Animation Support for a Conceptual Modelling Language *†

Thorsten Hartmann, Ralf Jungclaus, Gunter Saake

Abt. Datenbanken, Techn. Universität Braunschweig

POBox 3329, D–38023 Braunschweig

E-mail {hartmann|jungclau|saake}@idb.cs.tu-bs.de

Abstract

The first phase in developing information systems is often called conceptual modelling phase. The model constructed is the first formal document describing the desired system. The conceptual model must deal with structural *and* behavioural aspects of the world. The complexity of such a model often prevents a detailed understanding by the customers. One way to estimate if (s)he really gets what (s)he wants are prototype systems constructed by conventional programming. We propose a way to construct prototypes out of formal specifications by means of transformation into a kernel language that can be executed in a suitable distributed runtime environment. We introduce the modelling language, the basic execution mechanism, and the basic architecture of the runtime system.

1 Introduction

A large number of software systems can be characterized as information systems. The purpose of an information system is to realize parts of the world usually called Universe of Discourse (UoD) in a computer system. An information system is *embedded* in the real world and therefore the first steps in designing such systems is the integrated modelling of the parts of the UoD that are to be computerized later on and the environment using this system. Following [8] the relevant aspects of the UoD are static and dynamic. Usually this phase of system design is called *conceptual modelling*.

On the one hand the conceptual model must be depicted in a formal way since it will be used by system developers as the *basis for following implementation steps*. Thus the derivation of knowledge for the design process must be possible. On the other hand it serves as the contract between system developers and

*This work was partially supported by the CEC under ESPRIT-2 BRA WG 3023 IS-CORE (Information Systems – COrrectness and REusability). The research work of Ralf Jungclaus and Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3.

†To appear in: Proc. DEXA'93, Prag, 1993, Springer LNCS Series.

users and must therefore be *readable by domain specialists.* These requirements contradict each other.

In general it seems not to be possible to have a declarative, formal specification of a conceptual model developed by a computer scientist that is also easily understandable by customers. A way out of this dilemma can be given by *early prototyping* or *animation* of the model, providing help to understand the specification by way of testing prototypes generated from the specification.

The formalisms usually adopted for conceptual modelling like the ER-model and its extensions [2, 6], semantic data models [22] as well as informal approaches like data flow diagramms or JSD [12] are not sufficient for a detailed modelling of real world aspects. Either they only provide the structural components of a UoD or they do not integrate the structural and behavioural aspects. Only very few modelling resp. knowledge representation approaches address the problem explicitly, among them TAXIS [21], RML [7], Object/Behaviour Diagramms [15], and CMSL [27].

One way to overcome the problem of integrating the structural and behavioural part of a system description is an *object oriented viewpoint.* Regarding a system as communicating objects seems to be a natural way to represent the relevant aspects of the real world [3]. Objects, encapsulating an internal state and providing an external operation interface may be used to describe the system as a whole when they are connected via communication links.

In our approach the specification of a UoD can be depicted as a system of interacting components. The characteristics of this approach can be summarized as follows: All system components have a *local state.* Thus, state information is localized in the components. The components evolve during time, i.e. their state changes over time. Autonomous components do not influence each other. All components of a system may themselves be structured and composed of smaller subparts.

The approach to conceptual modelling of dynamic information systems described here integrates formalisms known from semantic data models, object oriented methods and formal approaches to software specification. The goal is to define a declarative formalism for system specification using results from algebraic specification of abstract data types [28], algebraic specification of processes [11, 20] and logical modelling of reactive systems [18].

The language TROLL [14] developed on the basis of [26] will only be sketched here. Key concepts can also be found in [13, 9].

2 Animation vs. Implementation

Following the specification of the desired information system, the next phase is described as *design.* In the design phase, a declarative specification is transformed into an operational description that can then be implemented on a computer system. The design process can and should be guided by information derived from the conceptual model specification. The derived knowledge can also be used to automatically construct parts of the design. We therefore distinguish between two orthogonal directions: *Animation*: automatic construction of executable specifications from conceptual model specifications on the same level of abstraction. *Implementation*: realisation of abstract concepts with concrete ones using services provided by the underlying sub-systems like database systems, operating systems and communication software.

On each level in the stepwise design process an animation must be possible. That is, starting with the conceptual model which contains mainly declarative components and can therefore only be executed to some extend, we reach more and more operational specifications during the design/implementation process.

During the conceptual modelling phase the animation is used to *validate* a conceptual model developed so far. This means, that the customer, together with the specifier, can test the specification with respect to the desired functionality. In the successive design steps again the animation can serve as a test of the functionality of the more concrete system specification.

This does not mean however that animation is a way to *verify* parts of the design. We think, that the main function of animation is a verification against *informal requirements* as described in some kind of requirements analysis *before* any formal conceptual model is constructed. In contrast to design steps formulated in a formal language that can be formally checked against the previous, more abstract level, the original conceptual model can only be *tested* against the requirements of the customer by means of experiment.

For animation we transform constructs from the conceptual model specification into equivalent executable constructs where this is possible. One may say that this approach is near to an automatic implementation constructed from the conceptual model. But this is not quite right. For animation purposes it is irrelevant to ask for efficiency, ergonomic user interfaces, security etc. The main task of the animation is to show the functionality of the modeled UoD.

Design and implementation on the other hand serves the purpose to construct an efficient, ergonomic and secure system. The abstractly described data and communication structure of the conceptual model is refined into concrete data structures and communication between the computational processes described on a lower level of abstraction. During design and implementation the transition from a problem oriented descripton to a solution oriented description is taken. Nevertheless, some techniques used for the automatic translation of abstract specifications into executable ones can be adopted for the design and implementation.

3 TROLL Concepts and Language Features

3.1 Basic Concepts

Nowadays, the object oriented paradigm is not only used in the realm of programming languages but also for the design of software, databases and information systems [16, 3]. We can identify the following basic ideas adopted from the areas programming languages and databases [1].

Objects are entities encapsulating an internal state and providing an external interface. In databases we can find a varying degree ranging from fully encapsulating the internal structure to a completely visible state in form of attributes. Objects are uniquely identifiable, i.e. we may distinguish between objects that are in the same state and between different states of an object. Objects can communicate with each other by sending messages. These messages are sometimes answered by the receiver. An object locally determines how to react on the receipt of a message. Objects can be classified due to similar structure and observable behaviour. In programming languages classes are usually described as the *type* of objects whereas in databases classes serve as templates for instances as well as containers for instances of the class [1]. Classes are embedded in an inheritance hierarchy. In programming languages this feature is used to describe code reuse. More generally we can also look at the *concrete properties* of an instance that can be inherited by subinstances.

For the conceptional modelling approach adopted for the language TROLL, we have to abstract from the operational point of view of programming languages respectively the access oriented view of databases and introduce declarative constructs to describe the possible *evolution* of objects and thereby the evolution of the system as a whole. The concepts of TROLL can be characterized as follows:

- Objects are the basic building blocks to describe a system of interacting components.
- The internal state of an object can only be changed by means of *events*.
- Objects may be persistent.
- The evolution of an object is described by the possible sequences of event occurrences and communication with other objects.
- The internal state of an object is encapsulated. Parts of this state however can be observed by means of *attributes*.

Additionally to these concepts TROLL introduces various abstractions that help to structure specifications:

- Objects are classified in *classes* with a unique name space.
- Objects may be *aggregated* to *composite objects* [9].
- Object may be connected via *relationships* [13].

For the specification of objects in TROLL we use a variety of formalism that are integrated into the object oriented framework. For the description of various properties we use a sorted first order logic. The behaviour in terms of possible event sequences and possible attribute evolutions is described with a linear first order temporal logic dialect [23] respectively a CSP [11] based language.

3.2 Basic Language Features

The specification of object systems in TROLL [14] starts with the specification of *templates* which together with a suitable identification mechanism are used to describe object classes. We do not only describe the static aspects of objects but also their *behaviour*.

As a starting point we will specify an object class account. The description of a class containing accounts starts with an identification mechanism introducing *external* names for account objects (here natural numbers) and continues with the specification of a suitable template:

```
object class Account
  identification No:nat;
  template
    attributes
      constant Holder:|BankCustomer|;
      Balance:money; CreditLimit:money;
    events
      birth open; death close;
      new_credit_limit(in Amount:money);
      withdraw(in Amount:money); deposit(in Amount:money);
    constraints
      initially CreditLimit = 0 and Balance = 0;
      (Balance < 0) ==> sometimef(Balance >= 0);
    valuation   variables m:money; C:|CashCard|;
      { m > CreditLimit } ==> [new_credit_limit(m)]CreditLimit = m;
      [withdrawal(m)]Balance = Balance - m;
      [deposit(m)]Balance = Balance + m; ...
    behavior
      permissions   variables t,t1:UpdateType; m,m1,m2:money;
        { Balance = 0 } close;
        { not sometime(after(update_request(t1,m1)))
          since last(after(update_failed) or
                    after(deposit(m2)) or
                    after(withdrawal(m2)))} update_request(t,m);
end object class Account;
```

The declaration of attributes and events defines the *signature* of the object instances of class `Account`. Attributes are strongly typed. Data types itself are not specified together with the specification of an object society but are assumed to be predefined with a suitable formalism for specification of abstract data types. A special data type is `|BankCustomer|` which denotes internal identifiers of objects.

In the event section the state changing operations on objects are declared. We distinguish between *birth*, *death*, and *update* events that create, destroy or update objects respectively. Optional event parameters are used to define more specific state changes as well as data interchange between objects during communication.

In the constraint section we introduce constraints on attribute evolutions. In this example, we specify that the initial value of the attributes `CreditLimit` and `Balance`, i.e. the value after the birth event occured, must be zero. The second rule formulated in temporal logic describes, that an account in the red must be non-red sometime in the future.

The valuation section describes effects that events have on attribute values. The rules in this section are formulae in a restricted positional logic [5]. These formulae have a rather operational meaning, namely that after the occurrence of an event for example `withdraw` instantiated with the value `m` the attribute `Balance` has the value `Balance-m`. The rules may be applicable in only certain object states which is described with a first order condition.

In the behaviour section we describe the allowed event sequences on the object state *over time*. In this example we only introduce *permissions*, i.e. *preconditions* on event occurrences. These preconditions are formulated as first order condition on the object states or temporal logic rules referencing the past evolution of the object at hand. The second rule states that a requested update operation is only allowed to occur after the preceding request is followed by an `update_failed`, `withdraw`, or `deposit`.

For a more detailed specification of behaviour it is also possible to specify events and event combination that must be part of a life cycle: *obligations* or *liveness properties* and events that occur by the initiative of the object itself: *commitments*.

3.3 Structuring Object Societies

Objects and object classes can be in various relation to each other. They can be structured in *composite objects*, they can be embedded in a class hierarchy (*specialization*) and they may be connected with *relationships*. Since the specialization hierarchy and relationships are not needed for the description of animation and the execution model in the rest of the paper, we will only introduce composite objects and communication relationships inside composite objects.

A composite object in TROLL can be defined using *component constructors* in the components section of a template description. Let us suppose a fragment of a specification of a bank object that is constructed from a manager object taken from class `Person` (not further specified here), a set of `Account` objects, and a list of `Person` objects:

```
object Bank
  template
    components
      Manager:Person; Accounts:SET(Account); ServQueue:LIST(Person);
    events
      birth establish; death close_down;
      open_account(in No:nat); close_account(in No:nat);
    behavior
      permissions   variables n:nat;
        { not Accounts.IN(Account(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
    interaction   variables n:nat;
      open_account(n) >> Accounts(Account(n)).open;
      open_account(n) >> Accounts.INSERT(Account(n));
end object Bank;
```

With the specification of *components* special events and attributes to alter and observe the structure of the composite object are introduced. The composition is *event driven*, i.e. special events are used to introduce and delete component objects to/from the composition. The components may be *single valued*, *set valued*, or *list valued* (see example).

Important in this discussion is the *interaction* section. In the interaction part we define the communication relationships inside the complex object. Events occurring in the composite object force other events to occur. E.g. the event `open_account` *calls* for the events `Accounts.INSERT` and `open`. The calling relationship denotes *synchronous*, *directed* communication between objects.

4 Towards Animation of Conceptual Models

On the one hand a modelling language like TROLL should be sufficiently powerful to represent the relevant details of a given UoD. It should be declarative, that is, it must show *what* the customer wants to have and not *how* it can be realized. It has been argued that such modelling or specification language need not (necessarily) be executable [10]. Here we want to stress that some kind of execution capability is necessary to *validate* the constructed model of the real world in terms of what the customer intended to have. The problem is merely to *identify* parts of the model that cannot be executed (in finite time) and warn the user of an animation system if he tries to execute these constructs.

The declarative language constructs in TROLL are mainly the conditions for allowed object evolution in terms of event sequences and attribute evolution sequences (temporal constraints) formulated using temporal logic.

In particular temporal logic preconditions for events may involve checking the history back to the birth event. Since this approach is not practical, we tried to find a reduced set of information about the history that is sufficient to evaluate these conditions [25]. Some of the techniques applied there may also be used to check the conditions on the future attribute evolution. Techniques as for example *integrity preserving transactions* have been worked out in [17].

If the conditions used in the specification are of an appropriate form or if the user is warned of conditions that may not be handled this way the problem of animating a model can be tackled. We adopt the well known idea of a *kernel language* over which the higher level language constructs are defined. Thus the problem of executing the language TROLL reduces to executing the kernel constructs and warn the user if such a transformation cannot be given. The kernel language contains only restricted first order formulae, no inheritance, only simple forms of complex objects etc.

Also the kernel language contains declarative features, for example the behavioural component dealing with communication of objects. Event calling can be regarded as (synchronous) message passing. The model applied is more powerful than one could see in the first examples. See for example a decimal counter object [24]. This object represents the digits of a decimal counter as attributes. Upon counting the digits are incremented starting with the leftmost digit Digit(1) and successively higher digits if necessary:

```
object Counter
  template
    attributes Digit(nat):nat;
    events birth CreateCounter; Count; IncDigit(nat);
    constraints variables n:nat; Digit(n) <= 9;
    valuation variables n:nat;
      [ CreateCounter ] Digit(n) = 0;
      { Digit(n) < 9 } ==> [ IncDigit(n) ] Digit(n) = Digit(n) + 1;
      { Digit(n) = 9 } ==> [ IncDigit(n) ] Digit(n) = 0;
    interaction variables n:nat;
      Count >> IncDigit(1);
      { Digit(n) = 9 } ==> IncDigit(n) >> IncDigit(n+1);
end object Counter
```

This object specification contains some language features not introduced up to now. In the attributes section we introduce an infinite set of attributes, one for each natural number or equivalently an infinite array. These attributes are set to zero with the first valuation rule. This is an example of a non executable construct. The only way to circumvent this problem is to delay the execution to the first *read* on this attribute. This can be done by transforming the attributes into suitable objects with initialization for their values upon first read.

More problematic in the sense of execution are the conditions posed on the valuation and calling rules. The conditions for the calling rules only say that the rules are only evaluated if their conditions hold in the current state. The event to increment single decimal places IncDigit calls (recursively) for an arbitrary number of other IncDigit events depending on the current state of the object. In this case it can be derived easily (at least for a human being) that all IncDigit-instantiations are different, the argument strictly increases. This may not always be the case. Consider e.g. the same event being called *twice*. Conceptually, calling is synchronous, even more, all events in such an event chain occur in parallel, they transform one object state into the next state. Does such an event occur *twice*? Another questions concerns the valuation rules. Some events in such an event chain defined by calling may change the values of *the same* attribute. What is the new attribute value in the next state? Clearly we need some form of an *execution model* for event occurrences and calling relationships.

If we want to execute a specification, we must analyze such calling rules. Since the problem in general is undecidable due to first order conditions for calling and valuation rules this can only be done at execution time, i.e. we try to detect conflicts and possibly also infinite calling chains. We will now introduce an informal execution algorithm that can be implemented using traditional programming language technology. The general problem is to execute an event e in an object o. *Conceptually* the following algorithm defines the execution model:

1. Collect the set E of events called by e. This may involve other (part-)objects of o and transitive calling of events. This step is problematic since it may

not terminate. Introducing a set `E` insures the absence of duplicates of events, i.e. events do not occur *twice* in a calling chain.

2. Check conflicts in terms of valuation rules. For every $x \in E$ the set of affected attributes A_x is determined. These sets must be *pairwise* disjoint. If conflicts are encountered: exit.
3. Check if all preconditions for events in `E` hold. If not: **abort**.
4. Evaluate all valuation rules and determine the next (temporary) object state in terms of new attribute bindings.
5. Check integrity rules for all rules containing attributes affected in 2. If one or more rules are violated: restore the old state and exit.

The execution of a calling chain must therefore be encapsulated as an *ACID* transaction. For an implementation of this execution model we do not want to be restricted to a single workstation architecture. So the objects involved in communication may be distributed over a network.

5 Animation Support

The operational properties of TROLL partly coincide with the capabilities of object-oriented programming languages and databases. We therefore translate the TROLL-kernel language to *Sather*, a language developed at ICSI (Berkeley) that is based on *Eiffel* [19]. Additionally we build a runtime system dealing with communication in a network of workstations. The runtime package handles the distributed management of state information on different database systems, the automatic routing of event callings, and the necessary transaction protocol.

The structure of the system is depicted in Figure 1. Each node in the system

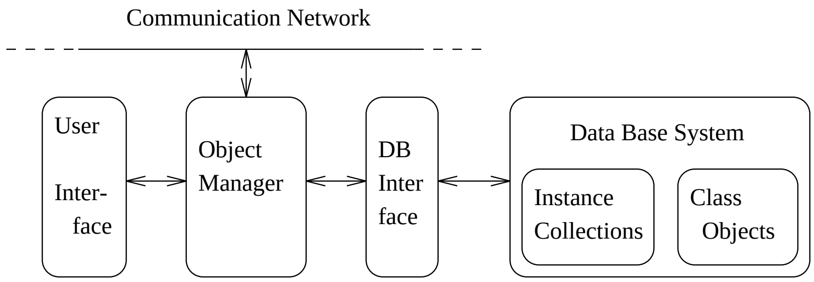

Figure 1: Architecture of Runtime/Animation System

contains an object manager that provides protocol for object communication and access. As long as we have no mechanism for object migration, the class objects on each node manage the actual existing class population. For migration of instances some further concepts like e.g. duplication of class objects are needed.

For each node there exists a database component with a simple interface to the runtime system. This interface provides protocol for object read write and transaction management. It allows an easy integration of different database systems.

The user interface can use the protocol of the object manager to *access* objects via their external names handled in the class objects, *trigger* events, or read attributes. Currently a simple graphical interface is under development showing existing classes (on a node), lists of their instances, mouse driven selection of instances, event triggers, and observing actions.

The implementation of event calling chains can be sketched in the following way. Event calls are routed via the object managers in the network (algorithm step 1) the set of events is constructed iteratively. Information about the instances and their location can be found in the object manager class-location tables and in the class objects.

The event calling is described as a tree. The runtime conflict detection, the detection of violated preconditions, computation of new attribute values, and checking of constraints is done in the objects itself that have an executable form (compiled Sather objects). Therefore the set of events constructed so far as well as the set of affected attributes must be contained in the communication of the different object managers. This method is not always efficient but sufficient for *prototyping*.

The transactional behaviour of the algorithm is realised with the transaction protocol provided by the database interface. The transaction must run with a *two phase commit protocol*. The object managers use an asynchronous communication protocol to really achieve a parallel execution of events.

6 Conclusion and Future Work

Prototyping or animation eases the *validation* of a constructed conceptual model. It must be performed together with the customer. The quality of the conceptual model is increased and possible flaws are found early during the modeling and design process. The animation is not restricted to the modeling phase but can be applied in any stage during the design of the software or information system. Animation however, means not *verification* of design. It is only meant to support the understanding between domain specialists (customers) and modelling specialists (computer scientists).

After having defined the executable language kernel and implementing the first animation prototype, we will formalise the transformation TROLL to TROLL-kernel. In fact the kernel was developed with respect to the needed functionality of the full language and ideas for the transformation.

We are aware of the restrictions of our simple execution model. Up to now we may execute simple event calling chains. For latter stages in the design process we have to *refine* or *implement* events on the conceptual level with *event sequences* on a more detailed level. These event sequences must be executed as (sub)transactions in their own right. Therefore we will have to integrate a nested transaction facility into the execution model.

Acknowledgements

We are grateful to Cristina Sernadas who developed the basic features of TROLL with us. For many fruitful discussions on the language we are grateful to all other members of IS-CORE, especially to Hans-Dieter Ehrich, Amílcar Sernadas, José Fiadeiro. For implementation work we thank our students Jan Kusch and Urs Thürmann.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The O-O Database System Manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *DOOD*, North-Holland, 1990, pages 40–57.

[2] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM TODS*, 1(1):9–36, 1976.

[3] D. de Champeaux and P. Faure. A Comparative Study of Object-Oriented Analysis Methods. *JOOP*, 3(2):21–33, 1992.

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

[5] 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 (4th IFIP WG 2.6 Working Conference DS-4)*, pages 243–284, Amsterdam, 1991. North-Holland.

[6] M. Gogolla and U. Hohenstein. Towards a Semantic View of an Extended Entity-Relationship Model. *ACM TODS*, 16:369–416, 1991.

[7] S. Greenspan, A. T. Borgida, and J. Mylopoulos. A Requirements Modelling Language and its Logic. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 471–502. Springer, 1986.

[8] J. J. van Griethuysen. Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.

[9] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In O. Lehrmann Madsen, editor, *Proc. ECOOP'92*, pages 57–77. Springer, LNCS 615, Berlin, 1992.

[10] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. Technical Report UMCS-89-12-1, Univ. of Manchester, 1989.

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[12] M. A. Jackson. *System Development.* Prentice-Hall, New Jersey, 1983.

[13] R. Jungclaus, T. Hartmann, and G. Saake. Relationships between Dynamic Objects. In H. Kangassalo, editor, *Seminar on Information Modelling and Knowledge Bases*. IOS Press, Amsterdam, 1993. *To appear.*

[14] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Technical Report 91-04, TU Braunschweig, 1991.

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

[16] W. Kim and F. H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, New York, 1989.

[17] U. W. Lipeck. Transformation of Dynamic Integrity Constraints into Transaction Specifications. *TCS*, 76:115–142, 1990.

[18] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification.* Springer-Verlag, New York, 1992.

[19] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[20] R. Milner. Operational and Algebraic Semantics of Concurrent Processes. In J. van Leeuwen, editor, *Formal Models and Semantics*, pages 1201–1242. Elsevier, 1990.

[21] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Interactive Database-Intensive Applications. *ACM TODS*, 5(2):185–207, 1980.

[22] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, 1988.

[23] G. Saake. Descriptive Specification of Database Object Behaviour. *DKE*, 6(1):47–74, 1991. North-Holland.

[24] G. Saake. Objektorientierte Spezifikation von Informationssystemen: Konzepte und Sprachvorschläge. Habilitationsschrift, TU Braunschweig, 1992

[25] S. Schwiderski and G. Saake. Monitoring Temporal Permissions using Partially Evaluated Transition Graphs. In U. Lipeck and B. Thalheim, editors, *Proc. Modelling Database Dynamics, Volkse 92*, pages 196–217. Workshops in Computing, Springer, Berlin, 1993.

[26] 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 VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.

[27] R. J. Wieringa. A Conceptual Model Specification Language (CMSL Version 2). Technical Report IR–248, Vrije Universiteit, Amsterdam, 1991.

[28] M. Wirsing. Algebraic Specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 675–788. Elsevier, Amsterdam, 1990.