i.e., type checking and inferencing flows from ROCK to ROLL code, and vice-versa, without any restrictions. Figure 4.1 depicts most of the interactions and configurations described so far in this section.

It should be noted that the present version of the ROCK & ROLL system does not comprise a component to enforce integrity constraints. There is no impediment on ROLL expressing integrity constraints and the methods and techniques developed for integrity constraint enforcement in the context of DRDBs are applicable to ROLL with minimal adaptations (see (Das, 1992), Chs. 12-13 for a thorough review of such methods and techniques). However, since an extension of ROCK & ROLL with active database functionality is scheduled, along the lines described in Chapter 5, it is conceivable that rather than adapting those methods and techniques, focus will change to more powerful functionality, such as that provided by repair actions.

The next three sections discuss in some detail how the axiomatic approach determines features and properties of OM, ROCK and ROLL. Note that the sections below are not meant to be descriptions of any of the three above pieces of software, instead they concentrate on how the axiomatic approach acquires concreteness in each of them.

The discussion uses a running example adapted from (Goñi, 1995). It is one version of the *bill of materials* problem, where parts can be base parts or composite parts, the latter being, recursively, ordered arrangements of parts. Parts have names. Base parts have mass, cost, a supplier, and may be imported. Composite parts have an assembly cost and a mass increment associated with them.

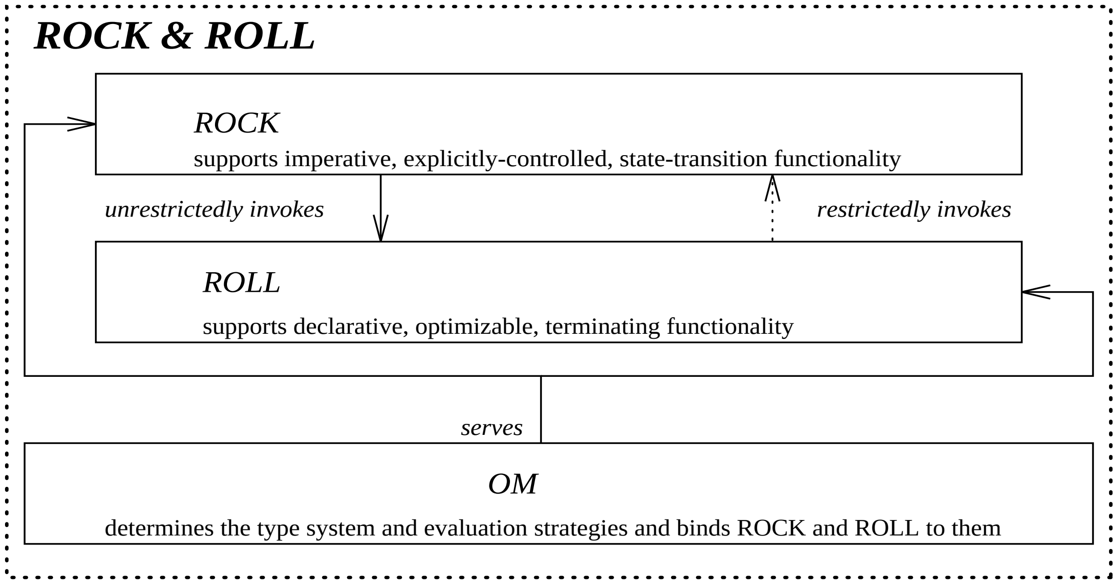


Figure 4.1: ROCK & ROLL Interactions

There is a need to compute mass and cost for parts, base parts and composite parts, which implies recursively computing the mass and cost of components whenever the part is composite. There is also the need to know whether a part is a component of another part. A partial diagram of the application is given in Figure 4.2. The information contained in Figure 4.2 is expressed in the form of axiom sets in Figures 4.3, 4.5 and 4.7 below, while Figures 4.4, 4.6 and 4.8 are exhibits of ROCK & ROLL code corresponding to those axiom sets.

4.2 OM: A Object Store for the Underlying OODM

OM (Paton and Abdelmoty, 1993) is a software component implementing a version (Fernandes et al., 1992a) of the OODM defined in Chapters 1 and 2 on top of a persistent store (Carey et al., 1990) that provides minimal database functionality such as that required to support concurrency and recovery. The role of OM is to provide an intermediate level of abstraction between the persistent store and the language interpreting components of the ROCK & ROLL system. This strategy allows the interpreters of ROCK and of ROLL to express requests that involve interaction with the persistent store in terms that are defined by the OODM (e.g., create a new class, retrieve an instance of a class, add a new property of a class, add a new specialization of a class, add a new object to the construction of an object, etc.) as opposed to terms that come predefined from lower levels (e.g., allocate persistent storage space, create a pointer to, etc.).

In terms of the implementation of an OODM, the axiomatic approach has some practical advantages to implementors, insofar as the characterization of the OODM as a mathematical structure can be seen as the formal specification of a particular *abstract data type* (ADT) (or a collection thereof), as follows:

1. The principal base sets of the OODM determine the minimal collection of ADTs that must be present. Additional structures will be required, e.g., indexes, but in a supporting role only.

2. The typical characterization of the OODM determines the minimal properties and interface of each ADT in the collection, insofar as it determines precisely all the relationships in which each base set is involved. Again, additional properties and interface (e.g., to make inverse

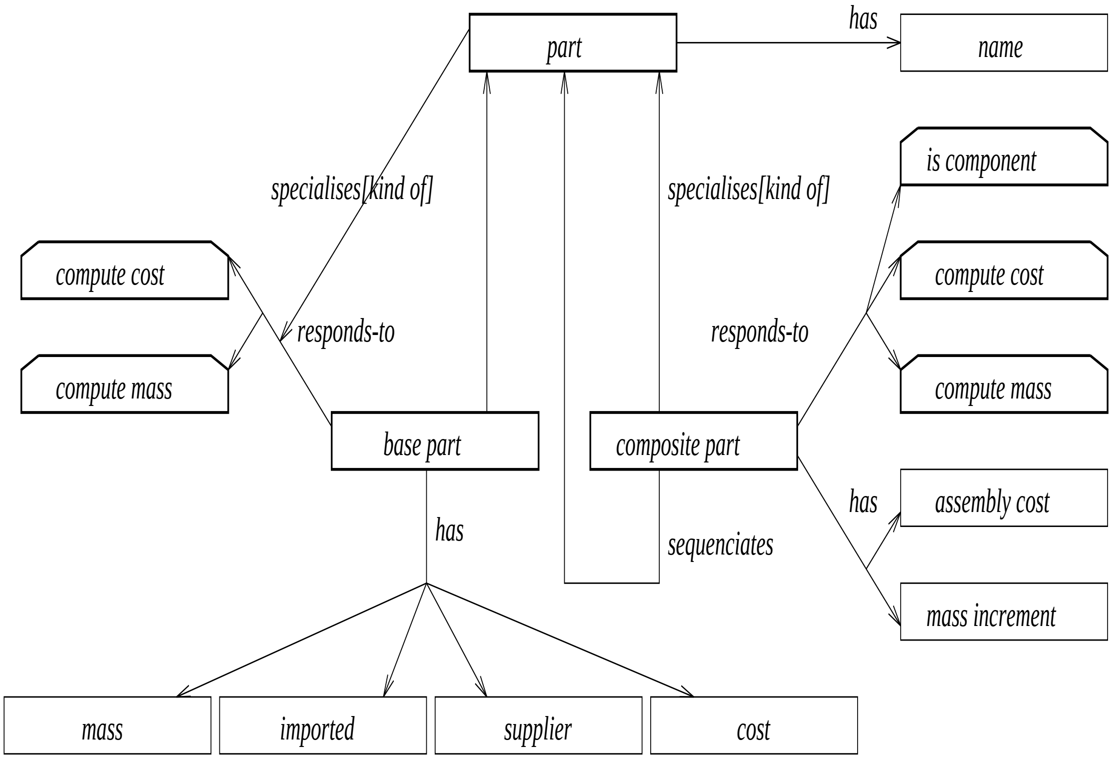


Figure 4.2: The *bill of materials* Example in Diagrammatic Form

pointers available), and operations (e.g., sequential access to unordered collections under some default ordering) have only a supporting role.

3. Finally, the mathematical axiom of the OODM characterizes and constrains the creation, access and update operations in the implementation of each ADT in the collection.

This means that OM implementors can profit from viewing the OODM almost directly as an ADT (or a collection thereof) with formally defined structure and operations to support the creation, retrieval, and update of schema and instance-level data. The axiomatic approach simplifies the task of engineering a robust object store by providing for free an abstract, high-level, formal specification for the store, thereby allowing implementors to concentrate on matters related to concrete representation and performance.

(Paton and Abdelmoty, 1993) describes the design of an object store for the OODM described in Chapters 1 and 2 as a collection of persistent C++ classes. OM, the ROCK & ROLL store, is an implementation of that design using E (Richardson and Carey, 1989; Richardson et al., 1993), a persistent version of g++, the GNU implementation of C++. E uses EXODUS (Carey et al., 1990) as the provider of lower-level functionality related to storage management.

4.3 ROCK: An Imperative, Procedural, Object-Oriented Database Language

In terms of its role in the overall architecture of the ROCK & ROLL system, ROCK is the language component that supports imperative, explicitly controlled, state transition programming on top of

OM. Strategically, its function is to ensure that complete, complex DOOD applications can be developed without jeopardizing the desirable metalogical properties of pure ROLL.[2] Tactically, it also takes on the role of being the DDL and DML of the database system.

Since ROCK is served by OM, it follows that the underlying type system of ROCK is that defined by the OODM as implemented by OM. Hence, the role of the axiomatic approach in determining the type system of ROCK is indirect, via OM. The evaluation paradigm of ROCK is also disciplined, in its essentials, by the OODM, insofar as the latter defines a message-based model of computation that abides by encapsulation and caters for overloading, overriding and late binding.

In its role as the DDL and DML of the system as a whole, ROCK defines a concrete syntax to express DOOTs. ROCK distinguishes between a *type* and a *class* as follows: a type defines the structure of, and the interface to, objects that are instances of the type, while a class declares and defines the operation implementations with which objects that are instances of the unique type to which the class corresponds can be manipulated. Roughly, type declarations constitute the DDL portion of ROCK, while class declarations constitute (the bulk of) the DML portion of ROCK.

Consider Figures 4.3 and 4.4. They show how the axiomatic presentation maps in a straightforward way to a block structured syntax that is much friendlier to users. (ROCK syntax also introduces annotations that effect finer-grained encapsulation, such as the categorization of properties as either private or public.)

---

specific_class(part)
    has(part,name)*
        generic_class(name)
            is_a(name,*`specialization`*,string,represented_as)*
                generic_class(string)
    responds_to(part,compute_mass,real)
                generic_class(real)
    responds_to(part,compute_cost,real)

    implemented_in_by(part,compute_mass,real,compute_mass_1)
    responds_with(X,compute_mass_1,0.0) ←

    implemented_in_by(part,compute_cost,real,compute_cost_1)
    responds_with(X,compute_cost_1,0.0) ←

Figure 4.3: State Axioms for *part*

Notice that Figure 4.3 comprises clausal definitions of *compute_mass_1* and *compute_cost_1* that in Figure 4.4 happen to, but need not, be expressed as pure ROCK code. Notice also that because ROCK fixes a set of generic types (viz., `int`, `real`, `string`, and `bool`) the declaration of a derived type can be reduced to type aliasing, denoted by a colon (e.g., `name:string`). Note also that ROCK adopts the convention that the value returned as the result of evaluating a method is the value of the last evaluated expression in the body of the method. This is why, in the implementations of `compute_mass` and `compute_cost` on `part`, which only require the setting of a default value, the constant `0.0` occurs as the last evaluated expression and is, therefore, the value returned in response to an invocation of either of those methods on `part`.

Figures 4.5 and 4.6 show more details of how ROCK provides concrete syntax and default functionality for DOOTs.

The noteworthy features of Figure 4.6 are the expression of a type-supertype relationship and the use of automatically generated operations to retrieve and update the properties of a type. Thus,

[2] By 'pure ROLL' is meant ROLL with no ROCK-defined predicates.

```
type part:
    properties:
      public:
        name: string;
    ROCK:
        compute_mass(): real,
        compute_cost(): real;
end-type

class part
  public:
    compute_mass(): real
    begin
        0.0
    end

    compute_cost(): real
    begin
        0.0
    end
end-class
```

Figure 4.4: The Type and Class `part` in ROCK

specific_class(base_part)
    is_a(base_part,*`specialization`*,part,kind_of)*
    has(base_part,mass)*
        generic_class(mass)
            is_a(mass,*`specialization`*,real,represented_as)*
    has(base_part,imported)*
        generic_class(imported)
            is_a(imported,*`specialization`*,bool,represented_as)*
                generic_class(bool)
    has(base_part,supplier)*
        generic_class(supplier)
            is_a(supplier,*`specialization`*,string,represented_as)*
    has(base_part,cost)*
        generic_class(cost)
            is_a(cost,*`specialization`*,real,represented_as)*
    responds_to(base_part,compute_mass,real)
    responds_to(base_part,compute_cost,real)

    implemented_in_by(base_part,compute_mass,real,compute_mass_2)
    responds_with(X,compute_mass_2, Y) $\leftarrow$
        property(X,mass, Y)

    implemented_in_by(base_part,compute_cost,real,compute_cost_2)
    responds_with(X,compute_cost_2, Y) $\leftarrow$
        property(X,cost, Y)

Figure 4.5: State Axioms for *base_part*

```
type base_part:
    specialises: part;
    properties:
      public:
        mass    : real,
        imported: bool,
        supplier: string,
        cost    : real;
    ROCK:
        compute_mass(): mass;
        compute_cost(): cost;
end-type

class base_part
  public:
    compute_mass():cost
    begin
        get_mass()@self
    end

    compute_cost():cost
    begin
        get_cost()@self
    end
end-class
```

Figure 4.6: The Type and Class base_part in ROCK

to every property (and aggregation coordinate) there correspond two methods whose name is the name of the property (or aggregation coordinate) prefixed, respectively, by `get_` and `put_`. As an example, Figure 4.6 depicts the use of `get_mass` and `get_cost` to implement `compute_mass` and `compute_cost`, respectively, on class `base_part`.

This is indicative of the general strategy by means of which ROCK provides, through implicit, i.e., built-in, methods, the basic functionality explicitly defined by the axiomatic characterization of classes, objects, operations and methods. Each of these is modelled, proof-theoretically, as a set of positive atoms in the language of DOOTs. In very broad terms, the minimal requirement is for a built-in retrieval method and a built-in update method for each argument of each form of atom that can occur in the state axioms of DOOTs. In the particular case of ROCK requests of the form `get_`y`()@`x and `put_`$y(z)$`@`x, they, respectively, retrieve and update the argument denoted by z in predicates of the form $property(x, y, z)$ or of the form $component(x, y, z)$, when y denotes a specific class.

This general strategy shows how the characterization of the OODM as a mathematical structure induces the operations needed to map between structures of the species, i.e., between DOODs (or, equivalently, between states of a DOOD). ROCK provides a rich set of built-in methods that bootstrap the linguistic expression of more complex methods over DOODs. The complete set of built-in methods in (Goñi, 1995) extends that defined in (Fernandes et al., 1992a).

Notice, finally, the special denotation, `self`, of the object that is to respond to the message send.

The final pair of definitions is given in Figures 4.7 and 4.8.

---

specific_class(composite_part)
    is_a(composite_part,*`specialization`*,part,kind_of)*
    has(composite_part,assembly_cost)*
        generic_class(assembly_cost)
            is_a(assembly_cost,*`specialization`*,real,represented_as)*
    has(composite_part,mass_increment)*
        generic_class(mass_increment)
            is_a(mass_increment,*`specialization`*,real,represented_as)*
    made_of(composite_part,*`ordering`*,part)*
    responds_to(composite_part,compute_mass,real)
    responds_to(composite_part,compute_cost,real)
    responds_to(composite_part,is_component,part)

    implemented_in_by(composite_part,compute_mass,real,compute_mass_3)
    implemented_in_by(composite_part,compute_cost,real,compute_cost_3)
    implemented_in_by(composite_part,is_component,part,is_component_1)
    responds_with(X,is_component_1,Y) ←
        component(Y,N,X)
    responds_with(X,is_component_1,Y) ←
        component(Y,N,Z),
        is_component(X) @ Z

Figure 4.7: The State Axioms for *composite_part*

Note that the versions of `compute_mass` and `compute_cost` on `composite_part` strictly require changes in the domain of the theory while the computation is being carried out, as indicated by the use of destructive assignments of intermediate results in the ROCK code displayed in Figure 4.8. This is because each successive recursive call has as its argument an object that is not identical to the one supplied as argument to the preceding call, and need, therefore, to be constructed in some sense.

```
type composite_part:
    specialises: part;
    properties:
      public:
        assembly_cost : real,
        mass_increment: real;
    public [part];
    ROCK:
        compute_mass(): real,
        compute_cost(): real
    ROLL:
        is_component(part);
end-type

class composite_part
   public:
    compute_mass(): real
    begin
        var tot: real;
        foreach p in self do
        begin
            tot := tot + compute_mass@p;
        end
        tot + get_mass_increment()@self
    end

    compute_cost(): real
    begin
        var tot: real;
        foreach p in self do
        begin
            tot := tot + compute_cost@p;
        end
        tot + get_assembly_cost@self
    end

    is_component(part)
    begin
        is_component(P)@Self :-
            isin(P)@Self;
        is_component(P)@Self :-
            isin(Q)@Self,
            is_component(P)@Q;
    end
end-class
```

Figure 4.8: The Type and Class composite_part in ROCK

This functionality can be accommodated in a purely proof-theoretical setting by the notion of *volatile objects*, which only exist while the proof that requires them is being carried out. Once the proof is concluded they are dismissed. This is legitimate, from a logical point of view, because objects are sets of axioms, and hence volatile objects are formally viewed as explicit, but temporary, assumptions that are required to compose the proof but need not become a permanent part of the database theory. A detailed description of this technique, including its implementation as an extension of a top-down evaluation method based on SLDNF-resolution (Das, 1992) can be found in (Fernandes et al., 1994b).

As characterized in this dissertation, the language of DOODs cannot express any state transition, and the one described above is no exception. In the context of ROCK & ROLL, the design decision is, likewise, not to allow ROLL to effect state transitions since this would introduce significant metalogical complications. Instead, it is left to ROCK to handle the creation of objects, along with all other functions that effect transitions of state.

Other noteworthy features in Figure 4.8 are the following. The square brackets around `part` in the type declaration indicate that a composite part is made of parts by ordering (were it association or aggregation, curly and angle brackets, respectively, would have been used). The interfaces of ROCK-defined methods are distinguished (by the header `ROCK:`) from ROLL-defined ones (headed by `ROLL:`). The reason behind this distinction is that ROCK, as a procedural language, distinguishes a return argument, while ROLL, as a logic programming language, does not. From the methods comprising the class `composite_part` one can gauge some of the expressiveness of ROCK as an imperative, computationally complete language, e.g., assignment statements, block (wrapped by `begin` ... `end`) and control structures (e.g., `foreach` ... `in` ... `do` ..., as an iterator over class extents, or constructions that are type-homogeneous collections). Note the use of another built-in function, viz., `is_in(`x`)@`y to retrieve constructions other than aggregation, and corresponding, therefore, to predicates of the form $component(y, z, x)$ when y is an association or an ordering, i.e., z is either a vacuous indexical (e.g., identical to x) or a proper index.

Finally, note in Figure 4.8 how the syntax of ROCK class definitions simplifies and enforces, through lexical scope, the relationships between the DOOT trio of predicates *responds_as_to*, *implemented_in_by* and *responds_with*. The method is tied to the class named in the header; the tying of method to interface achieved by *implemented_in_by* is obtained by scoping, and scoping also caters for the tie to *responds_with*. The invocation of ROLL-coded methods by ROCK-coded methods, and vice-versa, is discussed later.

This brief discussion of the relationship between DOOTs and ROCK shows that, as in the case of OM, the axiomatic approach provides a smooth path for the specification of practical DDL and DML, up to the definition of primitive operations.

4.4 ROLL: A Declarative, Deductive, Object-Oriented Database Language

In terms of its role in the overall architecture of the ROCK & ROLL systems, ROLL is the language component that supports declarative, optimizable, terminating programming on top of OM. Strategically, its function is to ensure that DOOD applications can make extensive use of the facilities associated with declarative programming. Both query termination and optimization opportunities are guaranteed (except that if ROCK-defined predicates occur in a ROLL query than the termination of the latter is dependent on the undecidable termination of the former). These guarantees result from the fact that the language, on the one hand, operates over a fixed finite domain, as does Datalog, and, on the other, is completely declarative (i.e., order independent). Tactically, ROLL also takes on the role of being the DQL of the database system. It can also be used to implement methods that do not require explicit control or state transition.

Consider again Figures 4.7 and 4.8. It can be easily seen how ROLL provides syntax to code an operation such as `is_component` that, in DOOTs, is written in standard Horn clausal syntax. Features to note are the standardization of the syntax of atoms to the request form described in Chapter 3, and the inheritance of ROCK syntax for built-ins such as `is_in`.

In Figure 4.8 the standard convention of capitalizing logical variables was adopted. Note that the essential semantic properties of the code bracketed by `is_component(part) begin ... end` are those of Datalog. In fact, only in the presence of overriding and late binding would semantic differences in behaviour be noticeable.

For examples of how ROLL is used from ROCK, and vice-versa, assume that the database has been populated. Then, consider the following ROCK statement:

```
var heavies := [ {P} | compute_mass@P == Mass, Mass >= 2.0 ];
```

This declares and assigns a state variable `heavies`, whose type (viz., an association of `parts`) is inferred at compile time, and assigns to it the result of the query enclosed in square brackets. To the left of the vertical bar, the logical variable `P` from the right hand side is being projected out, so bindings for it will become the elements in the construction of (the association of `parts` denoted by) `heavies`. The curly brackets around `P` indicate that all answer substitutions are of interest. The query (i.e., the expression to the right of the vertical bar, which a user would be able to type into a top-level user query interface) invokes the ROCK-coded `compute_mass` on `parts` and is true only of those whose computed `mass` is greater than or equal to `2.0`. Overriding and late binding of methods according to the binding values retrieved for `P` are certain to occur, but the programmer need not be aware of the fact. The symbol '`==`' denotes unification, in this case of the returned value of `compute_mass@P` with the logical variable `Mass`.

Notice that, as exemplified in the ROCK statement above, object creation is the sole responsibility of the ROCK interpreter. The ROLL interpreter optimizes and evaluates queries under the assurance that no side-effects are possible while the ROLL interpreter is running.

Notice also that the query invokes a ROCK-coded method `compute_mass` (as a ROLL-coded method also could), which is legitimate as long as the method does not produce side effects on nonlocal data. However, such invocations of ROCK make it impossible, in general, to guarantee that the query will terminate. For any query in which only pure ROLL methods are invoked, then this guarantee of termination exists. Finally, notice that the above is a ROCK statement containing an invocation of the ROLL interpreter and such a statement could legitimately appear wherever any other ROCK statement could.

The level of integration of the imperative and declarative languages can be gauged from the fact that a ROCK statement that could legitimately follow the statement above is:

```
foreach p in heavies do
begin
    write get_name@p, " is heavy.",  nl;
end
```

which would print out the names of all the heavy parts in the database, as collected and assigned to the state variable `heavies` after the query has been evaluated.

Results of queries are available for further manipulation. For example, suppose that, in possession of heavy parts one should want to know which are the heavy components of each composite part. The following nested iteration achieves the desired result:

```
foreach q in composite_part do
begin
    var heavy_components := [{Q} | isin(Q)@!q, isin(Q)@!heavies];

    foreach r in heavy_components do
    begin
        write get_name@r, " is a heavy component of ", get_name@q, nl;
    end
end
```

Note the character '`!`' prefixing the ROCK variable names `q` and `heavies` in the ROLL query expression. This indicates that the query is evaluated by the ROLL interpreter with those arguments bound to the values that `q` and `heavies` have at the point that ROLL evaluation is requested. Naturally, this offers opportunities for optimizing the query. Note, finally, that all the above code can be statically type checked, and that the lack of type annotation indicates the important role of type inferencing. Nonetheless, type annotations can be provided for documentation purposes, and, in some cases, *must* indeed be provided since the type inferencing algorithm of (Goñi, 1995) may fail to infer a unique type.

This brief discussion of the relationship between DOOTs and ROLL, as well as the integration of the two languages, shows that the axiomatic approach lays a solid foundation for the development of a practical DOOD system. The independent definition of an OODM guarantees type- and evaluation-strategy compatibility between languages adopting different paradigms of expression. This is crucial in terms of guaranteeing that each required functionality can be provided by the component that is most suited for it. Thus, querying is supported by a logically pure language which is optimizable and guaranteed to terminate. State transition and explicit control over evaluation is supported by an imperative, computationally complete language without the need to compromise the logical semantics of the DOOT.

From the point of view of ROLL alone, the most important contribution of the axiomatic approach is the fact that it ensures that DOOTs are reducible to DDTs, and hence that virtually all the technology that has been deployed to engineer DRDBs in the recent past can be deployed to implement ROLL using almost off-the-shelf techniques. Like Datalog and related languages, ROLL can be evaluated bottom-up or top-down. Bottom-up methods have traditionally been preferred in most DRDB implementations because they merge more easily with query optimization and evaluation techniques used by commercial, nondeductive RDBs. In the ROCK & ROLL system, ROLL is parsed into *system graphs* (Kifer and Lozinskii, 1990) that represent internally queries and the methods required to answer them. System graphs are then translated into *processing trees*, a form of query execution plan (Lanzelotte and Valduriez, 1992). Processing trees are globally optimized using an extension of the *static filtering* method (Kifer and Lozinskii, 1990), and can be locally optimized, using a cost estimate function that governs the reordering of nodes within tree sections. The adaptation of system graphs, processing trees and query execution plans to ROLL is described in (Dinn et al., 1995).

The practicability of the axiomatic approach is at its most evident when one realizes that all the above techniques are well understood and have been applied in DRDB/OODB research prototypes. The implementors of ROLL were able to select and apply them with an engineering effort that was less demanding than had hitherto been thought likely. That the validity and adequacy of those techniques could be judged by the implementors in a relatively straightforward way is due in large part to the congruence properties of the metalogics of DOOTs and RDTs/DRDTs.

Summary of Chapter 4

This chapter had the main goal of showing that the axiomatic approach provides a practical, effective route to the implementation of a fully functional and expressive DOOD system

This chapter had four sections.

Section 4.1 discussed how the axiomatic approach facilitated the design and implementation of ROCK & ROLL.

Sections 4.2, 4.3 and 4.4 concentrated on the three major software components of the system, describing how the DOODs defined in Chapters 1 to 3 inherently determine an object store OM and induce a set of minimal operations for creation, access and updating of references representing the state of objects and their classes. OM can then act as a server for a declarative, deductive, object-oriented database querying language ROLL that provides the surface syntax of the language of DOOTs and DODTs defined in Chapter 3. Finally, ROLL is wrapped up by ROCK, an imperative, procedural, object-oriented database programming language for which OM plays the role of a persistent store. Since ROCK has comprehensive facilities for both explicit control and for effecting state transitions, the system has all the functionality required of practical DBMSs.

ROCK & ROLL is concrete evidence for the claim as to the practicality of the axiomatic approach to the characterization of DOODs, in the sense that it reduces the effort required to engineer a practical DOOD system without renouncing any major feature of the paradigms undergoing integration. Chapter 5 provides similar evidence to substantiate claims as to extensibility.

Chapter 5

Extending DOODs Characterized by the Axiomatic Approach

The purpose of this chapter is to substantiate the claim concerning the extensibility of the axiomatic approach to the characterization of DOODs by describing two exemplary extension paths: the first can be used to capture more domain independent semantics; the second, to capture more functionality. In particular, the chapter contributes a detailed treatment of the problem of providing a logic-based extension of the passive DOODs defined so far with active behaviour modelled by event-condition-action (ECA) rules.

This chapter illustrates two paths for extending DOODs that are defined axiomatically:

1. The first method is that of strengthening the mathematical structure. This implies increasing the axiomatic content of the theory of the species, i.e., of the model of data. This method changes the source (but not the specification) of all the mappings emanating (possibly transitively) from the OODM node. Hence, in comparison with Figure 0.2, each node after, and including, the OODM node, appears inflated until the mappings sink into the DDT node. Figure 5.1 is a graphic depiction of this kind of extension path.

2. The second method is that of embedding DDTs, and hence DOOTs (or DRDTs), in a stronger theory. This implies defining a logical characterization for the axioms in DOOTs as constructs within the stronger theory. It is then possible, among other things, to use the stronger theory to reason about DOODs. This method changes neither the source nor the specification of the mappings emanating from the OODM node. Instead, it involves mapping DOOT axioms into the stronger theory. Figure 5.2 depicts one particular instance of this kind of extension path in which the stronger, embedding theory is an event theory (EVT).

Evidence will be presented to substantiate the claim that the axiomatic approach offers important opportunities of extension when the semantic content or the functionality of the database theory has to be tailored, and possibly enhanced, in a nondisruptive way.

This chapter has two sections.

Section 5.1 describes how increasing the axiomatic content of theories suffices to capture many kinds of domain independent semantic knowledge not always available in typical data modelling formalisms. It is shown how to extend DOODs with finer-grained ways of specifying inheritance of structure and behaviour and with means to express participation and cardinality constraints on relationships.

Section 5.2 contributes a thorough and detailed logic-based account of active behaviour that extends the functionality of DOODs characterized by the axiomatic approach, thereby increasing

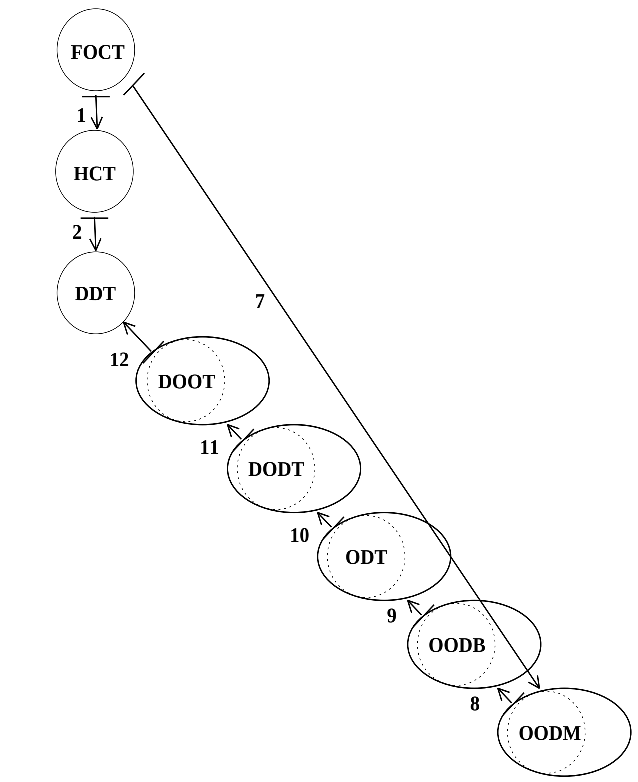


Figure 5.1: Extension Possibilities: Increasing the Axiomatic Content of Theories

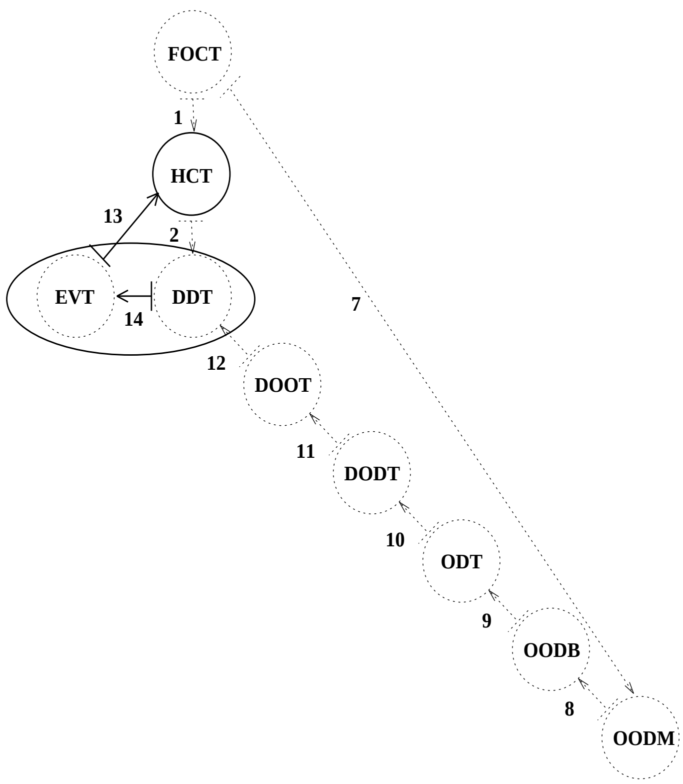


Figure 5.2: Extension Possibilities: Embedding a Theory in a Stronger Theory

their degree of self-sufficiency. The section describes how DOODs can be extended with a notion of state transition based on the Kowalski-Sergot event calculus (Kowalski and Sergot, 1986; Kowalski, 1992), which provides the foundations for the subsequent development of *active database* (ADB) functionality.

5.1 Capturing More Domain Independent Semantics

5.1.1 Variations on Inheritance Granularity

Declaring subclass-superclass relationships can be seen as a concise way of reusing specifications, an important concern in the principled engineering of very large software systems. However, the traditional style of declaring such relationships may not allow an adequate level of granularity, insofar as declaring a subclass typically implies inheriting *all* the structural and behavioural features of the superclass. Finer granularities seem to be desirable, insofar as opportunities for reuse would then arise that are not possible if one can only declare inheritance relationships in too blunt a way.

In this context, many possibilities can be envisaged of finer-grained (or even coarser-grained) declarations of inheritance relationships. The following are some, nonexhaustive, examples:

aliasing can be defined as the declaration of a new class that, at all times, has identical extension to the class whose name is being aliased. The implication is that every instance of the new class has the same structure and behaviour of an instance of the aliased class.

property-confined inheritance can be defined as the declaration of a class that has the same properties as another, but possibly distinct construction and interface.

construction-confined inheritance can be defined as the declaration of a class that has the same construction as another, but possibly distinct properties and interface.

interface-confined inheritance can be defined as the declaration of a class that has the same interface as another, but possibly distinct properties and construction.

In order to capture the semantics of the above relationships one extends the typical characterization of the theory of the species with more relations, e.g., **aliasing**, **p_inheritance**, **c_inheritance**, and **i_inheritance**. These are then defined as elements of the power set $\mathcal{P}(\mathbf{C} \times \mathbf{C}^*)$. Finally, the axioms below define the extension of these relations:

aliasing

$$\langle x, y\rangle \in \textbf{aliasing} \;\Leftrightarrow\; \langle y, x\rangle \in \textbf{aliasing}$$

$$\langle x, y\rangle \in \textbf{aliasing} \;\wedge\; \langle y, z\rangle \in \textbf{attribution}^* \;\Rightarrow\; \langle x, z\rangle \in \textbf{attribution}^*$$

$$\langle x, y\rangle \in \textbf{aliasing} \;\wedge\; \langle y, u, v\rangle \in \textbf{composition}^* \;\Rightarrow\; \langle x, u, v\rangle \in \textbf{composition}^*$$

$$\langle x, y\rangle \in \textbf{aliasing} \;\wedge\; \langle y, w, u_1, \ldots, u_n\rangle \in \textbf{interface} \;\Rightarrow\; \langle x, w, u_1, \ldots, u_n\rangle \in \textbf{interface}$$

$$\langle x, y\rangle \in \textbf{aliasing} \;\wedge\; \langle z, y\rangle \in \textbf{classification} \;\Rightarrow\; \langle z, x\rangle \in \textbf{classification}$$

property-confined inheritance

$$\langle x, y\rangle \in \mathbf{p\_inheritance} \wedge \langle y, z\rangle \in \mathbf{attribution}^* \Rightarrow \langle x, z\rangle \in \mathbf{attribution}^*$$

construction-confined inheritance

$$\langle x, y\rangle \in \mathbf{c\_inheritance} \wedge \langle y, u, v\rangle \in \mathbf{composition}^* \Rightarrow \langle x, u, v\rangle \in \mathbf{composition}^*$$

interface-confined inheritance

$$\langle x, y\rangle \in \mathbf{i\_inheritance} \wedge \langle y, w, u_1, \ldots, u_n\rangle \in \mathbf{interface} \Rightarrow \langle x, w, u_1, \ldots, u_n\rangle \in \mathbf{interface}$$

Adding the axioms above to the OODM induces OODBs that capture more domain independent semantics. This enhancement is made effective in ODTs, DODTs and DOOTs by retracing the technical development of Chapters 2 and 3 but with the axiom of the species now including the new conjuncts above. From a proof-theoretical point of view, no other intervention is necessary and, in particular, the metalogical properties of DOODs are preserved.

Notice that the preceding remarks only apply if the extended axiom of the species is such that in the mappings that take OODBs into DDTs nothing is introduced that changes an essential property of the theory, as given in Table 0.6 in the Introduction. An example of change to an essential property would be the introduction of function symbols in the language. In this case, DODTs would be mappable into HCTs, rather than DDTs.

5.1.2 Participation and Cardinality Constraints on Relationships

Another example of domain independent semantics that is often useful to support at the level of a data model is that of constraints on the participation and cardinality of entities in relationships. In the OODM, relationships are modelled as objects built by aggregation. For example, the relationship *employed_by* between a *company* and each one of its many *employee*s can be modelled as an aggregation.

For example, assume one wants to assert that no unemployed person may exist in the database, and no company may exist in the database unless it has at least one employee. Furthermore, no employee in the database can be employed by more than one company.

A simplified version of this kind of modelling problem is solved by the same sequence of steps used to establish finer-grained declarations of inheritance relationships, as follows. First, two relations, **participation** and **cardinality** are introduced. Then, two new auxiliary base sets are introduced: **PC** (for **participation_constraints**) with extension { `mandatory`, `optional`}, and **CC** (for **cardinality_constraints**) with extension { `single`, `multiple`}. This allows **participation** to be defined as an element of the power set $\mathcal{P}(\mathbf{C} \times \mathbf{PC} \times \mathbf{C})$, and **cardinality** to be defined as an element of the power set $\mathcal{P}(\mathbf{C} \times \mathbf{CC} \times \mathbf{C})$. Finally, the following axioms define the extension of these relations:

participation:

$$\begin{aligned}&\langle x, y, z\rangle \in \mathbf{composition}^* \wedge \langle z, \texttt{mandatory}, x\rangle \in \mathbf{participation} \Rightarrow \\ &\qquad (\forall u \langle u, z\rangle \in \mathbf{classification} \Rightarrow (\exists v w \langle v, x\rangle \in \mathbf{classification} \wedge \langle v, w, u\rangle \in \mathbf{construction}))\end{aligned}$$

cardinality:

$$\begin{aligned}&(\langle x, y, z\rangle \in \mathbf{composition}^* \wedge \langle z, \texttt{single}, x\rangle \in \mathbf{cardinality} \wedge \langle v, w, u\rangle \in \mathbf{construction} \wedge \\ &\qquad \langle v, w, u'\rangle \in \mathbf{construction} \wedge \langle u, z\rangle \in \mathbf{classification} \wedge \langle u', z\rangle \in \mathbf{classification} \Rightarrow u = u'\end{aligned}$$

The default, unconstrained case of declaring participation to be `optional`, or cardinality to be `multiple`, is vacuously true and requires no new axioms. Again note that, from a proof-theoretical point of view, no other intervention is necessary and, in particular, the metalogical properties of DOODs are preserved.

It can be seen from this and the previous subsection that the axiomatic approach is easily extensible with respect to capturing more domain independent semantics of important kinds. Of course, once systems are built the axiomatic content is, thereby, fixed to a great extent. Depending on the way the implementation is built, this may mean that the extensibility of the theory may not be matched by the extensibility of a system that embodies that theory, but this is orthogonal to the point at hand.

5.2 Capturing More Functionality

This section begins by describing an embedding of DDTs — and, by extension, of DOOTs (and DRDTs) — into a logical theory of change, viz., the Kowalski-Sergot event calculus (Kowalski and Sergot, 1986; Kowalski, 1992). It then uses these foundations to contribute a detailed formalization of active behaviour over the above classes of theories.

Very roughly, an *event calculus* is a logical system for the derivation of the information states induced by a set of time-stamped event descriptions. It is a calculus insofar as states, each of which is construed as a set of facts, are intensionally derived through a logical consequence relation based on a set of axioms, rather than explicitly and destructively manipulated. Event calculi use the language and rules of inference of HCL, and hence are categorized as acceptably efficient logical systems. The axiomatic content of an event calculus comprises:

1. A *history*, denoted by **H**, which is a set of *event descriptions* ordered by a *time line*.

2. A set of *event axioms*, denoted by $EvAx$, which determine how relationships among entities in the domain arise or change as a result of *event occurrences*, which are the notions that make their way into the history in the form of event descriptions.

When a new event occurs, its corresponding event description is added to **H**. This causes relationships to arise or change, thereby implicitly characterizing a new state by the force of the axioms in $EvAx$.

In the context of this dissertation, an event calculus provides the basis for modelling state transitions (hence capturing the functionality of updates) within the axiomatic approach to DOODs. The fact that DOODs and event calculi can both be mapped into HCL simplifies, in a significant way, the task of formally characterizing their logical relationship.

The crucial idea behind an event calculus is that states are not explicitly asserted, rather they arise as a result of the known succession of event occurrences. Correspondingly, instead of construing the set of facts that constitutes an EDB as being asserted explicitly, one can think of each EDB, at a particular point in time, as the set of logical consequences that are implicitly derived from the history of event occurrences, up to that point in time, through the application of the event axioms.

The methods and techniques described in this section are independent of the data model that underlies the database and also of the proof-theoretical form taken by it, i.e., the methods and techniques are applicable to relational or object-oriented, and to conventional or deductive databases. For this reason, the mapping into the event calculus is described taking the DDT node in Figure 5.2 as its source, since DDTs generalize both modelling assumptions and proof-theoretical forms.

5.2.1 Embedding DDTs into a Theory of State Change

This subsection begins by providing a specification of the mapping

$$EVT \mapsto_{13} HCT$$

in Figure 5.2. This specification follows closely the specialization of the event calculus for databases described in (Kowalski, 1992). Then, a specification of the mapping

$$DDT \mapsto_{14} EVT$$

in Figure 5.2 is provided. It shows how DDTs can be embedded in a stronger theory; in this case, an EVT. The specification makes explicit the logical relationship between the two classes of theories, and it is this relationship that embodies the essentials of the embedding process. In the case of this dissertation, the logical relationship is such that it can be used to capture the functionality of updates, and of integrity constraints on state transitions, in a logically homogeneous way with the standard functionality of DDBs.

A Formal Definition of the History

The notion of a history, i.e., a sequence of event occurrences, can be formalized as an extensional database of event descriptions, which underlies the derivation of database states, as follows.

Conceptually, a history is assumed to be a set of representations of time-stamped event occurrences. This means that, minimally, elements of a history have the form $\langle \tau, \delta \rangle$ where τ denotes a time point and δ describes an event occurrence.

The form assumed by δ and what that form denotes is partly dependent on the nature of the events that the history is supposed to record. Event types that are typically of interest include time events (e.g., system clock ticks, recurrence patterns, relative time events), structural events (e.g., insert, delete, update), behavioural events (e.g., method invocation, application start-up), transaction events (e.g., start, commit, abort) and environment events (e.g., device signal). A typical case is that of δ taking the form of a DML expression denoting a structural operation, such as that of setting the salary of an employee.

A more flexible, and more expressive, representation is obtained by providing unique, lifelong identifiers for event occurrences, resulting in the form $\langle \upsilon, \tau, \delta \rangle$. Asserting this is equivalent to asserting that the event occurrence denoted by υ happened at time point τ and is described by δ. The remainder of this dissertation assumes elements of a history to represent event occurrences in the latter form.

A *history* $\mathbf{H}$ is formalized as an EDB, i.e., a set of ground unit clauses. In this dissertation, the language $L(\mathbf{H})$, whose expressions are elements of event descriptions, belongs to the class of DDT languages. $L(\mathbf{H})$ is defined as follows. Let $EventId$, $TimeLine$, and $EventDescriptor$ be nonempty sets of symbols. In $\langle \upsilon, \tau, \delta \rangle$, let $\upsilon \in EventId$, $\tau \in TimeLine$, and let δ be an expression, said to be an *event description in surface form*, generated by a given domain specific alphabet and syntax. An event description in surface form is assumed to admit a translation, through parsing and dereferencing, into a first-order term, referred to as an *event description in internal form*, $a_0(a_1, \ldots, a_n)$, where $a_0, a_1, \ldots, a_n$ are elements in $EventDescriptor$; '(', ')', and ',' are punctuation symbols; and notions such as arity are assumed in the usual way when constructing internal forms.

This translation of event descriptions in surface form into internal form is necessarily application specific, and, therefore, is not described here, though some examples given later help sharpen intuitions about what the translation is supposed to achieve. Its main purpose is to remove syntactic sugar so that the result is closer to the clausal form in which the history is represented.

Each internal form $a_0(a_1, \ldots, a_n)$ can be translated (Deliyanni and Kowalski, 1979) into a set of clauses Δ with $n+1$ elements, each of which has the form $r_i(v, a_i)$, where $0 \leq i \leq n$, $v \in EventId$, and the r_i are domain specific symbols, interpreted as *roles*. The set of role denoting symbols is denoted *Roles* and is adjoined to $\mathcal{C}(\mathbf{H})$ to effect the translation of internal forms into clause sets. $\mathcal{C}(\mathbf{H})$ is also application specific.

A clause set which results from applying this translation procedure to an event description in internal form will be referred to as an *event description in Deliyanni-Kowalski form*, which is a special form of clause set. The advantages of representing complex expressions in this form are presented in (Deliyanni and Kowalski, 1979). Essentially, there are gains in flexibility, uniformity of representation, and more semantics is made explicit than would otherwise be the case.

As an exhaustive example of the stages in the translation process, consider the surface form $\delta =$'`put_salary(20)@self`'. It yields the internal form 'put(salary, !231, 20)' ,assuming that dereferencing `self`, which is a symbol in the DML but not in $L(\mathbf{H})$, has yielded '!231'. Then this internal form can be translated into the set of clauses:

referencing-act(ev321, put) $\leftarrow$
referenced-property(ev321, salary) $\leftarrow$
reference-source(ev321, !231) $\leftarrow$
reference-target(ev321, 20) $\leftarrow$

making use of 'ev231' $\in EventId$ and { 'referencing-act', 'referenced-property', 'reference-source', 'reference-target' } $\subseteq$ *Roles*. Note that, after parsing and dereferencing, every argument of a internal form, and, hence, of a Deliyanni-Kowalski form, is atomic, since nesting in the surface form is resolved into constant symbols before the translation into internal form.

Every event occurrence $\langle v, \tau, \delta \rangle$ can be represented in the history as a set of clauses of the form {'*event(v)$\leftarrow$', '*at(v, τ)$\leftarrow$'} $\cup\, \Delta'$, where Δ' is obtained from the Deliyanni-Kowalski form Δ of δ as follows: if '$r_i(v, a_i) \leftarrow$' $\in \Delta$ then '*happened(v, r_i, a_i)$\leftarrow$' $\in \Delta'$. An event occurrence represented as a set of clauses of the above form is said to be an *event occurrence in clausal form*. An *event description* is an event occurrence in clausal form. This last mapping ensures that the representation abstracts over application details by mapping application specific predicate symbols into individual constants that appear as arguments of (a fixed set of) application independent predicate symbols.

Thus, the event occurrence $\langle$ `ev321, 94.04.25.13.06.20.00, put_salary(20)@self` $\rangle$ is expressed in clausal form as follows:

*event(ev321) $\leftarrow$
*at(ev321, 94.04.25.13.06.20.00) $\leftarrow$
*happened(ev321, referencing-act, put) $\leftarrow$
*happened(ev321, referenced-property, salary) $\leftarrow$
*happened(ev321, reference-source, !231) $\leftarrow$
*happened(ev321, reference-target, 20) $\leftarrow$

A history is a set of event occurrences in clausal form. It follows that a history $\mathbf{H}$ is an extensional database over $\mathcal{C}(\mathbf{H}) = EventId \cup TimeLine \cup EventDescriptor \cup Roles$ and $\mathcal{P}(\mathbf{H}) =$ {*event/1, *at/2, *happened/3}. (Throughout this dissertation, p/n expresses the fact that the predicate symbol p has arity n.)

A formal definition for the store of event occurrences assumed by event calculi has now been provided as a particular form of EDB whose clauses are grouped into event descriptions that represent event occurrences. It is now shown how a history is used with the intensional axioms of event calculi to give rise to a sequence of EDBs, each of which represents an extensional state of the application domain.

The Event Calculi EC and EC$^+$

Over a history defined as a set of event descriptions in Deliyanni-Kowalski form, (Kowalski and Sergot, 1986) defined an *event calculus*. In (Kowalski and Sergot, 1986), the event calculus caters for relationships persisting indefinitely into the past as well as into the future. Recently, (Kowalski, 1992) reduced the event calculus to the database case, in which persistence into the past is not assumed. This simplifies the calculus and is the basis for the version, denoted by **EC**, described below. In contrast with the definition in (Kowalski, 1992), two assumptions made in **EC** are that events have no extended duration and that the relationships that they give rise to hold in the period initiated by the event and contain the said event. These assumptions simplify the formulation and implementation of **EC**, but, otherwise, nothing essential hinges on them and different conventions could be catered for.

Given a history **H**, such as that defined above, the event calculus **EC** is jointly determined by **H** and by the following set of *event axioms*, denoted by $EvAx$, each of which is a set of clauses:

1. The **persistence axiom** AxP defines the atoms in the application domain that hold at a given time point through the predicate $*holdsAt/2$. It is the singleton containing the following clause:

   *holdsAt(Atom, TimePoint2) ←
   starts(EventId, Atom),
   happenedAt(EventId, TimePoint1),
   TimePoint1 ≤ TimePoint2,
   ∼(brokenPersistence(EventId, Atom, TimePoint1, TimePoint2))

   i.e., *Atom* holds for an open period containing $TimePoint1$ at which the event $EventId$ that gave rise to the relationship denoted by *Atom* occurred, unless at some $TimePoint2$ subsequent to $TimePoint1$ the persistence of *Atom* initiated by $EventId$ at $TimePoint1$ is broken. As in any other part of this dissertation, '∼' can be interpreted as negation-by-finite-failure (Clark, 1978). The predicates $starts/2$, $happenedAt/2$, and $brokenPersistence/3$ are defined by axioms AxS, $AxTP$, and $AxBP$, respectively, given below.

2. The **time-point axiom** $AxTP$ defines the $happenedAt/2$ predicate. It is the singleton containing the following clause:

   happenedAt(EventId, TimePoint) ←
   *at(EventId, TimePoint)

   where $*at/2$ is the extensional predicate symbol in $L(\mathbf{H})$ defined in Subsection 5.2.1. This shows one circumstance in which **EC** is dependent on **H**.

3. The **broken-persistence axiom** $AxBP$ defines the $brokenPersistence/3$ predicate. It is the singleton containing the following clause:

   brokenPersistence(EventId1, Atom, TimePoint1, TimePoint3) ←
   interferesWith(EventId2, Atom),
   happenedAt(EventId2, TimePoint2),
   EventId2 ≠ EventId1,
   TimePoint1 ≤ TimePoint2,
   TimePoint2 ≤ TimePoint3

   i.e., the persistence of *Atom* started by $EventId1$ at $TimePoint1$ is broken at $TimePoint3$ if a distinct event $EventId2$, that happened at $TimePoint2$ between $TimePoint1$ and $TimePoint3$, interferes with *Atom*.

4. The **interference axiom** $AxIW$ defines the *interferesWith*/2 predicate. It is the set containing the following implicitly disjoined clauses:

   interferesWith(EventId, Atom1) ←
   ends(EventId, Atom1)
   interferesWith(EventId, Atom1) ←
   starts(EventId, Atom1)
   interferesWith(EventId, Atom1) ←
   ends(EventId, Atom2),
   incompatible(Atom1, Atom2)
   interferesWith(EventId, Atom1) ←
   starts(EventId, Atom2),
   incompatible(Atom1, Atom2)

   i.e., *EventId* interferes with *Atom*1 if it ends or starts it, or else it ends or starts an *Atom*2 that is incompatible with *Atom*1. The predicates *starts*/2 and *ends*/2 are defined by the axioms AxS and AxE discussed below.

5. The *incompatible*/2 predicate is defined by axiom AxI, as follows. Intuitively, two atoms A and B are incompatible if they cannot both be true at the same time point, or, trivially, if they are in fact the same atom. The **incompatibility axiom** AxI is the set containing the following implicitly disjoined clauses:

   incompatible(Atom1, Atom2) ←
   excludes(Atom2, Atom1)
   incompatible(Atom1, Atom2) ←
   Atom2 == Atom1

   The *excludes*/2 predicate is defined by the axiom $AxEx$ discussed below.

6. The **start**, **end** and **exclusion axioms** of **EC**, denoted by AxS, AxE and $AxEx$, respectively, are application specific, and stipulate how event occurrences start and end relationships, and also whether the fact that one relationship holds excludes another from holding. They define the three predicates *starts*/2, *ends*/2 and *excludes*/2, respectively. The following examples are typical and give an idea of their general form. An element of AxS could be:

   starts(EventId, property(Employee, salary, Salary)) ←
   *event(EventId),
   *happened(EventId, referencing-act, put),
   *happened(EventId, referenced-property, salary),
   *happened(EventId, reference-source, Employee),
   *happened(EventId, reference-target, Salary)

   where the body uses the extensional predicate symbols $*event$ and $*happened$ in the language $L(\mathbf{H})$ of the history defined in Subsection 5.2.1, and, in particular, the roles introduced in the transformation of the surface form `put_salary(Salary)@Employee` to its corresponding Deliyanni-Kowalski form. The expression $property(Employee, salary, Salary)$ denotes an atom representing a relationship that *starts* to hold after the event occurrence denoted by *EventId*. In the example above, the atom belongs to $\mathcal{CL}^{\bullet}$, i.e., the language of DODTs described in Chapter 3. This point is discussed in more detail later in this section.

   The next examples show how AxE and AxS contain clauses that are often related by the fact that they have shared bodies. Let the surface form of the relevant event description be `make_redundant@Employee`, then:

ends(EventId, property(Employee, status, permanent-contract)) $\leftarrow$
  *event(EventId),
  *happened(EventId, referencing-act, make-redundant),
  *happened(EventId, reference-target, Employee)
starts(EventId, property(Employee, status, redundant)) $\leftarrow$
  *event(EventId),
  *happened(EventId, referencing-act, make-redundant),
  *happened(EventId, reference-target, Employee)

Finally, for properties that are single-valued, one could have in $AxEx$:

excludes(property(Object, Property, Value1), property(Object, Property, Value2)) $\leftarrow$
  Value1 $\neq$ Value2

These are all the axioms in $EvAx$. Note that $EvAx$ is dependent on a history $\mathbf{H}$, and hence should be more clearly denoted by $EvAx(\mathbf{H})$. This is left implicit in the rest of this dissertation.

An **event calculus EC** is determined by its event axioms $EvAx = AxP \cup AxTp \cup AxBP \cup AxIW \cup AxI \cup AxS \cup AxE \cup AxEx$. Note, however, that the $*holdsAt$ predicate, defined by AxP, can be used to define other predicates, each of which is referred to as **holdsAt-derived* (HAD). For instance, a HAD predicate $*holdsNow(Atom)$ can be used to point implicitly to the current state:

*holdsNow(Atom) $\leftarrow$
  clock(Now),
  *holdsAt(Atom, Now)

under the assumption that an environment specific predicate *clock*/1 exists that instantiates Now with the time point $\tau \in TimeLine$ that corresponds to the system clock at invocation time. It is also possible to define HAD predicates that define states relative to a reference state. In this case, one sets a time point as the origin 0 of an integer line $-m, \ldots, -1, 0, +1, \ldots, n$ and states can be defined relative to the origin by an offset. For example, if 0 is now, then -1 is the state that held between now and the time of the last event occurrence, -2 is the state that held immediately before the last event, and so on. Let HAD denote the set consisting of the $*holdsAt$ predicate and those derived from it, and let AxP^{+} denote the set of clauses whose heads have a predicate symbol in HAD.

A **reducible event calculus** $\mathbf{EC}^{+}$ is then determined by an enlarged set of event axioms $EvAx^{+} = AxP^{+} \cup AxTp \cup AxBP \cup AxIW \cup AxI \cup AxS \cup AxE \cup AxEx$. An HCT $\mathbf{S}^{\star}$ can be defined as $\mathbf{S}^{\star} = \mathbf{EC}^{+} \cup \mathbf{H}$.

Since there is no direct use for any predicate in $\mathbf{EC}^{+}$ other than those in HAD, and since all other predicates in HAD can be rewritten in terms of $*holdsAt$, the convention is adopted that the set of intensionally defined predicates $IPred(\mathbf{S}^{\star}) = \{*holdsAt\}$. Further, having shown how a reducible event calculus can be formulated and reduced, the remainder of the dissertation uses either of the calculi as best suits the context. In the light of the above, the set of all logical consequences of $\mathbf{S}^{\star}$ consists of facts of the form $*holdsAt(\alpha, \tau)$, each of which denotes that the relationship named by the atom α holds at time point τ.

Precisely which atoms are derived from $\mathbf{S}^{\star}$ depends on the domain specific definition of $(AxS \cup AxE \cup AxEx) \subset \mathbf{EC}^{+}$. Given $cons(\mathbf{S}^{\star})$ (i.e., the set of logical consequences of $\mathbf{S}^{\star}$, which is computable as described in (Ceri et al., 1990b)) and a time point τ, $\mathbf{DB}^{\star} = \{\alpha : *holdsAt(\alpha, \tau) \in cons(\mathbf{S}^{\star})\}$ denotes the set of ground atoms naming all facts that hold at τ. One can then define, for each $\tau \in TimeLine$, a set $\mathbf{DB}^{\star}(\tau) = \{\alpha : *holdsAt(\alpha, \tau') \in \mathbf{DB}^{\star}, \tau = \tau'\}$, called the *extensional database at* τ. By isomorphism with the natural numbers, this shows how $TimeLine$ induces a sequence of *extensional databases*, each term of which is a subset of the intensional Herbrand base

$IHB(\mathbf{S}^{\star})$, i.e., the logical consequences of $\mathbf{S}^{\star} = \mathbf{EC}^{+} \cup \mathbf{H}$.

Two event calculi have now been formally defined and it was shown how they give rise to a time-stamped sequence of database states. Taken together, Subsections 5.2.1 and 5.2.1 constitute a specification of the mapping

$$EVT \mapsto_{13} HCT$$

in Figure 5.2. They show how an event calculus can be expressed as an HCT. Next it is shown how DDTs can be embedded into EVTs.

Embedding DDTs into EVTs

Event calculi express how states change as a result of event occurrences. In particular, they can be used to express updates to databases. Given a history $\mathbf{H}$ and a time point τ, the state of a database at τ, denoted by $\mathbf{DB}^{\star}(\tau)$, is the subset of the logical consequences $cons(\mathbf{S}^{\star})$ of the HCT $\mathbf{S}^{\star} = \mathbf{EC}^{+} \cup \mathbf{H}$, each element of which holds at τ.

Now, the assertion into $\mathbf{H}$ of the description of a new event that has occurred at a time point τ', maps $\mathbf{H}$ into a new history $\mathbf{H}'$, from which it follows a new HCT $\mathbf{S}^{\star'} = \mathbf{EC}^{+} \cup \mathbf{H}'$ and a new state at τ', $\mathbf{DB}^{\star}(\tau')$. Notice that, in this framework, updates are performed in a nondestructive way, i.e., the monotonic appending of new event descriptions suffices to effect state transitions with no need for previously asserted facts to be retracted.

The crucial relationship that needs to be established between a DDT and an EVT is as follows. In EVT-atoms of the form

$$*holdsAt(\alpha, \tau)$$

the EVT-term α is a DDT-atom. This technique, which maps an atom (i.e., a truth-denoting expression) in one theory (the weaker one) into a term (i.e., an individual-denoting expression) in another theory (the stronger one), is a form of *embedding.* This yields a specification of the mapping

$$DDT \mapsto_{14} EVT$$

in Figure 5.2. Essentially, the mapping takes the set of all logical consequences of a DDT into the set of terms of the EVT which embeds it.

The essential details of how DDTs (and hence DOOTs, and DRDTs) are embedded in the event calculus, thereby explaining how their state transitions can be modelled, are as follows.

The language and rules of inference of the new logical systems are those of an event calculus. The set of event axioms is extended as follows:

1. Each element of the extensional Herbrand base (Ceri et al., 1990b) of the DDT must be embedded. This is achieved by rewriting each ground fact of the DDT EDB as an element of AxP of the form:

   $$*holdsAt(\alpha, \tau) \leftarrow$$

   where τ is the time point where α is first known to hold, with $\tau = 0$ for all the atoms determining the initial state.

2. Each element α of the intensional Herbrand base (Ceri et al., 1990b) of the DDT must also be embedded. This is achieved by rewriting each rule of the DDT IDB of the form

   $$H \leftarrow L_1, \ldots, L_n$$

   into an element of AxP of the form

   $$*holdsAt(H, \tau_H) \leftarrow *holdsAt(L_1, \tau_{L_1}), \ldots, *holdsAt(L_n, \tau_{L_n})$$

   where, typically, $\tau_H = \tau_{L_1} = \ldots = \tau_{L_n}$.

3. AxS, AxE and $AxEx$ are defined so that the second argument of $starts/2$, in AxS, and of $ends/2$, in AxE, as well as both arguments of $excludes/2$ in $AxEx$ have a predicate symbol in the DDT.

This gives rise to a succession of database states in the way described above. Notice that the above embedding not only provides a means of capturing updates, but also allows the expression of integrity constraints on state transitions, insofar as time points are reified and appear explicitly as arguments. Hence rules can be written which have a time point in the head that is later than the latest time point in the body.

As mentioned before, to effect an update it suffices to assert into the history the event occurrence (in clausal form) that gave rise to the change in circumstances. In the next section, a more comprehensive approach to the dynamic aspects of DOODs is described that builds upon the foundations laid in this section.

5.2.2 A Logic-Based Extension of DOODs with Active Rules

Using the constructs described in Subsection 5.2.1, this subsection shows how the passive DOODs characterized so far can be transformed into active DOODs.

Rules in database systems can be of at least two kinds: *deductive rules* (DRs), which are used to express knowledge about application domains in a purely declarative way, and *active rules* (ARs), which are used to express actions to be performed, depending on context, as a reaction to events that may be internal or external to the system. The incorporation of active capabilities into database systems is justified on several grounds (van der Voort and Kersten, 1991), of which the important examples are: improving interaction between the system and its users, enforcing constraints on database states, and managing the database environment dynamically.

The class of *active database systems* (ADBs) (Dayal et al., 1988; Díaz et al., 1991; Gatziu et al., 1991; Hanson, 1989; Lohman et al., 1991) in which active behaviour is expressed by sets of ARs known as *ECA rules* has recently attracted great interest. One appealing property of an AR-set is that its operational semantics is independent of the underlying data model. As discussed in (Paton et al., 1994), ADBs based on ARs and displaying significant functional similarity have been built over both relational and object-oriented databases. The question arises whether this independence property also holds for ADBs over DDBs. The answer is not straightforward because DDBs already have an operational semantics of their own, and this, in turn, raises the issue as to whether an AR-set and a DR-set can be given a unified operational semantics and what gains there are in doing so.

The different semantic accounts (Das, 1992) that can be given to DR-sets are special cases of similar accounts given to logic programs in general. The semantics of AR-sets is typically operational, with roots in research into rule-based expert systems (Forgy, 1982; Miranker, 1987). Much of the interest in investigating ways of combining the two kinds of rule sets lies in the semantic issues that arise from their acknowledged differences and purported similarities.

Typically, research into DRs has been carried out in almost complete isolation from research into ARs, and vice-versa. More recently, a strong interest in investigating whether there is any intuitive, clean and efficient way of reconciling the two kinds of rules in a single database system has resulted in two positions. The first (Widom, 1994) argues that ARs and DRs lie at opposite ends of a spectrum that progressively trades declarativeness for more powerful expression of active behaviour. Intuitively, this would seem to lead to the conjecture that AR systems could be constrained to run on sets of DRs. Unfortunately, no formal semantics has yet been proposed to ground this intuition. The second position (Zaniolo, 1994) argues that a fixpoint interpretation extending the classical operational semantics of DR-sets succeeds in providing a common view of DRs and ARs. Intuitively, this leads to the conclusion that DR systems can be naturally extended to run on

AR-sets. Unfortunately, it is clear that some of the expressive power of AR systems is not easily captured in a purely declarative framework (e.g., the possibility of specifying particular coupling modes (Dayal et al., 1988)).

This dissertation contributes a third position. It takes the view that the operational semantics of DDBs and that of ADBs are not intrinsically similar, although in some cases they can be made to be so either by unduly extending the expressiveness of the former or by constraining the expressive power of the latter. The intended interpretation of DR-sets is different from the intended interpretation of AR-sets. Some of the reasons for this are: active behaviour and deductive behaviour serve different purposes from the point of view of users; also, they have different conceptual bases, require different techniques and styles for the formalization of their operational semantics and are realized by different computational mechanisms. It is difficult to see how one kind of behaviour can be subsumed by the other, without the resulting whole becoming more restrictive than the parts that originally went into its making.

Paradigm merging enterprises are only justified if they deliver more than is provided by each paradigm in isolation and if the original paradigms still retain their distinctive strengths in the resulting merger. This is clearly the case in the DDB-OODB merger exemplified by ROCK & ROLL as described in Chapter 4; it seems not to be the case in the attempted DDB-ADB mergers reported so far (a survey of these is given in the next chapter). This dissertation, therefore, argues in favour of keeping the two operational semantics independent of one another. Instead of attempting to show how one operational semantics subsumes the other, it details how to integrate the two in a clean and coherent whole that respects the differences in purpose that users discern in AR and DR-sets. In particular, no major constraints are imposed on either of them. Rather than forcing one semantics to fit uneasily into the other, it is argued that the two can be cleanly integrated without renouncing their separate identities and properties.

The technical development below is centred on the idea of a history that generates, through an event calculus, an ordered set of database states. One contribution of this dissertation is to to show that this notion of a history suffices to endow the three specification languages of ADBs, viz., the event, condition, and action specification languages, with a precise semantics.

The remainder of this subsection is structured as follows. First, the syntax and semantics of ADBs is introduced and a general ADB architecture is proposed which is a novel variant of the classical architecture (van der Voort and Kersten, 1991) in that the role of the history of event occurrences is much more prominent. Then, the three specification languages of ADBs are defined. Finally, over the specification languages an AR language can be derived for which a simple operational semantics is given.

Syntactic and Semantics Aspects of ADBs

An overview of the syntax and semantics of ADBs (Bayer, 1993; Chakravarthy, 1993; Hanson and Widom, 1993; Paton et al., 1994; van der Voort and Kersten, 1991) is now given. Knowledge of the syntax and semantic aspects of DDBs is assumed at the level of (Ceri et al., 1990b), whose terminology and notation are followed.

In contrast with DDBs, for which the characterization given in (Ceri et al., 1990b) is standard, ADBs have no widely accepted characterization. There is broad agreement on the syntax and semantics of ARs but only now are detailed proposals concentrating on providing formal foundations for ADBs in general (Campin et al., 1995). A system independent characterization of syntactic and semantic aspects of ADBs to the extent needed for the rest of the chapter is, therefore, provided.

Active Rule Syntax An *active rule*[1] ρ can be characterized as a pair $\rho = \langle \kappa, \sigma \rangle$. κ, called the *control part of* ρ, determines the execution conditions of σ, called the *specification part of* ρ, by parameterizing for ρ the behaviour of the execution model of rule sets. Since there is no standard operational semantics for ARs, proposals vary with respect to what execution parameters can be set by κ, and as a consequence the form taken by the control part of ARs varies accordingly. By contrast, there is broad agreement on the form taken by the specification part σ. The specification part σ of an AR ρ is a triple $\langle E, C, A \rangle$, where E is an *event specification*, C is a *condition specification*, and A is an *action specification*.

The specification part of an AR is often referred to as an *ECA rule*. In this dissertation, an ECA rule $\sigma = \langle E, C, A \rangle$ is expressed as '$E : C \rightsquigarrow A$'. Such expressions are commonly taken to mean that when an occurrence of the event specified by E has been detected and the condition specified by C holds in the current state of the underlying data, the action specified by A is performed, possibly changing that state and thereby generating new event occurrences. An ECA rule $E : C \rightsquigarrow A$ is read as '`when (`E `and` C`) do` A'. On the basis of this reading, ECA rules can also be referred to as *when-do rules*.

Each of the constituent parts of an ECA rule $E : C \rightsquigarrow A$ is an expression in a *specification language*. There is thus a distinct specification language for each part of an ECA rule, viz., an *event* specification language (ESL), a *condition* specification language (CSL) and an *action* specification language (ASL). Since all three are dependent on the definition language of the data model underlying a given database B for their lexicon, one could be precise and speak of the ESL of B, denoting it $\mathbb{E}_B$. Similarly, for the CSL $\mathbb{C}_B$ and the ASL $\mathbb{A}_B$. However, in the remainder of this dissertation, the subscript B is often dropped because the underlying database is of no special concern.

If $\mathbb{Q}$ denotes the DQL and $\mathbb{M}$ the DML of the data model underlying B, one can choose to set, for every B, $\mathbb{C} = \mathbb{Q}$ and $\mathbb{A} = \mathbb{M}$, thereby identifying a condition specification with a query expression, and an action specification with an expression that manipulates the database environment. This might obviate the need for independently defined CSLs and ASLs. However, the characterization of ESLs is still an open issue (Chakravarthy and Mishra, 1993; Gatziu and Dittrich, 1994; Gehani et al., 1992), as existing data models make no provision for an independent representation of events.

A tentative syntactic characterization of ADBs can be made as follows. Let B be a database, whose DDL, DML and DQL are, respectively, $\mathbb{D}$, $\mathbb{M}$, $\mathbb{Q}$. Similarly, let $\mathbb{E}$, $\mathbb{C}$, $\mathbb{A}$ be the event, condition and action specification languages whose application dependent lexicon is induced from that of $\mathbb{D}$. Furthermore, let $\mathbb{C} \supseteq \mathbb{Q}$ and $\mathbb{A} \supseteq \mathbb{M}$. An *ECA rule set* B^+ *over* B is a set of expressions of the form $E : C \rightsquigarrow A$ such that $E \in \mathbb{E}$, $C \in \mathbb{C}$ and $A \in \mathbb{A}$. An *active rule program* B^* *over* B is a set of pairs $\langle \kappa, \sigma \rangle$ where κ is a set of parameters for the execution of σ and $\sigma \in B^+$. An *active database* is a pair $\langle B, B^* \rangle$ where B is a database and B^* is an AR-program over B. The syntax and semantics of the ESL, CSL and ASL of the class of logic-based ADBs proposed in this dissertation are given later in this section.

Active Rule Semantics Describing the operational semantics of an ADB $\langle B, B^* \rangle$ that is general enough to cover the gamut of existing proposals is a daunting task. In the context of this section, simplifying assumptions are made. This allows the focus to fall on explaining general features of ADBs, but inevitably excludes important aspects of the functionality provided by specific proposals. The assumptions, embodied in Figures 5.3 to 5.5, are the following:

1. It is assumed that procedures exist which, given an expression in a language, return either the bindings resulting from successful evaluation of the expression or an arbitrary symbol

[1] So far in this section no distinction has been made between active rules and event-condition-action rules. This and the next few paragraphs propose one such distinction, in which the latter kind of rule is a particular case of the former. However, in the sequel the two terms are sometimes used as synonyms if the distinction is not relevant in the context that the terms are being used.

(e.g., $\bigtriangledown$) to indicate failure to generate any bindings. Two such procedures are required: a procedure *detect* for *event detection* and a procedure *verify* for *condition verification.*

2. It is assumed that a transition effecting procedure *execute* exists. This procedure is responsible for *action execution.*

3. If $\mathbf{DB}^\star$ is the set of all possible database states, $B \in \mathbf{DB}^\star$ and $\mathbf{E}$ denotes the set of known event occurrences in the environment of B, then the process that constructed $\mathbf{E}$, up to the point in which the database state is B, is assumed to be well defined.

4. Let an event occurrence be called *driven* if all its implications have been carried out through as many runs as required of the rule set that specifies the event occurrence. An event occurrence is called *nondriven* if it is not yet driven. Then, it is assumed that a nonempty $\mathbf{E}$ can be partitioned into a set of nondriven event occurrences and a set of driven event occurrences, e.g., by manipulating and consulting a system-wide log. Let these partitions be denoted by, respectively, *nondriven(*$\mathbf{E}$*)* and *driven(*$\mathbf{E}$*).*

5. A total ordering on sets of known event occurrences, denoted by *PriorE*, is assumed to exist — for example, that induced by a time line that is discrete and linear into the future — along with a function *earliest* that returns the least element of the resulting ordered set.

6. A total ordering on ECA rule sets, denoted by *PriorR*, is assumed to exist — for example, that induced by order of occurrence in a textual representation of the set — along with a function *first* that returns the least element of the resulting ordered set.

7. An assignment function from the set $\mathcal{V}$ of variables in the specification languages to the set Val of values in $\mathbf{DB}^\star \cup \mathbf{E}$ is assumed to exist.

8. A function *bind* is assumed to exist that, given an expression and a substitution (Ceri et al., 1990b), uniformly and simultaneously replaces every variable in the expression that occurs in the domain of the substitution by the value of the variable in the codomain of the substitution.

The signatures of the *binding generation procedures* and the *transition effecting procedure* are:

$$
\begin{array}{rlll}
\textit{detect}: & 2^{\mathbf{E}} & \times\, \mathbb{E} \rightarrow & 2^{(\mathcal{V} \times Val)} \cup \{\bigtriangledown\} \\
\textit{verify}: & \mathbf{DB}^\star & \times\, \mathbb{C} \rightarrow & 2^{(\mathcal{V} \times Val)} \cup \{\bigtriangledown\} \\
\textit{execute}: & \mathbf{DB}^\star & \times\, \mathbb{A} \rightarrow & \mathbf{DB}^\star \times 2^{\mathbf{E}}
\end{array}
$$

Thus, *detect* takes as input a set of known event occurrences and an event specification, and returns either the variable/value pairs resulting from detecting that the specified event has indeed occurred or else the indication of failure to detect. Similarly, *verify* takes as input a database state and a condition specification, and returns either the variable/value pairs resulting from verifying that the specified condition holds or else the indication of failure to verify. Finally, *execute* takes as input a database state and an action specification, and returns a new database state and a new set of known event occurrences resulting from effecting the state transition.

In this dissertation, only an operational semantics for the specification part of ARs is formalized. The *operational semantics of an ECA rule set* B^+ *over a database* B *with respect to a set of known event occurrences* $\mathbf{E}$ is defined by the *evaluateECASet* procedure shown in Figure 5.3.

In Figure 5.3, *evaluateECASet* returns the *NewDBState* resulting from driving all the elements in *Events*, including those that may have been generated in the process of driving previous event occurrences. Note that a saturation state is guaranteed by the requirement that every nondriven event be driven, i.e., *NewEvents* must be empty for the procedure to stop. The procedure to drive a nondriven event is shown in Figure 5.4.

```
proc evaluateECASet
   input Rules, PriorR, PriorE, DBState, Events
   output DBState, Events
body
   begin
   while nondriven(Events) ≠ ∅   do
      driveEvent(Rules, PriorR, PriorE, DBState, Events, NewDBState, NewEvents);
      DBState := NewDBState;
      Events := NewEvents;
   od
   end
endproc
```

Figure 5.3: An Operational Semantics for ECA Rule-Sets (a)

```
proc driveEvent
   input Rules, PriorR, PriorE, DBState, Events
   output NewDBState, NewEvents
body
   begin
   RemainingRules := Rules;
   while RemainingRules ≠ ∅   do
      ToConsider := first(RemainingRules, PriorR);
      touchRule(ToConsider, DBState, Events, NewDBState, NewEvents);
      RemainingRules := RemainingRules \ {ToConsider};
      DBState := NewDBState;
      Events := NewEvents;
   od
   logAsDriven(earliest(nondriven(Events), PriorE);
   end
endproc
```

Figure 5.4: An Operational Semantics for ECA Rule-Sets (b)

In Figure 5.4, *driveEvent* considers each rule in turn. The rule *ToConsider* is the first (as defined by *PriorR*) among the rules not yet considered. It is processed by invoking *touchRule*, shown in Figure 5.5. *touchRule* returns a new (possibly intermediate) database state and a new (possibly intermediate) set of known event occurrences. As the new state and event occurrences are generated they are also taken into account in the next step of the iteration by means of which *driveEvent* proceeds through the entire rule set. The procedure *logAsDriven* simply records the fact that the earliest (under *PriorE*) known nondriven event has now been fully attended to.

```
proc touchRule
    input ToConsider, DBState, Events
    output NewDBState, NewEvents
body
    begin
    E := eventSpec(ToConsider);
    C := conditionSpec(ToConsider);
    A := actionSpec(ToConsider);
    detect(Events, E, BindingsFromEventDetection);
    if BindingsFromEventDetection ≠ {▽}
        then
            begin
                bind(C, BindingsFromEventDetection, boundC);
                verify(DBState, boundC, BindingsFromConditionVerification);
                if BindingsFromConditionVerification ≠ {▽}
                    then
                        begin
                            bind(A, BindingsFromConditionVerification, boundA);
                            execute(DBState, boundA, NewDBState, NewEvents);
                        end
                fi
            end
    fi
    end
endproc
```

Figure 5.5: An Operational Semantics for ECA Rule-Sets (c)

As shown in Figure 5.5, *touchRule* takes the rule *ToConsider* and fetches its specification parts, viz., E, C, A, using the functions *eventSpec*, *conditionSpec*, and *actionSpec*, respectively, for that purpose. Then, *detect* is requested to return bindings using E. If it succeeds in doing so, the bindings are applied to C, and *verify* is requested to return bindings using *boundC*. If it succeeds in doing so, the bindings are applied to A, and *execute* is requested to generate a new database state and a new set of known events using *boundA*.

A History-Centred ADB Architecture An ADB architecture is proposed now in which the role of the history is more prominent than in the classical case (van der Voort and Kersten, 1991). The notion of history that is used is the one that underlies event calculi as described earlier in this section. The precise sense in which the history notion is central is the following: event detection, condition verification and action execution are all definable in terms of that notion.

Figure 5.6 depicts the conceptual components of a history-centred ADB. Three data components are used: the first for *event occurrences*; the second for *database states*; and the third for internal control data, produced and required by the ECA rule interpreter, in the form of a *log*. Three groups of software components interact with the data stores: both the *environment* and *applications* append

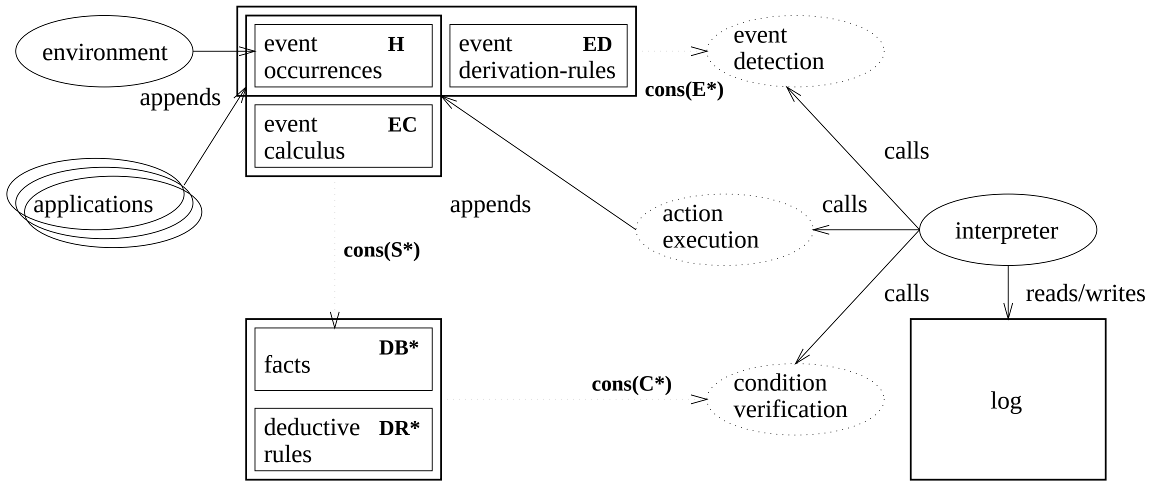


Figure 5.6: A History-Centred Active Database Architecture

event occurrences to the history as they occur, while the *ECA rule interpreter* checks and records its workings in the log while controlling *event detection*, *condition verification* and *action execution*. These three subcomponents of the ECA rule interpreter interact with the data stores as follows: the procedure for event detection scans event occurrences to check whether, as described in the event-specification part of ECA rules, any ECA rule is to be selected for possible scheduling; the procedure for condition verification requires the database to check whether, as described in the condition-specification part of ECA rules, any selected ECA rule is to be scheduled for execution, and, finally, the procedure for action execution of a scheduled ECA rule reads and writes into the database, records its having done so in the form of event occurrences, and records its internal affairs in the log.

Assume the history $\mathbf{H}$ previously defined. Then, two IDBs are defined on top of the history, as follows:

1. The first IDB, denoted by $\mathbf{ED}$, is made up of *event derivation rules* over $\mathbf{H}$. This determines a first DDB $\mathbf{E}^\star = \mathbf{H} \cup \mathbf{ED}$. Then, the set $\mathbf{E}$ of *known events* is immediately characterized in terms of the set of all logical consequences of $\mathbf{E}^\star$, denoted by $cons(\mathbf{E}^\star)$. Note that the elements of $\mathbf{E}$ are potentially complex, in the sense of being characterized by patterns of occurrence (and of recurrence), for instance. Such events are usually referred to in the literature as *composite events*.

2. The second IDB, denoted by $\mathbf{EC}$, over $\mathbf{H}$ is an event calculus as previously defined. This determines a second DDB $\mathbf{S}^\star = \mathbf{H} \cup \mathbf{EC}$. Then, the set $\mathbf{DB}^\star$ of *possible database states* is immediately characterized in terms of the set of all logical consequences of $\mathbf{S}^\star$. One can then write a set $\mathbf{DR}^\star$ of *data derivation rules* over $\mathbf{DB}^\star$. This makes $\mathbf{DB}^\star$ a genuine DDB $\mathbf{C}^\star = \mathbf{DB}^\star \cup \mathbf{DR}^\star$, against which conditions are evaluated.

By being used extensionally both by event derivation rules and by the event axioms, the history determines the set of known event occurrences and the set of all database states, and from the latter, the set of all (intensional and extensional) facts can be derived. It follows that for action executions to make events known or to effect state transitions they need only append event occurrences to the history. This, in fact, characterizes the precise sense in which the evaluation rules for the specification part of ARs only require the notion of a history.

A typical cycle in this architecture can be informally described as follows: the internal effect of a structural operation, e.g., `put_salary(25)@employee`, is that an event occurrence, e.g., that denoted by `[ev456, 94.04.20.16.44.55.25, put_salary(25)@!789)]` is appended to **H** in the clausal form described in Subsection 5.2.1. It follows from $\mathbf{H} \cup \mathbf{ED}$ that the set of known events now includes at least one event occurrence, viz., that named by `ev456`, and hence, from $\mathbf{H} \cup \mathbf{EC}$, that from `94.04.20.16.44.55.25` onwards, the `salary` property of employee `!789` has value `25`. When the interpreter reacts (or is made to react) to the structural operation it proceeds to coordinate the following sequence of computations: event detection selects rules whose event-specification part, taken as a goal against $\mathbf{E}^\star$, evaluates to true; condition verification schedules selected rules (if any) whose condition-specification part, taken as a goal against $\mathbf{C}^\star$, evaluates to true; and, finally, action execution fires the action specification of scheduled rules (if any). If this last step involves appending new event occurrences into **H** (which induce by logical consequence new known events in **E** and new facts in $\mathbf{DB}^\star$), then the interpreter must react (or be made to react) to these new happenings until a saturation state is reached in which all known events have been driven.

Specification Languages over the History

The ESL $\mathbb{E}$, the CSL $\mathbb{C}$ and the ASL $\mathbb{A}$ are now defined. Note that clausal form is uniformly used in this section as this is essential for the argument it presents. Since the semantics of all languages described in the remainder of this section are very well known, there are clear paths for adding syntactic sugar to the languages without compromising any semantic property. It is not in any way part of the approach described that the languages should *look*, as opposed to *be interpreted*, in the way they are in the remainder of the section.

An Event Specification Language over a History An ESL $\mathbb{E}$ can be characterized as the query language of a DDB $\mathbf{E}^\star$ whose EDB is a history **H** and whose IDB **ED** is a set of DRs, called *event derivation rules*, as follows.

The set of *known event occurrences* defined by event derivation rules naturally includes all the event occurrences explicitly recorded in the history. More importantly, it includes event occurrences that are inferred from explicitly asserted ones.

Among known event occurrences some correspond to so-called *composite events*. A typical composite event is the one usually denoted by $(E1, E2)$, with the intended interpretation that events $E1$ and $E2$ are known to have occurred, not necessarily in chronological succession. When it is ascertained that a relationship between event occurrences holds, the corresponding composite event is said to have been *detected*. Because such relationships are derived, in the sense that they are verified when the corresponding rule body (or query) is verified, the event occurrence they denote could also be said to be *derived*, and is *composite* in the sense that for its detection to be ascertained it is necessary that the detection (or, as it happens, the lack thereof) of other event occurrences is ascertained.

The event specification language $\mathbb{E}$ is the set of all queries over the DDB $\mathbf{E}^\star = \mathbf{H} \cup \mathbf{ED}$, whose extensional part **H** is a history and whose intensional part **ED** is defined as follows.

First, information explicitly asserted in the extensional part **H** is made available in the intensional part **ED** by including in the latter the following rules:

event(EventId) ←
    *event(EventId)
at(EventId, TimePoint) ←
    *at(EventId, TimePoint)
happened(ReferringEventId, Role, Referred) ←
    *happened(ReferringEventId, Role, Referred)

This introduction of new predicates is standard in DDBs and makes the set of intensionally defined predicates disjoint from the set of extensionally defined ones. This is largely a presentation issue and has the purpose of making a clear cut distinction between the set of asserted facts and the set of derived facts, thereby making it simpler to understand many concepts and techniques in the theory of DDBs.

Then, one extends the set of derived events by including rules in **ED** that use in their bodies:

- predicates in the set $\{event/1, at/2, happened/3\}$, and

- as expressive a set of auxiliary predicates as one chooses in the light of the trade-offs in semantic complexity that they imply. Typical auxiliary predicates are negation (e.g., $\sim /1$ interpreted as finite failure to prove), aggregates (e.g., *setof*, *count*, *sum*, *max*, *min*, *average*, all binary), arithmetic (e.g., $+, -, /, *, mod$, all ternary) and comparison (e.g., $<, >, \leq, \geq, =$, all binary) predicates. Access to the system clock through a predicate, e.g., $clock/1$, is also of obvious interest.

With these atoms, one can write rules that relate event occurrences and thus model composite events. To exemplify the approach, consider deriving the occurrence of two events having occurred irrespective of their chronological succession:

conjunction(E1, E2) ←
    event(E1),
    event(E2)

Disjunction and sequence can be similarly defined:

disjunction(E1, E2) ←
    event(E1)
disjunction(E1, E2) ←
    event(E2)

sequence(E1,E2) ←
    at(E1, TimePoint1),
    at(E2, TimePoint2),
    TimePoint2 > TimePoint1

The definition of more complex ways of composing events, such as those defined in (Chakravarthy and Mishra, 1993; Gatziu and Dittrich, 1994; Gehani et al., 1992), proceeds in the same predicate-based style, as opposed to the operator-based style of the above references. This is essentially a process of adding new predicates to the intensional part of $\mathbf{E}^{\star}$, and hence enlarging the set of logical consequences of $\mathbf{E}^{\star}$. An event specification is then a query over $\mathbf{E}^{\star}$, i.e., an ESL can be formally defined as

$$\mathbb{E} = \{\leftarrow E_1, \ldots, E_n : \forall 1 \leq i \leq n[F \sqsubseteq E_i \wedge F \in IHB(\mathbf{E}^{\star})]\}$$

where $L \sqsubseteq L'$ denotes that L is a substitution instance of L' as in logic programming in general (Ceri et al., 1990b). When an event specification holds against $\mathbf{E}^{\star}$ the corresponding event occurrence is said to have been *detected*. Let δ describe a known event occurrence that has been detected, then the known event occurrence can be represented as $\langle \upsilon, \tau, \delta \rangle$ where υ is system supplied, τ is the latest time point occurring in δ, and δ is the specification of the detected event occurrence. For instance, when $sequence(ev435, ev437)$ and $at(ev437, 94.05.12.15.54.19.25)$ hold, the following event occurrence has been detected `[ev453, 94.05.12.15.54.19.25, sequence(ev435, ev437)]`, which, in clausal form, is appended to the history as:

*event(ev453) ←
*at(ev453, 94.05.12.15.54.19.25) ←
*happened(ev453, event-kind, sequence) ←
*happened(ev453, sequence-first, ev435) ←
*happened(ev453, sequence-second, ev437) ←

This allows new known events to be verified taking previous known events into account. For instance, an event specification such as

disjunction(E3, E2),
happened(E2, event-kind, sequence)

now holds, and hence the corresponding event occurrence can be detected and appended to the history as described above.

The process of extending the set of intensional predicates of $\mathbf{E}^\star$ is limited only by the built-in expressiveness of the logic language one decides to employ in **ED**. For DRDBs, this expressiveness can range from pure Datalog (in which complex recurring patterns that require collecting, counting, summing, and so on, are not expressible) to a language such as $\mathcal{LDL}$ (Naqvi and Tsur, 1989) in which special language extensions (e.g., grouping, union, partition, choice) are available that can express aggregates on solutions.

The implications of choosing between pure Datalog and $\mathcal{LDL}$-like languages can be gauged from the thorough discussion in (Ceri et al., 1990b) of the added semantic complexity of $\mathcal{LDL}$ in comparison with pure Datalog. As an orthogonal alternative to this spectrum, one can, as well, represent $\mathbf{E}^\star$ as a DOOD. In the particular case of ROCK & ROLL this would make available a very expressive language in which to express known events.

A Condition Specification Language over a History A CSL $\mathbb{C}$ can be characterized as the query language over a DDB $\mathbf{C}^\star = \mathbf{DB}^\star \cup \mathbf{DR}^\star$, whose EDB is the set of logical consequences $cons(\mathbf{S}^\star) = \mathbf{DB}^\star$ of $\mathbf{S}^\star$ and whose IDB is a set of DRs, called *data derivation rules*, as follows.

The set of *derived data* defined by data derivation rules naturally includes every piece of data of every state that can be inferred by the event calculus $\mathbf{EC}^+$ from the event occurrences recorded in the history **H** and the data inferred from it, in turn. It is this property that makes $\mathbf{C}^\star$ a genuine DDB, in a way that is orthogonal to the active capabilities of the system as whole.

Let $IHB(D)$ and $EHB(D)$ denote those subsets of the Herbrand base associated with a database D that correspond, respectively, to $IDB(D)$ and $EDB(D)$. The aspects that distinguish $\mathbf{C}^\star$ from traditional DDBs, as defined in (Ceri et al., 1990b), are:

- The EDB of $\mathbf{C}^\star$ is obtained by computing the logical consequences of another DDB, viz., $\mathbf{S}^\star$. This means that to ascertain that a ground atom F is a fact in $EHB(\mathbf{C}^\star)$ one has to ascertain that F is in $IHB(\mathbf{S}^\star)$. This is so because one wants the states that are used for condition verification to reflect the event occurrences recorded in the history, and the event calculus, construed as an IDB, intervenes to determine how this reflection takes place.

- It follows from this and from the discussion in Subsection 5.2.1 that the EDB of $\mathbf{C}^\star$ can be construed as a sequence of time-stamped EDBs, so that there is an EDB at each time point τ for which an event occurrence is recorded in the history **H**.

With these two points in mind one can characterize the intensional part $\mathbf{DR}^\star$ of $\mathbf{C}^\star$ as follows. First, information inferred from $\mathbf{S}^\star$ is made available in $\mathbf{DR}^\star$ by including in the latter the following rule:

holdsAt(Atom, TimePoint) ←
  *holdsAt(Atom, TimePoint)

and similar ones for each $*holdsAt$-derived predicate, e.g.,

holdsNow(Atom) ←
  *holdsNow(Atom)

The set of specifiable conditions can be extended by including rules in $\mathbf{DR}^*$ that use in their bodies:

- the *holdsAt* predicate (as well as the HAD predicates defined in the particular reducible event calculus $\mathbf{EC}^+$ one is using), and
- as expressive a set of auxiliary predicates as one chooses, with the same caveat as in the case of event derivation rules.

A condition specification is then a query over $\mathbf{C}^*$, i.e., a CSL can be formally defined as

$$\mathbb{C} = \{\leftarrow C_1, \ldots, C_n : \forall 1 \leq i \leq n[F \sqsubseteq C_i \wedge F \in IHB(\mathbf{C}^*)]\}$$

The data that is derived in this approach is inherently time-stamped. In the typical case, this presupposes that the time point indicating when the atom derived in the head starts to hold is the same as that of the atoms used in the body of the corresponding rule. This characterization has to be done explicitly but a default to that effect can easily be imposed, as follows.

Given a rule of the form

holdsAt(Derived, TimePointD) ←
  holdsAt(Asserted$_1$, TimePointA$_1$),
  ...
  holdsAt(Asserted$_n$, TimePointA$_n$)

such a default would amount to adding to the body above the subgoals

  TimePointD = TimePointA$_1$,
  ...
  TimePointA$_{n-1}$ = TimePointA$_n$

This suffices to enforce that data is derived for a particular time point only from data that holds at *that* time point. If one further assumes, that

  clock(Now),
  TimePointD = Now

is also present in the body, the default is strengthened to derivation in the current state *only*. If both the above defaults are adopted, the original rule can be written as

holdsAt(Derived) ←
  holdsAt(Asserted$_1$),
  ...
  holdsAt(Asserted$_n$)

with the implicit time point being that of the system clock, in which case it would be clearer to write *holdsNow* instead of *holdsAt*. However, even the *holdsNow* predicate becomes vacuous, i.e., one can take a further step and write

Derived $\leftarrow$
  Asserted_1,
  ...,
  Asserted_n

which is just what one would write if multiple states were not involved.

If HAD predicates are defined that provide means to refer to states by indices instead of absolute time points, it becomes easier to write data derivation rules that involve, e.g., two consecutive states, and this means that some dynamic constraints over sequences of states can be expressed on top of the usual static ones.

An Action Specification Language over a History To complete the set of specification languages, an ASL $\mathbb{A}$ can be minimally defined in terms of monotonic appendage of event occurrences in clausal form to the history. On the one hand, this suffices to change the set of known events by changing the extensional base on which the logical consequences of $\mathbf{E}^\star$ are computed; on the other hand, by virtue of the event calculus, this suffices to induce a new state in $\mathbf{DB}^\star$ and hence change the logical consequences of $\mathbf{C}^\star$ that can be computed using $\mathbf{DR}^\star$. Note that, in this approach, when applications make changes to the database they too need only append event occurrences in clausal form to the history.

This means that an ASL $\mathbb{A}$ can be formally defined as the set of all expressions of the form $append(\delta)$, where δ is an event description in surface form that is mappable into an internal form, denoted by *internal(δ)*. With the assignment of a unique event identifier, the internal form becomes mappable into its corresponding Deliyanni-Kowalski form. This is a set of clauses denoted by *deliyanni-kowalski(internal(δ))*. Finally, with a time stamp, the Deliyanni-Kowalski form gives rise to an event occurrence in clausal form that can be appended to the history. This is a set of clauses denoted by *clausal(deliyanni-kowalski(internal(δ)))*. (All the above mappings, i.e., *internal*, *deliyanni-kowalski* and *clausal*, are as described in Subsection 5.2.1).)

Unlike ESL and CSL expressions, whose semantics is that of DDB queries (as described in (Ceri et al., 1990b)), the semantics of ASL is given by a simple set-theoretic expression. Given a history $\mathbf{H}$ and an event description δ in surface form, the result of evaluating an ASL expression $append(\delta)$ is the set $\mathbf{H}'$ of clauses defined by

$$\mathbf{H}' = \mathbf{H} \cup clausal(deliyanni\text{-}kowalski(internal(\delta)))$$

From a history $\mathbf{H}'$, one can derive anew both

- the set of known events $cons(\mathbf{E}^\star)'$ by the intervention of the event derivation rules $\mathbf{ED}$ over $\mathbf{H}'$, and
- the set of known facts $cons(\mathbf{C}^\star)'$ by the intervention of the data derivation rules $\mathbf{DR}^\star$ over the new base facts that are derivable using the event calculus $\mathbf{EC}^+$ over $\mathbf{H}'$.

Thus, the three specification languages are related by their definability in terms of the history.

Note that to facilitate a more concise expression of ECA rule sets, it may be useful to provide composition mechanisms over the atomic action $append(\delta)$. These mechanisms might range from simply sequencing of two atomic actions to making both $\mathbb{E} \subset \mathbb{A}$ and $\mathbb{C} \subset \mathbb{A}$, thereby opening the way for composite actions that are dependent on events and data. This can be realized by admitting statement forms such as *if-then* and *while-do* in the action part of ECA rules.

A Language for ECA Rules

It is now shown how $\mathbb{E}$, $\mathbb{C}$, and $\mathbb{A}$, viz., the previously defined ESL, CSL and ASL, respectively, can be used to define syntactically and semantically a language for ECA rules.

An ECA rule $E : C \rightsquigarrow A$ was previously defined as being the specification part σ of an AR ρ. Given the characterizations of $\mathbb{E}$, $\mathbb{C}$, and $\mathbb{A}$ above, it is now possible to strengthen the definition of the specification part of an AR as follows.

Given an AR $\rho = \langle \kappa, \sigma \rangle$, the specification part σ of ρ is an expression of the form $E : C \rightsquigarrow A$ where $E \in \mathbb{E}$, $C \in \mathbb{C}$, and $A \in \mathbb{A}$, and $\mathcal{V}(\mathbb{E}) = \mathcal{V}(\mathbb{C}) = \mathcal{V}(\mathbb{A})$. Note that, under this definition, information can flow from/to event detection, condition verification and action execution by first-order unification.

Consider an ECA rule that detects the event of a change in the salary of an employee, verifies whether the new salary exceeds that of the employee's manager, and, if so, executes the action of raising the latter's salary to that of the employee. This can be written as:

```
happened(EventId, referencing-act, put),
happened(EventId, referenced-property, salary),
happened(EventId, reference-source, Employee),
happened(EventId, reference-target, EmployeeSalary)
:
    holdsNow(property(Employee, manager, Manager)),
    holdsNow(property(Manager, salary, ManagerSalary)),
    ManagerSalary < EmployeeSalary
⇝
        append(put_salary(EmployeeSalary)@Manager)
```

It is not difficult to devise friendlier syntax to express the same semantics, e.g.,

```
happened(put_salary(EmployeeSalary)@Employee)
:
    get_manager@Employee == Manager,
    get_salary@Manager == ManagerSalary,
    ManagerSalary < EmployeeSalary
⇝
        put_salary(EmployeeSalary)@Manager
```

in which the event specification uses surface form rather than clausal form, and the condition specification uses both the syntax of ROCK & ROLL and the default that the current clock time gives the time point of interest for condition verification.

The reading associated with $E : C \rightsquigarrow A$ is thus *when* (E *and* C) *do* A. Given that E and C are query expressions over $\mathbf{E}^\star$ and $\mathbf{C}^\star$, respectively, and that $A = append(\delta)$ is a set union operation over two sets of clauses, viz., the history $\mathbf{H}$ and the clause set denoted by *clausal(deliyanni-kowalski(internal(δ)))*, the operational semantics of the specification part of an individual AR is intuitively defined as: if evaluating E over $\mathbf{E}^\star$ succeeds and evaluating C over $\mathbf{C}^\star$ succeeds, then append to the history the event occurrence defined in A; in any other case, do nothing. At each stage of the evaluation bindings flow bidirectionally by virtue of the set of variable symbols of $\mathbb{E}$, $\mathbb{C}$, and $\mathbb{A}$ being shared between them.

Recall the *evaluateECAset* procedure depicted in Figures 5.3-5.5. In *evaluateECAset*, *bind* becomes the unification procedure over terms in $\mathbb{E}$, $\mathbb{C}$ and $\mathbb{A}$. The binding generation procedures *detect* and *verify* and the transition effecting procedure *execute* can be more precisely characterized as follows:

- *detect* becomes the query evaluation procedure associated with the DDB $\mathbf{E}^\star$: it takes an event specification E and a DDB $\mathbf{E}^\star$ and returns the corresponding answer substitutions, i.e., pairs of the form $\mathcal{V} \times Val$ such that, for some substitution instance F of E, it is the case that $F \in cons(\mathbf{E}^\star)$. In line with the general signature given above, the ECA rule interpreter, as it reacts (or is made to react) to new events, logs which events it has driven, thereby making it possible to partition the EDB of $\mathbf{E}^\star$ accordingly.

- Similarly, *verify* becomes the query evaluation procedure associated with the DDB $\mathbf{C}^\star$: it takes a condition specification C and a DDB $\mathbf{C}^\star$ and returns the corresponding answer substitutions, i.e., pairs of the form $\mathcal{V} \times Val$ such that, for some substitution instance F of C, it is the case that $F \in cons(\mathbf{C}^\star)$. Note that, whereas the general signature given above considers only one database state, $\mathbf{C}^\star$ is capable of making all known states available for condition verification.
- Finally, *execute* need not effect transitions by directly mapping a database state into another, insofar as it can achieve the same effect simply by adding event occurrences to the history. These include any composite events detected in the process of evaluating event specifications. In any case, *execute* may generate new events in the process, as indicated by the general signature given above.

In summary, because E, C and A can be given a logical interpretation, *bind* reduces to first-order unification. In the case of *detect* and *verify*, their complexity and termination properties will be those of the query evaluation procedures associated with $\mathbf{E}^\star$ and $\mathbf{C}^\star$, and these in turn will depend on the expressiveness of the DRs that comprise their intensional parts. Finally, *execute* amounts to no more than a set union operation.

All the contributions of Section 5.2 have been prototyped using an an evaluator, written in Prolog, that implements the architecture of Figure 5.6 and processes ECA rules in the language described above (i.e., rules of the form $E : C \rightsquigarrow A$ where $E \in \mathbb{E}$, $C \in \mathbb{C}$, and $A \in \mathbb{A}$) using the operational semantics described in Figures 5.3 to 5.5.

Summary of Chapter 5

This chapter had the main goal of showing that the axiomatic approach, through its sharing a large part of the metalogic of logic programming, leads to DOODs that are extensible both in their semantic content and their functionality.

Section 5.1 showed how the axiomatic approach facilitates the task of capturing more application independent semantics in DOODs. It was shown how finer-grained forms of declaring inheritance relationships can be catered for, and how constraints on entity participation in relationships can be expressed.

Section 5.2 showed how the approach facilitates the provision of more functionality, e.g., to make DOODs generated by the approach more self-sufficient. It was first shown how DOODs can be embedded into a theory of state change, thereby characterizing a notion of updates and state transition. Then, a general account of how passive DOODs can be extended with active behaviour was detailed.

This chapter has provided concrete evidence to substantiate the claims that axiomatically characterized DOODs are extensible. It has described two typical extension paths by means of which more domain independent semantics and more functionality can be captured in DOODs in an unencumbered manner.

Summary of Part II

The second part of the dissertation is concluded. It treated the two questions of how the axiomatic approach can be applied and extended, as follows:

Chapter 4 substantiated the claim as to the practicality of the approach through a discussion of how it has been used in the development of ROCK & ROLL, a fully operational DOOD system developed at Heriot-Watt University.

Chapter 5 substantiated the claim as to the extensibility of the approach through a description of two extension paths that lead to DOODs that capture more domain independent semantics, on the one hand, and more functionality, on the other.

The third part, spanning Chapters 6 and 7 brings the dissertation to an end by surveying related work and highlighting the contributions and future directions resulting from the approach described, applied and extended in Parts I and II.

Chapter 6

A Survey of Related Work

This chapter surveys related work on the characterization of DOODs in order to highlight similarities and contrasts between the contributions reported in this dissertation with those of other researchers. The chapter also includes a separate discussion of alternative approaches to the problem of integrating deductive and active rules in databases.

The Introduction to this dissertation presented an overview of the most influential lines of DOOD research. However, a detailed discussion was postponed until the full technical development of the contributions of this dissertation had been elaborated. This development, spanning Chapters 1 to 5, is now complete, and a survey of related work can be presented against that background.

In the Introduction, attention was drawn to the breadth and depth of work on the logical foundations of object-oriented concepts. The Introduction argued for focusing comparison efforts on proposals that have a database slant and aim at a proof-theoretic approach to the integration of logic and databases, since much research has been done or is underway that bears more on issues of concurrent programming and of type theory than on the theory of databases. However, even under this more focused view the field of DOOD research is varied.

Research on DOODs has accumulated rapidly since the late 1980s [see e.g., (Ceri et al., 1993; Delobel et al., 1991; Kim et al., 1990; Mumick, 1993)]. Such a sudden surge in interest is one possible reason why the body of research looks, in the mid-1990s, somewhat unstructured, since there has scarcely been time for proposals to inform, and hence relate to, one another. Another reason for this state of affairs is the lack of consensus on what constitutes an answer to basic questions regarding the definition, scope and purpose of DOODs. In part, this variety of opinions has been inherited from the field of OODB research, which is generally recognized as having traditionally favoured a hands-on, implementation-driven approach in preference to a foundational, theory-driven one.

Because there are relatively few general trends and not enough evident similarities between proposals, the task of reviewing and objectively comparing proposals and approaches to the characterization of DOODs is not straightforward. This chapter focuses on proposals that share with the research reported in this dissertation the goal of characterizing DOODs as logical theories that model object-oriented concepts and admit an automated proof procedure.

This chapter has two sections.

Section 6.1 surveys related work on the characterization of DOODs and concentrates on ascertaining the presence of properties and functionalities. It uses a set of requirements (derived from Table 0.1 in the Introduction) to investigate the extent to which each individual proposal does or does not meet the strategic goals embodied in the concept of a DOOD system.

Section 6.2 surveys and compares work that, like that described in Chapter 5, endeavours to integrate deductive and active rules in database systems.

6.1 A Survey and Comparison of Proposals to Characterize DOODs

6.1.1 The Comparison Framework

The Introduction to this dissertation argued that DDBs and OODBs complement one another in providing the majority of functionalities expected of next generation DBMSs. Hence, DOODs promise not only to unify two influential research programmes on data modelling, but also to enhance the usability of database technology in a significant way.

To fulfill that promise, DOODs should remedy the shortcomings of RDBs that DRDBs and OODBs remedy, while preserving the qualities of RDBs that DRDBs and OODBs preserve. In other words, approaches to the characterization of DOODs can be assessed with respect to a set of requirements, summarized in Table 6.1, that corresponds to Table 0.1 in the Introduction.

If the DOODs characterized by a proposal satisfy all of R.1 to R.9, then that proposal remedies all the shortcomings of RDBs that DRDBs and OODBs remedy, and preserves all the qualities of RDBs that DRDBs and OODBs preserve.

Table 6.2 lists, in alphabetical order of authors, the individual proposals that are surveyed, in varying level of detail, in the next two subsections. Note that proposals whose reference is marked with a superscripted asterisk were not given names by their authors. The name that appears in the left column of Table 6.2 is invented for ease of reference.

The requirements in Table 6.1 imply that Table 6.2 is not as inclusive as it could be if a looser view of what counts as a DOOD were adopted.

For example, a declarative query language for an OODB is left out if it is not presented as (the language of) a database theory. Prominent among these are DQLs for OODBs that are either extensions of SQL or SQL-inspired (such as Object SQL (Annevelink et al., 1995), OQL[C++] (Blakeley, 1993; Blakeley, 1995) and UniSQL/X (Kim and Kelley, 1995)), and languages that place themselves within specializations of the logic programming paradigm. For example, equational or functional logic programming (such as (Alagić et al., 1993) or MaudeLog (Meseguer and Qian, 1993)), constraint logic programming (such as $\mathcal{QUIXOTE}$ (Yokota et al., 1993)), contextual logic programming (such as *Class&Instance* (Omicini and Natali, 1994)), and so on.

Some account of object identity and some form of support for nonatomic values are also considered to be mandatory. The first of these excludes proposals to extend a semantic data model with deductive rules such as ERL (Battista and Lenzerini, 1993; Battista and Lenzerini, 1989) or ERCLog (Ye et al., 1994), and the second excludes proposals such as C-Datalog (Greco et al., 1992).

Note, finally, that Table 6.2 only lists the latest member in any sequence of evolving languages. However, when it helps to highlight the points under consideration, the discussion below does make reference to predecessors in language sequences.

6.1.2 Related Proposals

IQL+ (Abiteboul, 1990) succeeds **COL** (Abiteboul and Grumbach, 1987) and **IQL** (Abiteboul and Kanellakis, 1989a) and is the latest member of this family of languages from INRIA, in France, all of which have attracted considerable attention in the DOOD research community. Since IQL+ is defined with reference to both COL and IQL, a brief description of their main features is now given.

COL extended Datalog with strong typing and type constructors and introduced the notion of multivalued data functions (i.e., an extensional or intensional database relation that satisfies a functional dependency) for manipulating constructed types. Only a fixpoint semantics was described and it required stratification (Ceri et al., 1990b) not only because of negated atoms in the

R.1	**formal basis**	The availability of a mathematical characterization of the OODM underlying the DOOD.
R.2	**user empowering**	The possibility of declaring, manipulating and querying the database in a way that, through optimization and tractable evaluation, is guaranteed not to impose excessive penalties on casual users.
R.3	**implicit information**	The provision of means to express more complex kinds of implicit information (e.g., derived data) that can be used to extend the information content of explicit states of application domains.
R.4	**integrity constraints**	The provision of means to express more complex kinds of general information (e.g., business rules) that can be used to constrain the set of possible states of application domains.
R.5	**recursion**	The possibility of using recursion to model structural and behavioural knowledge.
R.6	**identity**	The possibility of formally distinguishing the existence of an object from the state it happens to be in at any point in its existence.
R.7	**complex structures**	The provision of means to model properties that have nonatomic values.
R.8	**semantic orientation**	The provision of means to model relationships between types, and, in particular, of inclusion and composition.
R.9	**behaviour orientation**	The possibility of formally assigning specific behaviour to particular types, and, in particular, of modelling the dynamic aspects of the interaction with elements of a type using the notions of encapsulation, overloading, overriding and late binding.

Table 6.1: A Set of Desired Properties of DOODs

Proposal	*Reference*
IQL+	(Abiteboul, 1990)*
Datalog^{meth,flex}	(Abiteboul et al., 1993)
DTL	(Bal and Balsters, 1993)
loopl	(Bertino and Montesi, 1992)*
Obj-U-Datalog	(Bertino et al., 1994a)
Logres	(Cacace et al., 1993)
LAURE	(Caseau, 1991)
OOLP+	(Dalal and Gangopadhyay, 1990)
Peplom^{d}	(Dechamboux and Roncancio, 1994b)
Gulog	(Dobbie and Topor, 1993)
LIVING IN A LATTICE	(Heuer and Sander, 1992)
Orlog	(Jamil and Lakshmanan, 1993)
ConceptBase	(Jarke et al., 1993)
F-logic	(Kifer et al., 1993)
LLO	(Lou and Ozsoyoglu, 1991)
Noodle	(Mumick and Ross, 1993a)
Coral++	(Srivastava et al., 1993)
OIL	(Zaniolo, 1990)

Table 6.2: A Selective List of DOOD Proposals

body of rules but also because of set constructors and of the introduced notion of data functions. COL has been implemented at the level of a prototype.

IQL extended COL with a subset of the structural aspects of the object-oriented paradigm, viz., object identity and structural inheritance. It added union and intersection types (but no notion of type extents being partitions) to the type system inherited from COL. IQL also stressed the need to separate the notion of a schema from that of an instance, thereby making clearer the typed nature of database entities already present in COL. The only semantics given to IQL was inflationary (Ceri et al., 1990b). One distinguishing feature of IQL was its option for supporting both the notion of a class (as a set of object identifiers) and that of a relation (as a set of values). This provided a choice between an identity-based and a value-based style of expression. The authors point out that this option made IQL fully compatible with Datalog.

IQL was an elegant and influential proposal, although its elegance was sometimes only achieved by constraining the modelling constructs it supported, (e.g., class extensions must be disjoint). The COL proposal carried with it significant metalogical complications (e.g., those caused by data functions) and represented a loss, with respect to Datalog, of desirable metalogical properties (e.g., the finiteness of interpretation domains, and hence of the guarantee of decidable evaluation procedures). IQL did do away with data functions, but only because they could be modelled using the IQL primitive for the invention of object identifiers, and this feature caused metalogical complications of its own, including the possibility of nonterminating programs.

IQL+ inherits its approach to constructed types from COL and adds a list constructor. It revives the data functions used by COL and dropped by IQL, and, in addition, allows them to be single-valued. IQL+ inherits its approach to object identity, classes and inheritance from IQL, but disallows multiple inheritance. In comparison with COL and IQL, the main novelties of IQL+ are its approach to explicit control of execution and to behaviour modelling.

IQL+ aims at self-sufficiency as a database programming language in a way that neither COL nor IQL did. Although it relies on calls to external functions for certain tasks (e.g., aggregates such as *count*, *average*, etc.), it includes a notion of methods, is capable of effecting state transitions, and introduces a control part into rule programs where constructs are provided to sequence rules, to group them into procedures, and to wrap them with structures such as *if-then* and *while-do* that stipulate the flow of evaluation.

The IQL+ notion of methods encompasses the distinction between interface and implementation, but does not support encapsulation. It provides built-in functions to access elements in collection types, to dereference objects into their values and to project fields of tuples. IQL+ extends COL with an equality predicate which defines a single-valued data function in addition to multivalued ones. Single-valued data functions make possible the derivation of inconsistent facts, and since detecting this property statically is undecidable, the consistency of derivations has to be checked at run time. This is at best, inefficient, and at worst, unsafe (since the proposed measure to address this problem is a run time inconsistency check that can only cause execution to be aborted and the user to be notified). Overriding is also logically understood in IQL+ as a form of inconsistency. However, like negation, multivalued data functions and state transitions, it is handled as a nonmonotonic feature of the logical system requiring either stratification or explicit control structures, or both. Late binding is not addressed explicitly by IQL+.

Updates have an impact on the requirements of the underlying logic that goes beyond the nonmonotonicity problems caused by the possibility of the underlying domain undergoing changes during the computation of logical consequences. Specifically, the notion of updates in IQL+ requires the underlying logic to have a temporal dimension, in the sense that it is semantically relevant *when* a fact holds with respect to all the other facts that hold concomitantly with it. If, as the result of evaluating a single-valued data function, two distinct ground atoms are derivable at *different* times, then the latterly derived atom is construed as an update of the earlier one. However, if two distinct ground atoms are derivable at the *same* time, then the run time consistency check referred to above fails and evaluation is aborted.

It can be seen that IQL+ is semantically very complex. There seems to be no simple way of avoiding requiring of programmers that they devote significant attention to control of execution. Matters are made more difficult by the lack of a formal semantics for all but the purely logical aspects of IQL+. IQL+ is conceived of as having an operational semantics on three levels: an essentially declarative *local* level in which the fixpoint of a stratified rule set is computed; an essentially imperative *top* level in which rule liveness and priorities are relevant (in a way that resembles ADBs); and a semantically opaque *external* level which extends the local and top levels with the functionality of the language in which rules occur embedded. Only the semantics of the local level has been informally defined with reference to those of COL and IQL. It is unclear what the formal semantics of the top level is, and the prospects for the immediate future are not encouraging, judging by the state-of-the-art in ADB research (Campin et al., 1995; Paton et al., 1994). By definition, the formal semantics of the external level is devolved, in the sense that IQL+ does or does not have one depending on whether or not the host language has one. There is no record of either IQL or IQL+ having been implemented even at the level of a prototype.

With respect to Table 6.1, the adoption of the IQL data model means that IQL+ only partially satisfies R.1, in that the OODM that underlies IQL+ is structurally, but not behaviourally, object-oriented. IQL+ fails requirement R.2 because query evaluation may not terminate and DRDB-style optimization is made virtually impossible by the degree of explicit control that is required. R.3 and R.4 are both satisfied. Since recursion is fully supported, R.5 is satisfied. IQL+ fulfills requirements R.6 and R.7 very comprehensively, and still retains the possibility of a value-based approach by supporting the distinction between classes and relations. With respect to requirement R.8, IQL+ fulfills it to an acceptable extent. Semantic relationships are catered for to the extent that set, list and tuple types are acceptable equivalents of association, ordering and aggregation. Note, however, that there is no explicit distinction between attribution and aggregation, so that the OODM underlying ROCK & ROLL behaves more flexibly under inheritance than the IQL model. Finally, the IQL+ treatment of behavioural aspects of object-orientation only partially fulfills R.9. On the one hand, particular types can be assigned specific behaviour, on the other hand the treatment of overriding is such that abortive run time errors can arise from it. More importantly, there is no support for encapsulation.

Datalog^{meth,flex} (Abiteboul et al., 1993) is a proposal to extend Datalog with the behavioural aspects of the object-oriented paradigm. This makes it a member of a very small group of proposals that address those aspects, and suffices to make the proposal noteworthy. Another aspect of the proposal that is of interest in the context of this dissertation is that the language is given a semantics by a mapping into Datalog with negation. This, of course, has similarities with the approach described in this dissertation and embodied in ROCK & ROLL. The main drawback of Datalog^{meth,flex} is that it *only* addresses behavioural aspects. In particular, the class of domains and the type system that underlie Datalog^{meth,flex} are those of the RDM. Datalog^{meth,flex} terms can only denote atomic values, and schema declaration provides no semantic modelling capabilities other than classification and behavioural inheritance.

The inheritance graph is described as a forest and not, as usual, a partial order. Inheritance is single. It is claimed, however, that support for a partial order graph and for multiple inheritance would be straightforward. As in IQL, class extensions are partitions, and, unlike IQL+, Datalog^{meth,flex} has no support for union or intersection of class extensions. Because the operational semantics of Datalog^{meth,flex} is a function of the rewriting into Datalog with negation, inheritance is only addressed by syntactic means, viz., the explicit inclusion of application independent axioms that determine the way in which the inclusion and the classification relationships interact. (The axioms are those expressing the transitivity of the inclusion relationship and the transitivity, over inclusion, of the classification relationship.) There is not much explicit knowledge about, and few constraints on, the inheritance graph, and hence the treatment given to the consequences of inheritance is less uniform and comprehensive than would be desirable, as discussed below.

In the tradition of COL/IQL/IQL+, a functional constraint is implicit for every method. This is handled in such a way that methods, even though defined by clauses, cannot be invoked with varying instantiation patterns, as usual in the logic programming paradigm. Predicates can be defined that do have the flexibility above, but they have global scope, i.e., they are not construed as being attached to any class, and hence, cannot be said to be part of the behavioural modelling process.

Overloading and overriding are adequately catered for by the rewriting into Datalog with negation. Essentially, the approach is to plant negated class membership predicates into certain elements of the set of rules defining a given overloaded predicate in such a way that overriding falls out from the evaluation process of Datalog with negation. There is support for late binding but, in line with the functional view of methods, a method call must be ground for late binding to take place. Late binding is unsafe, in the following sense: because there are no guarantees, in certain cases, that the inheritance graph is well formed, late binding may fail (causing, presumably, evaluation to abort, though this is not discussed in the proposal). Late binding is also inefficient: again because of the lack of a comprehensive and exhaustive characterization of the formal properties of the inheritance graph, it seems that the whole hierarchy has to be recomputed at each resolution step for late binding to be possible, although this undesirable feature might be done away with at the implementation level.

Because Datalog^{meth,flex} does not adopt a structurally object-oriented approach, the notion of a state that is modelled independently of behaviour is not really supported. Since there is no independent notion of state, it is not surprising that the proposal, makes no mention of encapsulation or of the need to distinguish between the interface and the implementation of operations. There is no attempt at self-sufficiency; evolution towards it could presumably be achieved indirectly through extensions of Datalog with negation. There is no record of Datalog^{meth,flex} having been implemented even at the level of a prototype, but, by its very nature, it should require little implementation effort to develop a prototype on top of an engine to process Datalog with negation.

With respect to Table 6.1, since Datalog^{meth,flex} is not structurally object-oriented, requirement

R.1 is only partially satisfied by the rewriting into Datalog with negation. R.2 is compromised by the unsafeness of late binding, but this should not be very difficult to overcome in the context of a more comprehensive and more uniform treatment of inheritance. R.3, R.4 and R.5 are satisfied. However, it should be noted that the proposal of Datalog^{meth,flex} seems to prohibit negation in rule bodies, even though the language is mapped into Datalog with negation. If this is confirmed, then the expressiveness of the language for use in practical situations is significantly undermined. Since there is a notion of object identity, R.6 is catered for. However, the notion is almost vacuous when, as is the case, there is no independent notion of state. The more significant shortcomings of Datalog^{meth,flex} are brought into focus by its failure to meet requirements R.7 completely and R.8 partially. The type system is that of the RDM, and hence there is no support for nonatomic values. This characteristic, in the context of a complete lack of self-sufficiency, means that the embedding of Datalog^{meth,flex} in typical computationally complete languages would run the risk of incurring a severe type-related impedance mismatch. Inclusion and classification are catered for, but type relationships that model other structural aspects of application domains are completely absent. Inheritance is supported, but not uniformly and not comprehensively. Finally, R.9 is the strongest point of Datalog^{meth,flex}.

loopl (Bertino and Montesi, 1992) is the culmination of a proposal to incorporate a subset of behavioural object-oriented concepts into a logic programming language in three stages. Taking a subset of $\mathcal{LDL}$ (Naqvi and Tsur, 1989) as its starting point, the first stage introduces notions of objects and of message passing to yield, in the terminology of the authors, a logical object-based programming language (which, in this dissertation is denoted by **lobpl**). The second stage introduces the notion of a class template into **lobpl** to yield a logical class-based programming language (denoted by **lcbpl**). Finally, in stage three, a notion of inheritance is introduced and yields a logical object-oriented programming language (denoted by **loopl**).

loopl is concerned with expressing state transition by annotated subgoals (formed from a set of action predicate symbols that is disjoint from the standard ones) in the bodies of rules. These follow the style, and have the update semantics of, $\mathcal{LDL}$, which is, in turn, formalized using dynamic logic. The available annotations allow the insertion and deletion of ground facts. There is an expressiveness limitation regarding updates though, in that one can write a transition effecting rule into the IDB but cannot issue a transition effecting query.

Like Datalog^{meth,flex}, **loopl** does not model complex structures, but unlike the former, it does have a notion of state. In the context of this dissertation, a noteworthy characteristic of **loopl** is that, like the languages of the theories described in Chapter 3, and unlike IQL+ and Datalog^{meth,flex} (as well as the majority of proposals reviewed in this chapter), it construes objects as theories and not as terms.

This opens the way for contrasting proposals with respect to whether they adopt a *theory granularity* or a *term granularity* to model objects. This is an important choice, since a theory granularity corresponds to the logical modelling of an object as the set of logical consequences that can be derived from the known facts about the real world entity of which the object is a representation, and hence, an object identifier denotes a theory. Term granularity corresponds to the logical modelling of an object as an expression which denotes a real world entity. Syntactically speaking, the latter is a simpler approach, but it is semantically more complex in that it introduces, among others, problems such as those of object identifier invention and of infinite Herbrand domains. It also reduces the extensibility of the approach, insofar as application semantics cannot but become inextricable from the purely syntactic formation rules for terms. For example, the semantics of association, ordering and aggregation are hardwired into the formation rules for set, list and tuple terms, respectively. This means that one is barred from enriching the set of axioms that determine how associations, orderings and aggregations function from the point of view of a database designer. Chapter 5 provided an example of why one might want to do that when it considered how participation and cardinality constraints can be imposed on relationships.

With a theory granularity, the clause base can be partitioned into labelled theories, where the labels are object identifiers. This allows the sending of a message to an object to be elegantly modelled by **lobpl** (and by the DOOTs/DODTs characterized in this dissertation) as the parameterizing of a goal by the partition of the clause base within which resolution is in scope. In the case of **loopl**, the evolution of the theory as a consequence of updates models transitions in the corresponding object state.

What distinguishes **lobpl** from **lcbpl** is the fact that the latter replaces object identifiers with class templates and replaces asserted facts by fact templates. This is, intuitively, a move from an instance-based viewpoint to a schema-based one. A class template assumes the role of denoting one of the theories into which the clause base is partitioned. Templates are essentially labelled tuples of terms. A class template determines, via the fact templates that occur in it, the structure imposed on the instances of the class it denotes.

This approach to classes requires the introduction of a built-in predicate *new* that invents a new object identifier, places it in the extension of the class denoted by the class template passed as argument, and, at the same time, instantiates (with the arguments with which *new* was invoked) the state of the object denoted by the new object identifier. This instantiation has the structure imposed on it by the class template. Associated built-ins are introduced to *remove* an object from the extension denoted by a class template, and to define the classification relation between objects and classes.

An unfortunate consequence of this approach to classes is that one of the advantages associated with a theory granularity that was exhibited by **lobpl** cannot be preserved in **lcbpl**, viz., in **lcbpl** the metalogical complications of object identifier invention (as exhibited by IQL and IQL+) resurface intact. Note that, by supporting updates and dynamic class extensions, this family of languages relinquishes declarativeness (and hence optimization opportunities) and decidability (insofar as termination of evaluation cannot be guaranteed).

loopl, the final language in the family, extends **lcbpl** with syntactic means to specify how classes inherit from one another. This process of inheritance is given informal meaning in terms of how theories inherit axioms from one another. There are two specifiable inheritance modes. The first mode is referred to as *extended*, the second mode is referred to as *overriding*. The first amounts, roughly, to taking the union of the axioms of the theory and the supertheory; the second amounts to taking the intersection of the two axiom sets. (There are qualifications to the preceding descriptions, but they are not very relevant for this survey). Rules are assigned global names that must be unique within any theory but may occur in more than one theory. This plays a part in exactly how intersection of axioms sets is performed.

It is important to note that inheritance is not seen as an inclusion relationship between class extensions, but rather as a set of operations on sets of axioms. Certain aspects in this view of inheritance are rather underspecified in the published proposal (e.g., class templates contain variables but nothing is said about their scope, and the examples supplied in the paper do not throw light on the issue). Insofar as union and intersection of axioms are purely syntactic operations, one would wish to have guarantees that inheritance in **loopl** is sound, especially since this is a very unconventional approach to capturing the notion of inheritance.

This family of languages embodies many ideas of interest. However, it completely lacks a formal semantics. The paper contains no semantic treatment beyond a remark to the effect that **lobpl** without actions can be given a semantics derived from that of L&O (McCabe, 1992) and that updates can be given the semantics of $\mathcal{LDL}$. Nothing is said about **lcbpl** and **loopl** apart from a remark as to the added difficulty in providing them with a formal semantics. There is no record of any member of the family having been implemented even at the level of a prototype.

With respect to Table 6.1, **loopl** completely fails to satisfy requirement R.1. Because it is order dependent and capable of effecting state transition it is less optimizable and safe to evaluate than would be expected for the language to be usable by casual users. Hence, requirement R.2 is also not met. Requirements R.3, R.4 and R.5 are met in the standard way for logic programming languages. Requirement R.6 is met, but the proposal does not fully develop answers to the problems of object identifier invention, especially in the context of side effecting goals. All the values in **loopl** are atomic, and hence it fails requirement R.7. The only semantic relationships considered are classification and inheritance, with the result that it only partially satisfies R.8. Even then, its treatment of inheritance is quite distinct from that of other proposals and little can be said of its merit without a more detailed development. Requirement R.9 is partially fulfilled, but the conspicuous absence of an underlying type system is manifest in the lack of a distinction between interface and implementation. Encapsulation is not modelled even though a message-passing style of communication is supported. The treatment of overriding is prejudiced by the lack of a more robust treatment of inheritance. Late binding is not considered.

Obj-U-Datalog (Bertino et al., 1994b) reuses many of the ideas put forth in **lobpl**. In fact, in many respects, the proposal can be seen as **lobpl** revisited for the purposes of providing a formal semantics to updates and transactions. This is achieved by a translation into **U-Datalog** (Montesi, 1993), for which a bottom-up interpreter is available. U-Datalog supports update atoms in the bodies which can insert and delete atoms from the EDB within transactions. A transaction is a query containing update atoms. A transaction is processed in two phases: in the first phase, each update is collected but not performed; in the second phase, the updates collected in the first phase are performed as a single batch, provided they are consistent. If an inconsistency is detected, no update is performed and the transaction is aborted. Stripped of updates and transactions, U-Datalog reduces to pure Datalog, i.e., it contains no object-oriented notions of structure or behaviour.

Obj-U-Datalog extends U-Datalog with a notion of objects as theories that is, in practical terms, that of **lobpl**. The proposal of Obj-U-Datalog is much more intent on modelling cooperating sets of objects (whose states may evolve as a result of that cooperation) than on the management of data using the deductive and object-oriented paradigms. While Obj-U-Datalog corresponds to **lobpl**, there has been no attempt to extend Obj-U-Datalog with classes and with inheritance, and hence there are no extensions of Obj-U-Datalog that correspond to **lcbpl** and **loopl**. Obj-U-Datalog has been implemented at the level of a prototype via a translation into U-Datalog, for which a bottom-up interpreter exists over a persistent Prolog system. There is no report of query optimization having been implemented.

With respect to Table 6.1, Obj-U-Datalog fails requirement R.1. Of the concepts and functionalities associated with an OODM it only caters for object identity and cooperation through message passing. The proposal is best seen as an attempt to model cooperation in a DRDB context than as a DOOD proposal. With respect to requirement R.2, the remarks on **loopl** are also applicable here: Obj-U-Datalog is order dependent and capable of effecting state transition, and therefore it is less optimizable and safe to evaluate than one would wish for if the language is to be usable by casual users. Requirements R.3, R.4 and R.5 are met in the standard way for logic programming languages. Requirement R.6 is met, but, like **loopl**, the proposal does not fully develop answers to the problems of object identifier invention. All the values in Obj-U-Datalog are atomic, and hence it fails requirement R.7. It caters for no semantic relationships (not even classification and inheritance), and hence it fails requirement R.8. Requirement R.9 is not fulfilled, except that the language provides for a message-passing style of communication between objects modelled as theories. Encapsulation, understood as the transparency to users of implementation choices, is not catered for, but the transaction mechanism does protect the state of objects from being acted upon by anything other than other objects. Since classification and inheritance are not catered for, the notions of overloading, overriding and late binding are not applicable.

Logres (Cacace et al., 1990; Cacace et al., 1992; Cacace et al., 1993) builds upon **Algres** (Ceri et al., 1988; Ceri et al., 1990a), though the relationship between them is more manifest in the implemented Logres prototype that in any formal treatment of Logres. The Logres data model is strongly inspired by that of IQL (e.g., in the approach to the invention of object identifiers, in retaining distinct support for classes and relations, in the definition of complex types by recursive application of type constructors, in the provision of data functions, etc.). Similarly, some shortcomings of Logres resemble those of IQL, among which the most important is the concentration on the structural aspects of the object-oriented paradigm to the complete exclusion of behavioural ones.

Logres has, however, some distinctive features. Of these, the most interesting are: system-generated referential integrity constraints and user-specified update modes. In order to capture some semantic aspects of relationships that bind the schema and instance levels, the Logres data model defines a small number of schema dependent conditions which, when satisfied, cause active rules to be appended to the rule base explicitly defined by application designers. These appended rules are designed to ensure that classification interacts properly with inclusion, composition and aggregation (called in Logres, *is-a*, *part-of* and *association*, respectively). The semantics of updates in Logres (syntactically expressed by object identifier invention in the case of insertion and by negated rule heads in the case of deletion) is such that the appended rules can be seen as repair actions that are triggered by the normal querying and updating of a Logres database. For example, whenever an object is deleted from the database, tuples that refer to the deleted object are also deleted.

However, there are problems with this approach. For example, a sequence of updates may fail to terminate or may contain inconsistent database states, in which case the fixpoint computation flounders. This stands in contrast with, e.g., U-Datalog, where such a sequence of updates is rolled back without further consequences to the user. Another unsatisfactory aspect of the Logres proposal is the fact that many semantic properties that one might expect would cause rules to be added, do not. For example, the inclusion relationship is, as usual, defined to be a partial order, but no active rule enforces antisymmetry. Analogously, the asymmetry of composition is not enforced. In short, only constraints on references are considered.

The second distinctive feature of Logres is its support for user-defined update modes. In standard DDB theory, the semantics of a database D is given in terms of a function from the power set of the extensional Herbrand base of D to the power set of the intensional Herbrand base of D. If parameterized to a goal G, the target is restricted to instances of G. In Logres, a database contains, besides an intensional part IDB (which may cause side effects) and an extensional part EDB, the active part ADB discussed in the preceding paragraphs. The approach to the definition of the semantics of a Logres database is different from the standard approach applied to a Datalog database. The semantics is defined in terms of a mapping that takes as parameters not just a goal G, but also a set of rules R and/or a set of type declarations T. Evaluation is defined, again as in IQL, in terms of an inflationary fixpoint operator. The idea is that T and R, together, can define a context (called an *application* in Logres) for the computation over the database $D = \langle EDB, IDB, ADB \rangle$. For example, if D is a `road` network that `connects cities`, then T might define a type `path`, R might contain rules defining instances of `path` as elements of the transitive closure of `connects`, and G might be a query over the extension of `path`. User-defined update modes come into play with the need to determine the persistence of any side effects produced during evaluation. Thus, it is possible to add, delete or keep invariant the IDB, and it is possible to make updates to the EDB persist or not. The authors note that this mechanism allows, if a user so wishes, the retention of a purely declarative semantics for Logres databases.

The Logres proposal, like that of IQL+, is intent on defining DOODs that are extensions of DRDBs. In many ways, the technical development of Logres and IQL+ is heavily influenced by the fact that not only do they take Datalog as the foundation in terms of which extensions are defined, but the DOODs characterized by Logres and IQL+ coexist with DRDBs through their class-relation dichotomy. However, the Datalog source is nowhere more manifest than in the fact that Logres, like IQL, does not address the *integrated* modelling of structure *and behaviour* that is the hallmark of OODBs. Logres has been implemented on top of the Algres database engine.

With respect to Table 6.1, Logres satisfies requirement R.1 but only with respect to structural aspects. It only partially satisfies R.2, insofar as queries can be optimized (after translation into Algres) but evaluation is not safe. Also, catering for updates implies renouncing opportunities for optimization insofar as some degree of order dependence is unavoidable. Requirement R.3 is satisfied in the standard way for DDBs. R.4 is also satisfied and can recourse to repair actions at the price of nonterminating programs. Requirement R.5 is also catered for as usual in DDBs. The need for object identity and complex structures is handled as in IQL+, i.e., Logres fulfills requirements R.6 and R.7, and still retains the possibility of a value-based approach by supporting both classes and relations. Again like IQL+, Logres fulfills requirement R.8 insofar as set, list and tuple types are acceptable equivalents of association, ordering and aggregation. Logres also has a multiset type constructor. Note, however, that in common with all approaches based on type constructors (IQL+ included), Logres falls short of the modelling capabilities of a typical semantic data model in being unable to assign properties to constructed types. This is, in data modelling terms, a failure to distinguish, and hence provide independent support for, both attribution (i.e., the assignment of properties, which can be unrestrictedly accumulated by inheritance) and composition (i.e., the definition of a type as being constructed from one or more types, which can only very restrictively be specialized through inheritance). Finally, Logres completely fails R.9. The Logres notion of an application associates types and rules with a specific query. However, R.9 requires types to have specific behaviour, states to be encapsulated and message sending to be the computational model relying on behavioural inheritance, overriding and late binding. Logres supports none of the above.

OOLP+ and **OOLP** are a pair of languages proposed in (Dalal and Gangopadhyay, 1990). OOLP+ shares two important tactical decisions with ROCK & ROLL: it adopts a theory granularity (as does **loopl**), and its semantics is expressible by a rewriting (into Prolog, whereas those of ROCK & ROLL and Datalog^{meth,flex} are into Datalog with negation).

The proposal is quite streamlined, perhaps excessively so from a database viewpoint. As suggested by its choice of acronym, the proposal has some features that characterize OOLP+ more as a programming language than as a database one. For example, in the tradition of untyped general purpose programming languages, OOLP+ is uniform (i.e., a class is also an object), which leads to a conflation of the notions of a schema and an instance (or type and token), the distinction of which is standard in the database field.

Another aspect in which OOLP+, like Datalog^{meth,flex}, is defective as a database language is the dearth of support for semantic data modelling concepts. It supports classification, attribution and inheritance, but not composition or aggregation. Its support for collections is limited to multivalued properties. Like **loopl**, it construes inheritance as an operation on sets of axioms but no detailed development of the consequences of this approach is offered (the OOLP+ proposal is even more laconic on this point than the **loopl** proposal). In short, in OOLP+ either one is constrained to atomic values or else one has to accommodate the unconstrained power of first-order terms in its entirety.

OOLP does not support overriding (i.e., behavioural inheritance is cumulative only). This is because the treatment of overriding is based on the explicit control of the search space, and the rewriting target of OOLP is pure, and hence order independent, Horn clauses. In order to capture overriding, OOLP gives rise to OOLP+ with a change of rewriting target to Prolog, in such a way that the *cut* extra-logical construct has to be relied upon to prune the search space in the intended manner. The programmer controls overriding by annotating the definition of methods in the surface language with *before* and *after* tags (the intuitive meaning being that a method annotated with the former is executed before any inherited method with the same name, and correspondingly for the latter). While the metalogic of OOLP is sufficiently simple to be of use in a database context, that of OOLP+ is as complex as that of Prolog with cuts. Somewhat surprisingly, the translation to Prolog is seen by the authors as advantageous from the point of view of self-sufficiency as well, insofar as they consider Prolog to provide an adequate framework for state transition.

The adoption of a theory granularity allows message sending to be elegantly modelled as context-specific resolution as mentioned above for **loopl** and ROCK & ROLL. There is no record of OOLP+ having been implemented even at the level of a prototype, but by its very nature it should require little implementation effort to develop a prototype on top of a Prolog engine.

With respect to Table 6.1, OOLP+ cannot be said to satisfy R.1, not only because its model lacks many essential features one expects in an OODM but also because the rewriting into Prolog is not adequate from a database point of view. Requirement R.2 is also not satisfactorily met, insofar as the metalogic of Prolog does not offer assurances of efficiency and termination as comprehensive as required by casual database users. R.3, R.4 and R.5 are satisfied in the standard way. R.6 is catered for but there is no provision to reconcile state transition with the lack of mechanisms for object identifier invention. R.7 is not satisfied, insofar as OOLP+ can only support nonatomic values if one models every complex structure with first-order terms. However, this is too expressive for database purposes and leads to nonterminating evaluation procedures. Requirement R.8 is minimally met by classification and inheritance. Here, as in **loopl**, a more thorough development of the notion of inheritance as operations on sets of axioms would be desirable. Finally, R.9 is only partially satisfied. Overriding is supported but has to rely on the Prolog cut. Late binding is not applicable, insofar as the language is essentially untyped. Encapsulation is not considered.

LIVING IN A LATTICE (Heuer and Sander, 1990; Heuer and Sander, 1991; Heuer and Sander, 1992) is a rule-based query language over a structurally object-oriented data model. The LIVING IN A LATTICE proposal lies in the confluence of many lines of development in DOOD research that concentrates on purely structural models. For example, its model of complex values is similar to those of IQL and Logres, its approach to object identifier invention refines that of IQL, and its formalization of inheritance is influenced by the semantic data model IFO (Abiteboul and Hull, 1987). Its main interest lies less in contributing new features and functionalities than in clarifying, through a deeper and more comprehensive formalization, features and functionalities of previous proposals. LIVING IN A LATTICE has not been implemented.

With respect to Table 6.1, the most important shortcoming of LIVING IN A LATTICE is the lack of any support for behavioural modelling, i.e., it completely fails to satisfy R.9. As to requirements R.1 to R.8, the summary given above for IQL and Logres is largely applicable to LIVING IN A LATTICE, apart from two points: LIVING IN A LATTICE does not support a value-based style as do IQL and Logres, and a minimal model semantics suffices for LIVING IN A LATTICE, whereas a more complex inflationary semantics is necessary for IQL and Logres.

F-logic (Kifer et al., 1993) is the latest proposal of a logical system capturing object-oriented concepts to come from the SUNY at Stony Brook, USA, in the wake of **O-logic** (Kifer and Wu, 1989b) and **C-logic** (Chen and Warren, 1989). The idea that binds them together is that of a *logic of objects*. This idea has made F-logic one of the most influential proposals in DOOD research. However, the approach to the characterization problem for DOODs embodied in F-logic (and the other object logics) is quite distinctive, as discussed in more detail in Chapter 7.

The metalogic of F-logic is the most complex among existing proposals for the characterization of DOODs. Although very comprehensive in the spectrum of object-oriented features for which a logical account is provided, the account given to some of those features in F-logic is less illuminating than one would wish in terms of informing the implementation of practical, effective DOOD systems. Of all proposals for the characterization of DOODs, F-logic seems to offer the longest and most difficult route to an implemented DOOD system.

Several features of F-logic are inherited from, or best understood against the background of, C and O-logic. For this reason, a brief account is now given of these two proposals. C-logic is a typed logic for complex objects with identity. Like all object logics, it relies on the notion of *labels* (sometimes also called *attributes*, or *features*) as another class of symbols in the logical alphabet. The proposal that inspired the use of labels in object logics was LOGIN (Aït-Kaci and Nasr, 1986). LOGIN used labels in the syntax of terms to capture inheritance directly by term unification. However, labels are used for a different purpose in object logics. In these systems, labels are predicate symbols and allow a more flexible syntactic representation of the state of objects than the classical one based on the fixed arity of function symbols and on the positional occurrence of terms.

In comparison with O and F-logic, C-logic has a much simpler metalogic (based on the availability of a straightforward translation into HCL). Syntactic similarities between the three proposals are largely misleading, insofar as a crucial difference between C-logic and the other object logics is that, in the former, a label is semantically construed as a binary predicate over objects, whereas in O and F-logic a label is semantically construed as a partial function from objects to objects (or to sets of objects), i.e., a functional constraint is implicit on the relation denoted by the label.

When labels are interpreted as binary predicates, the syntactic representation of an object can be thought of as syntactic sugar for a theory conjoining an atomic formula for each label. (In this view, a typing assertion can be construed as a unary predicate). This makes it possible for C-logic to have the standard proof and model-theories of HCTs. In contrast, the metalogic of O and F-logic is made much more complex by several changes of interpretation, prominent among which is their notion of labels as partial functions.

The practical consequences of these changes of interpretation are not slight. For example, notice that while an implementation of C-logic can, by reduction to HCL, share the optimization and evaluation techniques developed within the theory of classical logic programming, one must restart largely from scratch the development of such techniques for O and F-logic.

It is worth noting that, in its concern with metalogical simplicity, C-logic is closer to the axiomatic approach contributed in this dissertation than O and F-logic are. Unfortunately, however, C-logic has never been developed beyond its first, very elegant but excessively Spartan, proposal. C-logic was not in fact intended as a characterization proposal for DOODs. For example, it supports multivalued properties but not lists, it has no schema/instance distinction, no semantic data modelling capabilities, not even classification or inclusion, and no support for behavioural modelling.

O-logic[1] can largely be seen as a preparatory step for the proposal of F-logic. O-logic is a proper superset of C-logic: it allows labels to have set, and not just single, values and it introduces denotations for classes. However, the more significant differences between the two lie at the metalogic level. O-logic not only gives a semantic interpretation to labels that is, in the way described above, different from the one given in C-logic, but, more importantly, it also changes the interpretation that was given in LOGIN to the expressions that are formed with labels. In LOGIN, labels are used to construct ψ-terms. A ψ-term is a typed extension of a first-order term, and hence occurs in the argument positions of predicates. In O-logic, however, labels are used to construct *O-terms*, which are not terms (i.e., individual-denoting expressions) at all. Building on the notion of an *id-term*, which can, though not necessarily and not uniquely, denote an individual in the domain of interpretation, an O-term is a formula that asserts the existence and the value of (not necessarily all) the properties in the state of the object denoted by the id-term that governs it in a functor-like manner. Note that an object need not be fully determined by a unique O-term, i.e., more than one O-term may play a part in defining the state of an object. Moreover, O-terms may nest inside O-terms, in which case the nesting is equivalent to a conjunction of the formulas denoted by each O-term involved. Finally, and more surprisingly, an id-term is itself an O-term, which effectively means that a term can sometimes denote a truth value.

[1] Unless it appears qualified, by *O-logic* is meant what some prefer to call *revisited O-logic*. The distinction arises because the original proposal of O-logic is (Maier, 1986). This proposal was never widely publicized because it contained some technical problems summarized later in this section. The version in (Kifer and Wu, 1989b) is not only an extension but also a correction of (Maier, 1986), and has, as a consequence, superseded the latter for all technical purposes. When necessary, the proposal in (Maier, 1986) is referred to as the *original O-logic*. To further complicate matters (Kifer and Wu, 1989b) is quite distinct from (Kifer and Wu, 1989a), but since the latter is never referred to in the discussion that follows no confusion should arise.

The lack of full appreciation of the issues that arise from these changes of interpretation lie at the root of the problems that plagued the original O-logic (Maier, 1986). Broadly, the problems are related to the fact that, by definition, all variables in a clause, are universally quantified, and this is not straightforward to reconcile with the interpretation that an O-term is a formula that asserts the existence of a single object. Also, if an O-term appears as the head of a rule, then satisfying the body may imply asserting the existence of an object whose state is distinct from any other object that can be deduced from the state of database at that point in the proof, thereby raising the issue of object identifier invention. Finally, equality is an inherent part of O-logic and can, sometimes implicitly, lead to a database being globally inconsistent (i.e., the theory associated with it has no models).

O-logic addresses object identifier invention by making an id-term a Skolem function on the variables that determine the state denoted by the O-term. This solution has the drawback that it suffices to make Herbrand universes infinite and programs possibly nonterminating. O-logic only partially addresses the possibility of inconsistency by introducing the notion of "inconsistency-tolerant functional attributes", which attempts to allow the programmer to localize inconsistency in particular attributes. For every inconsistency-tolerant functional attribute, a pair of axioms is implicitly asserted into the database theory. The first axiom says that if a single-valued attribute of a given object has two values then it has value $\top$ (which is a constant specially introduced into the alphabet for this purpose). The second axiom says that if a single-valued attribute has value $\top$ then it has every possible value. Even if one conceded that consistency is not a necessary property of databases that are meant for practical use, this solution would have, among others, the drawback that an answer to a query may consist of the entire Herbrand universe associated with a database state, and hence is an infinite answer.

O-logic has a proof theory but, significantly, no proof procedure. Furthermore, there are reasons to believe that obtaining an acceptably efficient one is not straightforward. Consider SLD, the classical proof procedure for definite logic programs (van Emden and Kowalski, 1976). Its efficiency relies on the following two properties of the proof theory of HCL, among others. First, only one inference rule, is needed, i.e., resolution. This means that at every proof step there are no alternative rules to be considered, and hence the search space is maximally constrained by one of the parameters that determine its size. Second, first-order resolution relies on first-order term unification for which an efficient algorithm exists. This means that each step can be efficiently computed once the clauses involved have been selected for application.

In contrast, the proof theory of O-logic requires six additional rules, with the result that the search space is much bushier than that of HCTs. Even disregarding the computational complexity of the additional rules themselves, O-logic unification is much less efficient than that of HCTs because of set-valued labels. Adopting the $\mathcal{LDL}$ strategy of rewriting set-valued attributes as rules at compile time is possible, but the number of unification steps required remains exponential. In the light of the above characteristics, it seems that designing efficient algorithms to implement the proof theory of O-logic will not be a question of simply reusing the methods and techniques developed for HCTs and DDTs.

The additional metalogical complexity of O-logic compared to other DOOD proposals is evident. In spite of paying this price, O-logic still has many shortcomings from the point of view of a DOOD characterization proposal. Examples include the following. There is no support for ordered collections. There may exist objects which are not instances of any class (and hence, strong type checking is not feasible). Because the is-a hierarchy is a preorder only, two distinct classes may both be super and subclass of one another (i.e., inclusion is not antisymmetric). Any form of inconsistency seems indefensible from the point of view of day-to-day database usage, and the O-logic tolerance of it seems no exception. Finally, there is no distinction between a primitive value and other objects and, in particular, one is free to model any primitive value in a domain dependent way. In other words, O-logic is unsorted and, again, this seems perverse from the point of view of day-to-day database usage.

F-logic extends O-logic in the following major ways: a more comprehensive treatment of inheritance, a stronger notion of typing based on signatures, and an account of behavioural modelling based on generalizing the notion of attributes to that of methods. Signatures are used in F-logic to define typing constraints on the arguments and the result of methods. Through signatures, F-logic supports overloading, the two forms of which are more specifically referred to as (inclusion) polymorphism and arity polymorphism. The attributes of O-logic are generalized to the methods of F-logic, insofar as an attribute is simply a method with no arguments. Methods, like the labels of the original O-logic and the attributes of the revisited one, are interpreted as partial functions and may be single (i.e., functional) or set-valued.

Unfortunately, to the complexities of O-logic, F-logic adds several of its own. The major ones can be summarized as follows:

- F-logic collapses the notion of schema and instance, in the sense that, in F-logic, the set of class-denoting symbols is not disjoint from that of object-denoting symbols, as it was in O-logic. This conflation is akin to the conflation in O-logic of primitive values and objects, and their motivation seems to be the same, viz., to keep as simple as possible a metalogic that is not short of sources of complexity. The counterargument to the obvious drawbacks of this approach is that, at the implementation stage, a sorted version of F-logic can be adopted if so desired. There is no detailed consideration, however, of whether the metalogical complexity of sorted F-logic, which seems to have motivated the conflation in the first place, has a negative impact or not on the practicality of F-logic as a DOOD proposal.

- From a programmer's viewpoint, inheritance in F-logic is more involved than in any other proposal because of the conflation of objects and classes. This conflation means that, not only classes, but also objects may inherit properties. However, only classes can further propagate properties down the is-a hierarchy. The complexity which the programmer must face arises from the fact that it is hard in F-logic to nail down an id-term as being the denotation of an object or of a class, since it is allowed to be both.

- The conflation of objects and classes also means that subtle distinctions exist as to where the signature of a method should be declared if it is to be inherited correctly. In particular, there are cases in which an ad-hoc class has to be invented by the programmer solely for the purpose of making signature inheritance work properly.

- The generalization of attributes into methods is yet another conflation of concepts in F-logic. Essentially, it amounts to a refusal to distinguish between the structure and the behaviour of objects. The fact that F-logic does not distinguish the modelling of the structure of an object from the modelling of its behaviour means that F-logic does not acknowledge the need to support both. Note that this means, in turn, that the standard notion of encapsulated states becomes meaningless, which explains why the F-logic proposal chooses to view encapsulation as a typing issue at the metalogical level and not as a data modelling principle.

- An id-term can denote, depending on its position, an individual and a method in the same syntactic expression. The reason why this is possible is that both expressions are interpreted as functions. This is rather subtle because two senses of the word *function* are being conflated here: the first sense is model-theoretic; the second, computational. Whether this is an advantage or not is not obvious, but it is certainly consistent with the F-logic approach of making as few semantic distinctions between syntactic forms as possible.

- There is no type inference in F-logic, yet the proposal shows examples of queries that only produce correct results if type annotations, which can sometimes be very subtle, are explicitly provided by the programmer. This is another indication that the current state of metalogical development of F-logic is such that were the proposal to be implemented, its use in practical situations would be significantly hampered by the lack of features such as a type inferencing algorithm comparable to the one provided by ROCK & ROLL.

With respect to the proof theory of HCL, the situation is even worse in F-logic than in O-logic. Whereas O-logic needs six additional inference rules, F-logic needs fourteen. Even if some inference rules may be compiled away, it seems inevitable that the bushy search space of O-logic becomes bushier in F-logic.

With respect to Table 6.1, F-logic cannot be said to satisfy requirement R.1 as enunciated there. F-logic eschews the question of independently formalizing an OODM and chooses instead to define from scratch an entire system of logic. Insofar as there is a precise characterization of the interpretation structures of F-logic programs, one could say that one could reverse engineer the OODM of which those interpretation structures could be taken as induced instances. The fact remains, however, that no OODM is defined and no need has been felt of defining one. Requirement R.2 can only be said to be satisfied or not once algorithms exist for optimization and evaluation, which at present they do not. However, it can be said that F-logic programs have infinite domains and may not terminate. Moreover, an F-logic program may be inconsistent without it being possible to detect this condition statically, and thus it may flounder at run time. F-logic satisfies R.3, R.4 and R.5 in the standard way. Requirement R.6 is satisfied using Skolem functions as id-terms. Satisfaction of R.7 is partial, insofar as F-logic does not support list-valued properties, but only single and set-valued ones. With respect to requirement R.8, F-logic supports classification and inheritance, but the conflation of objects and classes makes for a more involved notion of inheritance than most other proposals. As to requirement R.9, F-logic preempts the very idea of behaviour orientation by not distinguishing the modelling of behaviour from that of structure. The result is that in F-logic the state of an object is the implementation of its interface, and vice-versa, i.e., there is no way of distinguishing between the two notions. This approach leads, for example, to an account of encapsulation that is only very remotely related to the way the notion is supported in classical object-oriented programming languages.

F-logic seems much harder to implement than any other DOOD proposal. The reason for this conjecture is the almost unchecked expressiveness that results from the malleability of its semantic foundations. However, this very malleability and expressiveness makes F-logic an ideal foundation from which derived logics can be defined that constrain, extend, or are simply inspired by it (or by logics that inspired it such as HiLog and LOGIN) in several ways. Several such proposals are known, among which DOODLE, Gulog, LLO, Noodle, Orlog and OIL provide, in different ways, the most fruitful insights into F-logic.

DOODLE (Cruz, 1993c; Cruz, 1993a; Cruz, 1993b) was not listed in Table 6.2 because it is not really a proposal for the characterization of DOODs. It represents F-logic terms visually as a step towards blending them with visual objects and classes in a visual query language, whose semantics are ultimately defined in terms of F-logic semantics. DOODLE shows well how the malleability of F-logic can be taken advantage of in novel developments. DOODLE is not yet implemented and, of course, it cannot rely on an F-logic implementation either. A research question of some interest is how much of its F-logic foundations will subsist in DOODLE, if and when it is implemented.

Gulog (Dobbie and Topor, 1993), in its turn, discards most of the semantic malleability of F-logic (e.g., the conflation between schema and instance and that between primitive values and objects), enforces strong typing, compiles away set-valued properties in the $\mathcal{LDL}$ way, discards equality, proscribes programs for which overriding is not stratified and programs in which single-valued methods have distinct values. Gulog principally retains from F-logic certain aspects of the inheritance model and the signature-based approach to methods which generalize attributes.

The overall effect of these changes is to make it possible to endow Gulog with a bottom-up and a top-down evaluation procedure. That these procedures exist at all is more relevant for the purposes of this discussion than whether the class of programs on which they operate is interesting enough or whether their computational cost is acceptable (both of which remain open questions). Gulog can be seen as a practical version of F-logic, in the sense that Gulog is possibly the limit of how far the F-logic approach can go while still admitting a practical implementation. The proposal has merit in its own right and is elegantly developed.

LLO (Lou and Ozsoyoglu, 1991) is another object logic, though less directly related to F-logic than DOODLE or Gulog. It is more influenced, as F-logic was, by HiLog (Chen et al., 1993), a proposal to introduce second-order syntax in logic programs while still retaining their first-order semantics. LLO makes crucial use of second-order syntax in characterizing inheritance, and overriding in particular. The most original feature of LLO is its use of type variables whose instantiation patterns determine the inheritance hierarchy associated with a program and model a form of parametric polymorphism. The price for this expressiveness is paid in the unification algorithm. It is also debatable, from the majority of examples quoted, whether the subtleties of second-order syntax will not turn out to be an impediment for casual users.

LLO is yet another proposal that eschews the F-logic conflation of schema and instance, choosing instead to enforce the distinction clearly. The semantics of LLO is defined in terms of the semantics of COL, which as previously mentioned, may incur inconsistencies that cannot be detected statically. These are caused by the LLO *dereference* operation (to retrieve the denotation of an id-term) which is equivalent to the data functions of COL.

Noodle (Mumick and Ross, 1993a; Mumick and Ross, 1993b) is a difficult proposal to survey due to lack of information. In particular, there is no published formalization of Noodle. It is intended to be the query-only language of O++, the imperative language of the ODE OODB system (Agrawal and Gehani, 1989). Like F-logic and LLO, Noodle is influenced by HiLog in its reliance on second-order syntax, which makes schema browsing possible. Like LLO, but unlike F-logic, it enforces a clear distinction between schemas and instances.

A departure from the object-logic approach to collections in Noodle is that collections are not modelled by set-valued attributes, but as objects with an identifier of their own. Because Noodle is designed to be supported by O++, like ROLL is supported by ROCK, it can afford to be extremely streamlined. However, whereas the data model underlying ROCK is formally defined and suitably constrained, O++ is but a persistent C++. Furthermore, because Noodle and O++, unlike ROCK and ROLL, do not share the same model of data, impedance mismatches are likely to arise.

Orlog (Jamil and Lakshmanan, 1993) draws some inspiration from F-logic but, in the effort to characterize itself as a practical proposal, changes so many characteristics of F-logic that, as a DOOD proposal, it turns out to have more differences than similarities with the latter.

In fact, the concern with practicality leads Orlog to share certain tactics with the approach described in this dissertation, as follows: it is the logical formalization of a (in the Orlog case, semantic) data model, it adopts the view of objects as theories, its metalogic is reduced by mapping (like that of OOLP+) to that of a relational language (in the Orlog case, $\mathcal{LDL}$) and involves the generation of a set of domain independent rules (in this case, to enforce the underlying inheritance model).

Orlog adopts a second-order syntax in the style of F-logic but enforces the distinctions between schema and instance and between primitive values and object. Orlog also has a notion of extensional state that is distinct from the intensional state characterized by the methods defined on the object as in F-logic. One important improvement in Orlog with respect to F-logic is the support for relationships as objects. However, like F-logic, Orlog does not support ordered collections.

OIL (Zaniolo, 1990), like Orlog, is defined by a mapping into $\mathcal{LDL}$. However, it goes back, past HiLog and F-logic, to LOGIN for inspiration. OIL departs from it, though, in replacing the treatment of inheritance within the unification algorithm with one that relies on the above mentioned rewriting into $\mathcal{LDL}$. Its lack of ambition allows it be very elegantly formulated, but it is not straightforward to see how it could be more comprehensively developed.

As can be seen from the ways the six proposals above have constrained, extend or been inspired by it (or by its sources of inspiration), F-logic lies in one extreme of semantic malleability that is no doubt inspirational in its freedom from modelling assumptions but offers, on the other hand, very few opportunities for practical implementation. More than any other proposal surveyed in this section, F-logic accords privilege to expressiveness over efficiency, and hence is the most likely to remain, in its pure form, a theoretical construct only.

ConceptBase (Jarke et al., 1993; Jarke et al., 1990; Jarke and Staudt, 1993) is the outcome of one of the longest running research efforts to integrate deductive and object-oriented features in database systems and is fully implemented.

The ConceptBase system implements O-Telos, a variant of the Telos data model (Mylopoulos et al., 1990). O-Telos can be reduced to Datalog with negation, and this explains, in part, how ConceptBase is capable of successfully implementing it. Like ROCK & ROLL, ConceptBase can implement O-Telos with reference to the methods and techniques to optimize and evaluate queries that were developed for DRDBs.

ConceptBase also shares with the approach described in this dissertation the goal of exploiting axioms to define domain independent semantic relationships, such as classification, attribution and inclusion, without leaving the metalogic of DDTs. Like ROCK & ROLL, ConceptBase takes advantages of such axioms to define a special purpose object store that corresponds to OM in the ROCK & ROLL system. One crucial difference, however, between the axiomatic approach described here and that of ConceptBase is that the latter is only structurally object-oriented, i.e., there is no attempt whatsoever to support behavioural modelling in ConceptBase. Its authors recognize this as an important distinction by referring to ConceptBase as a *deductive object* database, as opposed to *deductive and object-oriented*. In the classification scheme adopted in Chapter 3, Concept Base only implements DODTs, not DOOTs.

Furthermore, the class of DODTs in ConceptBase is significantly less expressive than the one described in this dissertation. For example, ConceptBase has no support for collections, i.e., DODTs in ConceptBase do not model the associations and orderings described in Chapter 1.

Another significant difference between ConceptBase and ROCK & ROLL is that although the former has counterparts for OM and ROLL, it lacks any counterpart to ROCK. The chosen strategy to extend ConceptBase towards the computational completeness required for the development of complete applications is to use a client-server architecture centred around two generic operations: ASK (for querying) and TELL (for updating). Fully fledged imperative languages embed ASK operations and can TELL the server to effect state transitions. However, this architecture incurs impedance mismatches while that of ROCK & ROLL avoids them completely.

With respect to Table 6.1, ConceptBase only partially satisfies R.1 in that its data model is object-based but not object-oriented. Requirement R.2 is satisfied in that the reduction of O-Telos to DODTs opens the way for tractable evaluation procedures. Requirements R.3, R.4 and R.5 are met in the standard way, and the ConceptBase system already includes integrity checking functionality. Requirement R.6 is met. ConceptBase fails requirement R.7, since it cannot model complex structures. It partially satisfies R.8 in that it has inheritance and attribution but it does not support abstraction mechanisms to construct bulk types. Finally, ConceptBase is purely structural, and has no support for behaviour modelling, thereby failing requirement R.9.

ROCK & ROLL (Barja et al., 1995b; Barja et al., 1994; Barja et al., 1995a; Fernandes et al., 1993) was discussed in Chapter 4 as an application of the axiomatic approach to DOODs developed in Chapters 1 to 3. It comprises three tightly integrated software components: the OM object store, the interpreter for the ROLL language and the interpreter for the ROCK language. The effectiveness of ROCK & ROLL as a DOOD system arises not from any isolated component but rather from the synergy generated by the interactions of these tightly integrated components.

The OM object store implements an earlier version (Fernandes et al., 1992a) of the OODM formalized in Chapter 1. This means that ROCK & ROLL satisfies requirement R.1.

The ROLL interpreter implements a superset of the logically pure languages, $\mathcal{CL}^{\bullet}$ and $\mathcal{CL}^{\circ}$, described in Chapter 3. Recall that $\mathcal{CL}^{\bullet}$ is the Datalog-equivalent language that results from the logical reconstruction of the OODM described in Chapter 1. $\mathcal{CL}^{\circ}$ differs from $\mathcal{CL}^{\bullet}$ in providing an account of overloading, overriding and late binding in the context of deductive database languages, and is reducible to $\mathcal{CL}^{\bullet}$ in the way described in Chapter 3. In addition to $\mathcal{CL}^{\circ}$ (which can be referred to as *pure ROLL*), the ROLL interpreter supports calls to ROCK-defined predicates in queries and in the bodies of rules.

As discussed in Chapter 4, pure ROLL is far from self-sufficient. Pure ROLL is Datalog with negation over a structurally and behaviourally object-oriented model of data expressed as a set of domain independent rules and integrity constraints. Like Datalog, pure ROLL cannot effect state transitions (not excluding input and output operations) and does not have predicates such as *sum, average, count*, etc. Thus, pure ROLL is clearly not expressive enough for the development of complete applications. Like Datalog, pure ROLL relinquishes the ability to effect state transitions, in order to achieve order independence, thereby providing comprehensive opportunities for query optimization. Like Datalog, pure ROLL operates over finite, static domains and is equipped with tractable query evaluation procedures. Hence, pure ROLL satisfies requirement R.2. However, when ROCK-defined predicates are allowed in queries and in rule bodies, only a weaker claim can be made, as discussed below.

In the current version of the ROCK & ROLL system, ROLL is parsed into system graphs (Kifer and Lozinskii, 1990) which are an internal representation of queries. System graphs are translated into processing trees, a form of query execution plan (Lanzelotte and Valduriez, 1992). Processing trees are globally optimized using an extension of the static filtering method (Kifer and Lozinskii, 1990). Local optimization of processing tree sections is possible (e.g., using a query optimizer generator), but ROCK & ROLL does not yet include a local optimizer.

Because it is Datalog-equivalent, ROLL also satisfies requirements R.3 to R.5. Note that, as previously remarked, the current version of the ROCK & ROLL system is not equipped with a dedicated component to maintain integrity constraints. There is no technical obstacle to providing such a component. However, it is likely that ROCK & ROLL will evolve towards the provision of active database functionality (Paton et al., 1994), which subsumes the functionality required for integrity maintenance as traditionally understood in DDBs (Das, 1992).

The ROCK interpreter is the third component of ROCK & ROLL. It implements the ROCK language described in (Goñi, 1995). ROCK allows complete applications to be developed insofar as it is computationally complete. In particular, it possesses the means to effect state transitions (including updates, input and output operations, etc.) and to express functions such as *sum, average, count*, etc.

Several opportunities for the combined use of the two languages arise. The discipline that is enforced for such a combined use has largely been described in Chapter 4. One point to stress in the present context is the impact of allowing ROCK-defined predicates to appear in queries or in the bodies of rules. When this occurs, one switches from pure ROLL to (full) ROLL. The main consequence of this shift is that the termination of query evaluation can no longer be guaranteed, since it is not generally possible to guarantee that the evaluation of a ROCK-defined predicate will terminate.

Note, however, that the ROCK & ROLL system statically enforces the following constraint: a ROCK-defined predicate can only appear in a ROLL expression if it does not side-effect nonlocal data, i.e., no ROCK-defined predicate which appears in a ROLL expression is expressive enough to change the global state during query evaluation. Being capable of enforcing this constraint suffices to ensure that ROLL (and not just pure ROLL) is order independent and optimizable. The disadvantage of enforcing this limitation on the expressiveness of ROLL is that ROLL is a language over strictly static domains. In summary, one needs to weaken the claim made above that ROCK & ROLL fully satisfies R.2, insofar as query optimization of (full) ROLL is still comprehensive but termination is dependent on the undecidable termination of ROCK-defined predicates whenever such predicates occur in a query or in the body of a rule.

Both ROCK and ROLL support all of R.6 to R.9, as follows. An object has an identity that is completely independent of its state and its behaviour, and hence ROCK & ROLL satisfies R.6. With respect to R.7, complex structures are supported, insofar as an object can be characterized as set-like (i.e., as association), sequence-like (i.e., as an ordering) or as record-like (i.e., as an aggregation). Because all such structures are required to be flat (in the way described in Chapter 4), no function symbols need to be introduced in the logic language. This approach does not imply any lack in expressiveness compared to other proposals, although it has the disadvantage that the modelling of complex types is more verbose than in most proposals. The crucial advantage of this approach is that it retains the finite interpretation domains of DRDBs. With respect to R.8, both ROCK and ROLL support inclusion as the basis for structural and behavioural inheritance, and a set of abstraction mechanisms to define structures, including a distinction between the attribution of properties and composition mechanisms, of which the following are available: association, ordering and aggregation. With respect to R.9, both ROCK and ROLL model behaviour using the standard object-oriented approach of associating specific behaviour to types. This defines the protocol through which instances of the type are queried and manipulated. Both ROCK and ROLL also support a distinction between the interface and the implementation of an operation, as well as the notions of encapsulation, overloading, overriding and late binding.

Summary

Table 6.3 summarizes the results of the above survey. The notational conventions are the following: $+$ denotes full satisfaction of a requirement, $\pm$ denotes that the requirement is largely satisfied, $\mp$ denotes that the requirement is satisfied to a small extent, $-$ denotes failure to satisfy a requirement. When there is not enough information to ascertain the satisfaction or not of a requirement this is indicated with '?'.

Note that some proposals that appear in Table 6.2 do not appear in Table 6.3 because they are less related to the work described in this dissertation, although they are related to its realization in ROCK & ROLL. The proposals that are missing in Table 6.3 are discussed in the next subsection.

6.1.3 Other Less-Related Proposals

Some proposals that fall somewhat outside the scope of the research reported in this dissertation are briefly mentioned below, for the sake of extending the coverage of the survey given in the previous subsection.

DTL (Bal and Balsters, 1993) is proposed as a query language for databases specified in the TM specification language (Bal et al., 1992), whose foundations lie in a typed λ-calculus, as opposed to classical FOL. DTL is surface syntax for FDTL. The relationship between DTL and FDTL is, therefore, much the same as that between ROLL and the language of DOOTs and DODTs

	Requirements from Table 6.1								
Proposals from Table 6.2	R.1	R.2	R.3	R.4	*R.5*	*R.6*	*R.7*	*R.8*	*R.9*
IQL+	$\pm$	$-$	$+$	$+$	$+$	$+$	$+$	$\pm$	$\pm$
Datalog^{meth,flex}	$\pm$	$\pm$	$+$	$+$	$+$	$+$	$-$	$-$	$+$
loopl	$-$	$-$	$+$	$+$	$+$	$\pm$	$-$	$\mp$	$\pm$
Obj-U-Datalog	$-$	$-$	$+$	$+$	$+$	$\pm$	$-$	$-$	$-$
Logres	$\pm$	$\pm$	$+$	$+$	$+$	$+$	$+$	$\pm$	$-$
OOLP+	$-$	$-$	$+$	$+$	$+$	$\pm$	$-$	$\mp$	$\mp$
LIVING IN A LATTICE	$\pm$	$-$	$+$	$+$	$+$	$+$	$+$	$\pm$	$-$
F-logic	$-$	?	$+$	$+$	$+$	$+$	$\pm$	$\mp$	$\pm$
Gulog	$-$	$\pm$	$+$	$+$	$+$	$+$	$\pm$	$\mp$	$\pm$
LLO	$+$	$\pm$	$+$	$+$	$+$	$+$	$\pm$	$\mp$	$\pm$
Noodle	?	$\pm$	$+$	$+$	$+$	$+$	$+$	$+$	$\pm$
Orlog	$\pm$	$\pm$	$+$	$+$	$+$	$+$	$+$	$\pm$	$\pm$
OIL	$\pm$	$\pm$	$+$	$+$	$+$	$+$	$\pm$	$-$	$\pm$
ConceptBase	$\pm$	$+$	$+$	$+$	$+$	$+$	$-$	$\mp$	$-$
ROCK & ROLL	$+$	$\pm$	$+$	$+$	$+$	$+$	$+$	$+$	$+$

Table 6.3: DOOD Proposals Against Requirements

defined in Chapter 3. The DTL/TM proposal is quite distinct from the proposals reviewed in the previous subsection. DTL is more concerned with type-theoretic (and model-theoretic) issues than proof-theoretic ones. Notice also that TM is only a specification language, i.e., there is no formal description of the database instances induced by a TM specification. In particular, there is no description of a database instance as a proof-theoretic construct. This is made more evident by the fact that the only formal semantics published for DTL is denotational, thereby offering very few indications about how to optimize and evaluate DTL queries with reference to typical database engines and scenarios. It is also indicative of the lack of concern for proof-theoretic issues (and automated theorem proving ones, in particular) that having sets as first-class values is seen as an important contribution of the proposal. Although this feature is scarcely discussed in (Bal and Balsters, 1993), one should note that the set unification problem is known to be NP-complete, which curtails any simple attempt to cater for set-valued terms within unification-based approaches to query evaluation.

LAURE (Caseau, 1991; Caseau, 1990) is the result of several years of development effort. It is fully implemented and has been applied to complex problems (Caseau et al., 1993). In contrast with all other proposals surveyed in this section, the LAURE system started its life as an object-oriented programming language, called **LORE** (Caseau, 1988). A logic language, called L_3 (Caseau, 1991), was later added to the LAURE system. As a result of this development history, many features of LAURE place it somewhat outside the scope of this survey. For example, the relationship between LAURE and L_3 is such that, strictly speaking, it is difficult to relate their definition to the standard proof theory of DDTs. Moreover, the metatheory of LAURE/L_3 has quite a different flavour from that of the other proposals considered in the section, in spite of the fact that both backward and forward-chaining evaluation procedures, as well as a translation into relational algebra, have been defined for L_3. As a result, it is more difficult to think of a LAURE database as a theory than of any other proposals surveyed in this section. This is largely due to the fact that the data model underlying LAURE is binary, i.e., all relationships between objects and classes must be modelled by binary relationships (which may or may not have a functional constraint imposed on them). This means that L_3 can be thought of as an extension of binary Datalog with object identity and function symbols (or as a restricted form of O-Logic without negation and with more constraints on object identifier invention). In other words, LAURE/L_3 map into a kind of DRDB that is clearly outside the mainstream, thus accounting, at least in part, for the nonstandard feel of the metatheory of LAURE/L_3. Finally, note that LAURE only models structural knowledge, and even then offers no support for lists or nonbinary tuples.

Peplom^d (Dechamboux and Roncancio, 1994b; Dechamboux and Roncancio, 1994a) is not a proposal to characterize DOODs from a proof-theoretic point of view. It is a proposal for an object-oriented database programming language which has been extended with deductive capabilities. It was developed independently of, but has ended up remarkably similar to, ROCK & ROLL, insofar as Peplom^d shares with ROCK & ROLL most of the strategic and tactical decisions that helped shape the latter.

On the strategic level, Peplom^d also adopts a two-language architecture which tightly integrates an imperative language (principally, for explicit control and state transition) and a deductive language (principally, for intensional definitions and data retrieval). Like ROCK & ROLL, Peplom^d achieves this without major impedance mismatches. However, Peplom^d differs from ROCK & ROLL in not having an independently defined and independently implemented model of data, i.e., there is no separate object store in Peplom^d that corresponds to OM in ROCK & ROLL. This may be a consequence of the fact that the imperative language was the first component of Peplom^d to be fully developed, and the only one to be fully implemented to date. As a result, the data model of a Peplom^d database is indistinguishable from the type system of a Peplom^d persistent application. This is, of course, an effective approach, but there is more flexibility in the ROCK & ROLL strategy, insofar as an independent definition and implementation of the data model as an ADT makes the system as a whole more resilient to extensions, such as those described in the first section of Chapter 5.

On the tactical level, Peplom^d also provides for a mode of integration between the imperative and the deductive language components in which the result of any query can be assigned a name in the state of the imperative program and, thereafter, manipulated like a collection value. Methods can be defined declaratively or imperatively. Unlike ROLL, the declarative language of Peplom^d has not yet been implemented. Thus, it is impossible to ascertain how the two languages will perform in tight integration. This is especially so because, again unlike ROCK and ROLL, neither the imperative nor the deductive language component of Peplom^d have been formalized, and hence there is no basis for the development of a metatheory. It can be said, however, that the Peplom^d deductive language is much harder to implement than ROLL. In particular, it is not Datalog equivalent, insofar as it allows function symbols, and hence infinite Herbrand domains which means that query evaluation cannot be guaranteed to terminate even in the absence of predicates that effect state transition.

In summary, the independent proposal of two DOOD systems with remarkably similar architectures and with largely equivalent functionalities can be taken as evidence of a convergence among researchers of the practical issues involved in developing DOOD systems, a theme to which attention has often been called [see, e.g., (Abiteboul, 1992)]. It should be noted that, at present, Peplom^d is a representative of the hands-on approach that has characterized OODB research since the early 1980s. There is no formal characterization of what a Peplom^d DOOD is, whereas all the components of the ROCK & ROLL implementation are formally described either in this dissertation or in (Goñi, 1995; Dinn et al., 1995; Paton and Abdelmoty, 1993).

Coral++ (Srivastava et al., 1993) builds on the DRDB technology developed for Coral (Ramakrishnan et al., 1993; Ramakrishnan et al., 1994). It is not, in fact, a DOOD proposal, but rather an attempt to embed a computationally complete, imperative language (in this case, C++) into a purely deductive language, thereby reversing the commonest direction of the embedding, which is to embed the deductive language in an imperative one. This is no improvement in terms of preventing the impedance mismatches caused by embeddings in the opposite direction. The reason is, of course, that Coral and C++ have wholly distinct data models. In ROCK & ROLL (as in Glue/Nail and in Peplom^d) a shared data model enforces a single type discipline and a single evaluation strategy, thereby ensuring that impedance mismatches do not arise, as they do in Coral++.

Coral++ can, therefore, be characterized as a proposal to provide an interface from a deductive language into an object-oriented imperative one as cleanly as possible. Although it offers a pragmatic solution to the problem of making some of the functionality associated with the object-oriented paradigm available within DDBs, Coral++ does not qualify as a characterization proposal for DOODs.

6.2 A Survey and Comparison of Proposals to Integrate Deductive and Active Rules

This section compares some of the more recent work on integrating ADBs and DDBs with the approach described in Chapter 5. This latter approach derives from the perception that, in all but the most trivial cases, ARs and DRs are more different in semantic terms than their syntactic similarities would suggest. To substantiate this, consider the following major contrasts between DRs and ARs, and the impact these have on the operational semantics of the two kinds of rule sets.

A DR-set is an axiomatic logical theory (Gallaire et al., 1984; Reiter, 1984). In particular, syntactically speaking, a DR inherits the (inverted) form of a kind of first-order logic formula known as *material implication*. This means that, evaluating a DR yields a value of type *Boolean*, since an expression of the form $H \leftarrow B$ is semantically equivalent to $H \vee \neg B$.[2] The crucial notion upon which the operational semantics of DDBs is constructed is that of classic logical consequence, and this presupposes a single fixed state for the evaluation of expressions.

The above properties of DRs stand in contrast to comparable properties of ARs as follows. An AR-set cannot be taken to be a logical theory of the classic kind. The form of an AR is that of an imperative command that is meant, in principle, to effect a state transition, as the operational semantics given in Figures 5.3-5.5 shows. This is the first major contrast with the deductive case: an AR expresses a state transition, i.e., a relation *between* consecutive states, whereas a DR expresses a relation that holds *within* any state.

Each AR $E : C \rightsquigarrow A$, considered in isolation, is equivalent in some sense to the program fragment **if** $(E \wedge C)$ **then** **do** A **od** **fi** with the consequence that the type of E and C is **Boolean**, the type of A is **statement**, and so is the type of $E : C \rightsquigarrow A$. This is not to claim, however, that the operational semantics of an AR *set* $\{\rho_1, \ldots, \rho_n\}$ is not significantly different from the one that would be obtained from simply wrapping all ARs in the set to yield a block such as **begin** $\rho_1; \ldots; \rho_n$ **end**. In fact, even assuming a fixed predefined order, some aspects of the flow of control in the execution of AR-sets are implicit and may not be gauged entirely from the textual representation alone, as they depend on the sequence of event occurrences and this is not determinate. Further, the explicit part of the control stretches beyond the normal flow of execution of simple condition statements (e.g., when it interacts with the transaction model). This is the second major contrast with the deductive case: both DRs and ARs assume a general operational semantics but only the latter assumes, in addition, an implicit (and indeterminate) flow of control. The third major contrast is that only the operational semantics of AR-sets has to encompass control parameters that govern the interaction of the AR evaluation process with other system components, e.g., the transaction manager.

The contrasts above seem to justify reconsidering the view that the operational semantics of DR

[2] More precisely, the evaluation of a DR yields bindings for the variables in it that characterize which instances of the rule are true in the current state.

and AR-sets intuitively lie in a single, simple, continuous spectrum, and can therefore be unified by a simple extension, or restriction as the case may be, of one point in the spectrum into the other.

Two recent proposals typify this view at opposing ends. In (Widom, 1994) a very general operational semantics of ADBs is presented informally and at a very high level of abstraction. It admits a simple restriction that, ultimately, should allow DRs to be processed by the same engine that processes ARs. In (Zaniolo, 1994) a new operational semantics for DDBs, based on *X-Y stratification*, is formalized (see also (Zaniolo et al., 1993)). It admits a simple extension that, ultimately, should allow ARs to be processed by the same engine that processes DRs. The next paragraphs discuss and compare each of these proposals with the one presented in this dissertation in terms of the contrasts highlighted in the beginning of this section.

The central argument in (Widom, 1994) is that there is a basic computational engine for rule processing that, by being subject to slight modifications at each point in a spectrum, leads from purely deductive Datalog rules to fully fledged active rules. However, the very starting point of the spectrum takes Datalog to be no more than a view definition language, clearly ignoring the fact that DRs can uniformly express other forms of invariant relations between elements of the application domain (e.g., business rules, data model invariants, integrity constraints). Unless one is prepared to accept a more restricted perspective of DDB technology than that already provided by existing implementations, the initial assumption in (Widom, 1994) is somewhat unsatisfactory. Also, it is arguable that some of the required modifications from one point of the spectrum to the next are not as slight as one would wish. Being a position paper, (Widom, 1994) does not provide a detailed formal semantics, and this makes it impossible to gauge to what extent the three major contrasts between DRs and ARs highlighted in the beginning of this section are reconciled by its approach.

The thrust of (Zaniolo, 1994) is that there is a basic computational engine that can process pure DRs and yet simulate the effect of active behaviour. The contrast between single and multiple states is addressed in a way that is similar to the event calculus in that it defines multiple states by distinguishing them in logical, and not physical, terms and differs from the event calculus in that it incorporates the information that is necessary to distinguish logical states by rewriting the existing rules and introducing names for the strata that model logical states, whereas the event calculus achieves a similar effect by simply adjoining additional rules (those given in Chapter 5) to constrain the logical consequences of the existing ones. The problem of control is not addressed in (Zaniolo, 1994). Since the argument stresses that the new semantics is fixpoint-based, and hence essentially declarative, it seems that incorporating explicit control has the potential to disrupt the foundations of that approach. Overall, (Zaniolo, 1994) seems to take a restrictive view of active behaviour in the light of current ADB technology. Its undeniable elegance is achieved by avoiding issues such as the means for specifying complex composite events and for interleaving AR execution with the transaction model orthogonally supported by the database system.

The approach in this dissertation differs from that of both (Widom, 1994) and (Zaniolo, 1994) in that it does not aim at a single computational engine that can process both DRs and ARs. The two kinds of rules, as argued above, are sufficiently different in semantic terms that attempts to unify, as opposed to integrate, them are probably bound to produce a distorted picture of either or both. In contrast with (Widom, 1994), in this dissertation the full power of DDB technology is preserved, and in contrast with (Zaniolo, 1994), there is no barrier to reach the level of current ADB technology in complex event specification and transaction dependent control. While both (Widom, 1994) and (Zaniolo, 1994) see the task as one of *unification*, this dissertation argues for *integration* on the grounds that semantic distinctions block the way to full scale unification. Nevertheless, a logical basis for the integration need not be renounced. In fact, Chapter 5 has shown that using a logical conception of a history it is possible to construct a simple but very expressive characterization of ECA rules.

The discussion so far has concentrated on the two proposals that represent extreme positions emerging from the two paradigms of concern. Other work addressing the integration of active and deductive behaviour includes (Bayer and Jonker, 1994; Ceri et al., 1994; Harrison and Dietrich, 1994), which are now briefly compared with the approach presented in this dissertation.

As in (Zaniolo, 1994), the proposal put forward in (Harrison and Dietrich, 1994) extends the operational semantics of DDBs to capture some form of active behaviour. However, in this case, the only events that can be specified are structural operations over data, viz., updates characterized by difference relations, and, as in (Zaniolo, 1994), there is no indication that the formalism is capable of incorporating interactions with a transaction manager.

The position expressed in (Bayer and Jonker, 1994) is that of wrapping pure DDBs with a fully fledged model of active behaviour. As a result, it does not provide any logical framework for integration, i.e., it is loosely rather than tightly coupled. With similar objectives, Chimera (Ceri et al., 1994) is built on top of Algres (Ceri et al., 1990a). The proposal has similarities to the one described in this dissertation, and it builds on the strengths of Algres in terms of expressing dynamic behaviour. Its capabilities for specifying events are limited to structural operations only and, to the best of our knowledge, a unified logic-based account of active behaviour has not been made public. In this case, as in that of (Bayer and Jonker, 1994) the integration is loose when contrasted with that achieved in Chapter 5.

Summary of Chapter 6

This chapter had two sections.

Section 6.1 used a set of requirements to survey how each individual proposal can be deemed to realize the goal of comprehensively addressing the shortcomings of current generation commercial DBMSs while exhibiting the qualities associated with next generation ones.

Section 6.2 surveyed work related to the contribution, in Chapter 5, to the problem of integrating deductive and active rules in database systems

The DOODs characterized in this dissertation satisfy all the requirements that constitute the core motivation for DOOD research. Other research proposals for the characterization of DOODs surveyed in this chapter have been shown to come short of such a comprehensive solution. Chapter 7 closes this dissertation by summarizing the contributions reported in it and some future research directions that it suggests.

Chapter 7

Contributions, Conclusions and Future Directions

The purpose of this chapter is to recapitulate the technical development of the research reported in this dissertation and, in that light, to draw some conclusions with respect to the contributions of the research and future directions that are suggested by it.

In particular, it highlights, through the identification and comparison of three alternatives to the axiomatic approach to the characterization of DOODs, some desirable properties that approaches might be expected to have. The approaches identified in the chapter are then briefly analyzed as to the extent to which they have the expected properties. This analysis is the basis for the conclusions with which this chapter is mostly concerned.

This chapter has four sections.

Section 7.1 recapitulates the technical development of the axiomatic approach by means of a brief overview.

Section 7.2 reviews and compares three alternatives to the axiomatic approach. The section contrasts the theory space that each approach constructs.

Section 7.3 recalls the central question addressed in this dissertation, describes the main features of the positive solution to that question that the dissertation contributes, describes the main results obtained, and highlights how the DOODs characterized in this dissertation meet the requirements whose satisfaction has been the main motivation of DOOD research.

Section 7.4 concludes the dissertation with some remarks about the consequences of its results and about future directions suggested by them.

7.1 An Overview of the Technical Development of the Axiomatic Approach

Figure 7.1[1] depicts the technical development of the axiomatic approach to the characterization of DOODs, as follows:

mathematical characterization — This is denoted by the μ mappings in Figure 7.1. Using FOCTs (equivalently, FOL), a mathematical characterization of a model of structural and behavioural data capturing object-oriented concepts was described, in Chapter 1, in two stages:

[1] The representation of the space adopted in Figure 7.1 is different from, but equivalent to, that used in Chapters 1 to 3 in a way that is explained later in this chapter.

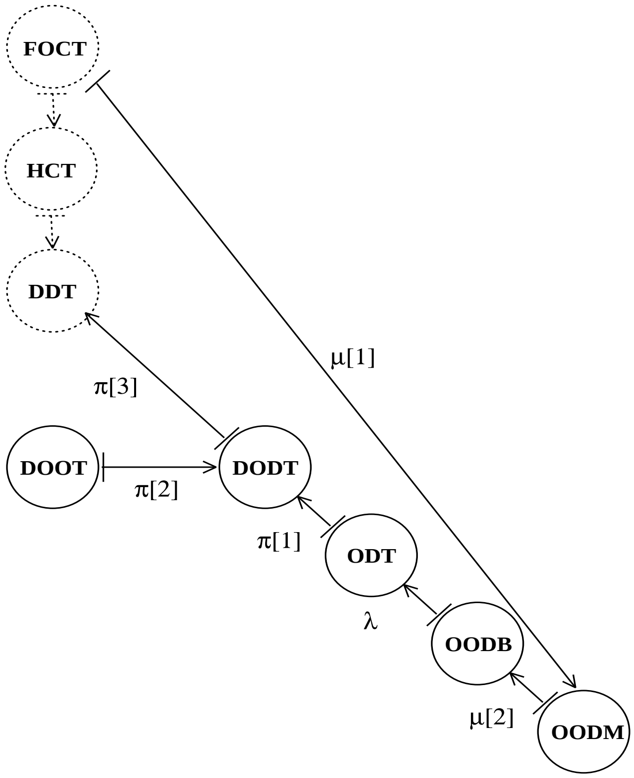


Figure 7.1: The Space of Theories Constructed by the Axiomatic Approach

[1] — An OODM was characterized as a Bourbakian species of mathematical structures assuming standard set theory.

[2] — The OODBs that an OODM induces were characterized as mathematical structures of the OODM species.

logical reconstruction — This is denoted by the λ mapping in Figure 7.1. As characterized above, the OODM and the OODBs induced by it are mathematical systems, i.e., fully rigorous but not fully formal (in the strict sense that they do not formalize the notion of proof within the system itself). Therefore, Chapter 2 reconstructed the class of *mathematical theories* denoted by the OODB node in Figure 7.1 as the class of (first-order) *logical (database) theories* denoted by the ODT node.

proof-theoretical derivation — This is denoted by the π mappings in Figure 7.1. Because ODTs are cast as specializations of a FOL calculus, they are, in that presentation, not amenable to the kind of automated theorem proving approach that underpins DDBs via the theory of logic programming. Chapter 3 derived, from ODTs, theories that possess mechanizable proof procedures, in three stages:

[1] — DODTs are derivable from ODTs by the application of algorithms that transform ODT sentences into DODT clauses.

[2] — In contrast with DODTs, DOODs were shown to incorporate a proof-theoretic account of encapsulation, overloading, overriding and late binding.

[3] — Finally, DOOTs were shown to be reducible to DDTs, thereby guaranteeing that most methods and techniques used to implement DDBs can be adapted and applied in the implementation of DOODs.

Note that Figure 7.1 differs from Figure 0.2 in that the maps $DODT \mapsto DOOT \mapsto DDT$ in the latter are replaced by the maps $DOOT \mapsto DODT \mapsto DDT$ in the former. This alternative topology is equally well defined, since it was shown in Chapter 3 that any DOOT literal can be mapped into a DODT literal. Because it is better suited for the comparison purposes of this chapter, it is assumed in the sequel without further remark.

As described in Chapter 4, the axiomatic approach facilitates implementation. As described in Chapter 5, the axiomatic approach is extensible. As to self-sufficiency, even though much can be achieved by extensions such as described in Chapter 5, the axiomatic approach does not attempt to cater for full self-sufficiency. There seem to be hard limits to what can be achieved without disrupting the metalogical properties that are thought to be desirable in database contexts. Hence, there is a strong argument in favor of reducing the characteristic impedance of multilanguage architectures rather than preempting the issue with a single-language architecture based upon a fully self-sufficient language. The axiomatic approach does indeed facilitate the impedance-free integration of two complementary language components that, taken together, provide a fully self-sufficient platform for the development of applications. As described in Chapter 4, ROCK & ROLL is a realization of this strategy.

7.2 A Comparison of Approaches to the Characterization of DOODs

This section focuses on how different approaches give rise to theory spaces with different topologies from the one contributed in this dissertation and depicted in Figure 7.1.

To contrast approaches to DOOD characterization, a comparison framework is adopted that can be motivated as follows. The proof-theoretic view of databases construes database states as sets of axioms. It is this viewpoint that underlies the extension of a conventional database with a notion of deduction, thereby opening the way for a unified, automated theorem proving approach to query answering, integrity maintenance and implicit data derivation.

Recall that, in the process of describing the context for the contributions reported here, the Introduction to this dissertation argued that proposals which seem radically different from one another can be compared by fixing a logical system of reference and then describing how each component part of each proposal (viz., language, axiom system and inference system) compares with the corresponding component part of the logical system of reference. This brings to light the loci and, to a certain extent, the levels of intervention that would be necessary on the logical system of reference to make it perform as the logical system that the proposal characterizes. More importantly, it makes it more explicit how the metalogical properties of the DOODs characterized by each proposal compare with those of the logical system of reference. In this way, some evidence is produced as to what properties the DOODs characterized by the proposal are likely to exhibit.

The obvious choice of a logical system of reference is Datalog (Ceri et al., 1990b), for the following main reasons:

- It is the system underlying DRDBs.
- Its metalogic is well defined, well developed and well known.
- It has very desirable properties with respect to the termination and efficiency of the available algorithms to compute models and proofs.
- Methods and techniques to implement these algorithms in areas such as query optimization, query evaluation, implicit data derivation and integrity constraint maintenance have been studied and tested at the level of research prototypes at least.

Consider now the question of identifying desirable properties of approaches. Although not exhaustive, the following properties of approaches to the characterization of DOODs seem of interest: *practicality*, *extensibility* and *impedance level*.

The task of investigating these properties formally was not in the scope of the research reported here. They are invoked only insofar as they throw light on the contributions of this dissertation. An informal definition of the properties is given in Table 7.1.

P.1: **practicality**	An approach will be said to be more, or less, **practical** depending on whether it simplifies, or not, the task of engineering a fully functional DOOD system.
P.2: **extensibility**	An approach will be said to be more, or less, **extensible** depending on whether it simplifies or not, the task of extending the semantic content of a system or its functionality or both, without disrupting its metalogical properties.
P.3: **impedance level**	A characterization proposal could be said to be more, or less, **self-sufficient** depending on whether it unifies more, or less, language subcomponents of databases among the standard ones, viz., definition, querying, constraining, view definition, and state transition. If a characterization is not self-sufficient, then other components must complement its functionality for practical use. When this need arises, an approach will be said to have a low, or high, **impedance level** depending on whether it provides the means with which to impose on the language components a common type and evaluation discipline.

Table 7.1: Desiderata of Approaches to the Characterization of DOODs

In anticipation of a more detailed discussion later in the chapter, some illustrative examples are now drawn to give a flavour of the contrasts that the properties in Table 7.1 allow one to draw.

A *more practical* approach reduces the effort required to engineer a practical DOOD system without renouncing any major feature of the paradigms undergoing integration. The axiomatic approach is practical, as demonstrated in the engineering of ROCK & ROLL described in Chapter 4.

A *less practical approach* either increases the engineering effort or is only capable of reducing that effort through the renunciation of features. The restart approach[2] is less practical than the axiomatic approach. For example, the metalogic of F-logic (Kifer et al., 1993) bears little resemblance with that of DRDBs, with the result that the effort required to engineer an F-logic-based DOOD system is larger than the effort spent in engineering the ROCK & ROLL DOOD system, due to the total absence of existing methods and techniques that could otherwise be chosen for reuse.

A *more extensible* approach allows extensions to the modelling power of a DOOD system to be engineered more easily. The axiomatic approach is extensible, as demonstrated in Chapter 5 by the incorporation of finer-grained specifications of inheritance relationships and of participation and cardinality constraints on relationships. This is largely due to the fact the most of the modelling semantics is captured by axioms, and hence capturing more semantic content does not require

[2]This and other approaches are discussed in detail later in this chapter.

changing the metalogic of DOODs.

A *less extensible* approach is less amenable to the incorporation of new modelling constructs. The restart approach is less extensible than the axiomatic approach. For example, in order for a proposal such as F-logic to incorporate participation and cardinality constraints on relationships, its model and proof theories, as well as any concrete system that realizes them, would have to be changed.

If a given proposal is not self-sufficient, then achieving it will require complementation by means of additional components. In this case, the particular approach that the proposal adopts plays a part in determining the severity of the impedance mismatches that the complementation task incurs.

An approach has *lower impedance-level* when it is capable of imposing a type and evaluation discipline across more components, and hence of reducing the occurrence and severity of impedance mismatches. The axiomatic approach has a very low impedance level insofar as it allows a common type and evaluation discipline to be imposed on all the components of the DOOD system, as exemplified by ROCK & ROLL.

An approach with *higher impedance-level* is less capable of attenuating the occurrence and severity of impedance mismatches. For example, the mapping approach has a higher impedance-level than the axiomatic approach because the lack of an independently defined model of structural and behavioural data means that type and evaluation disciplines are, in a sense, hardwired into each language component, and these may not match.

The notion of levels of intervention in component parts of a logical system of reference allows the following observations to be made, in broad terms, which link them with the desiderata of Table 7.1:

- The lower the level of intervention in the language and inference system, the greater the sharing of metalogical results. The greater the sharing of metalogical results that have given rise to effective and efficient methods and techniques, the more *practical* the approach.
- The lesser the extent in which the language and the inference system are relied upon to capture modelling semantics, the more *extensible* the approach.
- The higher the level of intervention in the language and inference system, the more self-sufficient a proposal characterized by the approach tends to be. If an approach intervenes in the logical system of reference to the extent that it leads to proposals that are not self-sufficient, then the question arises as to what level of impedance the necessary embedding incurs. The greater the extent in which modelling semantics is independent of its realization on any single language component, the lower the *impedance level* of the approach.

Besides the axiomatic approach contributed in this dissertation, three other approaches to the characterization of DOODs can be distinguished in the literature. One approach is to *restart* the metalogical development of DOODs from scratch and is the one adopted by F-logic (Kifer et al., 1993) and several languages that are inspired by it. The restart approach is essentially characterized by a rejection of the family of relational languages derived from FOL as a source in the metalogical development of DOODs. Another approach is exemplified by proposals such as Datalog^{meth,flex} (Abiteboul et al., 1993), **loopl** (Bertino and Montesi, 1992) , OOLP+ (Dalal and Gangopadhyay, 1990), Orlog (Jamil and Lakshmanan, 1993) and OIL (Zaniolo, 1990), among others. These proposals attempt to capture object-oriented concepts in a source language which can be *mapped* to an existing relational language acting as a target. For example, the targets for Datalog^{meth,flex}, **loopl** and $OOLP+$ are, respectively, Datalog with negation, $\mathcal{LDL}$ and Prolog. In this approach, the metalogic of DOODs is developed by reduction to that of the target language. Proposals such as IQL+ (Abiteboul, 1990), Obj-U-Datalog (Bertino et al., 1994a), Logres (Cacace et al., 1993) and LIVING IN A LATTICE (Heuer and Sander, 1992), among others, postulate Datalog as a foundation and proceed to *extend* it with features that try to capture object-oriented concepts with logical machinery. In this extension approach, the metalogic of DOODs is developed by extending that of Datalog. The following three subsections comment more extensively on the three approaches identified above.

7.2.1 The Restart Approach

This approach to the DOOD characterization problem assumes that the paradigmatic properties of object-orientation are different enough from those which underlie DRDBs to justify *restarting* the characterization of DOODs from scratch. This is taken to mean that only FOL is assumed and the DDB technology embodied in DRDBs (as expressed in Datalog and the various languages derivable from it) is neither a source nor a target in the characterization task. The proposal that most vividly typifies this approach is F-logic (Kifer et al., 1993).

If one attempts to explain proposals that fall under the restart approach in terms of the interventions that would be required in Datalog for any extension of the latter to perform as nearly as possible to the former, one realizes that the language and inference system of a restart proposal have to be subject to a high level of intervention. Typically, this is most manifest in (as yet) insufficiently studied metalogics, of which, nevertheless, it is possible to say that they are much more complex to understand and implement efficiently than that of DRDTs. In particular, methods and techniques for optimization and evaluation are underdeveloped in comparison with those available for DRDBs.

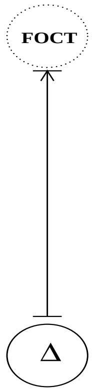


Figure 7.2: The Space of Theories Constructed by the Restart Approach

Figure 7.2 depicts the theory space constructed by the restart approach. Here, and in the rest of the chapter, Δ is the node that denotes the characterization of DOODs obtained in the approach. Its most striking feature is its sparseness. It has no OODM and OODB nodes, indicating that the restart approach does not acknowledge the need for an independent definition of OODBs that could be studied in isolation from proof-theoretical issues. More importantly, the absence of OODM and OODB nodes means that proposals in the restart approach have no answer to the question of which OODBs the DOODs it characterizes are supposed to extend and how these extensions can be described. Also, no clear understanding exists of how DOODs characterized by the restart approach compare and relate to DRDBs. Hence, there is no defined path for the transition from one technology to the other.

The topology of the space shows a single edge from the DOOD characterization to FOCTs. One consequence of this topology is that metalogical results can only be shared with FOCTs, which are intractable logical systems,[3] and hence, barren ground for the reuse of existing methods and techniques for query optimization and evaluation in database contexts, which are stricter with respect to decidability requirements. In the case of F-logic, even that connection is not easily understood

[3] The classification of logical systems into intractable, acceptably efficient and tractable is that proposed in the Introduction to this dissertation.

due to the presence of an equality predicate with which, unlike DDTs, it cannot dispense.

In broad terms, because of the very high level of intervention in both the language and the inference system, the restart approach seems to lead to proposals that imply a greater development effort to attain the level of fully functional DOOD systems than approaches that allow the sharing of results with acceptably efficient or tractable logical systems. This detracts from its practicality. For the same reason, the restart approach is less extensible. In contrast, the approach may lead more easily to self-sufficient characterizations (e.g., research on transaction logic programming (Bonner and Kifer, 1992) addresses the problem of making F-logic self-sufficient). Notice, however, that no existing proposal using a restart approach is fully self-sufficient if it is mapped into FOCTs, as these lack a notion of dynamic state. The impedance-level is very high for all existing proposals as they currently stand.

7.2.2 The Mapping Approach

This approach to the DOOD characterization problem assumes that the paradigmatic properties of object-orientation are best captured in a surface language (possibly, but most often not, associated with a particular data model) which is then *mapped* to an existing relational (in the sense, of predicate-based) language which, in this indirect way, provides the surface language with a metalogic. Several DOOD proposals, such as Datalog^{meth,flex}, **loopl**, OOLP+, Orlog and OIL, adopt this approach.

Proposals that fall under the mapping approach can be explained in terms of the interventions that would be required in Datalog by defining the target of the map. Example targets are Datalog with negation (in the case of Datalog^{meth,flex}), $\mathcal{LDL}$ (in the case of **loopl**, Orlog and OIL), or Prolog (in the case of OOLP+). The extent of the intervention can then be gauged by the extent of intervention (which is assumed to be known) involved in extending Datalog to the target. Thus, for Datalog^{meth,flex} the level of intervention is minimal. For proposals mapped into $\mathcal{LDL}$ the level of intervention is high in both the language and the (implicit part of the) inference system. As for OOLP+, one must bear in mind that the metalogic of Prolog is inadequate for DDBs. The perception of this inadequacy was, after all, the driving force behind the conception of Datalog. Retreating to a position in which a DOOD is defined by its mapping into Prolog seems clearly unsatisfactory.

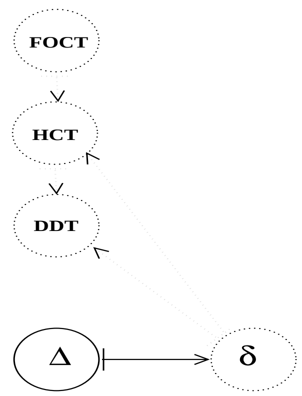


Figure 7.3: The Space of Theories Constructed by the Mapping Approach

Figure 7.3 depicts the theory space constructed by the mapping approach. Here, δ denotes the target of the map that characterizes this approach. Depending on the particular δ it is possible to have a DDT or an HCT sink. The former is the case with $\Delta =$Datalog^{meth,flex} and $\delta =$ Datalog with negation; the latter is the case with $\Delta \in \{$**loopl**, Orlog, OIL$\}$ and $\delta = \mathcal{LDL}$. Although less sparse than the restart space of Figure 7.2, the map space of Figure 7.3 also lacks OODM and OODB nodes, thereby indicating that proposals in the mapping approach also have no answer to the question of which OODBs the DOODs it characterizes are supposed to extend and how these extensions can be described. Orlog is a possible exception insofar as it purports to have an associated data model, but there is doubt as to the extent to which support for the structural and, especially, behavioural features of the object-oriented paradigm is adequate. In any case it is worth noting that because a relational language is the target, there can be, in comparison with the restart approach, a clearer understanding of how DOODs characterized by the mapping approach compare and relate to DRDBs. Hence, either there is a defined path for the transition from one technology to the other or one can easily be foreseen.

The topology of the space shows an edge into either an acceptably efficient or a tractable logical system. One consequence of this topology is that metalogical results can be shared that are of interest for query optimization and evaluation in database contexts. Datalog^{meth,flex} can take advantage of all the methods and techniques for optimization and evaluation developed for DRDBs. **loopl**, Orlog and OIL are restricted to that subset that is applicable to $\mathcal{LDL}$. In the best case, therefore, the sharing is maximal. In broad terms, the mapping approach seems to imply a small development effort to attain the level of fully functional DOOD systems. This makes it a practical approach. Extendibility is hard since it requires redefining an already complex mapping. Self-sufficiency is not easily achievable and may imply changing the sink of the space from a tractable to an acceptably efficient, or worse, logical system. If self-sufficiency is not achieved the impedance-level will tend to be high.

7.2.3 The Extension Approach

This approach to the DOOD characterization problem assumes that the paradigmatic properties of object-orientation are best captured by *extending* a paradigmatic relational language, the metalogic of which is successively developed into a metalogic for DOODs. IQL+ (Abiteboul, 1990), Obj-U-Datalog (Bertino et al., 1994a), Logres (Cacace et al., 1993) and LIVING IN A LATTICE (Heuer and Sander, 1992), among others, take Datalog as a metalogical source and add features to it that try to capture object-oriented concepts with logical machinery.

There is a large degree of commonality among extension proposals as to what interventions need to be carried out. Typically, the language undergoes the most extensive intervention. Function symbols are introduced, and hence the finite interpretation domains of DRDBs are lost. The inference system may also suffer a low level of intervention, typically in its implicit part (e.g., the axioms that determine the properties of complex types such as sets, lists and records are typically hidden in one or more assumptions about the well-formedness of interpretation domains).

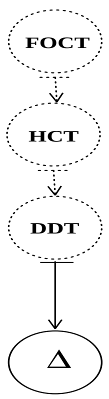


Figure 7.4: The Space of Theories Constructed by the Extension Approach

Figure 7.4 depicts the theory space constructed by the extension approach. Like the restart space of Figure 7.2 and the map space of Figure 7.3, it also lacks OODM and OODB nodes. Note that some of the proposals above do define their specific model of data. However, rather than being reconstructed as a proof-theoretic concept, such a model of data remains the definition of the class of interpretation structures used by the DOODs characterized. Furthermore, much of the semantics implied by the model is captured in the implicit part of the inference system, rather than explicitly as done in this dissertation. There is, in the case of the extension approach, a clear understanding of how DOODs characterized by the mapping approach compare and relate to DRDBs, since that is exactly the development path that characterizes the approach. Hence, there is a well defined path (although not necessary an effortless one) for the transition from one technology to the other.

The topology of the space shows an edge from DDTs, a tractable logical system. One consequence of this topology is that as Δ, at each successive extension step, moves further and further away from its DDT source, less and less metalogical results can be shared. In broad terms, the extension approach seems to imply a large development effort to attain the level of fully functional DOOD systems. This makes it a less practical approach. Extendibility is hard but the difficulty is usually reduced by making each incremental step small and well defined. Self-sufficiency is typically achievable but has to be traded off against the loss of desirable metalogical properties. If self-sufficiency is not achieved, there is little scope to alleviate the severity of impedance mismatches.

Summary

Table 7.2 summarizes the characteristic properties of the approaches discussed. The presence of these properties in the axiomatic approach is discussed in the next section.

Approach	*Practicality*	*Extendibility*	*Impedance-Level*
restart	very small	very small	very high
mapping	large	small	very high
extension	small	small	high

Table 7.2: Properties of Approaches to Characterization

7.3 A Summary of Contributions

The main contribution of this dissertation is an approach to the characterization of DOODs as axiomatically presented OODBs. The central research question addressed in the dissertation is whether this definition of a DOOD can be strengthened to the extent that the property of the metalogic of DOODs being predominantly shared with DRDBs can be said to hold. The positive answer to this question involved defining DOODs as a class of logical systems with the following properties:

language — A DOOD is a function-free theory of HCL. Hence, metalogical properties that are language-determined are totally shared between DOODs and DRDBs.

axiom system — The axiom system of DOODs differs from that of DRDBs only in the replacement of relational by object-oriented modelling assumptions. Hence, metalogical properties that are determined by the axiomatic content of the logical system are maximally shared between DOODs and DRDBs.

inference system — The inference system of DOODs differs from that of DRDBs only in taking into account encapsulation, overloading, overriding and late binding in the development of proofs. This is achieved without any significant overburdening of the computational complexity of proof procedures defined for DRDBs. Hence, DRDB proof procedures are largely shared with DOODs.

More specifically, the main research results comprising the technical development of the solution described above and presented in detail in Chapters 1 to 3, are the following:

- a formalization of an object-oriented data model as a Bourbakian species of mathematical structures,
- its reconstruction as an applied first-order predicate calculus,
- the conversion of this applied calculus into a class of logic programs without function symbols,
- a logical analysis of the dynamic aspects of object-orientation based on a simple and intuitive extension to the standard inference engine of DDBs, and
- a logic-based extension of DDBs with active behaviour.

Since the DOOD research enterprise is mainly motivated by the idea of preserving the qualities and remedying the shortcomings of current generation databases (see Table 0.1) through the integration of the deductive and the object-oriented paradigms for databases, it is crucial for a characterization proposal that the DOODs it defines preserve as many qualities, and remedy as many shortcomings, of current generation databases as possible. The DOODs characterized by the axiomatic approach do meet all the requirements (see Table 6.1) that follow from the observations, as follows:

- They have formal foundations, i.e., the OODM that underlies them has a mathematical characterization.
- They empower casual users, in the sense that they make it possible, through optimization and tractable evaluation, to guarantee that querying the database will not incur excessive performance penalties.
- They support the definition of data by deductive rules, i.e., provide means to express more complex kinds of implicit information that can be used to extend the information content of explicit states of application domains.
- They support the expression of integrity constraints by deductive rules, i.e., they provide means to express more complex kinds of general information (e.g., business rules) that can be used to constrain the set of possible states of application domains.
- They support the use of recursion to model structural and behavioural knowledge.
- They support object identity, i.e., they make it possible to distinguish the existence of an object from the state it happens to be in at any point in its existence.
- They support complex structures, i.e., they provide the means to model properties that have nonatomic values.
- They have a semantic orientation, i.e., they provide the means to model relationships between types, and, in particular, of inclusion and composition.
- They have a behaviour orientation, i.e., they provide the means to assign specific behaviour to particular types, and, in particular, of modelling the dynamic aspects of the interaction with elements of a type using the notions of encapsulation, overloading, overriding and late binding.

As summarized in Table 6.3, the survey of related work presented in Chapter 6 indicates that several related proposals fail, in varying degrees, one or more of the above requirements.

The axiomatic approach is unique in constructing a space of theories (see Figure 7.1) that is homeomorphic, in the object-oriented case, to the space of theories, in the relational case, that emerged from the intersection of research into the theory of databases and the theory of logic programming (cf. Figures 0.1 and 0.2). This means that the OODBs that the DOODs it characterizes are supposed to extend are formally defined. Furthermore, the relationship between DOODs and DRDBs is formally characterized, and hence there is a precise definition of the path that leads from one technology to the other.

Since the level of intervention in the language and inference system in the axiomatic approach is minimal, the sharing of metalogical results with DDTs is maximal. Hence, the axiomatic approach is very practical. Since it relies on axioms to capture most of the modelling semantics, rather than on the language and the inference system, the axiomatic approach is very extensible. Since the modelling semantics is explicit and independent of its realization on any single language component, the axiomatic approach can impose a typing and an evaluation discipline on language components and has a very low impedance level. As summarized in Table 7.1, the discussion of alternative approaches presented in the previous section indicates that they do not meet as many, or as extensively, these desiderata.

This dissertation has shown that the deductive and object-oriented paradigms for databases can be integrated in a clean and well founded way. The feasibility of this integration was far from obvious, as illustrated by the doubts expressed in (Ullman, 1991) as to whether a DOOD could be constructed at all. Ullman's conception of a DOOD is closer to that described for proposals classified under the extension approach. This lead him to identify problems with the proliferation of objects due to the fact that their existence might be ascertained by distinct proofs, with the decision as to what behaviour a type that is created in an ad-hoc manner is to have, and, finally, with the optimization of user-defined (and presumably unconstrained) methods. However, these problems only arise if one expects the deductive language to cater single-handedly for all the functionality that the DOOD system will ultimately exhibit. In other words, Ullman did not consider the possibility of a multilanguage DOOD system, presumably because he saw no immediate route for mismatch-free integration.

In the axiomatic approach, as implemented in ROCK & ROLL, the problems raised in (Ullman, 1991) simply do not arise (see (Barja et al., 1994) for a detailed discussion of how each of the objections raised by Ullman does not arise in ROCK & ROLL). This shows that a significant contribution of the axiomatic approach presented in this dissertation is to demonstrate that such mismatch-free integration is possible, and, as shown by ROCK & ROLL, practical.

7.4 Conclusions and Future Directions

The main conclusions to be drawn from this dissertation are:

- This dissertation addressed the question of whether a class of DOODs can be formally characterized whose metalogic is predominantly shared with the class of DRDBs. The positive solution to the question has resulted in the development of an approach to the characterization of DOODs which both captures the structural and behavioural aspects of the object-oriented paradigm with proof-theoretic means (axioms, most noticeably) and provokes no disruption of the tractability properties of the metalogic of reference, viz., that of DRDBs.

- The axiomatic approach demonstrates that there is no need to incur the penalties resulting from attempts to capture object-oriented concepts by interfering with the language or the inference system of DRDBs. In particular, many metalogical complications such as infinite interpretation domains and nonmonotonic phenomena related to the dynamic interpretation domains can be avoided.

- The logic-based extension of DOODs with active rules presented in the dissertation demonstrates that the axiomatic approach is extensible in a well founded manner.

- The design and implementation of ROCK & ROLL demonstrates that the approach is practical and that self-sufficiency can be achieved with a two-language architecture that incurs no impedance mismatch, insofar as an independently defined OODM imposes both a typing and an evaluation discipline on both the language components. Note also that routes to self-sufficiency adopted for DRDBs, such as those surveyed in (Ceri et al., 1990b), can also be adopted in the present context.

- A space of theories that connects traditional logic and traditional database research can be built for the object-oriented case that is homeomorphic to the relational one. This homeomorphism is a valuable tool in understanding what is involved in the paradigmatic choice both between RDBs and OODBs and between DRDBs and DOODs. Furthermore, it demonstrates that the extensibility of databases with deductive mechanisms is independent of their adopted model of data.

Some possible directions for further research are:

- The development of a theory of what constitutes good design for DOODs by analogy with relational dependency theory, which is made possible by the availability of a formally defined OODM.

- The development of a theory for schema evolution and for schema integration, relying on the fact that the semantics of modelling relationships is explicitly available as a set of axioms.

- The development of semantics-based query optimization methods that make use of the explicit modelling axioms.

- The study of useful domain independent modelling constructs that can be axiomatized and incorporated into the OODM, by analogy with the finer-grained specification of inheritance exemplified in this dissertation.

- The exploration of issues related to expressing constraints on state transitions, made possible by the visibility of past database states resulting from the embedding of DDBs in the event calculus.

- The exploration of alternative execution models for the extension of DDBs with active rules.

- The exploration of the applicability of the methodology used to integrate the deductive and object-oriented paradigms for databases to the integration of other paradigms, e.g., of the deductive and functional paradigms.

- The axiomatization of alternative proposals for formal OODMs and the derivation of the corresponding theory spaces.

This concludes this dissertation.[4]

[4] Attention is drawn to the appendices of this dissertation, viz., Appendix A contains the full set of axioms presented less rigorously in Section 1.1. Appendix B describes in detail the construction of a witness element in the species of mathematical structures $S(\Sigma^*)$ defined in Chapter 1. Appendix C contains the full formalization, as sentences of a first-order language, of the axioms in Appendix A. Appendix D contains the complete set of Horn clauses obtained by translating into clausal form the sort, completion and exception axioms of ODTs, as presented in Appendix C. Finally, Appendix E is a glossary of acronyms and symbols for the benefit of the reader.

Appendix A

The Mathematical Axiom: Informal Version

This appendix lists all the axioms used, in Section 1.2, to define the species of mathematical structures over object-oriented concepts contributed by Chapter 1. They correspond one-to-one with the more informal presentation given in Section 1.1.

Axiom A.1 **inclusion** is induced from **inclusion*** by disregarding the inclusion mode and inheritance hierarchy. In more detail, **inclusion** is a restriction of **inclusion*** to the latter's first and third arguments, which, in case the inclusion mode is `generalization`, swap positions in **inclusion**.

Axiom A.2 **inclusion** is reflexive over **classes** and transitive.

Axiom A.3 **attribution*** is subsumed by **attribution**. In more detail, **attribution** contains **attribution***, but not necessarily vice-versa.

Axiom A.4 **attribution** propagates through **inclusion**. In more detail, if x *has* a y and z *is_a* x, then z *has* a y.

Axiom A.5 **composition** is induced from **composition*** by disregarding the composition mode. In more detail, **composition** is a restriction of **composition*** to the latter's first and third arguments.

Axiom A.6 **composition** propagates through **inclusion**. In more detail, if x is *made_of* y and z *is_a* x, then z is *made_of* y.

Axiom A.7 **interface** propagates through **inclusion** by being annotated with the source class, thereby inducing **interface***. In more detail, if y *responds_to* z with argument classes $w_1, \ldots, w_n$ and x *is_a* y, then x *responds_as* y *to* z with argument classes $w_1, \ldots, w_n$.

Axiom A.8 **classification** propagates through **inclusion**. In more detail, if x is an *instance_of* y and y *is_a* z, then x is an *instance_of* z.

Axiom A.9 **index_ordering** is reflexive over **proper_indexes** and transitive.

Axiom A.10 All sorts are pairwise disjoint. In more detail, no two base sets among **C**, **Gc**, **O**, **Go**, **Op**, **Gop**, **M**, **Gm**, **I**, **IM**, **IH**, **CM** have an element in common.

Axiom A.11 There is a set **classes** (respectively, **objects**; **indexicals**) that is the union of **specific_classes** and **generic_classes** (respectively, **specific_objects** and **generic_objects**; **specific_classes**, **objects** and **proper_indexes**). In more detail, **classes** = **specific_classes** $\cup$ **generic_classes**, **objects** = **specific_objects** $\cup$ **generic_objects**, and **indexicals** = **specific_classes** $\cup$ **objects** $\cup$ **proper_indexes**.

Axiom A.12 There are no **inclusion_modes** other than `specialization`, `generalization` and `partition`. In more detail, if $x \in$ **inclusion_modes**, then either $x =$ `specialization` or $x =$ `generalization` or $x =$ `partition`.

Axiom A.13 There are no **composition_modes** other than `association`, `ordering` and `aggregation`. In more detail, if $x \in$ **composition_modes**, then either $x =$ `association` or $x =$ `ordering` or $x =$ `aggregation`.

Axiom A.14 **inclusion*** determines a functional relationship from the first, third and fourth arguments to the second. In more detail, if x *is_a** y on the inheritance hierarchy z by both inclusion mode u and inclusion mode u', then $u = u'$.

Axiom A.15 **inclusion** is antisymmetric.

Axiom A.16 **inclusion** relates elements in the same subset of its domain. More detailedly, if x *is_a* y and y belongs to **specific_classes** then so does x; similarly, if x *is_a* y and y belongs to **generic_classes** then so does x.

Axiom A.17 **inclusion** is weakly directed.

Axiom A.18 Every element of **specific_objects** is **classification** related to some element of **specific_classes** and every element of **generic_objects** is **typing** related to some element of **generic_classes**. In more detail, if $x \in$ **specific_objects**, then, for some $y \in$ **specific_classes**, x is an *instance_of* y, and if $x \in$ **generic_objects**, then, for some $y \in$ **generic_classes**, x is *of_type* y.

Axiom A.19 In `generalization` **inclusion*** on a given hierarchy, the extension of the generalized class is the iterated union of the extension of its subclasses. In more detail, if x is an *instance_of* y and y generalizes $z_1, \ldots, z_n$, then, for at least one $1 \leq i \leq n$, x is an *instance_of* z_i.

Axiom A.20 In `partition` **inclusion*** on a given hierarchy, the extension of the partitioned class is the iterated union of the extension of its subclasses and the iterated intersection of the latter is empty. In more detail, if x is an *instance_of* y and y is partitioned by $z_1, \ldots, z_n$, then, for exactly one $1 \leq i \leq n$, x is an *instance_of* z_i.

Axiom A.21 **attribution*** is not weakly directed through **inclusion**.

Axiom A.22 **composition*** determines a functional relationship from the first and third arguments to the second. In more detail, if x is *made_of* z by both composition mode u and composition mode v, then $u = v$.

Axiom A.23 `association` and `ordering`-**composition*** determine a functional relationship from the first and second arguments to the third. In more detail, if x is an `association` (or an `ordering`) of both u and v, then $u = v$.

Axiom A.24 **composition** is asymmetric.

Axiom A.25 Elements of **composition*** with distinct first arguments that are directed through the **inclusion** relation to a common sink must have pairwise equal second and third arguments. In more detail, if y and z are distinct, x is both a y and z, y is *made_of** y' by composition mode y'', and z is *made_of** z' by composition mode z'', then both $y' = z'$ and $y'' = z''$.

Axiom A.26 Proper **classification** related **description** references exist in all instances of a class for all **attribution** defined properties of the latter. In more detail, if a is an *instance_of* x and x *has* a y, then there is an *instance_of* y, say b, such that b is the y of a.

Axiom A.27 `association` **construction** references are indexed by an identity mapping and **classification** related. In more detail, if a is an *instance_of* x, x is an `association` of z and a has a *component* b with index b', then b is an *instance_of* z and $b = b'$.

Axiom A.28 `ordering` **construction** references are indexed by proper indexes and **classification** related. In more detail, if a is an *instance_of* x, x is an `ordering` of z and a has a *component* b with index b', then b is an *instance_of* z and b' is a proper index..

Axiom A.29 `aggregation` **construction** references exist and are class indexed and **classification** related. In more detail, if a is an *instance_of* x and x is an `aggregation` of z, then there is an *instance_of* z, say b, that is a *component* of a with index z.

Axiom A.30 An element in **specific_classes** responds to at least one **interface***. In more detail, if $x \in \mathbf{C}$ then there is a z defined on y with argument classes $w_1, \ldots, w_n$ such that x responds as y to z.

Axiom A.31 The class annotations in **interface*** must always be **inclusion** comparable. In more detail, for distinct y and y', if x responds as y to z and x responds as y' to z (in both cases with argument classes $w_1, \ldots, w_n$), then either y *is_a* y' or y' *is_a* y.

Axiom A.32 There is no element that is **interface** related and not **implementation** related, and **implementation** determines a functional relationship from its last argument to all others. In more detail, if y *responds_to* z with argument classes $w_1, \ldots, w_n$, then there exists exactly one r such that z with argument classes $w_1, \ldots, w_n$ is *implemented_in_by* by r.

Axiom A.33 The execution of an implementation relates objects that conform with the signature of the interface the implementation is associated with. In more detail, if z at w_1 with argument classes $w_2, \ldots, w_n$ is *implemented_in_by* r and, upon being executed for a_1, r *responds_with* $a_2, \ldots, a_n$, then, for $1 \leq i \leq n$, a_i is an *instance_of* w_i.

Axiom A.34 Every coercion pattern for a built-in is evaluable. In more detail, if z with argument classes $w_1, \ldots, w_n$ *coerces_to* r, then there exists an evaluable r that relates $a_1, \ldots, a_n$, such that, for every $1 \leq i \leq n$, a_i is *of_type* w_i.

Axiom A.35 **index_ordering** is antisymmetric.

Axiom A.36 $\langle$**proper_indexes**, **index_ordering** $\rangle$ is a linear order.

This appendix listed all the axioms in the definition of the species of mathematical structures over object-oriented concepts contributed by Chapter 1. The formal version of the axioms given here can be found in Appendix C.

Appendix B

The Construction of a Witness Mathematical Structure

This appendix describes in all the necessary detail the construction of a witness element in the species of mathematical structures $S(\Sigma^*)$ defined in Chapter 1. This suffices to prove that the theory of the species is consistent.

A witness mathematical structure $W = \langle D^W, R^W \rangle$ that satisfies **Axioms A.1-A.35** is now constructed. The domain D^W is set by assigning values to each of the sets that comprise D^W, as follows. The principal base sets are:

$$
\begin{aligned}
\mathbf{C} &= \{A, B, C, D, E, F, G, H\} \\
\mathbf{Gc} &= \{I\} \\
\mathbf{C}^* &= \{A, B, C, D, E, F, G, H, I\} \\
\mathbf{O} &= \{!1, !2, !3, !4, !5\} \\
\mathbf{Go} &= \{\top, \bot\} \\
\mathbf{O}^* &= \{!1, !2, !3, !4, !5, \top, \bot\} \\
\mathbf{Op} &= \{O\} \\
\mathbf{Gop} &= \{|\} \\
\mathbf{M} &= \{P, Q, R\} \\
\mathbf{Gm} &= \{|'\} \\
\mathbf{I} &= \{'0,'1\} \\
\mathbf{I}^* &= \{A, B, C, D, E, F, G, H, I, !1, !2, !3, !4, !5, '0, '1\}
\end{aligned}
$$

The auxiliary base sets are:

$$
\begin{aligned}
\mathbf{IM} &= \{\texttt{specialization}, \texttt{generalization}, \texttt{partition}\} \\
\mathbf{IH} &= \{U, V, W, X\} \\
\mathbf{CM} &= \{\texttt{association}, \texttt{ordering}, \texttt{aggregation}\}
\end{aligned}
$$

This sets the domain D^W. It satisfies **Axioms A.10-A.13**. The tuple of relations R^W is set by assigning values to each of the sets that comprise R^W, as follows.

$$
\begin{aligned}
\mathbf{inclusion}^* = \{ \; & \langle B, \texttt{specialization}, A, U \rangle, \\
& \langle B, \texttt{generalization}, D, V \rangle, \\
& \langle B, \texttt{generalization}, E, V \rangle, \\
& \langle F, \texttt{specialization}, C, W \rangle, \\
& \langle G, \texttt{partition}, E, X \rangle, \\
& \langle H, \texttt{partition}, E, X \rangle \\
\}
\end{aligned}
$$

$$
\mathbf{attribution}^* = \{ \; \langle C, I \rangle \; \}
$$

$$\textbf{composition}^* = \{ \langle D, \texttt{association}, G\rangle, \langle C, \texttt{aggregation}, E\rangle, \langle G, \texttt{ordering}, H\rangle \}$$

$$\textbf{interface} = \{ \langle A, O, A\rangle, \langle C, O, C\rangle, \langle F, O, F\rangle \}$$

$$\textbf{implementation} = \{ \langle A, O, P\rangle, \langle C, O, Q\rangle, \langle F, O, R\rangle, \}$$

The sets above are best thought of as schema-level application specific explicit information. They satisfy **Axioms A.14, A.20-A.22, A.24, A.31**.

$$\textbf{inclusion} = \{ \langle B, A\rangle, \langle D, B\rangle, \langle E, B\rangle, \langle F, C\rangle, \langle G, E\rangle, \langle H, E\rangle, \langle A, A\rangle, \langle B, B\rangle, \langle C, C\rangle, \langle D, D\rangle, \langle E, E\rangle, \langle F, F\rangle, \langle G, G\rangle, \langle H, H\rangle, \langle I, I\rangle, \langle D, A\rangle, \langle E, A\rangle, \langle G, B\rangle, \langle G, A\rangle, \langle H, B\rangle, \langle H, A\rangle \}$$

$$\textbf{attribution} = \{ \langle C, I\rangle, \langle F, I\rangle \}$$

$$\textbf{composition} = \{ \langle D, G\rangle, \langle C, E\rangle, \langle C, F\rangle, \langle G, H\rangle, \langle F, E\rangle \}$$

$$\textbf{interface}^* = \{ \langle A, A, O, A\rangle, \langle B, A, O, A\rangle, \langle C, C, O, C\rangle, \langle D, A, O, A\rangle, \langle E, A, O, A\rangle, \langle F, F, O, F\rangle, \langle F, C, O, C\rangle, \langle G, A, O, A\rangle, \langle H, A, O, A\rangle \}$$

The sets above are best thought of as schema-level application specific implicit information, derived through the use of **Axioms A.1-A.7**. They satisfy **Axioms A.15, A.16, A.23, A.29, A.30**.

$$\textbf{classification} = \{ \begin{array}{l} \langle !1, G\rangle, \\ \langle !2, H\rangle, \\ \langle !3, D\rangle, \\ \langle !4, A\rangle, \\ \langle !5, F\rangle, \\ \langle !1, E\rangle, \\ \langle !2, E\rangle, \\ \langle !1, B\rangle, \\ \langle !2, B\rangle, \\ \langle !3, B\rangle, \\ \langle !1, A\rangle, \\ \langle !2, A\rangle, \\ \langle !3, A\rangle, \\ \langle !5, C\rangle \end{array} \} \quad \textbf{description} = \{ \langle !5, I, \perp\rangle \} \quad \textbf{construction} = \{ \begin{array}{l} \langle !1, '1, !2\rangle, \\ \langle !3, !1, !1\rangle, \\ \langle !5, E, !2\rangle \end{array} \}$$

The sets above are best thought of as instance-level application specific explicit information (although the last nine elements in the enumeration of **classification** are best thought of as being derived through the application of **Axiom A.8**). They satisfy **Axioms A.17-A.19** and **A.25-A.28**.

$$\textbf{execution} = \{ \begin{array}{l} \langle P, !1, !1\rangle, \\ \langle P, !2, !1\rangle, \\ \langle P, !3, !1\rangle, \\ \langle P, !4, !1\rangle, \\ \langle Q, !5, !5\rangle, \\ \langle R, !5, !5\rangle \end{array} \}$$

The set above is best thought of as being computed by a given operational semantics of method execution. It satisfies **Axiom A.32**.

$$\textbf{typing} = \{ \begin{array}{l} \langle \perp, I\rangle, \\ \langle \top, I\rangle \end{array} \} \quad \textbf{coercing} = \{ \langle |, I, I, I, |'\rangle \}$$

$$\textbf{evaluation} = \{ \begin{array}{l} \langle |', \top, \top, \perp\rangle, \\ \langle |', \top, \perp, \top\rangle, \\ \langle |', \perp, \top, \top\rangle, \\ \langle |', \perp, \perp, \top\rangle \end{array} \} \quad \textbf{index\_ordering} = \{ \begin{array}{l} \langle '0, '0\rangle, \\ \langle '1, '1\rangle, \\ \langle '0, '1\rangle \end{array} \}$$

The sets above are best thought of as being defined by computations that are delegated to the environment in which the theory is realized. One can think of I as being the set of Boolean values, of $\top$ and $\perp$ as denoting *truth* and *falsity*, respectively, of $|$ as Sheffer's stroke denoting *alternative denial*, and of $|'$ as its (typically computed) truth table. They satisfy **Axioms A.9, A.17, A.34**, and **A.35**.

This sets the relations R^W and completes the construction of the witness mathematical structure $W = \langle D^W, R^W\rangle$. It satisfies **Axioms A.1-A.35**.

This appendix described the construction of a witness mathematical structure, thereby proving that the theory of the species $\mho$ is consistent.

Appendix C

The Mathematical Axiom: Formal Version

This appendix lists all the axioms used, in Chapter 2, for the purposes of logically reconstructing the species of mathematical structures over object-oriented concepts contributed by Chapter 1.

Axiom C.1 **inclusion** is a restriction of **inclusion*** to the latter's first and third arguments, which, in case the inclusion mode is `generalization`, swap positions in **inclusion**. Formally,

$$(is\_a^*(x, y, u, v) \wedge (y = \texttt{specialization} \vee y = \texttt{partition})) \Rightarrow is\_a(x, u) \tag{1}$$

$$(is\_a^*(x, y, u, v) \wedge y = \texttt{generalization}) \Rightarrow is\_a(u, x) \tag{2}$$

Axiom C.2 **inclusion** is reflexive over **classes** and transitive. Formally,

$$class(x) \Rightarrow is\_a(x, x) \tag{3}$$

$$(is\_a(x, y) \wedge is\_a(y, z)) \Rightarrow is\_a(x, z) \tag{4}$$

Axiom C.3 **attribution** contains **attribution***, but not necessarily vice-versa. Formally,

$$has^*(x, y) \Rightarrow has(x, y) \tag{5}$$

Axiom C.4 If x *has* a y and z *is_a* x, then z *has* a y. Formally,

$$(has(x, y) \wedge is\_a(z, x)) \Rightarrow has(z, y) \tag{6}$$

Axiom C.5 **composition** is a restriction of **composition*** to the latter's first and third arguments. Formally,

$$made\_of^*(x, y, z) \Rightarrow made\_of(x, z) \tag{7}$$

Axiom C.6 If x is *made_of* y and z *is_a* x, then z is *made_of* y. Formally,

$$(made\_of(x, y) \wedge is\_a(z, x)) \Rightarrow made\_of(z, y) \tag{8}$$

Axiom C.7 If y *responds_to* z with argument-classes $w_1, \ldots, w_n$ and x *is_a* y, then x *responds_as* y *to* z with argument-classes $w_1, \ldots, w_n$. Formally,

$$(responds\_to(y, z, w_1, \ldots, w_n) \wedge is\_a(x, y)) \Rightarrow responds\_as\_to(x, y, z, w_1, \ldots, w_n) \tag{9}$$

Axiom C.8 If x is an *instance_of* y and y *is_a* z, then x is an *instance_of* z. Formally,

$$(instance\_of(x, y) \wedge is\_a(y, z)) \Rightarrow instance\_of(x, z) \tag{10}$$

Axiom C.9 **index_ordering** is reflexive over **proper_indexes** and transitive. Formally,

$$proper\_index(x) \Rightarrow precedes(x, x) \tag{11}$$

$$(precedes(x, y) \wedge precedes(y, z)) \Rightarrow precedes(x, z) \tag{12}$$

The right hand side of expressions (i)-(xxxii) in Section 2.1 is used (instead of the more usual left hand side) in this and the next axiom because the expressions would otherwise become unacceptably long. This has no semantic consequence whatever.

Axiom C.10 No two base sets among **C**, **Gc**, **O**, **Go**, **Op**, **Gop**, **M**, **Gm**, **I**, **IM**, **IH**, **CM** have an element in common. Formally:

$$\begin{aligned} x \in \mathbf{C} \Rightarrow (&\neg(x \in \mathbf{Gc}) \wedge \neg(x \in \mathbf{O}) \wedge \neg(x \in \mathbf{Go}) \wedge \neg(x \in \mathbf{Op}) \wedge \neg(x \in \mathbf{Gop}) \wedge \\ &\neg(x \in \mathbf{M}) \wedge \neg(x \in \mathbf{Gm}) \wedge \neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \end{aligned} \tag{13}$$

$$\begin{aligned} x \in \mathbf{Gc} \Rightarrow (&\neg(x \in \mathbf{O}) \wedge \neg(x \in \mathbf{Go}) \wedge \neg(x \in \mathbf{Op}) \wedge \neg(x \in \mathbf{Gop}) \wedge \neg(x \in \mathbf{M}) \wedge \\ &\neg(x \in \mathbf{Gm}) \wedge \neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \end{aligned} \tag{14}$$

$$\begin{aligned} x \in \mathbf{O} \Rightarrow (&\neg(x \in \mathbf{Go}) \wedge \neg(x \in \mathbf{Op}) \wedge \neg(x \in \mathbf{Gop}) \wedge \neg(x \in \mathbf{M}) \wedge \neg(x \in \mathbf{Gm}) \wedge \\ &\neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \end{aligned} \tag{15}$$

$$\begin{aligned} x \in \mathbf{Go} \Rightarrow (&\neg(x \in \mathbf{Op}) \wedge \neg(x \in \mathbf{Gop}) \wedge \neg(x \in \mathbf{M}) \wedge \neg(x \in \mathbf{Gm}) \wedge \neg(x \in \mathbf{I}) \\ &\wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \end{aligned} \tag{16}$$

$$\begin{aligned} x \in \mathbf{Op} \Rightarrow (&\neg(x \in \mathbf{Gop}) \wedge \neg(x \in \mathbf{M}) \wedge \neg(x \in \mathbf{Gm}) \wedge \neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \\ &\neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \end{aligned} \tag{17}$$

$$\begin{aligned} x \in \mathbf{Gop} \Rightarrow (&\neg(x \in \mathbf{M}) \wedge \neg(x \in \mathbf{Gm}) \wedge \neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \\ &\neg(x \in \mathbf{CM})) \end{aligned} \tag{18}$$

$$x \in \mathbf{M} \Rightarrow (\neg(x \in \mathbf{Gm}) \wedge \neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \tag{19}$$

$$x \in \mathbf{Gm} \Rightarrow (\neg(x \in \mathbf{I}) \wedge \neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \tag{20}$$

$$x \in \mathbf{I} \Rightarrow (\neg(x \in \mathbf{IM}) \wedge \neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \tag{21}$$

$$x \in \mathbf{IM} \Rightarrow (\neg(x \in \mathbf{IH}) \wedge \neg(x \in \mathbf{CM})) \tag{22}$$

$$x \in \mathbf{IH} \Rightarrow \neg(x \in \mathbf{CM}) \tag{23}$$

Axiom C.11 **classes** = **specific_classes** ∪ **generic_classes**, **objects** = **specific_objects** ∪ **generic_objects**, and **indexicals** = **specific_classes** ∪ **objects** ∪ **proper_indexes**. Formally:

$$((x \in \mathbf{C} \vee x \in \mathbf{Gc}) \Rightarrow x \in \mathbf{C}^*) \wedge (x \in \mathbf{C}^* \Rightarrow (x \in \mathbf{C} \vee x \in \mathbf{Gc})) \tag{24}$$

$$((x \in \mathbf{O} \vee x \in \mathbf{Go}) \Rightarrow x \in \mathbf{O}^*) \wedge (x \in \mathbf{O}^* \Rightarrow (x \in \mathbf{O} \vee x \in \mathbf{Go})) \tag{25}$$

$$((x \in \mathbf{C} \vee x \in \mathbf{O}^* \vee x \in \mathbf{I}) \Rightarrow x \in \mathbf{I}^*) \wedge (x \in \mathbf{I}^* \Rightarrow (x \in \mathbf{C} \vee x \in \mathbf{O}^* \vee x \in \mathbf{I})) \tag{26}$$

Axiom C.12 If $x \in$ **inclusion_modes**, then either $x = \texttt{specialization}$ or $x = \texttt{generalization}$ or $x = \texttt{partition}$. Formally:

$$inclusion\_mode(x) \Rightarrow (x = \texttt{specialization} \lor x = \texttt{generalization} \lor x = \texttt{partition}) \quad (27)$$

Axiom C.13 If $x \in$ **composition_modes**, then either $x = \texttt{association}$ or $x = \texttt{ordering}$ or $x = \texttt{aggregation}$. Formally:

$$composition\_mode(x) \Rightarrow (x = \texttt{association} \lor x = \texttt{ordering} \lor x = \texttt{aggregation}) \quad (28)$$

Axiom C.14 If x *is_a** y on the inheritance hierarchy z by both inclusion mode u and inclusion mode u', then $u = u'$. Formally:

$$(is\_a^*(x, u, y, z) \land is\_a^*(x, u', y, z)) \Rightarrow u = u' \quad (29)$$

Axiom C.15 **inclusion** is antisymmetric. Formally:

$$(is\_a(x, y) \land is\_a(y, x)) \Rightarrow x = y \quad (30)$$

Axiom C.16 If x *is_a* y and y belongs to **specific_classes** then so does x; similarly, if x *is_a* y and y belongs to **generic_classes** then so does x. Formally:

$$(is\_a(x, y) \land specific\_class(y)) \Rightarrow specific\_class(x) \quad (31)$$

$$(is\_a(x, y) \land generic\_class(y)) \Rightarrow generic\_class(x) \quad (32)$$

Axiom C.17 **inclusion** is weakly directed. Formally:

$$(is\_a(x, y) \land is\_a(x, z) \land \neg(y = z)) \Rightarrow \exists w(is\_a(y, w) \land is\_a(z, w)) \quad (33)$$

Axiom C.18 If $x \in$ **specific_objects**, then, for some $y \in$ **specific_classes**, x is an *instance_of* y, and if $x \in$ **generic_objects**, then, for some $y \in$ **generic_classes**, x is *of_type* y. Formally:

$$specific\_object(x) \Rightarrow \exists y(instance\_of(x, y)) \quad (34)$$

$$generic\_object(x) \Rightarrow \exists y(of\_type(x, y)) \quad (35)$$

Axiom C.19 If x is an *instance_of* y and y generalizes $z_1, \ldots, z_n$, then, for at least one $i, 1 \leq i \leq n$, x is an *instance_of* z_i. Formally:

$$(instance\_of(x, y) \land is\_a^*(y, u, z, v) \land u = \texttt{generalization}) \Rightarrow \exists z'(instance\_of(x, z') \land is\_a^*(y, u, z', v)) \quad (36)$$

Axiom C.20 In $\texttt{partition}$ inclusion on a given hierarchy, the extension of the partitioned class is the iterated union of the extension of its subclasses and the iterated intersection of the latter is empty. Formally:

$$(instance\_of(x, y) \land is\_a^*(z, u, y, v) \land u = \texttt{partition}) \Rightarrow \exists z'(instance\_of(x, z') \land is\_a^*(z', u, y, v)) \quad (37)$$

$$(instance\_of(x, y) \land is\_a^*(z, u, y, v) \land u = \texttt{partition} \land instance\_of(x, z) \land instance\_of(x, z') \land is\_a^*(z', u, y, v)) \Rightarrow z = z' \quad (38)$$

Axiom C.21 **attribution*** is not weakly directed through **inclusion**. Formally:

$$has^*(x,z) \wedge has^*(y,z) \Rightarrow \neg(\exists w(is\_a(w,x) \wedge is\_a(w,y))) \tag{39}$$

Axiom C.22 If x is *made_of* z by both composition mode u and composition mode v, then $u = v$. Formally:

$$(made\_of^*(x,u,z) \wedge made\_of^*(x,v,z)) \Rightarrow u = v \tag{40}$$

Axiom C.23 If x is an `association` (an `ordering`) of both u and v, then $u = v$. Formally:

$$\begin{aligned}&(made\_of^*(x,y,u) \wedge (y = \texttt{association} \vee y = \texttt{ordering}) \wedge \\ &made\_of^*(x,y,v)) \Rightarrow u = v\end{aligned} \tag{41}$$

Axiom C.24 **composition** is asymmetric. Formally:

$$made\_of(x,y) \Rightarrow \neg(made\_of(y,x)) \tag{42}$$

Axiom C.25 If y and z are distinct, x is both a y and z, y is *made_of** y' by composition mode y'', and z is *made_of** z' by composition mode z'', then both $y' = z'$ and $y'' = z''$. Formally:

$$\begin{aligned}&(\neg(y = z) \wedge is\_a(x,y) \wedge is\_a(x,z) \wedge made\_of^*(y,y',y'') \wedge made\_of^*(z,z',z'')) \Rightarrow \\ &(y' = z' \wedge y'' = z'')\end{aligned} \tag{43}$$

Axiom C.26 If a is an *instance_of* x and x *has* a y, then there is an *instance_of* y, say b, such that b is the y of a. Formally:

$$(instance\_of(a,x) \wedge has(x,y)) \Rightarrow \exists b(property(a,y,b) \wedge instance\_of(b,y)) \tag{44}$$

Axiom C.27 If a is an *instance_of* x, x is an `association` of z and a has a *component* b with index b', then b is an *instance_of* z and $b = b'$. Formally:

$$\begin{aligned}&(instance\_of(a,x) \wedge made\_of^*(x,y,z) \wedge y = \texttt{association} \wedge component(a,b',b)) \Rightarrow \\ &(instance\_of(b,z)) \wedge b' = b)\end{aligned} \tag{45}$$

Axiom C.28 If a is an *instance_of* x, x is an `ordering` of z and a has a *component* b with index b', then b is an *instance_of* z and b' is a proper index. Formally:

$$\begin{aligned}&(instance\_of(a,x) \wedge made\_of^*(x,y,z) \wedge y = \texttt{ordering} \wedge component(a,b',b)) \Rightarrow \\ &(instance\_of(b,z) \wedge proper\_index(b'))\end{aligned} \tag{46}$$

Axiom C.29 If a is an *instance_of* x and x is an `aggregation` of z, then there is an *instance_of* z, say b, that is a *component* of a with index z. Formally:

$$\begin{aligned}&(instance\_of(a,x) \wedge made\_of^*(x,y,z) \wedge y = \texttt{aggregation}) \Rightarrow \exists b(component(a,z,b) \wedge \\ &instance\_of(b,z))\end{aligned} \tag{47}$$

Axiom C.30 If $x \in \mathbf{C}$ then there is a z defined on y with argument classes $w_1, \ldots, w_n$ such that x responds as y to z. Formally:

$$specific\_class(x) \Rightarrow \exists y,z,w_1,\ldots,w_n(responds\_as\_to(x,y,z,w_1,\ldots,w_n)) \tag{48}$$

Axiom C.31 For distinct y and y', if x responds as y to z and x responds as y' to z (in both cases with argument classes $w_1, \ldots, w_n$), then either y *is_a* y' or y' *is_a* y. Formally:

$$\begin{aligned}&(responds\_as\_to(x,y,z,w_1,\ldots,w_n) \wedge responds\_as\_to(x,y',z,w_1,\ldots,w_n) \wedge \neg(y = y')) \Rightarrow \\ &(is\_a(y,y') \vee is\_a(y',y))\end{aligned} \tag{49}$$

Axiom C.32 If y *responds_to* z with argument classes $w_1, \ldots, w_n$, then there exists exactly one r such that z with argument classes $w_1, \ldots, w_n$ is *implemented_in_by* by r. Formally:

$$responds\_to(y, z, w_1, \ldots, w_n) \Rightarrow \exists r(implemented\_in\_by(y, z, w_1, \ldots, w_n, r)) \tag{50}$$

$$(implemented\_in\_by(y, z, w_1, \ldots, w_n, r) \wedge implemented\_in\_by(y, z, w_1, \ldots, w_n, r')) \Rightarrow r = r' \tag{51}$$

Axiom C.33 If z at w_1 with argument classes $w_2, \ldots, w_n$ is *implemented_in_by* r and, upon being executed for a_1, r *responds_with* $a_2, \ldots, a_n$, then, for $1 \leq i \leq n$, a_i is an *instance_of* w_i. Formally:

$$\begin{aligned}&implemented\_in\_by(w_1, z, w_2, \ldots, w_n, r) \Rightarrow \exists a_1, \ldots, a_n(responds\_with(a_1, r, a_2, \ldots, a_n) \wedge \\ &instance\_of(a_1, w_1)) \wedge \ldots \wedge instance\_of(a_n, w_n))\end{aligned} \tag{52}$$

Axiom C.34 If z with argument classes $w_1, \ldots, w_n$ *coerces_to* r, then there exists an evaluable r that relates $a_1, \ldots, a_n$, such that, for every $1 \leq i \leq n$, a_i is *of_type* w_i. Formally:

$$\begin{aligned}&coerces\_to(z, w_1, \ldots, w_n, r) \Rightarrow \exists a_1 \ldots a_n(evaluates\_to(r, a_1, \ldots, a_n)) \wedge \\ &of\_type(a_1, w_1) \wedge \ldots \wedge of\_type(a_n, w_n))\end{aligned} \tag{53}$$

Axiom C.35 index_ordering is antisymmetric. Formally:

$$precedes(x, y) \wedge precedes(y, x) \Rightarrow x = y \tag{54}$$

Axiom C.36 $\langle$**proper_indexes**, **index_ordering** $\rangle$ is a chain. Formally:

$$(proper\_index(x) \wedge proper\_index(y) \wedge \neg(x = y)) \Rightarrow (precedes(x, y) \vee precedes(y, x)) \tag{55}$$

This appendix listed all the axioms used in the logical reconstruction of the OODM. The informal version of the axioms given here can be found in Appendix A.

Appendix D

The Clausal Form of the Object Modelling Axioms

This appendix presents the complete set of Horn clauses obtained by translating into clausal form the sort, completion and exception axioms of ODTs, as presented in Chapter 2.

Each axiom is presented in a different section. The order in which the clauses are listed is (roughly) alphabetical on the clause head for the sort and completion axioms, and on the leftmost subgoal in the body for the exception axiom. This is for legibility only and has no semantic implication.

D.1 The Clausal Form of the Sort Axiom of ODTs

violation_1 $\leftarrow$
    is_a*(X,Y,U,V),
    $\sim$ class(X).
violation_1 $\leftarrow$
    is_a*(Y,X,U,V),
    $\sim$ inclusion_mode(X).
violation_1 $\leftarrow$
    is_a*(Y,U,X,V),
    $\sim$ class(X).
violation_1 $\leftarrow$
    is_a*(Y,U,V,X),
    $\sim$ inclusion_hierarchy(X).

violation_2 $\leftarrow$
    is_a(B,C),
    $\sim$ class(B).
violation_2 $\leftarrow$
    is_a(C,B),
    $\sim$ class(B).

---

violation_3 $\leftarrow$
    has_a*(B,C),
    $\sim$ specific_class(B).
violation_3 $\leftarrow$
    has_a*(C,B),
    $\sim$ specific_class(B).

violation_4 $\leftarrow$
    has_a(B,C),
    $\sim$ specific_class(B).
violation_4 $\leftarrow$
    has_a(C,B),
    $\sim$ specific_class(B).

---

violation_5 ←
made_of(B,C),
∼ specific_class(B).
violation_5 ←
made_of(C,B),
∼ specific_class(B).

violation_6 ←
made_of*(C,X,Y),
∼ specific_class(C).
violation_6 ←
made_of*(X,C,Y),
∼ composition_mode(C).
violation_6 ←
made_of*(X,C,Y),
∼ specific_class(C).

---

violation_7 ←
responds_to(C,X,Y),
∼ specific_class(C).
violation_7 ←
responds_to(X,C,Y),
∼ specific_operation(C).
violation_7 ←
responds_to(X,Y,C),
∼ class(C).

violation_8 ←
instance_of(B,C),
∼ specific_object(B).
violation_8 ←
instance_of(C,B),
∼ specific_class(B).

---

violation_9 ←
property(C,X,Y),
∼ specific_object(C).
violation_9 ←
property(X,C,Y),
∼ specific_class(C).
violation_9 ←
property(X,Y,C),
∼ object(C).

violation_10 ←
component(C,X,Y),
∼ specific_object(C).
violation_10 ←
component(X,C,Y),
∼ index(C).
violation_10 ←
component(X,Y,C),
∼ object(C).

---

violation_11 ←
implemented_in_by(X,Y,U,V),
∼ specific_class(X).
violation_11 ←
implemented_in_by(Y,X,U,V),
∼ specific_operation(X).
violation_11 ←
implemented_in_by(Y,U,X,V),
∼ class(X).
violation_11 ←
implemented_in_by(Y,U,V,X),
∼ specific_method(X).

violation_12 ←
of_type(B,C),
∼ generic_object(B).
violation_12 ←
of_type(C,B),
∼ generic_class(B).

```
violation_13 ←
    coerces_to(C,X,Y),
    ~ generic_operation(C).
violation_13 ←
    coerces_to(X,C,Y),
    ~ generic_class(C).
violation_13 ←
    coerces_to(X,Y,C),
    ~ generic_method(C).
```

```
violation_14 ←
    responds_as_to(X,Y,U,V),
    ~ specific_class(X).
violation_14 ←
    responds_as_to(Y,X,U,V),
    ~ specific_class(X).
violation_14 ←
    responds_as_to(Y,U,X,V),
    ~ specific_operation(X).
violation_14 ←
    responds_as_to(Y,U,V,X),
    ~ class(X).
```

```
violation_15 ←
    responds_as_to(X,Y,U,V),
    ~ specific_class(X).
violation_15 ←
    responds_as_to(Y,X,U,V),
    ~ specific_class(X).
violation_15 ←
    responds_as_to(Y,U,X,V),
    ~ specific_operation(X).
violation_15 ←
    responds_as_to(Y,U,V,X),
    ~ class(X).
```

```
violation_16 ←
    responds_with(C,X,Y),
    ~ specific_object(C).
violation_16 ←
    responds_with(X,C,Y),
    ~ specific_method(C).
violation_16 ←
    responds_with(X,Y,C),
    ~ object(C).
```

```
violation_17 ←
    evaluates_to(B,C),
    ~ generic_method(B).
violation_17 ←
    evaluates_to(C,B),
    ~ generic_object(B).
```

```
violation_18 ←
    class(A),
    ~ specific_class(A),
    ~ generic_class(A).
```

```
violation_19 ←
    object(A),
    ~ specific_object(A),
    ~ generic_object(A).
```

```
violation_20 ←
    index(A),
    ~ specific_class(A),
    ~ object(A),
    ~ proper_index(A).
```

D.2 The Clausal Form of the Completion Axiom of ODTs

```
is_a(A,B) ←
    C==specialization,
    is_a*(A,C,B,D).
is_a(A,B) ←
    C==partition,
    is_a*(A,C,B,D).
is_a(A,B) ←
    C==generalization,
    is_a*(B,C,A,D).
is_a(A,A) ←
    class(A).
is_a(A,B) ←
    is_a(A,C),
    is_a(C,B).
```

```
has(A,B) ←
    has*(A,B).
has(A,B) ←
    has(C,B),
    is_a(A,C).
```

```
made_of(A,B) ←
    made_of*(A,C,B).
made_of(A,B) ←
    is_a(A,C),
    made_of(C,B).
```

```
responds_as_to(A,B,C,D) ←
    is_a(A,B),
    responds_to(B,C,D).
```

```
instance_of(A,B) ←
    instance_of(A,C),
    is_a(C,B).
```

```
precedes(A,A) ←
    proper_index(A).
precedes(A,B) ←
    precedes(A,C),
    precedes(C,B).
```

D.3 The Clausal Form of the Exception Axiom of ODTs

violation_33 ←
specific_class(A),
generic_class(A).
violation_33 ←
specific_class(A),
specific_object(A).
violation_33 ←
specific_class(A),
generic_object(A).
violation_33 ←
specific_class(A),
specific_operation(A).
violation_33 ←
specific_class(A),
generic_operation(A).
violation_33 ←
specific_class(A),
specific_method(A).
violation_33 ←
specific_class(A),
generic_method(A).
violation_33 ←
specific_class(A),
proper_index(A).
violation_33 ←
specific_class(A),
inclusion_mode(A).
violation_33 ←
specific_class(A),
inclusion_hierarchy(A).
violation_33 ←
specific_class(A),
composition_mode(A).

violation_34 ←
generic_class(A),
specific_object(A).
violation_34 ←
generic_class(A),
generic_object(A).
violation_34 ←
generic_class(A),
specific_operation(A).
violation_34 ←
generic_class(A),
generic_operation(A).
violation_34 ←
generic_class(A),
specific_method(A).
violation_34 ←
generic_class(A),
generic_method(A).
violation_34 ←
generic_class(A),
proper_index(A).
violation_34 ←
generic_class(A),
inclusion_mode(A).
violation_34 ←
generic_class(A),
inclusion_hierarchy(A).
violation_34 ←
generic_class(A),
composition_mode(A).

violation_35 ←
    specific_object(A),
    generic_object(A).
violation_35 ←
    specific_object(A),
    specific_operation(A).
violation_35 ←
    specific_object(A),
    generic_operation(A).
violation_35 ←
    specific_object(A),
    specific_method(A).
violation_35 ←
    specific_object(A),
    generic_method(A).
violation_35 ←
    specific_object(A),
    proper_index(A).
violation_35 ←
    specific_object(A),
    inclusion_mode(A).
violation_35 ←
    specific_object(A),
    inclusion_hierarchy(A).
violation_35 ←
    specific_object(A),
    composition_mode(A).

violation_36 ←
    generic_object(A),
    specific_operation(A).
violation_36 ←
    generic_object(A),
    generic_operation(A).
violation_36 ←
    generic_object(A),
    specific_method(A).
violation_36 ←
    generic_object(A),
    generic_method(A).
violation_36 ←
    generic_object(A),
    proper_index(A).
violation_36 ←
    generic_object(A),
    inclusion_mode(A).
violation_36 ←
    generic_object(A),
    inclusion_hierarchy(A).
violation_36 ←
    generic_object(A),
    composition_mode(A).

---

violation_37 ←
    specific_operation(A),
    generic_operation(A).
violation_37 ←
    specific_operation(A),
    specific_method(A).
violation_37 ←
    specific_operation(A),
    generic_method(A).
violation_37 ←
    specific_operation(A),
    proper_index(A).
violation_37 ←
    specific_operation(A),
    inclusion_mode(A).
violation_37 ←
    specific_operation(A),
    inclusion_hierarchy(A).
violation_37 ←
    specific_operation(A),
    composition_mode(A).

violation_38 ←
    generic_operation(A),
    specific_method(A).
violation_38 ←
    generic_operation(A),
    generic_method(A).
violation_38 ←
    generic_operation(A),
    proper_index(A).
violation_38 ←
    generic_operation(A),
    inclusion_mode(A).
violation_38 ←
    generic_operation(A),
    inclusion_hierarchy(A).
violation_38 ←
    generic_operation(A),
    composition_mode(A).

---

violation_39 ←
 specific_method(A),
 generic_method(A).
violation_39 ←
 specific_method(A),
 proper_index(A).
violation_39 ←
 specific_method(A),
 inclusion_mode(A).
violation_39 ←
 specific_method(A),
 inclusion_hierarchy(A).
violation_39 ←
 specific_method(A),
 composition_mode(A).

violation_40 ←
 generic_method(A),
 proper_index(A).
violation_40 ←
 generic_method(A),
 inclusion_mode(A).
violation_40 ←
 generic_method(A),
 inclusion_hierarchy(A).
violation_40 ←
 generic_method(A),
 composition_mode(A).

---

violation_41 ←
 proper_index(A),
 inclusion_mode(A).
violation_41 ←
 proper_index(A),
 inclusion_hierarchy(A).
violation_41 ←
 proper_index(A),
 composition_mode(A).

violation_42 ←
 inclusion_mode(A),
 inclusion_hierarchy(A).
violation_42 ←
 inclusion_mode(A),
 composition_mode(A).

---

violation_43 ←
 inclusion_hierarchy(A),
 composition_mode(A).

violation_44 ←
 class(A),
 ∼ specific_class(A),
 ∼ generic_class(A).
violation_44 ←
 specific_class(A),
 ∼ class(A).
violation_44 ←
 generic_class(A),
 ∼ class(A).

---

violation_45 ←
 object(A),
 ∼ specific_object(A),
 ∼ generic_object(A).
violation_45 ←
 specific_object(A),
 ∼ object(A).
violation_45 ←
 generic_object(A),
 ∼ object(A).

violation_46 ←
 index(A),
 ∼ specific_class(A),
 ∼ object(A),
 ∼ proper_index(A).
violation_46 ←
 specific_class(A),
 ∼ index(A).
violation_46 ←
 object(A),
 ∼ index(A).
violation_46 ←
 proper_index(A),
 ∼ index(A).

---

violation_47 ← inclusion_mode(A), ~ A==specialization, ~ A==generalization, ~ A==partition.	violation_48 ← composition_mode(A), ~ A==association, ~ A==ordering, ~ A==aggregation.
violation_49 ← (is_a*(E,X,Y,Z), is_a*(E,W,Y,Z)), ~ X==W.	violation_50 ← (is_a(B,C), is_a(C,B)), ~ B==C.
violation_51 ← (is_a(B,C), specific_class(C)), ~ specific_class(B).	violation_52 ← (is_a(B,C), generic_class(C)), ~ generic_class(B).
violation_53 ← (is_a(D,E), is_a(D,X), ~ E==X), ~ also_53(E,Y,X).	also_53(X,Y,Z) ← is_a(X,Y), is_a(Z,Y).
violation_54 ← specific_object(B), ~ also_54(B).	also_54(D) ← instance_of(D,E).
violation_55 ← generic_object(B), ~ also_55(B).	also_55(D) ← of_type(D,E).
violation_56 ← (instance_of(X,Y), is_a*(Y,Z,W,V), Z==generalization), ~ also_56(X,U,Y,Z,V).	also_56(Z,W,V,U,T) ← is_a*(V,U,W,T), instance_of(Z,W).
violation_57 ← (instance_of(X,Y), is_a*(Y,Z,W,V), Z==partition), ~ also_57(X,U,Y,Z,V).	also_57(Z,W,V,U,T) ← is_a*(V,U,W,T), instance_of(Z,W).
violation_58 ← (instance_of(X,Y), is_a*(Z,W,Y,V), W==partition, instance(X,Z), instance(X,U), is_a*(U,Y,Y,V)), ~ Z==U.	violation_59 ← (has_a*(D,E), has_a*(X,E)), is_a(Y,D), is_a(Y,X).

```
violation_60 ←
    (made_of*(D,E,X),
    made_of*(D,Y,X)),
    ~ E==Y.
```

```
violation_61 ←
    (made_of*(D,E,X),
    E==association,
    made_of*(D,E,Y)),
    ~ X==Y.
violation_61 ←
    (made_of*(D,E,X),
    E==ordering,
    made_of*(D,E,Y)),
    ~ X==Y.
```

```
violation_62 ←
    ~ made_of(B,C),
    made_of(C,B).
```

```
violation_63 ←
    (is_a(Y,Z),
    is_a(Y,W),
    ~ Z==W,
    made_of*(Z,V,U),
    made_of*(W,T,S)),
    ~ V==T.
violation_63 ←
    (is_a(Y,Z),
    is_a(Y,W),
    ~ Z==W,
    made_of*(Z,V,U),
    made_of*(W,T,S)),
    ~ U==S.
```

```
violation_64 ←
    (instance_of(D,E),
    has(E,X)),
    ~ also_64(Y,X,D).
```

```
also_64(X,Y,Z) ←
    property(Z,Y,X),
    instance_of(X,Y).
```

```
violation_65 ←
    (instance_of(X,Y),
    made_of*(Y,Z,W),
    Z==association,
    component(X,V,U)),
    ~ instance_of(U,W).
violation_65 ←
    (instance_of(X,Y),
    made_of*(Y,Z,W),
    Z==association,
    component(X,V,U)),
    ~ V==U.
```

```
violation_66 ←
    (instance_of(X,Y),
    made_of*(Y,Z,W),
    Z==ordering,
    component(X,V,U)),
    ~ instance_of(U,W).
violation_66 ←
    (instance_of(X,Y),
    made_of*(Y,Z,W),
    Z==ordering,
    component(X,V,U)),
    ~ proper_index(V).
```

```
violation_67 ←
    (instance_of(E,X),
    made_of*(X,Y,Z),
    Y==aggregation),
    ~ also_67(W,Z,E).
```

```
also_67(Y,Z,W) ←
    component(W,Z,Y),
    instance_of(Y,Z).
```

---

violation_68 ← specific_class(D), ∼ also_68(D).	also_68(X) ← responds_as_to(X,Y,Z,W).
violation_69 ← (responds_as_to(E,X,Y,Z), responds_as_to(E,W,Y,Z), ∼ X==W), ∼ is_a(X,W), ∼ is_a(W,X).	violation_70 ← responds_to(D,E,X), ∼ also_70(D,E,X).
also_70(X,Y,Z) ← implemented_in_by(X,Y,Z,W).	violation_71 ← (implemented_in_by(E,X,Y,Z), implemented_in_by(E,X,Y,W)), ∼ Z==W.
violation_72 ← implemented_in_by(X,U,Y,Z), ∼ also_72(W,X,V,Y,Z).	also_72(Z,W,V,U,T) ← responds_with(Z,T,V), instance_of(Z,W), instance_of(V,U).
violation_73 ← coerces_to(U,X,Y,Z), ∼ also_73(W,X,V,Y,Z).	also_73(Z,W,V,U,T) ← evaluates_to(T,Z,V), of_type(Z,W), of_type(V,U).
violation_74 ← (precedes(D,E), precedes(E,D)), ∼ D==E.	violation_75 ← (proper_index(B), proper_index(C), ∼ B==C), ∼ precedes(B,C), ∼ precedes(C,B).

This appendix presented the set of Horn clauses obtained through the translation into clausal form of the sort, completion and exception axioms of ODTs.

Appendix E

Glossary

Σ	a theory of sets
Σ^*	a theory containing Σ
$\mho$	the theory of the $S(\Sigma^*)$
Ω	an instance of $\mho$

A

$\mathbb{A}$	an action specification language
$A(S(\Sigma^*))$	the axiom of the species $S(\Sigma^*)$
ACP	the antitrust control problem from (Ceri et al., 1990b)
ADB	active database
ADT	abstract data type
AR	active rule
ASL	action specification language
$AxBP$	the broken-persistence axiom
$AxIW$	the interference axiom
$AxTP$	the time-point axiom

C

$\mathbb{C}$	a condition specification language
$\mathcal{C}$	a set of individual constants
$\mathcal{CA}$	the alphabet of a clausal language
CDB	the constraint part of a deductive database
$\mathcal{CL}$	the clausal language of ODTs
$\mathcal{CL}^\bullet$	$\mathcal{CL}$ extended with integrity constraint predicates
$\mathcal{CL}^\bullet$	the request language associated with $\mathcal{CL}^\bullet$
$clausal(DBAx)$	the clausal version of $DBAx$
$clausal(OMAx)$	the clausal version of SAx
CSL	condition specification language
CWA	the closed world assumption
$CWAx$	the closed world axioms

D

$\mathbb{D}$	a data definition language
$DBAx$	the database axioms

DBMS	database management system
DCA	the domain closure assumption
$DCAx$	the domain closure axiom
DDB	deductive database
$DDBAx$	the deductive database axioms
DDL	data definition language
DDT	deductive database theory
$DedAx$	the deductive axioms of a deductive database
DML	data manipulation language
$DODB$	the deductive object database axioms
DODT	deductive object database theory
DOOD	deductive object-oriented database
DOOT	deductive object-oriented theory
DQL	data query language
DR	deductive rule
DRDB	deductive relational database
DRDT	deductive relational database theory

E

$\mathbb{E}$	an event specification language
ECA	event-condition-action rule
EDB	extensional database
EDB	the extensional part of a deductive database
EHB	the Herbrand base associated with an EDB
ESL	event specification language
EVT	event theory

F

FOCT	first-order clausal theory
FOL	first-order logic

H

HAD	$*holdsAt$-derived
HCL	Horn clausal logic
HCT	Horn clausal theory

I

IC	the application specific integrity constraints
ID	the application specific rules characterizing intensionally defined data
IDB	intensional database
IDB	the intensional part of a deductive database
IHB	the Herbrand base associated with an IDB
IMP_DEC	the set of operation implementation declarations
IMP_DEF	the set of operation implementation definitions
INS	the instance-level axioms
INT_DEC	the set of operation interface declarations

IO	input and output

K

K	the class of first-order theories
K^{OLL}	the instance of K whose language is OLL
K^{OLL}_{DBAx}	K^{OLL} with proper axioms $DBAx$
K^{OLL}_{OMAx}	K^{OLL} with proper axioms $OMAx$

L

LIB	the *library* example from (Peckham and Maryanski, 1988)
LPS	the land parcel selection problem from (Abdelmoty et al., 1994)

M

$\mathbb{M}$	a data manipulation language
$\mathbf{M}$	the set of mathematical structures of a species
$M(S(\Sigma^*))$	the mathematical axiom of $S(\Sigma^*)$
MSS	the **Media Storage Services** example from (Zdonik and Maier, 1990a)

O

ODA	the object dynamics assumption
ODR	the object dynamics variant of the resolution principle
ODT	object database theory
OLL	Object-Logic Language, the language of ODTs
OM	Object Manager, the persistent object store of ROCK & ROLL
OMA	the object modelling assumptions
$OMAx$	the object modelling axioms
OODB	object-oriented database
OODM	object-oriented model of data

P

$\mathcal{P}$	a set of predicate symbols
PAx	the particularization axioms

Q

$\mathbb{Q}$	a data query language

R

RA	the representational assumptions
RAx	the representational axioms
RDB	relational database

RDM	relational model of data
RDT	relational database theory
RES	the resolution principle
RMA	the relational modelling assumptions
$RMAx$	the relational modelling axioms
ROCK	Rule-Object Computational Kernel, the imperative part of ROCK & ROLL
ROCK & ROLL	the DOOD comprising OM, ROCK and ROLL
ROLL	Rule-Object Logic Language, the declarative part of ROCK & ROLL

S

S	database state
$S(\Sigma^*)$	a species of mathematical structures over Σ^*
SAx	the state axioms
SCH	the schema-level axioms
SLD	linear resolution with selection function for definite clauses
SLDNF	SLD with negation-as-finite-failure

T

$\mathcal{T}$	a set of terms
$T(S(\Sigma^*))$	the typical characterization of $S(\Sigma^*)$

U

UNA	the unique name assumption
$UNAx$	the unique name axioms

V

$\mathcal{V}$	a set of individual variables

Bibliography

Abdelmoty, A. I., Paton, N. W., Williams, M. H., Fernandes, A. A., Barja, M. L., and Dinn, A. (1994). Geographic Data Handling in a Deductive Object-Oriented Database. In *(Karagiannis, 1994)*, pages 445–454.

Abdelmoty, A. I., Williams, M. H., and Paton, N. W. (1993a). Deduction and Deductive Databases for Geographic Data Handling. In Abel, D. and Ooi, B. C., editors, *Advances in Spatial Databases: Third International Symposium - SSD'93*, LNCS 692, pages 443–464, Singapore. Springer-Verlag. ISBN 3-540-56869-7, 0-387-56869-7.

Abdelmoty, A. I., Williams, M. H., and Quinn, M. (1993b). A Rule-Based Approach to Computerized Map Reading. *Information and Software Technology*, 35(10):587–602.

Abiteboul, S. (1990). Towards a Deductive Object-Oriented Database Language. *Data & Knowledge Engineering*, 5:263–287. Preliminary Version in (Kim et al., 1990), pages 453–472.

Abiteboul, S. (1992). Deductive and Object-Oriented Databases. Technical Report Verso Report 045, INRIA. Notes for a tutorial in the 1992 Joint International Conference and Symposium on Logic Programming, Washington.

Abiteboul, S. and Grumbach, S. (1987). COL: A Language for Complex Objects Based on Recursive Rules. In *Proceedings of the Workshop on Database Programming Languages*, pages 253–276, Roskoff, France. Complete version in (Abiteboul and Grumbach, 1990). See also (Abiteboul and Grumbach, 1988).

Abiteboul, S. and Grumbach, S. (1988). COL: A Logic-Based Language for Complex Objects. In *(Schmidt et al., 1988)*, pages 271–293. See also (Abiteboul and Grumbach, 1987; Abiteboul and Grumbach, 1990).

Abiteboul, S. and Grumbach, S. (1990). COL: A Logic-Based Language for Complex Objects. In Bancilhon, F. and Buneman, P., editors, *Advances in Database Programming Languages*, Frontier Series, chapter 21, pages 347–374. ACM Press/Addison-Wesley Publ. Co., New York, NY, USA. Abstract in (Abiteboul and Grumbach, 1987). See also (Abiteboul and Grumbach, 1988). ISBN 0-201-50257-7.

Abiteboul, S. and Hull, R. (1987). IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565.

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley Publishing Company, Reading, MA, USA.

Abiteboul, S. and Kanellakis, P. C. (1989a). Object Identity as a Query Language Primitive. In *(Clifford et al., 1989)*, pages 159–173. This is a preliminary version of (Abiteboul and Kanellakis, 1989b).

Abiteboul, S. and Kanellakis, P. C. (1989b). Object Identity as a Query Language Primitive. Technical Report 1022, INRIA. A preliminary version appears as (Abiteboul and Kanellakis, 1989a).

Abiteboul, S., Lausen, G., Uphoff, H., and Waller, E. (1993). Methods and Rules. In *(Buneman and Jajodia, 1993)*, pages 32–41.

Agrawal, R. and Gehani, N. (1989). ODE (Object Database and Environment): The Language and the Data Model. In *(Clifford et al., 1989)*, pages 36–45. Also available as AT&T Technical Report att-db-tr-89-2.

Aït-Kaci, H. and Nasr, R. (1986). LOGIN: A Logic Programming Language with Built-in Inheritance. *The Journal of Logic Programming*, 1986(3):185–215.

Alagić, S., Sunderraman, R., and Bagai, R. (1993). Declarative Object-Oriented Programming: Inheritance, Subtyping and Prototyping. In *(Ceri et al., 1993)*, pages 236–259.

Annevelink, J., Ahad, R., Carlson, A., Fishman, D., Heytens, M., and Kent, W. (1995). Object SQL – A Language for the Design and Implementation of Object Databases. In *(Kim, 1995)*, chapter 3, pages 42–68.

Apt, K. R. (1990). Logic Programming. In *(van Leeuwen, 1990)*, chapter 10, pages 493–574.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. B. (1990). The Object-Oriented Database System Manifesto. In *(Kim et al., 1990)*, pages 223–240.

Bal, R. and Balsters, H. (1993). A Deductive and Typed Object-Oriented Language. In *(Ceri et al., 1993)*, pages 340–359.

Bal, R., Balsters, H., de By, R. A., Bosschaart, A., Flokstra, J., van Keulen, M., Skwronek, J., and Termorshuizen, B. (1992). The TM Manual version 2.0 revision d. Technical Report INF92-81, Computer Science Department, University of Twente, Enschede, The Netherlands. Revision of 21/12/93.

Barja, M. L., Fernandes, A. A., Paton, N. W., Williams, M. H., Dinn, A., and Abdelmoty, A. I. (1995a). Design and Implementation of ROCK & ROLL: A Deductive Object-Oriented Database System. *Information Systems*, 20(3):185–211.

Barja, M. L., Paton, N. W., Fernandes, A. A., Williams, M. H., and Dinn, A. (1994). An Effective Deductive Object-Oriented Database Through Language Integration. In Bocca, J., Jarke, M., and Zaniolo, C., editors, *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 463–474, Santiago, Chile. Morgan Kaufmann Publishers, Inc.

Barja, M. L., Paton, N. W., Williams, M. H., and Fernandes, A. A. (1995b). A Database Programming Language for a Deductive Object-Oriented Database. To appear in *Journal of Information Science and Technology.*

Battista, G. D. and Lenzerini, M. (1989). A Deductive Model for Entity-Relationship Modeling (Extended Abstract). In Apers, P. M. and Wiederhold, G., editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 13–21, Amsterdam, The Netherlands. Morgan Kaufmann Publishers.

Battista, G. D. and Lenzerini, M. (1993). Deductive Entity-Relationship Modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):439–450.

Bayer, P. (1993). State-of-the-Art Report on Reactive Processing in Databases and Artificial Intelligence. *The Knowledge Engineering Review*, 8(2):145–171.

Bayer, P. and Jonker, W. (1994). A Framework for Supporting Triggers in Deductive Databases. In *(Paton and Williams, 1994)*, pages 316–329.

Beeri, C. (1990). A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5:353–382. Also in (Kim et al., 1990), pages 405-430.

Bertino, E., Guerrini, G., and Montesi, D. (1994a). Deductive Object Databases. In *(Tokoro and Pareschi, 1994)*, pages 213–235. A preliminary version of (Bertino et al., 1994b).

Bertino, E., Guerrini, G., and Montesi, D. (1994b). Towards Deductive Object Databases. Manuscript. Final version of (Bertino et al., 1994a).

Bertino, E. and Montesi, D. (1992). Towards a Logical-Object Oriented Programming Language for Databases. In Pirotte, A., Delobel, C., and Gottlob, G., editors, *Advances in Database Technology - EDBT'92, 3rd International Conference on Extending Database Technology*, LNCS 580, pages 168–183, Vienna, Austria. Springer-Verlag. ISBN 3-540-55270-7, 0-387-55270-7.

Blakeley, J. A. (1993). ZQL[C++]: Integrating the C++ Language and an Object Query Capability. In *(Mumick, 1993)*, pages 138–144. Abstract.

Blakeley, J. A. (1995). OQL[C++]: Extending C++ with an Object Query Capability. In *(Kim, 1995)*, chapter 4, pages 69–88.

Bocca, J. (1991). Megalog: A Platform for Developing Knowledge-Base Management Systems. In *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pages 374–380, Tokyo, Japan.

Bonner, A. J. and Kifer, M. (1992). Transaction Logic Programming (or, A Logic of Procedural and Declarative Knowledge. Technical Report CSRI-270, Computer Systems Research Institute, University of Toronto, Toronto, Canada. Revised June 1993.

Bourbaki, N. (1968). *Elements of Mathematics: Theory of Sets*. ADIWES International Series in Mathematics. Hermann, Publishers in Arts and Science/Addison-Wesley Publishing Company, Paris, France./Reading, MA, USA.

Buneman, P. and Jajodia, S., editors (1993). *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, Washington, DC, USA. ACM Press. SIGMOD Record, 22(2), June 1993.

Cacace, F., Ceri, S., Crespi-Reghizzi, S., Tanca, L., and Zicari, R. (1990). Integrating Object-Oriented Data Modeling with a Rule-based Programming Paradigm. In *(Garcia-Molina and Jagadish, 1990)*, pages 225–236.

Cacace, F., Ceri, S., Fraternali, P., Paraboschi, S., and Tanca, L. (1993). An Overview of the Logres System. In *(Mumick, 1993)*, pages 31–43.

Cacace, F., Ceri, S., Tanca, L., and Crespi-Reghizzi, S. (1992). Designing and Prototyping Data-Intensive Applications in the Logres and Algres Programming Environment. *IEEE Transactions on Software Engineering*, 18(6):534–546.

Campin, J., Paton, N. W., and Williams, M. H. (1995). A Structured Specification of an Active Database System. *Information and Software Technology*, 37(1):47–61.

Carey, M. J., DeWitt, D. J., Graefe, G., Haight, D. M., Richardson, J. E., Schuh, D. T., Shekita, E. J., and Vanderberg, S. L. (1990). The EXODUS Extensible DBMS Project: An Overview. In *(Zdonik and Maier, 1990b)*, pages 474–499.

Caseau, Y. (1988). A Model for a Reflective Object-Oriented Language. In Agha, G., Wegner, P., and Yonezawa, A., editors, *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 22–24, San Diego, CA, USA. ACM Press. SIGPLAN Notices 24(4), April 1989.

Caseau, Y. (1990). A Formal System for Producing Demons from Rules in an Object-Oriented Database. In *(Kim et al., 1990)*, pages 203–219.

Caseau, Y. (1991). An Object-Oriented Deductive Language. *Annals of Mathematics and Artificial Intelligence*, 3(2-4):211–258. Special Issue on Deductive Databases.

Caseau, Y., Guillo, P.-Y., and Levenez, E. (1993). A Deductive and Object-Oriented Approach to a Complex Scheduling Problem. In *(Ceri et al., 1993)*, pages 67–80.

Cattell, R. G. (1994). *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley Publ. Co., Reading, MA, USA., 2nd edition. ISBN 0-201-54748-1.

Ceri, S., Crespi-Reghizzi, S., Gottlob, G., Lamperti, F., Lavazza, F., Tanca, L., and Zicari, R. (1988). The Algres Project. In *(Schmidt et al., 1988)*, pages 551–555.

Ceri, S., Crespi-Reghizzi, S., Zicari, R., Lamperti, G., and Lavazza, L. A. (1990a). Algres: An Advanced Database System for Complex Applications. *IEEE Software*, 7(4):68–78.

Ceri, S., Fraternali, P., Paraboschi, S., and Tanca, L. (1994). Active Rule Management in Chimera. Manuscript.

Ceri, S., Gottlob, G., and Tanca, L. (1990b). *Logic Programming and Databases.* Surveys in Computer Science. Springer-Verlag, Berlin, Germany. ISBN 3-540-51728-6, 0-387-51728-6.

Ceri, S., Tanaka, K., and Tsur, S., editors (1993). *Deductive and Object-Oriented Databases (Third International Conference DOOD'93, Phoenix, Arizona, USA)*, LNCS 760, Berlin, Germany. Springer-Verlag. ISBN 3-540-57530-8, ISBN 0-387-57530-8.

Chakravarthy, S. (1993). A Comparative Evaluation of Active Relational Databases. Technical Report UF-CIS-TR-93-002, Department of Computer and Information Science, University of Florida, Gainesville, FL, USA.

Chakravarthy, S. and Mishra, D. (1993). Snoop: An Expressive Event Specification Language for Active Databases. Technical Report UF-CIS-TR-93-007, Department of Computer and Information Science, University of Florida, Gainesville, FL, USA. Revised and Extended Version of UF-CIS-TR-91-23, September 1991.

Chandra, A. K., editor (1989). *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, PA, USA. ACM Press.

Chang, C.-L. and Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics Series. Academic Press, New York, NY, USA. ISBN 0-12-170350-9.

Chen, Q. and Chu, W. W. (1990). HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases. In *(Kim et al., 1990)*, pages 431–452.

Chen, Q. and Gardarin, G. (1988). Nested Datalog: A Rule Language for Complex Objects. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL, USA. ACM Press.

Chen, W., Kifer, M., and Warren, D. S. (1989a). HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114. ISBN/ISSN 0-262-62064-2. Expanded Version in (Chen et al., 1993).

Chen, W., Kifer, M., and Warren, D. S. (1989b). HiLog as a Platform for Database Languages (or Why Predicate Calculus Is Not Enough). In Hull, R., Morrison, R., and Stemple, D., editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 315–329, Gleneden Beach, OR, USA. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-072-8. Expanded Version in (Chen et al., 1993).

Chen, W., Kifer, M., and Warren, D. S. (1993). HiLog: A Foundation for Higher-Order Logic Programming. *The Journal of Logic Programming*, 1993(15):187–230. Expanded version of (Chen et al., 1989b; Chen et al., 1989a).

Chen, W. and Warren, D. S. (1989). C-Logic of Complex Objects. In *(Chandra, 1989)*, pages 369–378.

Clark, K. L. (1978). Negation as Failure. In *(Gallaire and Minker, 1978)*, pages 293–322.

Clifford, J. and King, R., editors (1991). *Proceedings of the 1991 ACM SIGMOD International Conference on the Management of Data*, Denver, CO, USA. ACM Press. SIGMOD Record, 20(2), June 1991.

Clifford, J., Lindsay, B., and Maier, D., editors (1989). *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, OR, USA. ACM Press. SIGMOD Record, 18(2), June 1989.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387. Also in CACM 26(1) January 1983 p. 64–69.

Cruz, I. F. (1993a). Expressing Constraints for Data Display Specification: A Visual Approach. Technical Report CS-93-57, Department of Computer Science, Brown University, Providence, RI, USA.

Cruz, I. F. (1993b). User-Defined Visual Languages for Querying Data. Technical Report CS-93-58, Department of Computer Science, Brown University, Providence, RI, USA.

Cruz, I. F. (1993c). Visual Querying of Deductive and Object-Oriented Databases. In *(Mumick, 1993)*, pages 149–150. Abstract.

Dalal, M. and Gangopadhyay, D. (1990). OOLP: A Translation Approach to Object-Oriented Logic Programming. In *(Kim et al., 1990)*, pages 593–606.

Das, S. K. (1992). *Deductive Databases and Logic Programming.* Addison-Wesley Publ. Co., Wokingham, England, UK. ISBN 0-201-56897-7.

Date, C. J. (1990). *An Introduction to Database Systems*, volume 1. Addison-Wesley Publ. Co., Reading, MA, USA., 5th edition. ISBN 0-201-52878-9.

Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M., Livny, M., and Jauhari, R. (1988). The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):51–70.

Dechamboux, P. and Roncancio, C. (1994a). Integrating Deductive Capabilities into an Object-Oriented Database Programming Language. Manuscript. Presented at *10eme Journées Bases de Données Avancées*, September 1994.

Dechamboux, P. and Roncancio, C. (1994b). Peplom^d: An Object-Oriented Database Programming Language Extended with Deductive Capabilities. In *(Karagiannis, 1994)*, pages 2–14.

Deliyanni, A. and Kowalski, R. A. (1979). Logic and Semantic Networks. *Communications of the ACM*, 22(3):184–192.

Delobel, C., Kifer, M., and Masunaga, Y., editors (1991). *Deductive and Object-Oriented Databases (Second International Conference DOOD'91, Munich)*, Berlin, Germany. Springer-Verlag. ISBN 3-540-55015-1, ISBN 0-387-55015-1.

Derr, M. A., Morishita, S., and Phipps, G. (1993). Design and Implementation of the Glue-Nail Database System. In *(Buneman and Jajodia, 1993)*, pages 147–156.

Derr, M. A., Morishita, S., and Phipps, G. (1994). The Glue-Nail Deductive Database System: Design, Implementation and Evaluation. *The VLDB Journal*, 3:123–160.

Díaz, O., Paton, N. W., and Gray, P. M. (1991). Rule Management in Object-Oriented Databases: A Uniform Approach. In Lohman, G. M., Sernadas, A., and Camps, R., editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 317–326, Barcelona, Spain. Morgan Kaufmann Publishers.

Dinn, A., Paton, N. W., Williams, M. H., Fernandes, A. A., and Barja, M. L. (1995). The Implementation of a Deductive Query Language Over an Object-Oriented Database. To appear in *Proceedings DOOD'95*.

Dobbie, G. and Topor, R. (1993). A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems. In *(Ceri et al., 1993)*, pages 473–488.

Fernandes, A. A., Barja, M. L., Paton, N. W., and Williams, M. H. (1993). A Deductive Object-Oriented Database for Data Intensive Application Development. In Worboys, M. and Grundy, A., editors, *Advances in Databases: Proceedings of the 11th British National Conference on Databases (BNCOD 11)*, LNCS 696, pages 176–198, Keele, England, UK. Springer-Verlag. ISBN 3-540-56921-9, ISBN 0-387-56921-9.

Fernandes, A. A., Barja, M. L., Paton, N. W., Williams, M. H., Dinn, A., Diaz, I. P., and Abdelmoty, A. I. (1994a). ROCK & ROLL: A Deductive Object-Oriented Database System. In *(Olivé, 1994)*, pages 25–32.

Fernandes, A. A., Paton, N. W., and Williams, M. H. (1994b). A Logical Query Language for an Object-Oriented Data Model. In *(Paton and Williams, 1994)*, pages 234–250.

Fernandes, A. A., Williams, M. H., and Paton, N. W. (1992a). A Formal Abstract Definition of Objects as a Data Modelling Primitive. Technical Report TR92003, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland, UK. Revised June 1992.

Fernandes, A. A., Williams, M. H., Paton, N. W., Barja, M. L., and Dinn, A. (1994c). Object-Oriented Database Programming Languages Founded on an Axiomatic Theory of Objects. Workshop on Logical Foundations of Object-Oriented Programming (8th European Conference on Object-Oriented Programming - ECOOP 94).

Fernandes, A. A. A., Paton, N. W., Williams, M. H., and Bowles, A. (1992b). Approaches to Deductive Object-Oriented Databases. *Information and Software Technology*, 34(12):787–803.

Fong, E., Kent, W., Moore, K., and Thompson, C. (1991). X3/SPARC/DBSSG/OODBTG Final Report.

Forgy, C. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37.

Gallaire, H. and Minker, J., editors (1978). *Logic and Databases.* Plenum Press, New York, NY, USA. Symposium at Toulouse, November 1977, ISBN 0-306-40060-X.

Gallaire, H., Minker, J., and Nicolas, J.-M. (1984). Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–185.

Garcia-Molina, H. and Jagadish, H., editors (1990). *Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data*, Atlantic City, NJ, USA. ACM Press. SIGMOD Record, 19(2), June 1990.

Gatziu, S. and Dittrich, K. R. (1994). Events in an Active Object-Oriented Database System. In *(Paton and Williams, 1994)*, pages 23–39.

Gatziu, S., Geppert, A., and Dittrich, K. R. (1991). Integrating Active Concepts into an Object-Oriented Database Systems. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data*, pages 399–415, Nafplion, Greece.

Gehani, N., Jagadish, H., and Shmueli, O. (1992). Event Specification in an Active Object-Oriented Database. In *(Stonebraker, 1992)*, pages 81–90. Also available as AT&T Technical Report att-db-tr-92-6.

Geppert, A., Dittrich, K. R., Goebel, V., and Scherrer, S. (1993). The NO^2 Data Model. Technical Report TR 93.09, Institut für Informatik, Universität Zürich, Zürich, Switzerland.

Goñi, M. L. B. (1995). *A Database Programming Language for a Deductive Object-Oriented Database*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, UK.

Graham, I. (1994). *Object-Oriented Methods*. Addison-Wesley Publishing Company, Wokingham, England, UK., 2nd edition. ISBN 0-201-59371-8.

Grant, J. and Minker, J. (1989). Deductive Database Theories. *The Knowledge Engineering Review*, 4(4):267–304.

Gray, P. M., Kulkarni, K. G., and Paton, N. W. (1992). *Object-Oriented Databases: A Semantic Data Model Approach*. Prentice Hall International(UK) Ltd., Hertfordshire, England, UK. ISBN 0-13-630203-3.

Greco, S., Leone, N., and Rullo, P. (1992). COMPLEX: An Object-Oriented Logic Programming System. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344–359.

Hanson, E. N. (1989). An Initial Report on the Design of Ariel: A DBMS with an Integrated Production Rule System. *ACM SIGMOD Record*, 18(3):12–19.

Hanson, E. N. and Widom, J. (1993). An Overview of Production Rules in Database Systems. *The Knowledge Engineering Review*, 8(2):121–143.

Harrison, J. V. and Dietrich, S. W. (1994). Integrating Active and Deductive Rules. In *(Paton and Williams, 1994)*, pages 289–305.

Heuer, A. and Sander, P. (1990). Semantics and Evaluation of Rules over Complex Objects. In *(Kim et al., 1990)*, pages 473–492.

Heuer, A. and Sander, P. (1991). Preserving and Generating Objects in the LIVING IN A LATTICE Rule Language. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 562–569, Kobe, Japan. IEEE Computer Society Press. ISBN 0-8186-2138-9.

Heuer, A. and Sander, P. (1992). The LIVING IN A LATTICE Rule Language. *Data & Knowledge Engineering*, 9:249–286.

Hughes, J. G. (1991). *Object-Oriented Databases*. Prentice Hall International(UK) Ltd., Hertfordshire, England, UK. ISBN 0-13-629874-5.

Hulin, G., Pirotte, A., Roelants, D., and Vauclair, M. (1989). Logic and Databases. In *(Thayse, 1989)*, chapter 6, pages 279–350.

Hunter, G. (1971). *Metalogic: An Introduction to the Metatheory of Standard First Order Logic.* University of California Press, Berkeley and Los Angeles, CA, USA. ISBN 0-520-01822-2, 0-520-02356-0.

Itô, K., editor (1987). *Encyclopedic Dictionary of Mathematics.* The MIT Press, Cambridge, MA, USA. ISBN 0-262-09026-0.

Jamil, M. H. and Lakshmanan, L. V. (1993). Realizing Orlog in $\mathcal{LDL}$. In *(Mumick, 1993)*, pages 45–59.

Jarke, M., Eherer, S., Gallersdoerfer, R., Jeusfeld, M. A., and Staudt, M. (1993). Concept-Base - A Deductive Object Base Manager. Technical Report 93-14, Aachener Informatik-Berichte/RWTH Aachen, Aachen, Germany.

Jarke, M., Jeusfeld, M., and Rose, T. (1990). Software Process Modelling as a Strategy for KBMS Implementation. In *(Kim et al., 1990)*, pages 531–550.

Jarke, M. and Staudt, M. (1993). An Application Perspective to Deductive Object Bases. In *(Mumick, 1993)*, pages 17–29. Extended Abstract.

Kanellakis, P. C. (1990). Elements of Relational Database Theory. In *(van Leeuwen, 1990)*, chapter 17, pages 1073–1156.

Karagiannis, D., editor (1994). *Database and Expert System Applications - 5th International Conference, DEXA '94, Proceedings*, LNCS 856, Athens, Greece. Springer-Verlag. ISBN 3-540-58435-8.

Kemper, A., Moerkotte, G., and Peithner, K. (1992). Object-Orientation Axiomatized by Dynamic Logic. Technical Report 92-30, Karlsruhe/RWTH Aachen, Aachen, Germany.

Kiernan, G., de Maindreville, C., and Simon, E. (1990). Making Deductive Database a Practical Technology: A Step Forward. In *(Garcia-Molina and Jagadish, 1990)*, pages 237–246.

Kifer, M., Lausen, G., and Wu, J. (1993). Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 93/06, Department of Computer Science, State University of New York at Stony Brook (SUNY), Stony Brook, NY, USA. Revised June 1993.

Kifer, M. and Lozinskii, E. (1990). On Compile-Time Query Optimization In Deductive Databases By Means Of Static Filtering. *ACM Transactions on Database Systems*, 15(3):385–426.

Kifer, M. and Wu, J. (1989a). A Logic for Object-Oriented Logic Programming (Maier's O-Logic: Revisited). In *(Chandra, 1989)*, pages 379–383. Preliminary Report of (Kifer and Wu, 1989b).

Kifer, M. and Wu, J. (1989b). A Logic for Programming with Complex Objects. Technical report, Department of Computer Science, State University of New York at Stony Brook (SUNY), Stony Brook, NY, USA. Extended Version of (Kifer and Wu, 1989a).

Kim, W., editor (1995). *Modern Database Systems: The Object Model, Interoperability and Beyond.* ACM Press/Addison-Wesley Publ. Co., New York, NY, USA. ISBN O-201-59098-0.

Kim, W. and Kelley, W. (1995). On View Support in Object-Oriented Database Systems. In *(Kim, 1995)*, chapter 6, pages 108–129.

Kim, W., Nicolas, J.-M., and Nishio, S., editors (1990). *Deductive and Object-Oriented Databases (First International Conference DOOD'89, Kyoto)*, Amsterdam, The Netherlands. Elsevier Science Press (North-Holland). ISBN 0-444-88433-5.

Kowalski, R. (1979a). *Logic for Problem Solving.* Artificial Intelligence Series (The Computer Science Library). Elsevier North-Holland, Inc., New York, NY, USA. ISBN 0-444-00365-7.

Kowalski, R. (1992). Database Updates in the Event Calculus. *The Journal of Logic Programming*, 1992(1–2):121–146.

Kowalski, R. and Sergot, M. (1986). A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95.

Kowalski, R. A. (1979b). Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436.

Kozen, D. and Tiuryn, J. (1990). Logics of Programs. In *(van Leeuwen, 1990)*, chapter 14, pages 789–840.

Lanzelotte, R. and Valduriez, P. (1992). Optimization of object-oriented recursive queries using cost controlled strategies. In *(Stonebraker, 1992)*, pages 256–265.

Lécluse, C., Richard, P., and Velez, F. (1988). O_2, an Object-Oriented Data Model. In *(Zdonik and Maier, 1990b)*, pages 227–236.

Lloyd, J. W. (1987). *Foundations of Logic Programming.* Springer Series: Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, Germany., 2nd., extended edition. ISBN 3-540-18199-7, 0-387-18199-7.

Lloyd, J. W. and Topor, R. W. (1984). Making Prolog More Expressive. *The Journal of Logic Programming*, 1984(3):225–240.

Lohman, G. M., Lindsay, B., Pirahesh, H., and Schiefer, K. B. (1991). Extensions to Starburst: Objects, Types, Functions and Rules. *Communications of the ACM*, 34(10):94–109.

Lou, Y. and Ozsoyoglu, Z. M. (1991). LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance. In *(Clifford and King, 1991)*, pages 198–207.

Loveland, D. W. (1991). Near-Horn Prolog and Beyond. *Journal of Automated Reasoning*, 7:1–26.

Maier, D. (1986). A Logic for Objects. Technical Report CS/E-86-012, Oregon Graduate Center. Also in *Preprints of the Workshop on Foundations of Deductive Database and Logic Programming, Washington, DC, USA. August 1986.*

Maier, D. and Stein, J. (1987). Development and Implementation of an Object-Oriented DBMS. In *(Zdonik and Maier, 1990b)*, pages 167–185.

McCabe, F. G. (1992). *Logic and Objects.* Prentice Hall International(UK) Ltd., Hertfordshire, England, UK. ISBN 0-13-536079-X.

Mendelson, E. (1987). *Introduction to Mathematical Logic.* Mathematics Series. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, USA., 3rd. edition. ISBN 0-534-06624-0.

Meseguer, J. and Qian, X. (1993). A Logical Semantics for Object-Oriented Databases. In *(Buneman and Jajodia, 1993)*, pages 89–98.

Minker, J., editor (1988). *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA. ISBN O-934613-40-0.

Miranker, D. (1987). TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of AAAI*, pages 42–47.

Montesi, D. (1993). *A Model for Updates and Transactions in Deductive Databases.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy. Available as Technical Report TD-20/93.

Mumick, I. S., editor (1993). *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, Washington, DC, USA.

Mumick, I. S. and Ross, K. A. (1993a). Noodle: A Language for Declarative Querying in an Object-Oriented Database. In *(Ceri et al., 1993)*, pages 360–377.

Mumick, I. S. and Ross, K. A. (1993b). The Influence of Class Hierarchy Choice on Query Language Design. In *(Mumick, 1993)*, pages 152–154.

Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. (1990). Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362.

Naqvi, S. A. and Tsur, S. (1989). *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, USA. ISBN 0-7167-8200-6.

O. Deux et al. (1991). The O_2 System. *Communications of the ACM*, 34(10):35–48.

Olivé, A., editor (1994). *Fifth International Workshop on the Deductive Approach to Information Systems and Databases*, Aiguablava, Spain. Report de Recerca LSI/94-28-R, Universitate Politècnica de Catalunya.

Omicini, A. and Natali, A. (1994). Object-Oriented Computations in Logic Programming. In *(Tokoro and Pareschi, 1994)*, pages 194–212.

Paton, N. W. (1989). ADAM: An Object-Oriented Database System Implemented in Prolog. In Williams, M. H., editor, *Proceedings of the Seventh British National Conference on Databases (BNCOD 7)*, pages 147–161, Edinburgh, Scotland, UK. Cambridge University Press.

Paton, N. W. and Abdelmoty, A. I. (1993). An Object Store for the DOOD Object Model. Internal Report.

Paton, N. W., Díaz, O., Williams, M. H., Campin, J., Dinn, A., and Jaime, A. (1994). Dimensions of Active Behaviour. In *(Paton and Williams, 1994)*, pages 40–57.

Paton, N. W. and Williams, M. H., editors (1994). *Rules in Database Systems: Proceedings of the 1st International Workshop, Edinburgh, Scotland (30 August–1 September 1993)*, Workshops in Computing Series, London, England, UK. Springer-Verlag. ISBN 3-540-19846-6, 0-387-19846-6.

Peckham, J. and Maryanski, F. (1988). Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189.

Phipps, G., Derr, M. A., and Ross, K. A. (1991). Glue-Nail: A Deductive Database System. In *(Clifford and King, 1991)*, pages 308–317.

Plaisted, D. A. (1988). Non-Horn Clause Logic Programming Without Contrapositives. *Journal of Automated Reasoning*, 4:287–325.

Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P. (1993). Implementation of the CORAL Deductive Database System. In *(Buneman and Jajodia, 1993)*, pages 167–176.

Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P. (1994). The CORAL Deductive System. *The VLDB Journal*, 3:161–210.

Ramakrishnan, R. and Ullman, J. D. (1993). A Survey of Research on Deductive Database Systems. Technical report, Department of Computer Science, Stanford University.

Reiter, R. (1984). Towards a Logical Reconstruction of Relational Database Theory. In Brodie, M. L., Mylopoulos, J., and Schmidt, J. W., editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Topics in Information System Series, pages 191–233. Springer-Verlag, New York, NY, USA. ISBN 0-387-90842-0, 3-540-90842-0.

Richardson, J. E. and Carey, M. J. (1989). Implementing Persistence in E. In Rosenberg, J. and Koch, D., editors, *Persistent Object Systems: Proceedings of the Third International Workshop*, pages 175–199, London, England, UK. Springer-Verlag.

Richardson, J. E., Carey, M. J., and Schuh, D. T. (1993). The Design of the E Programming Language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534.

Schmidt, J. W., Ceri, S., and Missikoff, M., editors (1988). *Advances in Database Technology - EDBT'88, International Conference on Extending Database Technology*, LNCS 303, Venice, Italy. Springer-Verlag. ISBN 3-540-19074-0, ISBN 0-387-19074-0.

Scholl, M., Laasch, C., Rich, C., Schek, H.-J., and Tresch, M. (1992). The COCOON Object Model. Technical Report TR 192, Department of Computer Science, ETH Zurich, Zürich, Switzerland.

Soley, R. M. and Kent, W. (1995). The OMG Object Model. In *(Kim, 1995)*, chapter 2, pages 18–41.

Sripada, S. (1988). A Logical Framework for Temporal Deductive Databases. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 171–182, Los Angeles, CA, USA. Morgan Kaufmann Publishers.

Srivastava, D., Ramakrishnan, R., Seshadri, P., and Sudarshan, S. (1993). Coral++: Adding Object-Orientation to a Logic Database Language. In Agrawal, R., Baker, S., and Bell, D., editors, *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 158–170, Dublin, Ireland. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-152-X.

Stickel, M. E. (1988). A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4:353–380.

Stoll, R. R. (1979). *Set Theory and Logic.* Dover Publications, Inc., New York, NY, USA. ISBN 0-486-63829-4.

Stonebraker, M., editor (1992). *Proceedings of the 1992 ACM SIGMOD International Conference on the Management of Data*, San Diego, CA, USA. ACM Press. SIGMOD Record 21(2).

Straube, D. D. (1990). Queries and Query Processing in Object-Oriented Database Systems. Technical Report TR90-33, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Su, S. Y., Guo, M., and Lam, H. (1993). Association Algebra: A Mathematical Foundation for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):775–798.

Thayse, A., editor (1988). *From Standard Logic to Logic Programming: Introducing a Logic Based Approach to Artificial Intelligence.* John Wiley & Sons, Chichester, England, UK. ISBN 0-471-91838-5, Contributors: André Thayse, Paul Gochet, Eric Grégoire, Pascal Gribomont, Georges Louis, Eduardo Sanchez, Dominique Snyers, Pierre Wodon and Philippe Delsarte.

Thayse, A., editor (1989). *From Modal Logic to Deductive Databases:Introducing a Logic Based Approach to Artificial Intelligence.* John Wiley & Sons, Chichester, England, UK. ISBN 0-471-92345-1, Contributors: André Thayse, Paul Gochet, Eric Grégoire, Pascal Gribomont, Guy Hulin, Alain Pirotte, Dominique Roelants, Dominique Snyers, Marc Vauclair, Pierre Wolper, and Philippe Delsarte.

Tokoro, M. and Pareschi, R., editors (1994). *Object-Oriented Programming – 8th European Conference, ECOOP '94*, LNCS 821, Bologna, Italy. Springer-Verlag. ISBN 3-540-58202-9, ISBN 0-387-58202-9.

Ullman, J. D. (1988). *Principles of Database and Knowledge-Base Systems*, volume 1 of *Principles of Computer Science Series.* Computer Science Press, Rockville, MD, USA. ISBN 0-716-78158-1.

Ullman, J. D. (1991). A Comparison Between Deductive and Object-Oriented Database Systems. In *(Delobel et al., 1991)*, pages 263–277.

Vaghani, J., Ramamohanarao, K., Kemp, D. B., Somogyi, Z., Stuckey, P. J., Leask, T. S., and Harland, J. (1994). The ADITI Deductive Database System. *The VLDB Journal*, 3:245–288.

van der Voort, M. and Kersten, M. (1991). Facets of Database Triggers. Technical Report CWI-CS-R9122, CWI, Amsterdam, The Netherlands. Revised, April 1993.

van Emden, M. H. and Kowalski, R. A. (1976). The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742.

van Leeuwen, J., editor (1990). *Handbook of Theoretical Computer Science – Volume B: Formal Models and Semantics.* Elsevier Science Publishers B.V./The MIT Press, Amsterdam, The Netherlands./Cambridge, MA, USA. ISBN O-444-88074-7, 0-262-22039-3.

Widom, J. (1994). Deductive and Active Databases: Two Paradigms or Ends of a Spectrum? In *(Paton and Williams, 1994)*, pages 306–315.

Williams, M. H., Paton, N. W., Abdelmoty, A. I., Fernandes, A. A., and Barja, M. L. (1993). Applying Advanced Database Techniques to Geographic Information Systems. In M.J. Cardwell et al., editor, *JFIT Technical Conference Digest*, pages 221–228, University of Keele. DTI/SERC.

Wos, L., Overbeck, R., Lusk, E., and Boyle, J. (1984). *Automated Reasoning: Introduction and Applications.* Prentice-Hall, Inc., Englewood Cliffs, NJ, USA. ISBN O-13-054446-9.

Ye, X., Parent, C., and Spaccapietra, S. (1994). Derived Objects and Classes in DOOD Systems. In *(Olivé, 1994)*, pages 79–94.

Yokota, K., Tsuda, H., and Morita, Y. (1993). Specific Features of a Deductive Object-Oriented Database Language $\mathcal{QUIXOTE}$. In *(Mumick, 1993)*, pages 89–99.

Zaniolo, C. (1990). Object Identity and Inheritance in Deductive Databases – An Evolutionary Approach. In *(Kim et al., 1990)*, pages 7–24.

Zaniolo, C. (1994). A Unified Semantics for Active and Deductive Databases. In *(Paton and Williams, 1994)*, pages 271–287.

Zaniolo, C., Arni, N., and Ong, K. (1993). Negation and Aggregates in Recursive Rules: The $LDL++$ Approach. In *(Ceri et al., 1993)*, pages 204–221.

Zdonik, S. B. and Maier, D. (1990a). Fundamentals of Object-Oriented Databases. In *(Zdonik and Maier, 1990b)*, pages 1–32.

Zdonik, S. B. and Maier, D., editors (1990b). *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA. ISSN 1046-1698, ISBN 1-55860-000-0.