in: U.W. Lipeck, G. Vossen (eds.), Proc. Workshop "Formale Grundlagen für den Entwurf von Informationssystemen", Informatik-Bericht 03/94, Universität Hannover, pp. 197–206, 1994.

An Elementary Logic for Object Specification and Verification *

Stefan Conrad & Hans-Dieter Ehrich

Technische Universität Braunschweig, Informatik, Abt. Datenbanken

Postfach 3329, D–38023 Braunschweig, Germany

e–mail: {conrad|ehrich}@idb.cs.tu-bs.de

Abstract

In the present paper we sketch an elementary logic for specifying information systems as systems of dynamic objects. First, we informally present the ingredients which should be included in a logic for abstract object specifications. Afterwards we turn to a calculus which realizes a basic part of such a logic. For this calculus a simple proof system is provided which permits to gain experience in verifying object properties. We conclude by briefly pointing out future work.

1 Introduction

For the development of reliable information systems formal specification is indispensable. Based on a formal specification crucial properties of the objects in an information system can be proved as early as possible. After having convinced oneself that the specification of the information system properly describes the intended behavior, a formal and provably correct implementation should be carried out.

Currently, we have reached a stage where the theoretical foundations of object specification for information systems are well-understood, e.g., see [SSE87, EDS93]. Several object specification languages originate in this theoretical work, for instance `TROLL` [JSHS91], OBLOG [SSG+91], and TROLL *light* [CGH92, GCH93]. Moreover, it turned out that an adequate logic is also necessary. Therefore, a lot of effort has been made in that direction.

Several variants of object calculi have emerged of which some are based on deontic concepts [FM91a, FM91b]. A particular feature of the *Object Specification Logic* OSL [SSC92] is the treatment of inheritance among objects and of aggregation of objects.

*Work reported here has been partly supported by the German Ministry for Research and Technology (BMFT) under Grant No. 01 IS 203 D (KorSo project) and by the CEC under Grant No. 6071 (IS-CORE) and under Grant No. 6112 (COMPASS).

These approaches are specification oriented, i.e., they offer expressive concepts for easy specification. Verification calculi, on the other hand, tend to be more elementary in order to make the proof system small and workable. We have been working on a proof system for TROLL *light* [Con94a, Con94b]. In this paper, we combine the advantages of specification and verification calculi by giving an elementary but powerful enough specification logic, together with a close to complete proof system for verification. A distinguishing feature of our logic is that all interpretations are state-independent.

In particular, it is more and more important to cope with normative and non-normative behavior of objects in specifying information systems. This is due to the fact that integrity constraints may temporarily be violated, for instance, during transactions. But there is always the obligation that all integrity constraints must be fulfilled after committing a transaction. It is obvious that deontic logic can help a lot (cf. [WM93]).

The reminder of this paper is structured as follows. First, we give an overview of ingredients of a logic for abstractly specifying dynamic objects. Afterwards, we present a calculus for verification purposes which provides a first realization of essential parts of such a logic. We conclude by pointing out future work.

2 Abstract Template Specification

Templates for classes and objects are the basic specification units in the `TROLL` languages [JSHS91, CGH92]. Templates describe structure and behavior of some kind of objects. We employ an abstract notation using equational logic for data and deontic logic for actions. This notation is closer to logic notation than to `TROLL` notation, it might serve for semantic description of the latter.

Example 2.1: Variables in imperative programming languages are objects keeping values of some data sort `d`. Variables can be created (`cr`) and killed (`kl`), and they allow for writing (`w(x)`) and reading (`r(x)`) values of sort `d`.

```
template VAR[d]
    data sort d
    actions  r(d), w(d), cr, kl
    vars     x, y : d
    axioms   [cr](▷w(x) ∧ ¬▷cr ∧ ▷kl)
             [w(x)]▷r(x)
             [kl](¬▷r(x) ∧ ¬▷w(x) ∧ ¬▷kl)
             ▷r(x) ∧ ▷r(y) ⇒ x = y
             ...
end VAR[d]
```

For each action α, $[\alpha]\varphi$ means that φ holds after α, i.e., in the next state after α occurred. $\triangleright\alpha$ means that α is permitted.

Thus, the above specification says that variables of sort `d` can be created and killed, and values of sort `d` can be read and written. After creation, any value of sort `d` can be written. The variable cannot be created again but it can be killed. After writing a value `x`, reading this value `x` is permitted (the variable "has value `x`"). After killing, no action is permitted any more. The fourth axiom says that a variable has at most one value. The dots indicate further axioms saying that actions do not change any state information unless explicitly specified. Such axioms are usually captured by "frame rules". □

In general, abstract template specifications have *components* in addition to actions. Components are named subtemplates that are specified elsewhere.

Example 2.2: Widely used standard components are *attributes* which are named variables keeping state information.

```
template PERSON
   data sort nat, text, ...
   comps     name: VAR[nat]
             age: VAR[0:127]
             father: VAR[id[PERSON]]
             ...
   actions   birthday, ...
   vars      n: nat
   axioms    birthday ⇒ age.w(age + 1)
             birthday ∧ ▷age.r(x) ⇒ age.r(x)
             ...
end PERSON
```

The `father` attribute has identities of sort `PERSON` as its data type. An action α in a formula denotes the proposition "α occurs". This holds, for instance, for `age.w(...)` in the first axiom. Using `age` as a term, however, as in ... `age + 1`..., is shortcut for $(\iota$ `x)age.r(x)` ("that x such that ..."), employing the particularization operator ι. Thus, evaluating an attribute always means that it is explicitly read. The second axiom states that the age is indeed read on a birthday, provided reading is permitted. According to these axioms, a film star disallowing reading her age will get an undefined age on her next birthday. □

Formally, let $D = (S_D, \Omega, E)$ be a data specification with sorts S_D, operators Ω, and axioms E. We may assume that S_D is partially ordered, but we do not need this feature here. Ω is an $S_D^* \times S_D$-indexed set family of operator symbols $\omega : s_1 \times \ldots \times s_n \to s_0$. For the axioms E, we employ some usual elementary logic, say, equational or conditional equational.

Let S_O be a set of *object sorts* disjoint from S_D. We may assume S_O to be partially ordered in order to express specialization inheritance, but we do not go into this issue here.

For each data sort $d \,\epsilon\, S_D$, we assume an object sort $\mathtt{VAR}[d] \,\epsilon\, S_O$ (cf. example 2.1). Conversely, for each object sort $b \,\epsilon\, S_O$, we assume two data sorts $\mathtt{id}[b], \mathtt{ac}[b] \epsilon\, S_D$, denoting the identifier and action symbols, respectively, for objects of sort b.

An object community is specified by a *schema* $Sch = (Sg, \Phi)$ consisting of a *schema signature* $Sg = (A, C)$ and a set of *schema axioms* Φ. A schema signature contains the actions A and components C in the system. A is an $S_O \times S_D^*$-indexed set family of *action generators* $\alpha : \mathtt{id}[b] \times s_1 \times \ldots \times s_n \to \mathtt{ac}[b]$. C is an $S_O \times S_D^* \times S_O$-indexed set family of *component generators* $\gamma : \mathtt{id}[b] \times s_1 \times \ldots \times s_n \to \mathtt{id}[c]$.

In object-oriented notation, a schema is grouped around object sorts. Formally, the action and component generators with the same first argument sort are grouped together, along with corresponding axioms. This way, we obtain a template $T[b] = (Sg[b], \Phi[b])$ for each object sort $b \,\epsilon\, S_O$, consisting of a template signature $Sg[b] = (A[b], C[b])$ and axioms $\Phi[b]$. $A[b]$ is the S_D^*-indexed set family of those action generators in A having $\mathtt{id}[b]$ as its first argument sort (and $\mathtt{ac}[b]$ as its range sort). $C[b]$ is the $S_D^* \times S_O$-indexed set family of those component generators in C having $\mathtt{id}[b]$ as its first argument sort.

It is common object-oriented practice to honor the special role of the first argument by a special prefix notation. For instance, we write `Jim.birthday` instead of `birthday(Jim)`. It is also common to omit corresponding variables since the meaning is clear from locality. For instance, we write `birthday`, `age`, … instead of `p.birthday`, `p.age`, … with a variable `p` of sort `id[PERSON]`.

Extending the data specification D by the schema signature Sch, adding appropriate identity constants and maybe some mechanism for generating new identities, and interpreting this extension freely, we obtain, for each object sort b, data sets $[\![\mathtt{id}[b]]\!]$ of identities and $[\![\mathtt{ac}[b]]\!]$ of actions.

Example 2.3: Referring to example 2.2, if `Jim` is an object identity of sort `PERSON`, so are `Jim.father`, `Jim.father.father`, etc. `Jim.birthday` is an action of sort $\mathtt{ac}[\mathtt{PERSON}]$, and so are `Jim.father.birthday`, `Jim.father.father.birthday`, etc. □

The axioms $\Phi[b]$ for objects of sort b should describe the latter's intended behavior as a generic process. We employ DAL, our variant of deontic action logic [FM91b, FM91a]. DAL is more elementary than the temporal logic used in [Ju93] for describing `TROLL` semantics.

Let $A = \{\alpha, \ldots\}$ be a set of actions. DAL extends the underlying data logic by formulae

$$\alpha\,,\ \triangleright\alpha\,,\ \diamond\alpha\,,\ [\alpha]\varphi$$

expressing that α occurs, α is permitted, α is obliged, and φ holds after α, respectively, where φ is an arbitrary DAL formula. As shown in example 2.2, we include the particularization operator ι in our logic. We note in passing that DAL has to be extended to include a scoping predicate $\surd\alpha$ in order to cope with dynamic roles [EJD93].

DAL is powerful enough to give formal semantics to languages like those of the `TROLL` family.

The semantics $[\![T[b]]\!]$ of template specifications $T[b] = (Sg[b], \Phi[b])$ is a process describing the dynamic behavior pattern of objects of sort b. An elementary but appropriate process model is the *life cycle* model: a process is a set $\Lambda = \{\lambda_1, \lambda_2, \ldots\}$ of life cycles where each life cycle is a finite or infinite sequence $\lambda = (\sigma_0, \sigma_1, \sigma_2, \ldots)$ of *situations*. Each situation is a state/step pair $\sigma_i = (z_i, t_i)$, $i \,\epsilon\, \{0, 1, 2, \ldots\}$, where a state $z_i = (Per_i, Obl_i)$ consists of two sets of actions, those permitted and those obliged, and a step is a set $t_i \subseteq Per_i$ of (simultaneous) actions.

For a life cycle $\lambda = (\sigma_0, \sigma_1, \sigma_2, \ldots)$, let λ^i be its rest sequence from position i on, i.e., $\lambda^i = (\sigma_i, \sigma_{i+1}, \ldots)$.

Satisfaction of DAL formulas can be defined in two steps, first defining satisfaction at the beginning, i.e., in the first situation of the life cycle.

$$
\begin{array}{lll}
\lambda \models_0 \alpha & \text{iff} & \alpha \,\epsilon\, t_0 \\
\lambda \models_0 \triangleright\alpha & \text{iff} & \alpha \,\epsilon\, Per_0 \\
\lambda \models_0 \diamond\alpha & \text{iff} & \alpha \,\epsilon\, Obl_0 \ \text{ and } \ \lambda^i \models_0 \alpha \ \text{ for some } \ i \geq 0 \\
\lambda \models_0 [\alpha]\varphi & \text{iff} & \text{whenever } \ \alpha \,\epsilon\, t_0\,, \ \text{ then } \ \lambda^1 \models_0 \varphi \\
\ldots & &
\end{array}
$$

Satisfaction of DAL formulae in an entire life cycle and in a set of life cycles is obvious:

$$
\begin{array}{lll}
\lambda \models \varphi & \text{iff} & \lambda^i \models_0 \varphi \ \text{ for each } \ i \geq 0 \\
\Lambda \models \varphi & \text{iff} & \lambda \models [0.3ex]\varphi \ \text{ for each } \ \lambda \,\epsilon\, \Lambda
\end{array}
$$

Processes are sets, so they are partially ordered by set inclusion. Reading inclusions as morphisms, the following proposition is obvious.

Proposition 2.4: *Each template specification has a final model.*

This final model can serve as a standard semantics, it displays the most liberal behavior allowed by the axioms: *everything is allowed unless stated otherwise.* There may be cases, however, where loose semantics is more appropriate, i.e., where it is preferable to accept *some* permissible behavior.

3 Basic Calculus and Proof System

In the sequel we briefly introduce a basic calculus as a first realization of ideas presented above. This calculus was essentially developed for the object specification language TROLL *light*. Hence, it mainly covers the concepts given in that language. Nevertheless, we have made sure from the beginning that this calculus can play an ideal starting point for the development of a more comprehensive calculus for verifying object specifications. In order to speed up availability for practical use, formulas have a very simple structure in this calculus (i.e., they are Gentzen clauses). Therefore, an elementary proof system can easily be determined by adopting inference rules of classical sequent calculi (for instance, cf. [Gal93]). This facilitates a quick implementation by means of a generic theorem prover like ISABELLE [Pau90].

First, we informally present the syntactical concepts of our calculus. A more comprehensive and formal treatment of this calculus is given in [Con94a].

Because we only allow a restricted form of formulas in the basic calculus we have to introduce *propositions* as an own syntactical category. Propositions may be built as follows:

$$\alpha, \qquad \triangleright\alpha, \qquad \neg P, \qquad [\alpha]P$$

where α is a term denoting an action and P is a proposition. In the basic version of our calculus we have omitted the $\diamond$ operator. However, it can be introduced quite easily as an extension.

Next, we can construct formulas in our calculus as Gentzen formulas resp. Gentzen clauses over propositions. Thus, if $P_1, \ldots, P_n, Q_1, \ldots, Q_m$ are propositions then

$$P_1, \ldots, P_n \ \rightarrow \ Q_1, \ldots, Q_m$$

is a formula which has to be understood in the usual way: provided P_1 and ... and P_n are satisfied at an instant of time (resp. at same state of the object community) then Q_1 or ... or Q_m must also be satisfied at that instant. We assume $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_m$ to be sets of propositions such that duplicates do not occur and the order in which the propositions are written does not matter. Of course, we allow each of these sets of propositions to be empty.

Example 3.1: Assuming an object specification for author objects with attributes and actions to be given, the following formulas describe properties of author objects:

(1) $\rightarrow \ [A.changeName(\text{“}\texttt{Codd}\text{”})]A.Name = \text{“}\texttt{Codd}\text{”},$

(2) $\neg \triangleright A.changeName(N) \ \rightarrow \ [A.publishBook(T)]\neg \triangleright A.changeName(N).$

The first formula states that in each state resp. at each instant of time it is true that after the occurrence of an action *changeName*(“Codd”) for some author object A the attribute *Name* of that object has the value “Codd”. That this is true at each instant is due to the fact that no proposition is given as a precondition on the left-hand side. Please note that $A.Name =$ “Codd” is here used as a shortcut for $\triangleright A.Name.r(\text{“}\texttt{Codd}\text{”})$.

Formula (2) describes the property that provided the action $changeName(N)$ is not permitted for some author object A at some moment then this action is also not permitted for that object at the moment after the occurrence of the action $publishBook(T)$ for A. □

As we have just seen, formulas are used to describe object properties, i.e., the logical description of an object (community) is given by a set of formulas. For the specification language TROLL *light* we have developed a transformation yielding a logical description for each specification [Con94a, Con94b]. Given such a set of formulas which logically describe an object community we can derive further formulas which are entailed by this set. Therefore, we need a proof system providing inference rules for the syntactical entailment of formulas.

The inference rules for our calculus are mainly adopted from classical sequent calculi. We have only added a rule for introducing positional operators into formulas. The following basic inference rules are essentially suitable for dealing with the propositional part of our logic (please note the correspondence to [FM91a, FM91b] which is due to syntactical similarity):

$$\frac{}{P \rightarrow P}\ \textbf{(axiom)} \qquad \frac{R, P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m}{P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m, \neg R}\ \textbf{(negR)}$$

$$\frac{P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m}{R, P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m, R'}\ \textbf{(ext)} \qquad \frac{P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m, R}{\neg R, P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m}\ \textbf{(negL)}$$

$$\frac{P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m, R \qquad R, P'_1, \ldots, P'_k \rightarrow Q'_1, \ldots, Q'_l}{P'_1, \ldots, P'_k, P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m, Q'_1, \ldots, Q'_l}\ \textbf{(cut)}$$

$$\frac{P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m}{[E]P_1, \ldots, [E]P_n \rightarrow [E]Q_1, \ldots, [E]Q_m}\ \textbf{(nex)}$$

With regard to soundness, only the rule **(nex)** is crucial because the propositions P_i and Q_j and the action E in the positional operator may refer to different objects. However, we can prove that this rule does not allow syntactical entailment of formulas which are not semantically entailed, i.e., our calculus is sound:

Theorem 3.2: *The calculus is sound.*

Sketch of proof: The main point in proving this theorem is the soundness of the rule **(nex)**. For the other rules we can easily adopt soundness proofs of other sequent calculi. This is due to the fact that these rules do not involve positional operators and, therefore, we can reduce soundness of these rules to the soundness of the corresponding rules for first order predicate calculi. For the soundness of the rule **(nex)** we may argue that the truth of $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$ is sufficient for the truth of $[E]P_1, \ldots, [E]P_n \rightarrow [E]Q_1, \ldots, [E]Q_m$. This results from the fact that the introduction of the positional operator can be seen as a restriction on the set of states which have to satisfy $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$. In [Con94a] we give a more detailed and formal proof of this theorem.

Up to now, there is no final result concerning the question of completeness. Whereas a weak form of completeness may easily be reachable, we assume that strong completeness cannot be achieved. Our assumption is based on a result of Abadi [Aba89] for a large class of temporal logics. He has shown that these logics cannot be strongly complete due to the fact that they are at least as expressive as Peano arithmetics which is known to be incomplete.

Another important property of our basic calculus is compositionality with regard to theories resp. to proofs. Thereby, we may carry out proofs of object properties as local as possible. For instance, having proved a property of author objects only in the context of their specification, this proof remains valid when we extend the context, say, when we want to consider the specification of a library in which we use the specification of authors. This kind of compositionality is necessary for reusing proofs in order not to prove the same properties again and again. Thus, lemmas can be introduced which can be proved independently.

This proof system has been implemented for the generic theorem prover Isabelle [Pau90] in a way that backward proving becomes possible. Besides the rules presented above, Isabelle already provides substitution and unification for dealing with the first-order version of our calculus as well as some basic *tacticals* for putting together elementary proof steps to more complex and more comfortable proof parts.

We are now in a position that we can carry out first proofs supported by this tool. Thereby, we can gain experience in case studies in order to see which improvements are necessary. Furthermore, a platform is at our disposal for testing modifications and extensions. For instance, we can experiment with additional inference rules, temporal operators, or even with other deontic concepts [MW93a].

4 Concluding Remarks

In this paper we have sketched an elementary logic for object specification and verification. Furthermore, we have presented a basic calculus as a first prototype of essential parts of this logic. We are going to enrich this calculus by adding further concepts in order to obtain a calculus covering all the concepts which are needed for specifying information systems as systems of dynamically interacting objects.

The introduction of deontic concepts (e.g., obligations) will be one of the most important extensions. A major aim of our future work is to obtain a calculus in which all concepts of a fully-fledged object specification language like `TROLL` are included. We started with a calculus for a proper subset of those concepts (i.e., as given by TROLL *light*). Thereby, we hope to avoid a long and misleading development. Especially, the possibility to gain practical experience by working with prototype implementations of the calculus promises to provide indications of necessary improvements and modifications very early.

Abstract template specifications and their semantics are, of course, only part of the theory needed for describing semantics of languages like `TROLL`. In a forthcoming paper, we will elaborate on the semantics of template *relationships* like specialization and roles, aggregation, interfacing, etc. and, last but not least, communication. The interested reader is referred to [Ju93] and [EJD93] where essential elements of a temporal logic semantics are given.

Acknowledgements:

We gratefully acknowledge inspirations, discussions, suggestions and criticisms that we have received within the ISCORE project and the Braunschweig information systems group. Special thanks are due to Amilcar and Cristina Sernadas and their group at INESC, their influence on our work has been great. We appreciate contributions by Gunter Saake and Martin Gogolla who are in charge of the `TROLL` and `TROLL` *light* projects, respectively.

References

[Aba89] M. Abadi. The Power of Temporal Proofs. *Theoretical Computer Science*, 65(1):35–83, 1989.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[Con94a] S. Conrad. *Ein Basiskalkül für die Verifikation von Eigenschaften synchron interagierender Objekte*. PhD thesis, TU Braunschweig, 1994. To appear: Fortschritt-Berichte Reihe 10, VDI-Verlag.

[Con94b] S. Conrad. On Certification of Specifications for TROLL *light* Objects. In F. Orejas, editor, *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (WADT/Compass'92)*. Springer, LNCS, 1994.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT'93: Theory and Practice of Software Development*, pages 453–467. Springer, LNCS 668, 1993.

[EJD93] H.-D. Ehrich, R. Jungclaus, and G. Denker. Object Roles and Phases. In U. Lipeck and G. Koschorrek, editors, *Proc. of the International Workshop on Information Systems – Correctness and Reusability, IS-CORE'93*, pages 114–121. Informatik-Bericht 01/93, Univ. Hannover 1993.

[FM91a] J. Fiadeiro and T. Maibaum. Temporal Reasoning over Deontic Specifications. *Journal of Logic and Computation*, 1(3):357–395, 1991.

[FM91b] J. Fiadeiro and T. Maibaum. Towards Object Calculi. In G. Saake and A. Sernadas, editors, *Information Systems — Correctness and Reusability, Workshop IS-CORE '91, London*, pages 129–178. Informatik-Bericht 91–03, TU Braunschweig, 1991.

[Gal93] J. Gallier. Constructive Logics — Part I: A Tutorial on Proof Systems and Typed λ-Calculi. *Theoretical Computer Science*, 110(2):249–339, 1993.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf.*

Design and Implementation of Symbolic Computation Systems (DISCO'93), pages 17–32. Springer, LNCS 722, 1993.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The `TROLL` Language. Informatik-Bericht 91–04, TU Braunschweig, 1991.

[Ju93] R. Jungclaus. *Modeling of Dynamic Object Systems, a Logic-based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

[MW93a] J.-J.Ch. Meyer and R.J. Wieringa. Deontic Logic: A Concise Overview. In Meyer and Wieringa [MW93b], pages 3–16.

[MW93b] J.-J.Ch. Meyer and R.J. Wieringa, editors. *Deontic Logic in Computer Science — Normative System Specification.* Wiley, Professional Computing, 1993.

[Pau90] L.C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[SSC92] A. Sernadas, C. Sernadas, and J.F. Costa. Object Specification Logic. Internal report, INESC, University of Lisbon, 1992.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.

[SSG+91] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, and J. Gouveia. OBLOG — Object-Oriented Logic: An Informal Introduction. Technical report, INESC, Lisbon, 1991.

[WM93] R.J. Wieringa and J.-J.Ch. Meyer. Applications of Deontic Logic in Computer Science: A Concise Overview. In Meyer and Wieringa [MW93b], pages 17–40.