Inheritance Conditions for Object Life Cycle Diagrams*

Gunter Saake & Peter Hartel & Ralf Jungclaus
Abt. Datenbanken, Techn. Universität Braunschweig
Postfach 3329, D-38023 Braunschweig, FRG

Roel Wieringa & Remco Feenstra
Faculty of Mathematics and Computer Science
Free University, NL-1081 HV Amsterdam, NL

8. April 1994

Abstract

Inheritance is the main principle in object-oriented design methods to support structuring and reuse of object behaviour descriptions. Most proposals restrict the (formal) use of inheritance to method interfaces and method effect specifications. We propose to extend the inheritance relation to cover whole object life cycles, i.e. to long term object behaviour. After sketching the basic idea of inheriting object life cycles, we give inheritance conditions and inheritance-preserving construction operators for a specific graphical notation for specifying life cycles.

1 Introduction

Inheritance in object-oriented design approaches mainly focuses on inheritance of method definitions. This inheritance relation is based on a subtype relation between object types, in particular of signatures. Some proposals extend this inheritance relation on method interfaces to an inheritance relation on method effects or on method preconditions by requiring these to be strengthened as we specialised. However, most of these approaches allow at the same time overriding of method effects, which contradicts the idea of inheriting method effect specifications. To conceptually model objects, method interface definitions are not enough — to fully capture object semantics, the design framework should support the specification of all aspects of object behaviour.

Object behaviour can be specified operationally, for example by declaring a state transition automaton, or more declaratively, for example by giving some logical constraints on object behaviour. Examples of operational specifications of object life cycles are the specification of process graphs in LCM [FW93, Wie91, WF93] or of state charts

*This work was partially supported by CEC under ESPRIT WG 6071 IS-CORE II (Information Systems – COrrectness and REusability), by Deutsche Forschungsgemeinschaft under grant no. Sa 465/1-3 and OBLOG Software S.A., Lisboa.

in OMT [RBP+90]. Examples of declarative specifications are the use of temporal logic to describe life cycle properties in TROLL and Oblog [SSE87, JSHS91, Jun93, Saa93]. Several proposals for inheritance conditions for declaratively specified life cycles can be found in the literature. They are based on implication relations between specification formulas, stating that for example a new permission for a method occurence inherits another permission, if the inherited permission is implied by the new formula. This corresponds to the idea that object behaviour can be modelled by a set of method occurrence traces. These trace sets are models of logical axioms that describe object behaviour. Inheritance is then modelled by a subset relation on trace sets (after projecting out newly defined methods), and this corresponds in the specification with a strengthening of axioms. This idea is explained in more detail in Section 2.

Our aim is to extend the framework developed for logical behaviour descriptions towards operational approaches. Many popular object-oriented design methods use graphical notations based on a kind of state transition automata, for example OMT [RBP+90] and others [Boo91, SM90]. Another variant is the use of (restricted) Petri nets, for example in the Object Behaviour Diagrams approach [KS91]. The specification languages Oblog [Esp93], TROLL *light* [CGH92] and LCM [FW93] use state transition automata or extensions thereof for specifying object life cycles. We will use state transition automata as an illustration of our approach.

2 Object Framework

A simplified concept of object can be summarized as follows. Objects have a local state that can be observed with attributes. (If the object is allowed to be nondeterministic, as in LCM, then there is a part of the state that is not observable by attributes but by events that may occur in the future. This is not relevant for our argument, so we do not pursue this here.) The local state can be changed by the occurrence of local events, i.e. attribute observations may change after the occurrence of events. An event occurrence may be initiated by the object itself (active event) or as a result of a communication with another object (event calling).

A sequence of events (or sets of synchronized events [HS93]) occurring in an object is called a life cycle of an object. The set of possible life cycles defines a process which is the formal description of object behaviour. Behaviour in this sense comprises the long term evolution of objects — not just their basic operations as in other object models. In this process oriented framework, an object state depends on the complete history of the object after creation. The set of possible life cycles of an object is what is called a trace set above, and can be represented by a finite state automaton. The set of (possibly infinite) words recognized by the automaton is the set of possible object life cycles. (Again, if we allow nondeterminism, the picture is more complicated but we ignore this here.)

This simple object concept can be used to formalize object-oriented design frameworks as well as corresponding specification languages like Oblog, TROLL and LCM [SSE87, JSHS91, Jun93, Saa93, FW93, WF93, Wie91]. In all these approaches object behaviour is characterised by a set of admissible object life cycles.

The intuitions which motivate our formalization of life cycle inheritance are as follows. Let object class C_s be a specialization of object class C_g. We call C_g the generalisation and C_s the specialisation. We first list the intuitions which (we hope) are uncontroversial:

- All attributes and events defined for instances of C_g are attributes and events of instances of C_s (but the instances of specialisation may have more).

- All integrity constraints defined for instances of C_g are constraints of instances of C_1 (but the instances of the specialisation may have more). In particular, we may add more restrictions to the range of possible attribute values. In this way, an instance of a specialisation always satisfies all constraints defined for instances of the generalisation.

Slightly more controversial is what happens if we specialise preconditions:

- An enabling precondition of a method is a necessary precondition for success, i.e. if the precondition is false, the method cannot be executed. This means that preconditions are strengthened as we specialise. This way, if m is a method defined for all instances of C_g and m occurs in the life of an instance o of C_s, then its is also a valid event occurrence in the life of o viewed as an instance of C_g.

Now, a life cycle is a set of possible processes executed by an object; let us call elements of this set life cycle executions. Then the intuitions we want to formalise in this paper are the following:

- Each property shared by all life cycles defined for instances of the generalisation is also a property shared by all life cycles defined for instances of the specialisation.

- If we restrict a life cycle defined for instances of a specialisation to events defined only for instances of the generalisation, we get a subset of the life cycle defined for instances of the genralisation. An execution of this life cycle is therefore a valid execution of the generalised life cycle. In other words, we want to preserve all properties of the inherited set of life cycles, but some of them may be not included in the new object [JSHS91, Saa93]. This correspond to the definition of object morphism as given for example in [SSE87, SE91, ESS92].

Note that this characterisation is stated in semantic terms, i.e. it is about object models. Elsewhere, it is shown that this characterisation can be lifted to the level of (temporal) logic specification by defining inheritance as a subset relation on theories, which in turn corresponds to a definition of specialisation as a strengthening of axioms [FSMS91]. In the rest of this paper, we present a way of lifting the above characterisation of life cycle inheritance to *operational* life cycle descriptions. Our aim is to give simple checking conditions for operational life cycle diagrams which determine whether a new (i.e. specialised) diagram *inherits* a given (i.e. generalised) diagram in the sense sketched above.

3 Life Cycle Diagrams and Inheritance

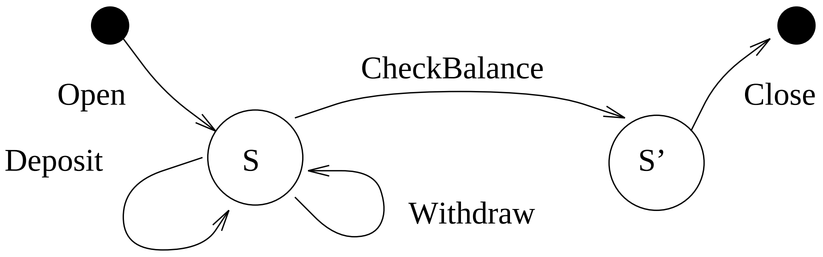
For exemplifying our approach using a popular formalism, we will use the OMT notation for the so-called *Dynamic Model* [RBP+90]. For each object class, a state chart [Har87] defines the possible life cycles for objects. The basic features of state charts can be summarized as follows:

- A state chart is a directed graph.

- Nodes of the graph are states of the object.
- State transitions are represented by edges labeled by action names.
- Birth and death transitions are transitions connected with a black dot (denoting the state of 'non-existence').
- Transition edges may be labeled by enabling conditions.

We will call such state charts *Life Cycle Diagrams* (LCD). Example life cycle diagrams are shown in the following figures.

We start with a small example of an LCD. Assume a class of `Account` objects with a very simple behaviour: accounts can be opened and closed, and during the life of an object we may deposit or withdraw money arbitrarily. The action `CheckBalance` checks the balance of an account before it can be closed. The corresponding LCD is very simple:



LCD Account

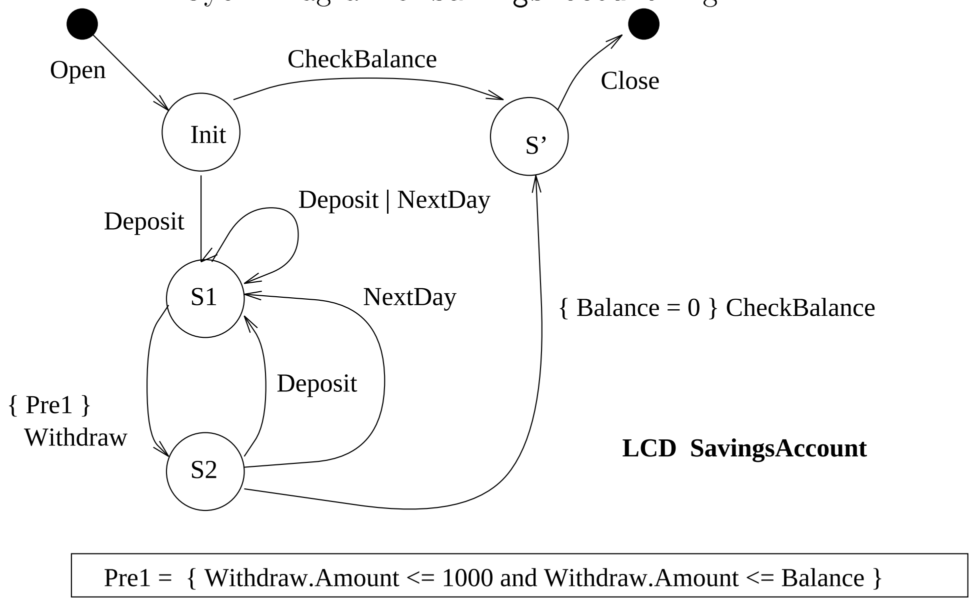
An account object has as a state attribute `Balance` storing the current balance. Both actions `Deposit` and `Withdraw` have a parameter `Amount`. Both parameters and attributes are not shown in the LCD.

To understand how we can specialise the account life cycle, we should realise that the diagram shows what local events can do in local states. For example, if we limit ourselves to events declared for Account objects, then CheckBalance is the only event that can take us from S to S'. This does not exclude the possibility that, if we specialise, there will be new events that may do different things in state S, provided that we respect our inheritance condition on life cycles.

A specialised version of an account is a `SavingsAccount`. However, we require that savings accounts should *inherit* the behaviour of simple accounts. A savings account is specialised with respect to several properties:

- Only one withdrawal per day is allowed. This property is modeled using a new action `NextDay`. An exception is if the holder makes a deposit after the withdrawal.
- There must be at least one deposit before the first withdrawn.
- The maximal withdrawal amount is 1.000, additionally the withdrawed amount must not be larger than the current balance.
- A savings account may only be closed after a withdrawal leaving the balance as zero.

The Life Cycle Diagram of `SavingsAccount` is given as follows:



Intuitively, the lives of saving accounts are specialised account lives — each life cycle of a savings account satisfies all restrictions we stated for simple accounts. The following section will give a formalization of this intuitive inheritance concept.

4 Inheritance Conditions for LCDs

In principle, we can insure a proper inheritance relation between LCDs using two criteria: we can define a check algorithm for this property, or we offer a set of LCD manipulations guaranteeing the inheritance property.

4.1 Inheritance as Graph Morphisms

We argued informally that inheritance on sets of life cycles corresponds to *"Each new life cycle when restricted to inherited events is contained in the original life cycle set"*. More formally, restriction to inherited events corresponds to a projection function on traces where new events are projected out, and the containment is defined as an injective function. Both together define a notion of *inheritance morphism* on object behaviour close to the definitions given in [SE91, ESS92].

The relation of trace sets to life cycle diagrams can be explained as follows: each path through an LCD defines a possible trace. The set of all traces can be generated by 'unfolding' the LCD. Since traces can be seen as special directed graphs, we can equivalently reformulate the relation using graph morphisms: 'A trace satisfies a LCD, if there exists a label-preserving embedding morphism mapping the trace onto the LCD'.

Now we have the following situation (the unfolding arrows denote graph morphisms in the opposite direction):

$$
\begin{tikzcd}
\textbf{LCD}_1 \arrow[d, "\text{unfold}"] & & \textbf{LCD}_2 \arrow[ll, dashed, "\text{inheritance between LCDs}"'] \arrow[d, "\text{unfold}"] \\
\textbf{traces}_1 & & \textbf{traces}_2 \arrow[ll, "\text{inheritance morphism}"']
\end{tikzcd}
$$

The main idea of lifting the inheritance condition onto the level of LCDs is the following: A set of traces can be equivalently seen as a branch connected at the start state. The injective function as heart of the inheritance morphism between trace sets is again a graph morphism between these branches. The diagram commutes (the dashed arrow) if we define an inheritance morphism between LCDs similar to inheritance between trace sets: *a projection followed by a graph morphism.*

4.2 Check Conditions

The first topic we handle is a *check* procedure which analyzes two given LCDs whether the second one inherits the first one. This check procedure can be part of an analyzing tool for object-oriented design.

The check procedure consist of two phases: the structure analysis and the analysis of the permissions (enabling conditions). The input of the check procedure are two LCDs lcd of object description o and lcd' of o' where the signature of o' is a superset of the signature of o (if necessary modulo renaming). The check procedure consists of two phases, the *structure analysis* and the *permission analysis*.

4.2.1 Structure analysis

1. Collapse state nodes in lcd' which are connected by transitions which are labeled with actions newly defined in lcd' (ie, not inherited from o). Adjust the edges to the new nodes.

2. Remove edges labelled by actions not inherited from o. The resulting LCD is called $lcd' \downarrow_o$ ('lcd' restricted to o').

   Step 1 and 2 realize the projection operator for LCDs mentioned in Subsection 4.1.

3. lcd' inherits the structure of lcd, if there exists a graph embedding morphism from $lcd' \downarrow_o$ to lcd. A graph embedding morphism maps nodes to nodes and edges to edges such that the graph structure is respected. Additionally, the action labels of transition edges must be respected and the birth and death nodes (the black bullets) are not identified with other nodes.

In our example, we have **LCD Accounts** as lcd and **LCD SavingAccounts** as lcd'. S_1 and S_2 of lcd' are collapsed into a new state S'' in Step 1 of the check, and $Init$ and S'' are mapped onto S of lcd in Step 3. Due to space restrictions we cannot give complete and more complex examples for the checking procedure.

4.2.2 Permission analysis

The permission analysis has to check whether the permissions in lcd' are more restrictive than the corresponding permissions in lcd.

1. Identify pairs of edges e and e' with each other, where e' from lcd' was mapped to e from lcd in the graph embedding.
2. Check if the permission $Cond$ of e logically implies the permission $Cond'$ of e', ie, if new permissions are more restrictive than the inherited conditions.

4.3 Safe LCD Manipulations

Besides a checking procedure, graph manipulations *guaranteeing* the inheritance condition are of special interest for tools supporting object-oriented design [LC93].

As an alternative to the checking procedure, we can offer safe graph manipulations guaranteeing the inheritance property. We distinguish *primitive operations* as base manipulations and *derived* operations consisting of a sequence of primitive operations.

4.3.1 Primitive LCD manipulations

Primitive manipulations guaranteeing the inheritance property are:

AddTransition Add a new transition from a node e to itself labeled with a new (i.e., not inherited) action name.

SplitNode Split a node into several nodes where all connected edges are copied for them. Cyclic edges are copied, too, and additionally connect all new nodes in both directions.

RemoveEdge Remove an edge.

RemoveIsolatedNode Remove an isolated node.

StrengthenCondition Strengthen a permission. Removing an edge can be seen as a special case of this operation (strengthen to **false**).

It can be easily checked, that these primitive operations conform to the inheritance condition. The resulting LCD is, however, in general a non-deterministic automaton which have to be made deterministic using the well-known techniques if necessary.

4.3.2 Derived LCD manipulations

The set of primitive operations is complete (compare [LC93]) but they do not have the right granularity to be operations in a design tool, for example as part of a syntax-directed editor for manipulating inherited LCDs. Therefore we define some useful derived operations. As derived manipulations consisting of a series of these basic steps we have the following transformations:

ExpandNode One node can be replaced by a complete subgraph:

1. Replace a node of *lcd* by a complete subgraph having only edges labeled with newly introduced action names.
2. Distribute copies of inherited edges leading to or origin from the original node over the new nodes (respecting the direction of the edges).

CopySubgraph In analogy to splitting a node, we can copy a complete subgraph (with doubling ingoing and outgoing edges).

RemoveSubGraphs In analogy to removing isolated nodes, we can remove subgraphs not reachable from the birth state. Since we are allowed to remove edges, we even can remove arbitrary subgraphs.

It can be easily shown that **ExpandNode** and **CopySubGraph** consist of a sequence of primitive operations (for example, **ExpandNode** performs the following steps: first adding cyclic edges with new actions, then node splitting, then edge removal).

5 Conclusions

5.1 Future Work

The inheritance relation on trace behaviour can be lifted to the level of graphical behaviour descriptions using graph morphisms. We have presented both a checking procedure for this inheritance condition as well as a safe graph manipulations based on this formalization. These techniques can be used in design environments supporting an object-oriented approach.

The OMT notation will be used as a graphical view on TROLL specifications in the TBENCH environment currently under development at the Technical University of Braunschweig [WJH+93]. The checking procedure is planned to be part of the analyzing tools whereas the graph manipulations can be integrated in the graphical editor. Here, we have to extend the framework to capture also combinations of LCDs (union and intersection) which decreases the modelling complexity for single LCDs. This problem also arises with multiple inheritance.

Theoretically, it is of interest to extend the framework to more advanced interpretation structures for object behaviour. The trace semantics is very natural for state transition diagrams, but too restricted for some kinds of concurrency.

Petri nets [Rei85] represent another popular formalism for the description of system dynamics. [OSS93] introduces a type of high-level Petri nets, Nested Relation/Transition nets to model procedures in complex object database applications. [Nem92] uses Predicate/Transition Petri nets to model dynamic aspects of objects and systems. An interesting topic for future investigation is how our ideas for inheritance on life cycle diagrams relate to such Petri net based behavior description techniques.

5.2 Comparison to Related Work

The problem of inheriting state transition diagrams is mentioned in several object-oriented design approaches (for example, see [RBP+90, Subsection 5.7]). In 1993, several research groups attacked this problem:

McGregor and Dyer [MD93] use state charts extended by hierarchical state decomposition and concurrency for modelling object behaviour. Their inheritance condition restricts possible modifications on pre- and post conditions of methods only. Long-term behaviour is not captured explicitly. Therefore, their modifications of state charts are more liberal than our definition — for example, they allow to connect two existing states by a new transition which violates our inheritance condition.

Lopes and Costa [LC93] use transition systems to describe object behaviour. Transition systems are similar to LCDs but they do not support explicit permissions as edge labels. Inheritance between transition systems is motivated by life cycle inheritance as in our approach. They present three basic graph rewriting rules corresponding to our operations **ExpandNode**, **RemoveEdge** and **RemoveIsolatedNode** and show that this rewriting system is complete and sound wrt inheritance.

Ebert and Engels [EE93] propose non-deterministic finite ϵ-automata for specifiying object behaviour. Again, they do not attach permissions to state transitions. Inheritance is expressed by automata homomorphisms. These homomorphisms are not only used to describe reuse via inheritance in subclasses, but also to define behaviour abstraction in view definitions.

References

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin / Cummings, Redwood City, 1991.

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

[EE93] J. Ebert and G. Engels. Dynamic models and behavioural views. Submitted for Publication, 1993.

[Esp93] Espírito Santo Data Informática, Lissabon. OBLOG CASE V1.0 – The User's Guide. Espírito Santo Data Informática, Av. Alvares Cabral 41-5, 1200 Lissabon, Portugal, 1993.

[ESS92] H.-D. Ehrich, G. Saake, and A. Sernadas. Concepts of Object-Orientation. In *Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany)*, pages 1–19. Springer IFB 303, 1992.

[FSMS91] J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, pages 243–284, Amsterdam, 1991. North-Holland.

[FW93] R.B. Feenstra and R.J. Wieringa. LCM 3.0: a language for describing conceptual models. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8:231–274, 1987.

[HS93] T. Hartmann and G. Saake. Abstract Specification of Object Interaction. Informatik-Bericht 93–08, Technische Universität Braunschweig, 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.

[Jun93] R. Jungclaus. *Modeling of Dynamic Object Systems—A Logic-Based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

[KS91] G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proc. Int. Conf. on Data Engineering*, pages 530–539, Kobe, 1991. IEEE Computer Society Press, 1991.

[LC93] A. Lopes and J. F. Costa. Rewriting for Reuse. In *Proceedings ERCIM Workshop, Nancy, November 2-4*, pages 43–55. INRIA, 1993.

[MD93] J. D. McGregor and D. M. Dyer. Inheritance and State Machines. *ACM SIGSOFT Software Engineering Notes*, 18(4):61–69, 1993.

[Nem92] Tibor Nemeth. Konzeptuelle Objektsysteme zur Modellierung von Informations- und Steuerungssystemen. *EMISA Forum*, (1):15–24, 1992.

[OSS93] A. Oberweis, P. Sander, and W. Stucky. Modelling the Behaviour of NF2-Database Applications. *EMISA Forum*, (1):15–34, 1993.

[RBP+90] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Rei85] W. Reisig. *Petri Nets.* Springer-Verlag, Berlin, 1985.

[Saa93] G. Saake. *Objektorientierte Spezifikation von Informationssystemen.* Teubner, Stuttgart/Leipzig, 1993. Habilitationsschrift.

[SE91] A. Sernadas and H.-D. Ehrich. What Is an Object, After All? In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, pages 39–70, Amsterdam, 1991. North-Holland.

[SM90] S. Shlaer and S. J.. Mellor. *Object Life Cycles: Modeling the World in States.* Yourdon Press, Englewood Cliffs, New Jersey, 1990.

[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.

[WF93] R.J. Wieringa and R.B. Feenstra. The university library document circulation system specified in LCM 3.0. Technical Report IR-343, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[Wie91] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.

[WJH+93] R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. OMTROLL – Object Modeling in TROLL. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, pages 267–283, 1993.