TROLL — A Language for Object-Oriented Specification of Information Systems

RALF JUNGCLAUS
Deutsche Telekom
and
GUNTER SAAKE
Universität Magdeburg
and
THORSTEN HARTMANN
Technische Universität Braunschweig
and
CRISTINA SERNADAS
Instituto Superior Técnico

TROLL is a language particularly suited for the early stages of information system development, when the universe of discourse must be described. In TROLL the descriptions of the static and dynamic aspects of entities are integrated into object descriptions. Sublanguages for data terms, for first-order and temporal assertions, and for processes, are used to describe respectively the static properties, the behavior, and the evolution over time of objects. TROLL organizes system design through object-orientation and the support of abstractions such as classification, specialization, roles, and aggregation. Language features for state interactions and dependencies among components support the composition of the system from smaller modules, as does the facility of defining interfaces on top of object descriptions.

Categories and Subject Descriptors: D.2.1. [**Software Engineering**]: Requirements/Specification—*languages*; D.3.2 [**Programming Languages**]: Language Classifications—*TROLL*; D.3.3. [**Programming Languages**]: Language Constructs and Features; H.1.0 [**Models and Principles**]: General

This work was partially supported by CEC under ESPRIT-III Basic Research Action Working Group no. 6071 IS-CORE II (Information Systems—COrrectness and REusability). The work of Ralf Jungclaus (until December 1993) and Thorsten Hartmann was supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3.

Authors' addresses: R. Jungclaus, Deutsche Telekom AG, Information Technology, P.O. Box 2000, D-53105 Bonn, Germany; email: Jungclaus@11.bonn02.telekom400.dbp.de; G. Saake, Institut für Technische Informationssysteme, University of Magdeburg, Universitätsplatz 2, D-39106 Magdeburg, Germany; email: saake@iti.cs.tu-magdeburg.de; T. Hartmann, Abt. Datenbanken, Technical Universität Braunschweig, Postfach 3329, D-38023 Braunschweig, Germany; email: hartmann@idb.cs.tu-bs.de; C. Sernadas, Departamento de Matemática—Instituto Superior Técnico, Av. Rovisco Pais, 1096 Lisboa Codex, Portugal; email: css@inesc.inesc.pt.

General Terms: Design, Languages

Additional Key Words and Phrases: Formal specification, information system design, object-oriented conceptual modeling

1. INTRODUCTION

Information systems can be characterized as being *reactive systems* that store, process, and produce information about a portion of the real world. This portion is usually called the *Universe of Discourse* (UoD). The UoD is a part of the *problem domain* which also includes phenomena that are part of the environment of the planned system. (By real-world phenomena we also mean historical or planned or abstract phenomena which are not really "real" [Kent 1978].) Information systems do not describe an input-output function like many programs do but are systems with a *state* that changes over time due to interactions with the environment. Such systems are called *reactive systems* [Pneuli 1977; 1986]. Therefore, relevant information objects in information systems are not only represented through values but also through behavior (i.e., changes over time including interactions).

The domains represented by information systems tend to become larger and larger and more and more complex (e.g., computer-integrated manufacturing, banking and stock exchange systems, or control systems). This fact and recent trends toward distributed information systems and federated information systems (without central control) make the need for modularization and the modeling of interactions between components significant.

One can often observe that design implementation issues are addressed already in the early stages of information systems design. It is well known, however, that design flaws in the analysis phase that are revealed during implementation are very costly to fix. Moreover, addressing implementation issues too early restricts choices that could be made in later design stages—the information system may not be of the quality expected by the user.

TROLL is designed for use in the *conceptual modeling* or *requirements specification* phase in the development of information systems [Dubois et al. 1992; Griethuysen 1982]. In this phase, we have already identified relevant phenomena in our UoD. The task, however, is to develop a structured representation of the aspects that are of relevance for the planned information system. This representation must have the following properties:

—it must be *unambiguous* and precise, i.e., the specification must have a consistent and rigorous semantics;

—it must be structured to support the management of the complexity of the model;

—it must be independent of concrete implementation as far as possible.

A model of the UoD should be *formal* in the sense that mathematical structures are used to represent real-world aspects. Only then it is possible to

assess a model formally [Cohen et al. 1986; Wing 1990]. Thus we need *formal specifications* in a formal language that denote such models early in the design process.

Unfortunately, formal approaches do not prevent system developers from making mistakes in the design that are due to misunderstandings between them and the users. Thus, it is essential to focus solely on *conceptual* issues in the early phases of design. That is, developers should think in terms of the application rather than in terms of the implementation.

Since system developers are in general not specialists in the application domain, the process of collecting knowledge about the concepts in the UoD should be interactive, i.e., the developer must work with the user. The result of this process is a representation of the relevant static and dynamic aspects in the UoD. In this phase, it is only relevant to know *which* aspects are relevant and *what* are their properties. Thus a modeling approach should support a *declarative* description.

We strive at supporting the conceptual modeling of information systems by providing a conceptual modeling language with a formal semantics. One can identify the following advantages:

—high-level language features along with a formal semantics help in a formalization of a conceptual model;

—object-oriented conceptual model specifications are structured since properties are localized according to the concept of object.

The goal of this article is the presentation of a *specification language* for the conceptual modeling of information systems. It aims at integrating a declarative, logic-based style of system specification (and knowledge representation) with structuring mechanisms known from semantic and object-oriented data models [Peckham and Maryanski 1988] and with approaches to model processes and concurrent systems (with some operational flavor) [Hoare 1985; Manna and Pnueli 1992].

TROLL has a formal semantics in terms of a translation into a temporal logic [Jungclaus 1993]. Thus, specifications have a precise and unambiguous meaning. Using inference rules of the logic, we may deduce further knowledge from a specification or prove assertions we might want to verify. Furthermore, validation by prototyping of specifications and generation of test data are made possible. The primary purpose of a conceptual model (playing the role of a reference model for implementations) can only be fulfilled by a formal specification, since only then are we able to prove that an implementation fulfills the specification.

In the TROLL approach, we have tried to combine elements of several different approaches to system specification and modeling. The basic concepts were taken from the fields of algebraic specification, from semantic data modeling, and from the specification of reactive and concurrent systems. The approach is trying to integrate those concepts using object-oriented structuring mechanisms. The result is a specification language that provides

—structuring concepts for entities/objects as known from semantics data models;

—formal structuring mechanisms for models as known from algebraic specification; and

—event- and process-oriented specifications as well as the specification of temporal evolution as known from the specification of concurrent and distributed systems.

2. FOUNDATIONS

There is a long research tradition in fields like *semantic data modeling* and *knowledge representation* for describing real-world aspects. Semantic data modeling (for a survey see Hull and King [1987] and Peckham and Maryanski [1988]) came up when several researchers noticed that "pure" data models were not suitable to be used in the design of database schemas [Kent 1978]. When designing a database schema, database designers should think of data as descriptions of *concepts* in the problem domain—a higher level of abstraction is needed for that task. Knowledge representation has traditionally focused on representing "knowledge" about an application domain like facts, rules, strategies, etc. [Levesque 1986]. An important contribution of semantic networks is the natural modeling of *is-a-*, *is-part-of-*, and *is-instance-of* relationships, but they do not distinguish between schema and data.

A major drawback of the approaches described above is the lack of constructs to represent behavioral or dynamic aspects. Only very few approaches address the problem explicitly, among them TAXIS and RML [Greenspan et al. 1986; Mylopoulos et al. 1980]. In these proposals we are able to specify *transactions* and even *scripts* that represent activities on entities (see also Borgida [1985]).

Approaches in other research fields have addressed the modeling of system dynamics, especially approaches toward the modelling of *concurrency* [Hoare 1985; Milner 1980; Pnueli 1986]. Another approach toward describing the behavior of reactive systems is *temporal logic* [Emerson 1990; Manna and Pnueli 1992]. Here, a logical approach to specifying the change of validity of propositions representing the state of a reactive system is presented. However, these approaches do not take into account structural aspects. Thus, very often one model is chosen to represent the structural aspects of the UoD, and another model is chosen to represent the behavioral issues. Object-oriented approaches like the ones described in Booch [1990] and Rumbaugh et al. [1991] aim at overcoming these problems but are mostly informal.

In our view object-oriented conceptual modeling should be integrating the advantages of work on algebraic specification of data types [Ehrig and Mahr 1985] and databases [Ehrich et al. 1988], process specification [Hoare 1985; Milner 1990], the specification of reactive systems [Emerson 1990; Pnueli 1986], and conceptual modeling [Engels et al. 1992; Mylopoulos et al. 1980; van Griethuysen 1982]. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e., *methods*), object-oriented databases put emphasis on the observable structure of objects (through *attributes*) [Atkinson 1989; Beeri 1990]. We propose to support both views in an equal manner (i.e., the structural properties of objects may be

observed through attributes, and the behavior of objects may be manipulated through *events* which are abstractions of methods).

An object description is usually regarded as a description of possible instances of the same kind which is similar to the notion of *type* in semantic data modeling. In object-oriented programming, the notion of type is closely related to (and sometimes even mixed up with) the notion of *class*. In our view, a class defines a *collection* of instances of the same type.

Objects can be composed from other objects (*aggregation*). Aggregation of objects imposes a *part-of* relation on a collection of object descriptions. Object descriptions may also be embedded in a *specialization* hierarchy. Usually, specialization implies reuse of specifications and allows us to treat instances both as instances of the base class and the specialization class. A related concept is *generalization* that allows us to treat conceptually different instances uniformly as instances of the generalization class. The concept of *role* as a (temporary) role played by a real-world object is another helpful structuring mechanism [Pernici 1991; Wieringa 1990; Wieringa and de Jonge 1991].

Besides the structuring mechanisms mentioned above, a means to describe the *interaction* of object instances is needed to specify system dynamics. For conceptual modeling, we have to abstract from implementation-related details that arise from using message passing.

Last, but not least a means to *relate* relatively independent objects is needed. It is not very natural to bury such relationships in objects instead of making them explicit in a system specification. Additionally, the hiding of relationships in object descriptions hinders reuse. Thus, a framework for object-oriented conceptual modeling should support *relationships* among objects [Rumbaugh 1991]. Relationships among dynamic objects are, however, not as restricted as relationships in, e.g., the Entity-Relationship model—here, we must also be able to specify *interactions* among objects [Jungclaus et al. 1993].

3. BASIC CONCEPTS OF TROLL

3.1 Concept of Object

Let us now briefly sketch the important features of the semantical framework underlying the TROLL language. In TROLL, objects are specified as possible sequences of *events* (basic state transitions) and *attributes* as observable properties changed by events. Thus objects can be characterized as *observable processes*. We will proceed by sketching the basic ideas of the object model including the concept of classes.

Identification. Each instance (of an object class) is uniquely identified by an identifier. An identifier is an element of the carrier set of an *identifier sort* which provides identifiers for instances of the same kind.

Interface. The interface of an instance is described by the elements of a *template signature*. A template signature contains (along with the identifier

sorts being used) the *attribute symbols* and the *event symbols* describing structural properties, and state transitions, respectively.

State and State Transitions. An *observation* is defined through a special predicate over attribute terms denoting the fact that an attribute can be observed as having a value. Observations may be restricted. Therefore, in object descriptions we must be able to state constraints on the observations and on the evolution of observations over time. The state of an instance is defined as a set containing predicates describing

—the possible observations in that state;
—the enabled events in that state; and
—the actually occurring events in that state.

In each state, a set of events *occur* which define a state transition. In order to occur, an event must be *enabled.*

Life Cycle. The *admissible behavior* of an instance over time is defined by formulas of a temporal logic which describe the evolution of local states in terms of the valid predicates in states. As explained later on, temporal logic serves exactly the purpose of describing how the validity of formulas evolves over time.

Object Relations. The mechanisms to relate instances are the following:

—*Specialization* describes a more detailed aspect of a conceptual entity. Technically, specialization means that the specification of the base template is conservatively extended, i.e., all formulas stated in the base template have to be valid for the specialized template.
—*Aggregation* describes a *composite object* which is composed from several base instances. Again, all properties of the incorporated components have to be valid in the aggregation, i.e., we may refer to local properties in the context of the aggregation.

3.2 Specification Units

The basic specification units of the TROLL language are the following:

Template. A *template* describes the structural and behavioral properties of possible object instances. A template consists of the following parts:

—Attribute and event symbols make up the logical signature.
—*Valuation rules* describe the effects of event occurrences on attributes.
—*Permissions* state preconditions that must be fulfilled in order to allow certain events to occur. Events are not allowed to occur in the current state if their precondition is not fulfilled.
—*Obligations* state completeness conditions for life cycles of objects. A life cycle of an object can only end if the obligations are fulfilled.
—*Patterns* describe explicit event sequences an object goes through.

Class. A *class specification* defines a class by providing an identification mechanism and a template for the possible instances.

Relating Objects. Various dependencies among objects can be specified in TROLL:

—A *role specification* describes a possible role an object can play. When playing a role, an object may have additional properties and may have restricted behavior. A role template defines a special view on the same conceptual object.

—A *specialization specification* is a role starting and ending with the creation of its base object. Roles and specialization describe *is-a* relations among objects.

—Specifications of *composite objects* describe *part-of* relations among objects.

—*Relationships* describe interaction relationships among objects as well as constraints over such related objects.

3.3 Specification Sublanguages

In TROLL, the properties of objects are specified using formal languages based on (temporal) logic.

Predicate Logic. The *data sublanguage* is used for data terms that denote a value of a particular data sort. In our approach, we use terms as they are used in algebraic specifications [Ehrig and Mahr 1985]. The basic building blocks of terms are constants, variables, and functions. The *state logic* sublanguage is a usual first-order logic. The basic building blocks are atomic formulas which are predicates over data terms. Atomic formulas can be connected using the usual first-order connectives **not**, **and**, **or**, and **implies** as well as quantifiers **forall** and **exists**.

Temporal Logic. In addition to the state logic, TROLL provides two *temporal logic dialects*: the past-tense logic to make assertions over the history of objects and the future-tense logic to state properties for evolution of attribute values. For both variants we assume *discrete linear time*, i.e., a time line isomorphic to the natural number line.

The past temporal logic contains operators like **always**, **sometime**, **previous**, **always sincelast**, and **sometime sincelast**; the latter two are bounded temporal quantifiers that state properties for a particular period of (abstract) time. Without giving a formal definition here we provide only an intuitive description for two frequently used operators.

always p holds at position j iff p holds at position j and all preceding positions.

sometime p **sincelast** q holds at position j iff q held in state i and p held in *some* state between states i and j (exclusively i, inclusively j). The state i is defined as the maximum state where q held in the past, or as $i = -1$ if q did not hold in the past.

For example we may use a formula **always** Balance >= 0 in an account object specification to describe that a balance attribute has never had a negative value. Additionally TROLL provides a predicate **after**; **after**(evt) is true in all states reached by occurrence of the event denoted by event term evt. For example we may use the formula (**sometime after**(deposit(m)) **sincelast** Balance < 0) **and** m > Balance as a condition for a withdrawal to state that we assume an account object has to be brought back to a positive balance before the next withdrawal can take place.

The future temporal logic contains operators like **alwaysf**, **sometimef**, **next**, **until**, and **before**; again the latter two state properties for a particular period of (abstract) time. The future-tense operators are dual to the past-tense operators and are used to describe possible *attribute evolutions*—the interested reader may refer to Jungclaus et al. [1991] for a detailed explanation of the temporal logic used in TROLL.

Process Declaration. Object processes can be described using an *explicit* process language that draws on *CSP* [Hoare 1985]. This process language contains the usual operators for *sequencing*, *choice*, and *recursion*. We will provide an example for this sublanguage in the next section.

Interaction. Synchronization of events is supported in TROLL by two language constructs: *event sharing* and *event calling*. Event sharing (denoted by ==) declares two event terms to denote the same event. Event sharing is a symmetric declaration. Event calling on the other hand is asymmetric, a declaration e1 ➤ e2 enforces e2 to occur each time e1 occurs but not vice versa [Hartmann and Saake 1993]. In fact, the temporal logic semantics of e1 ➤ e2 is given by the following formula:

$$\textbf{always}(\textbf{occurs}(e_1) \Rightarrow \textbf{occurs}(e_2))$$

where **occurs**(e) denotes the occurrence of e in a particular state.

4. OBJECT AND CLASS SPECIFICATION

4.1 Templates

In this section we are only concerned with the specification of single, isolated, noncomposite objects. All issues about semantic links or abstractions among descriptions, specification of communication, description of composite objects, and description of systems composed from components are postponed.

Simple templates have sections for specifying **data types**, **attributes**, **event**, **constraints**, effects of events on attributes (**valuation**), and **behavior**. Behavior is specified either with **permissions** and **obligations** or using an explicit process language (**patterns**). Templates may be given a name. Thus, we can *reuse* template specifications just by giving the name. Most of the beforehand-mentioned sections of a template are optional. The listed template sections are explained for a simple BankAccount template in the following subsections.

4.1.1 *Signature.* The **data types**, **attribute**, and **event** sections make up the *signature* of a simple template definition. The symbols declared or imported in the signature are the elements of the alphabet that can be used in specification formulas of the template.

In the **data types** section, we import the signatures of data type specifications. The data type signatures contain the constant symbols and the operation symbols along with their parameter types for a data type. Data types are imported using their unique name. As an example, suppose we want to import the (predefined) data types nat (for the natural numbers) and string (for strings of characters).

```
data types nat, string;
```

The next section of the signature of a template is the **attributes** section. Attributes (as mentioned earlier) are observable properties of objects. Each attribute takes values of a sort. As an example, consider attributes of a template that describes *accounts* maintained by a bank. We use a data type money that has the expected properties of data values describing amounts of money.

```
attributes
  Balance:money;  Red:bool;  CreditLimit:money;
```

We may declare also *constant* attributes by marking them with the key word **constant**. Consider for example the holder of an account—in our miniworld at hand, we may want to require that the holder of an account never changes (once it is assigned a value):

```
constant Holder:|BankCustomer|;
```

In this example the codomain of the attribute Holder is a special data type: it is the type of identifiers for instances of the class BankCustomer written as |BankCustomer|. Thus, the elements of this type are *values* that *identify* object instances of type BankCustomer. However, identifiers are not references to objects. To refer to the object properties of bank customers, for example, we have to use *components* or *relationships*. Components make it possible to access objects from other classes that are closely related whereas relationships are used to connect loosely related objects; see the relevant sections below.

The codomains of attributes can also be domains of *complex structured values* like sets or lists of values. Please recall that these are only data values. Consider for example a customer of a bank. Each customer may be the holder of a number of accounts. Thus, the specification of a template for bank customers may include the following attribute specification:

```
Accounts:set(|Account|);
```

Furthermore, attributes can be declared to be *derived*. The values of derived attributes are computed from the values of other attributes. In our example, we want accounts to be able to provide information about the

maximum amount that can be withdrawn from an account without overdrawing it:

```
derived MaxWithdrawal:money
```

The rules that specify the derivation of the values of derived attributes are specified in the **derivation** section. It is possible to state (optionally) *conditional* derivation rules. Conditional derivation rules are only applicable if the condition is valid in the current state. They have the following general form:

'{'⟨*condition*⟩'}' '= ⇒ ' ⟨*attribute_term*⟩ '= ' ⟨*data_term*⟩

That is, every data term yielding a result of the appropriate type (of the attribute) may be used to compute the value of a derived attribute.

As an example consider the following conditional **derivation** rules that describe the computation of the attribute MaxWithdrawal of the BankAccount template. Since the value of the attribute Balance is always positive even if the account is in the red, two rules are required that state the computation in the Red case and in the **not** Red case:

```
derivation
  { Red } = ⇒ MaxWithdrawal = CreditLimit − Balance;
  { not Red } = ⇒ MaxWithdrawal = Balance + CreditLimit;
```

Another type of symbol that can be declared in the **attributes** section is an *attribute generator*. An attribute generator is a *parameterized* attribute name. This way, we may specify arrays of attributes or even infinitely many attributes. Consider for example that we want to have an attribute for the balance of an account at the end of each year. We may model that by an indexed attribute symbol in the following way:

```
Balance(nat):money;
```

In the **events** section of a template, the event symbols of a template specification are introduced. Event symbols may have parameters. Similar to attributes, these may be referred to as *event generators* or simply events. Event parameters may be used to define the effects of an event occurrence on the current state of an object, or they may be used to describe the exchange of data during communications. Consider the **events** section of the BankAccount template:

```
events
  birth open(in Holder:|BankCustomer|, in Type:{checking, saving});
  death close;
  new_credit_limit(in Amount:money);
  accept_update(in Type:UpdateType, in Amount:money);
  withdrawal (in Amount:money); deposit (in Amount:money);
  update_failed;
```

For a template, we require that at least one creation event be specified. Creation events are marked with the keyword **birth**. The values of constant attributes that are not defined by constraints (see Section 4.1.2) must be provided as parameters of a **birth** event generator, here the attributes Holder and Type. The key words **in** and **out** refer to the data flow when events are

invoked. This is only useful in communications. Briefly, **in** parameters are set by the environment, i.e., the data flows into the object. The values of **out** parameters are set by the object, i.e., the value is delivered by the object.

Events that put the life of an object to an end (*death events*) are marked with the key word **death**. It is not required to declare death events—if no death events are specified, objects with this template cannot be destroyed.

Finally we may have *active events*. Active events represent initiatives, i.e., they can be invoked by the object on its own initiative. For a discussion about activity of objects see Section 4.1.5. Briefly, active events are the only events that can occur without being called by other events. Thus, for a society to be able to evolve at all, at least one active event must be declared in some component. Active events are declared by marking an event with the key word **active**.

4.1.2 *Constraints.* In the **constraints** section, we may impose restrictions on the observable states and on the evolvement of attribute values over time. Constraints that restrict the possible observable states are called *static* constraints or *invariants*. Constraints that restrict the possible evolutions of attribute values over time are called *dynamic* constraints. Static constraints are formulas of first-order logic. Dynamic constraints are formulas of future-tense temporal logic [Lipeck 1990; Saake 1991; Saake and Lipeck 1989; Sernadas 1980]. Constraints that are implicit restrict also the admissible behavior of objects in that certain state transitions (i.e., event occurrences) are not permitted.

For accounts, we may state the following:

```
constraints
  initially Red = false;
  initially CreditLimit = 0;
  initially Balance = 0;
  initially ((Balance > 100) before Red); /* (1)
  Red ⇒ (Balance < = CreditLimit);
  Red ⇒ sometimef(not Red); /* (2)
```

Constraints with the key word **initially** state conditions to be fulfilled relative to the initial state. Consider the initial constraint (1) which says that after an account has been opened, the balance must have been more than 100 once before it can be overdrawn. The formula (2) states that if an account is in "red conditions," it has to leave this condition sometime in the future.

4.1.3 *Effects of Events.* The values of attributes may change with the occurrence of events. Thus, to describe the change of objects over time, we have to describe how the occurrence of events affect the values of attributes. *Valuation formulas* in the **valuation** section of a template are based on a *positional logic* [Fiadeiro and Sernadas 1990]. The sentences in the valuation section refer to positions in a sequence of state transitions. Valuation rules have the following form (the condition is optional; there may be several effects; see below):

{guarding condition} = ⇒ [*event term*] *attribute term* = *data term*, . . . ;

The rule says: immediately after the event denoted by *event term* the attribute denoted by *attribute term* has the value denoted by *data term* evaluated in the previous state. In more traditional terms, this describes an assignment by an event occurrence. The term on the right side of the equality sign is evaluated in the state *before* the event occurred. Implicitly we use a *frame rule* saying that attributes for which no effects of events are specified do not change their value after occurrences of such events.

For our BankAccount template we may have valuation rules such as:

```
valuation
  variables m:money;
  [new_credit_limit(m)] CreditLimit = m;
  {  not Red and (m > Balance)  } ==>
      [withdrawal (m)] Balance = m − Balance, Red = true;
```

The first rule states that the event new_credit_limit(m) changes the value of attribute CreditLimit to the value denoted by m. The second rule is only applicable if the guard holds and changes two attribute values in a straightforward manner.

4.1.4 *Behavior Specification.* A major part of a template specification is made up of the behavior specification. By behavior specification we mean here the specification of the *permitted event sequences.* That is, we have to specify precedence relationships for events, completeness requirements for life cycles, and some form of activity. Each event trace must satisfy the behavior specification.

Permissions. Permissions ensure that nothing bad does occur in the life of an instance. They are stated in the **permissions** section of a template. The general form of permissions is

{(*temporal*)*condition*}*event_term*;

For convenience, it is possible to have a list of event terms after the condition if the same condition applies for all the event terms listed. Permissions may refer to the current observable state (*simple permissions*) or to the history of events that occurred in the life of an instance so far (*temporal permissions*). In simple permissions, the condition is a formula of first-order logic over the signature of a template. As an example for a simple permission look at the following rule that requires an account to be empty before it can be closed:

```
{  Balance = 0  } close;
```

Simple permissions may refer to the parameters of the event which is guarded by a permission condition. Here, for example, we have a precondition that prohibits withdrawals of more than the maximum withdrawal being the current value of the attribute MaxWithdrawal:

```
variables m:money;
  {  m < = MaxWithdrawal  }withdrawal (m);
```

By default all variables are quantified *universally*, i.e., this condition holds for all possible values of m. Temporal permissions have a condition that

refers to the history of the instance. Thus, the condition is a formula of past-tense temporal logic.

Very often, the condition refers to event occurrences in the past. The **after** predicate is used to state that an event just occurred. An example for a temporal permission is the rule that the credit limit can only be updated if there has been at least one deposit before:

```
variables m1:money;
  { exists(m:money)(sometime(after(deposit(m))))}new_credit_limit(m1);
```

We have to bind the variable m with an existential quantifier since we only require the precondition to be true for at least one value for m. Additionally, there are a number of implicit permissions that need not be specified, e.g., for birth events: they are not allowed to occur after a birth event has already occurred in the life of an instance.

Obligations. In the **obligations** section, we state completeness requirements for life cycles. These requirements must be fulfilled before the object is allowed to die. The simplest form of an obligation is an event term. This means that the events denoted by the event term must occur in the life cycle of an object. A simple example of this is the requirement that each account must be closed sometimes, i.e., an instance must die eventually:

```
obligations
  close;
```

Parameters of events are universally quantified implicitly if not otherwise stated. This is not always suitable. An obligation may have to be fulfilled for at least one element of a set. An example for this may be that a deposit must be made at least once:

```
exists(m:money)(deposit(m));
```

Obligations may also be whole processes that must occur in a life cycle of an object. Thus, also a process description may be stated as an obligation. For accounts, this may be that at least one deposit transaction must be performed successfully:

```
exists(m:money)(accept_update(deposit,m) → deposit(m));
```

We also admit combinations of simple obligations. Obligations may be combined disjunctively and conjunctively. For a disjunction, at least one of the alternatives must be fulfilled. As an example consider the obligation that at least one update (a withdrawal or a deposit) must occur in the life of an account:

```
exists(m:money)(deposit(m) or withdrawal(m));
```

For a conjunction all the alternatives must be fulfilled. Usually, obligations depend on the history of the object. The following requirement states that once an event accept_update(t, m) occurs this update must be completed

eventually by an occurrence of one of the events update_failed or deposit(m) or withdrawal(m):

```
variables t:{deposit, withdrawal}; m:money;
{  after(accept_update(t, m))  } ⇒
    deposit(m) or withdrawal(m) or update_failed;
```

A complete life cycle is only admissible if the obligations are finally satisfied before the death event.

Patterns. Patterns describe explicitly how events are ordered in the life cycle of an object. Usually, patterns are used to describe processes, i.e., sequences of events. In TROLL, patterns do only restrict the occurrences of events appearing in the process declaration: events that are not covered by a process definition may interleave the process defined by a pattern specification.

In principle it is possible to specify arbitrary behaviors with permissions and obligations. The motivation to introduce the specification of patterns as yet another means to specify behavior is convenience. If we have to specify a very *determined* behavior, then we must specify many permissions and obligations—the specification of patterns is just more concise and more natural. The **patterns** section of a template consists of variable and process declarations along with process definitions.

To illustrate the use of the process sublanguage consider the description of automatic teller machines (ATMs) (we only show the pattern specification here);

```
patterns
  variables n, p:nat; m:money; C:|CashCard|;
  process ATM_USAGE = read_card(C) → check_card_w_bank(n, p)
                        → GO_ON → eject → ready
    where process GO_ON = case
                            bad_card_msg;
                            card_accepted → case
                                              cancel;
                                              DO_IT;
                                            esac;
                          esac
      where process DO_IT = issue_TA(n, m) → case
                                               dispense(m);
                                               TA_failed_msg
                                             esac
    end process
  end process
end process;
```

The events denote the start of a service session (read_card), the request for checking a cash card (check_card_w_bank), various messages to the ATM user (card_accepted, bad_card_msg), the issuing of a transaction request to the bank (issue_TA), the ejection of the inserted card (eject), and the dispensing of cash.

After the card has been read, the session proceeds with the check_card_w_bank event. Through the variables we are able to use the values read by

the read_card event (i.e., variables are not universally quantified but chosen by the environment; they are existentially quantified implicitly). The key word **case** indicates such an *external choice*, i.e., the decision for one of the alternatives is left to the environment. The GO_ON, pattern starts with another external choice which depends on whether the inserted card is valid or not (a decision which is not made by the ATM). If the card is accepted, the process may be canceled or may go on with the launch of a transaction that can either be terminated with dispensed of cash or a failure message. That decision is again not made by the ATM but by the environment. In any case, the service session with an ATM ends by the ejection of the cashcard. An external choice requires communication with the environment that decides on how to proceed.

4.1.5 *Activity.* Let us now briefly comment on the specification of object activity. From our point of view we identified the following forms of object activity [Saake 1993]:

—An object *triggers* another object or a collection of other objects.

This is the classical case of objects being regarded as either active or passive entities. An active object may *initiate* something whereas a passive object *suffers* something. Note that the classification of objects using this criterion is valid only in the current instant of time since an object that triggers something can be requested to do something by another object immediately after having been active in this sense. The object that triggers has the authority to cause activity. A trigger can be regarded as a *directive*. In TROLL this kind of activity is modeled as *calling* events in other objects [Hartmann and Saake 1993].

—An object performs a number of actions as a *reaction* on being triggered.

This is the traditional computer system view of activity. Programs need to be triggered (i.e., started) to do a number of tasks that have been specified in a certain order (i.e., programmed) before. In TROLL this kind of activity is modeled with process patterns that are valid after events that cause such a reactive behavior, i.e., obligations with suitable past temporal logic conditions.

—An object can do something *on its own initiative* without being asked to do it.

While modeling the real world it is often the case that system components (e.g., users or models of entities whose behavior is unpredictable) may do something in a nondeterministic way. A formalism has to offer a means to describe such spontaneous activity. Spontaneous activity often arises in abstracting from causality of actions. Take for example a user of the library: if we say that he or she may spontaneously decide to borrow a book, we abstract from reasons that cause this behavior. Events with initiative have to be modeled as **active** events in TROLL.

4.2 Temporal Semantics of Template Specifications

In this section, we want to give some ideas toward the semantics of template specifications in terms of a temporal logic [Manna and Pnueli 1992]. In fact, we use a general temporal logic which includes the past-tense and future-tense temporal quantifiers. A possible object evolution can be described by a state sequence, and therefore linear temporal logic is appropriate for giving a logical semantics to object specifications. Here, we want to give some hints toward a translation of template specifications in sets of sentences over some temporal logic which we want to call a *temporal object specification* in the sequel.

The signature of a temporal object specification is composed from the data type signatures imported in the template specification and the signature of the local events and attributes (attribute generators). Let us now briefly sketch the translation of the sections of a template specification.

In the **derivation** section, we have stated equations between derived attributes and data terms. These equations are first-order formulas. They have to be valid in every state the object goes through—thus, they are translated into invariants. A derivation ϕ can be handled as a (static) constraint.

In the **constraints** section, we have stated formulas of future-tense temporal logic. Additionally, we are able to state initial constraints. Initial constraints do not have to be translated. Other constraints are invariant, i.e., they are valid for the whole lifetime of an object. Let ϕ be a constraint formula (or a derivation equation). It is translated into

alwaysf(ϕ).

Valuation rules describe the effects of event occurrences on attribute values depending on the old state. Let e an event term, a an attribute term and t a data term. Then a valuation rule

[e]a = t

is transformed into a temporal formula of the following form:

$\forall x$:**sort**(a):(**alwaysf**((t = x) ⇒ (**next**(**after**(e) → (a = x)))))

where the variable x is used to save the value of the term t evaluated in the old state. All free variables in valuation rules have to be bound explicitly by a universal quantifier. We use the notation **sort** to denote the sort of an attribute.

Recall that we employ a frame rule: each attribute which is not affected by an event occurrence does not change its value in such a transition. This has to be stated for all transitions e modifying a in a formula like

$\forall x$:**sort**(a):**alwaysf**(((a = x) ∧ **next**(a ≠ x)) ⇒ **next**(**after**(e) ∨ ...))

The disjunction "**after**(e) ∨ ..." lists all events which are allowed to change attribute a. Note that attributes are assumed to have valued **undefined** upon creation of an object if not explicitly initialized. **Undefined** is assumed to be element of all data sorts used in TROLL.

For the **behavior** section, the translation is more complicated. First of all, we have to ensure that the initial state is reached by a birth event transition and that no following transition includes a birth event. Let $s_1, \ldots, s_n$ be data sorts and $b(s_1, \ldots, s_n)$ be a birth event generator. Then the temporal object specification must include the following two formulas:

$$\exists x_1{:}s_1, \ldots, x_n{:}s_n{:}\mathbf{after}(b(x_1, \ldots, x_n))$$
$$\forall x_1{:}s_1, \ldots, x_n{:}s_n{:}\mathbf{next}(\mathbf{alwaysf}(\neg \mathbf{after}(b(x_1, \ldots, x_n))))$$

For death events, we must ensure that transitions including them will lead to an empty state, i.e., the only state where the constant **false** is valid. Let $s_1, \ldots, s_n$ be data sorts and $d(s_1, \ldots, s_n)$ be a death event generator.

$$\forall x_1{:}s_1, \ldots, s_n{:}s_n{:}\mathbf{alwaysf}(\mathbf{after}(d(x_1, \ldots, x_n))) \Rightarrow \mathbf{next}(\mathbf{false})$$

Permissions have the general form $\{ \phi \}$ e, where ϕ is a formula of past-tense temporal logic, and e is an event term. Permissions are translated into

$$\mathbf{alwaysf}(\neg \phi \Rightarrow \mathbf{next}(\neg \mathbf{after}(e)))$$

In a second step, all free variables must be universally quantified. In the temporal object specification, permissions are formulas combining past-tense and future-tense temporal operators.

For obligations, we have several forms. An event term e as an obligation is translated into

$$\mathbf{sometimef}(\mathbf{after}(e))$$

For an obligation of the form

$$\mathbf{exists}(x{:}asort) \ldots e(\ldots, x, \ldots)$$

with an event generator e the following temporal formula is created:

$$\exists x{:}asort{:} \ldots \mathbf{sometimef}(\mathbf{after}(e(\ldots, x, \ldots)))$$

Conjunctions and disjunctions of obligations are translated to conjunctions and disjunctions of the translations of the components, respectively. Conditional obligations are translated to an implication bound by **alwaysf** [Jungclaus 1993; Saake 1993].

We do not want to go into detail on the transformation of process definitions into temporal logic formulas. In general it is known that explicit process formalisms like CSP can be translated into temporal logic with the possible necessity to use an extra variable to encode the current state [Jungclaus 1993].

4.3 Classes

A class type defines the potential instances of a class of that type. Thus, it can be regarded as being the *schema* for a class. An instance is specified by a template identified by an identifier. Thus, a class type is defined by a template along with a set of possible identifiers (the *identification space*). An

instance identifier is a value of a special data type, the type of identifiers of a class type.

A class can be seen as a *container* for instances of the corresponding class type. The class type defines the possible extensions of a class. In general, class types may not be specified separately but their specification is part of a class specification.

Basically, a class type consists of a template and a data type for the possible identifiers of instances. Objects are identified by *values* in our framework. An object identifier remains unchanged for the whole lifetime of an instance. Moreover, it is unique in the system. For each type an associated identification space is defined. An identifier along with the class name then identifies an instance of a class uniquely.

For the definition of identification spaces, we use an approach that combines the advantages of key mechanisms known from the database field (e.g., Date [1986]) and surrogates (e.g., Codd [1979] and Kent [1978]). The approach is based on abstract data types. The idea is to characterize a type of invisible identifiers (i.e., surrogates) by operations that construct these values from key values. The basic advantages are:

—the identifier is not directly visible;

—the construction of identifiers is based on key data types;

—key values can be obtained from an identifier, and

—identifiers are *typed*.

For identification spaces, we use the following notational convention: for a class type class the identification space is notated as |class|. The name of the identification space can be used in template specification like an ordinary data type name.

We implicitly have some operations on a data type |class|, namely the operation class to construct a |class| value from key values and functions to map |class| values to the corresponding constructor key values. In the sequel, we will call these constructor values *key values*, their types *key types*, and the cartesian product of the key types *key domain*. The result of this is that

—the data type |class| is isomorphic to the type **tuple**(key-type_1, ..., key-type_n);

—if not explicitly specified in the **identification** section (see below) |class| is isomorphic to nat (no primary keys are specified);

—we do not say anything about the representation of |class|-values; and

—the carrier set of |class| is the set of possible identifiers.

Since we use key values to construct identifiers, the language features to specify identification spaces in TROLL are similar to the definition of key attributes in data definition languages. For a class type as explained above, we may specify one or more classes. The class type in general is specified along with the specification of the corresponding class. The class type then has the name of that class. As an example for a class definition let us give the specification of an object class Account that describes accounts. This object

class reuses the BankAccount template discussed in Section 4.1. Implicitly we define the class type Account with the identification space |Account|.

```
object class Account
  identification
    data types nat;
    No: nat;
  template BankAccount
end object class Account;
```

Single objects are special classes that have at most one instance. In TROLL, we have a special syntactic construction to specify single objects.

As an example consider a system which models exactly one bank. Then, we would specify a single bank object identified by a name like "BankOfTroll." We do not give a complete specification here, however.

```
object BankOfTroll
  template
    ...
end object BankOfTroll;
```

For single objects, the identification space consists only of the name of the object.

5. ROLES AND SPECIALIZATIONS

Up to now we only talked about object templates, individual objects, object identities, and object classes. But an object is more than a template with an attached identity. If we look at an object from different perspectives, we are able to see special facets or concentrate on special properties. This way, we can speak about *aspects* of objects [Ehrich and Sernadas 1991; Saake et al. 1992]. Different aspects may be specified separately in TROLL. We may start with some general information about a real-world entity and construct a basic object description for this entity. For example a person may be described by name, birthday, etc.

As a next step we concentrate on special aspects of this entity and construct specialized object descriptions. These aspects may be either static, a person can be looked at as male or female, or dynamic, a person may show specialized behavior for some time in its life (he or she may become a customer, car driver etc.). The former case is modeled as a *specialization* of person, the latter as a *role* of person.

Modeling real-world entities using specialization and roles imposes a structure on the specification. Structuring mechanisms in object-oriented languages are usually described using *inheritance hierarchies*. TROLL supports two different levels of inheritance:

—*syntactic inheritance* denotes reuse of specification code. This means, attribute and event *symbols* of the parent object are visible in its ancestors. For example the birthdate attribute is known by a role or specialization of person;

—*semantic inheritance* denotes reuse of the objects itself, meaning that the behavior, i.e., the dynamic part of a parent object, is known by its ances-

tors. For example, role and specialized objects can refer to attribute *values* and the *current state* of their parent objects.

The next two sections introduce the main abstraction mechanism supported in TROLL: roles and specialization as temporary and static aspects of objects.

5.1 Semantic Inheritance

The concept of *role* describes a dynamic (temporary) specialization of objects [Pernici 1991; Wieringa 1990; Wieringa and de Jonge 1991]. As an example consider the roles customer and employee of persons. For instance, a given person may be looked at as an employee or a customer for a given period of time. Even more, the person may play different roles at the same time. For role object specifications we require the following conditions to hold:

—The role object will optionally introduce a set of new attribute symbols. Since the inherited attribute symbols can be identified by qualifying their names with the name of the base object/class, we may avoid name conflicts. If there are no name clashes, the qualification can be omitted.

—The role object has birth and optional death events. An object may play a role several times. We require the birth and death events either to be inherited from the parent of the role or be locally defined. The two mentioned possibilities reflect that the creation may be under local control (first case) or due to communication with the environment (second case). As usual we require the birth event to be the first event of a life cycle and a death event the last event of the life cycle of a role object.

—Valuation rules may be specified as usual. They can only be applied to *locally* defined attributes (otherwise the update of an inherited attribute violates the locality of attribute updates). On the other hand, valuation rules may contain inherited events. The role object may suffer from events occurring in the base object and change state according to the state changes in the base object. As an example consider the birthday event of a person object which may have effects on the role employee of person (for example, a salary increase).

—However, update of attribute values of the parent object is still possible—but not directly! For this purpose, inherited events may be triggered by events of the role object. Note that the locality of attribute updates may not be violated in this way since state changes of the parent object take place under local control, i.e., may be prohibited by permissions.

—Attribute values from inherited objects may be read; they are visible. All data terms defined in the role object may therefore contain inherited attributes.

—Constraints may be specified for local and inherited attributes. An object playing a role can thus have a more restricted behavior. Some attribute updates may be allowed; others, violating the newly specified constraints, are forbidden. For example, we may have specified that an employee must not be older than 65 years.

—For permissions we can state the same as for constraints: the possible occurrences of events inherited from the parent object may be further constrained during the life time of the role instance.
—New interactions using calling can be added because an added interaction corresponds to stronger permissions.
—Obligations can be added, too (they can be seen as permissions for the death event).

These requirements follow from a monotonic inheritance relation as defined on a logical level: an object instance corresponds to a temporal theory defined as a consequence of some axioms, and inheritance means adding new propositions but keep all old logical consequences (strengthening of conditions).

5.2 Role Specification

After having discussed the general requirements for semantics inheritance, let us consider the following example. Assume, that we specified a template person having attributes like Name, Birthdate, Address, Age, etc. In a bank world, the person may play the role of a customer. The role object BankCustomer is described by the following specification:

```
object class BankCustomer
  role of Person;
  template
    data types nat, set(nat);
    attributes
      Accts:set(nat);
    events
      birth become_bank_customer;
      death cancel;
      open_account(out Acct:nat); close_account(in Acct:nat);
    constraints
    Age > = 14;
  valuation
    variables n:nat;
    [become_bank_customer]Accts = emptyset( );
    [open_account(n)]Accts = insert(n, Accts);
    {  in(n, Accts)  } = ⇒ [close_account(n)]Accts = remove(n, Accts);
  behavior
    permissions
      variables n:nat;
      {  sometime(after(open_account(n)))  } close_account(n);
end object class BankCustomer;
```

The role introduces new attribute and event symbols extending the signature of the original person object. The event become_bank_customer induces the object Person playing the role BankCustomer and creates the role object. Clearly, the newly introduced events for opening and closing accounts make sense only for bank customers. The only attribute Accts registers the current available accounts for this person.

In the **constraints** section we refer to an inherited property. Bank customers in our example at hand must be older than 13 years. The mentioned constraint may also prevent the birth event of the role to occur. The semantics of role specifications is formally given by seeing the parent object as an (inherited) part of the role object [Jungclaus 1993; Jungclaus et al. 1991; Saake 1993].

Recall that multiple inheritance can also be formulated in this context. On the level of the role class, we have an intersection of the class populations of the parent classes. Thus we require that all parent classes of a role inherit from some shared object class located higher in the inheritance graph. On the level of instances, we require that an inherited property can be traced back to exactly one node in the inheritance graph.

5.3 Specialization

Another case of an object aspect is *specialization*. A specialization is a special case of an object role, namely a role played for the whole life of an object. Therefore, we do not have birth and death events for a specialization. The general requirements introduced for role object specification also apply to the specification of specialized objects.

The concept leads to a structured specification, factoring out the common properties in the description of the base objects, describing the special properties in the specialized objects. In conceptual modeling this concept is known under the term *is-a* or *kind-of* relationship. In this context, roles may be looked upon as being event driven whereas specialization is value based (see below).

With the birth of an object we must be able to decide to which class the object belongs. This is done using a specialization condition, a predicate over the constant attributes and identification components. Restricting the specialization predicate to constant properties of the object implies that no change of object classes is possible for a specialized object.

As in the case of object roles, an object may belong to different specialized classes of objects at the same time; it may be a specialization of several objects/classes; and specialization may be specified for single objects and class types. For a specialization, as a special case of a role, the same conditions as stated above apply. The membership of objects to specialized classes is determined with the birth of the parent object; this way we need no local birth and death events for specialized objects.

As an example, consider a CheckingAccount, a specialization of our Account. A CheckingAccount has special properties in addition to its base template Account. With each checking account, a constant personal identification number PIN and a set of cashcards is associated:

```
object class CheckingAccount
  specializing from Account where Type = checking;
    template
      data types nat, |CashCard|, set(|CashCard|);
      attributes
        constant PIN:nat;
        Cards:set(|CashCard|);
```

```
    events
      assign_card(in C:|CashCard|); cancel_card(in C:|CashCard|);
    constraints
      initially Cards = emptyset;
    valuation
      variables C:|CashCard|;
      [assign_card(C)]Cards = insert(C, Cards);
      { in(C, Cards) } ⇒ [cancel_card(C)]Cards = remove(C, Cards);
end object class CheckingAccount;
```

With the creation of an Account (once and for all the life of the object) it is determined whether the instance belongs to the class CheckingAccount depending on the value of the (constant) attribute Type of the base class.

Since specialization is only a special case of a role, namely a role played for the whole life of the object, the semantics of specialized class specification can be directly transferred from the general case of object roles.

6. COMPOSITE OBJECTS

Composite objects are a means to describe aggregation of subobjects into structured objects. This construction known from semantic data models is especially useful for the description of nonstandard applications, e.g., in engineering and office automation. For composite objects the components are encapsulated in the sense that their state may only be altered by events local to the components. Their attribute values, however, are visible. The coordination and synchronization between the composite object and its components must be performed by communication.

The semantics of composite objects is based on the concept of *safe object embedding*. The concept of embedding defines the inclusion of an object into another object without violating any of its properties [Hartmann et al. 1992; Sernadas 1987]. The semantics is the following. On the one hand the signature of the composite object is obtained by disjoint union of the local signature of the composite object (without the components) and the local signature of the components. On the other hand we must also deal with the included instance, that is, the effects of this construction on the process and observations of the participating objects. For all possible life cycles and the associated observations of attribute values of the composite object we state the following properties:

—For any life cycle of the composite object we obtain a valid life cycle of a component object if we restrict this life cycle to the events local to the component. This means that a valid life cycle of the composite object *contains* the life cycle of a component as a "subprocess." Thus, the aggregation does not change the behavior of the parts; it may however restrict the set of life cycles of the part.

—A somewhat similar condition must hold for the observation of the parts. An observation for a given life cycle denotes the binding of attributes to associated data sort values. Suppose that we have an observation of the

composite object derived from one of its possible life cycles. Then we take into account only those attributes local to some subobject. We require this (subobject) observation to be the same as the observation of the part object derived from the life cycle of the composite object restricted to the events of the part. That is, an observation of the part does not change in the context of the composite object.

A more detailed formal treatment of embedding relationship and calling can be found in Ehrich et al. [1990], Hartmann et al. [1992], Saake [1993], and Saake and Jungclaus [1992a]. To guarantee the stated properties on the language level, it is not allowed to have valuation rules for inherited attributes. The set of possible life cycles of the embedded objects may be further constrained by means of communication if it is part of a composite object.

Communication is possible using *event calling*. Event calling denotes the synchronization of object life cycles by stating constraints on the occurrence of events in the components. This way calling expressions imply further restrictions on the behavior of the component objects: some of their life cycles may not be possible in the context of the enclosing composite object and the specified communication structure.

Object embedding is used as the basic mechanism to describe the semantics of other high-level language constructs throughout this article. In our framework it is the only way to relate different objects on the semantic level. We will now proceed with the syntactic representation of *components* (dynamic composite objects). Dynamic composite objects allow for high-level description of object composition. Components may be specified as *single components* as well as *sets* or *lists* of components. They have a life of their own and may be shared by other objects as well.

With the specification of a dynamic composite object denoted with the key word **components** we have implicitly defined events to update the composition. Additionally, we have implicit attributes to observe the composite object. This view of composite objects is *operational*. In another terminology we can say that dynamic aggregation is *event driven*.

We now give an example of a specification of a bank including accounts as a set component and a single instance of class Person denoting the manager of the bank. Some parts of the bank specification are omitted since we want only to deal with the important parts of a component specification:

```
object Bank
  template
    data types nat, |ATM|, UpdateType, money, |CashCard|;
    components
      Manager:Person;
      Accounts:SET(Account);
    events
      birth establish;
      death close_down;
      open_account(in No:nat); close_account(in No:nat);
      card_request(in AcctNo:nat); card_return(in C:|CashCard|);
      ...
```

```
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; C:|CashCard|;
    open_account(n) >> Accounts.INSERT(Account(n));
    open_account(n) >> Accounts(Account(n)).open; ...
  { Accounts.IN(Account(n)) and not in(C, Account(n).Cards)} =>
      card_request(n) >> Accounts(Account(n)).assign_card(C);
  card_return(C) >>
    Accounts(Account(CashCard(C).ForAccount)).cancel_card(C);
    ...
end object Bank;
```

Initially, the Bank has no component. To manipulate the set of component accounts, the events Accounts.INSERT(|Account|) and Accounts.REMOVE (|Account|) are automatically added to the signature of the Bank object. For set components a parameterized, bool-valued attribute, in this example Accounts.IN(|Account|):bool, is included in the signature of Bank.

For the behavior of the composite object the communication inside the composite object must be specified. Communication can take place among the component objects and between the composite object and the component objects. See for example the clauses

```
open_account(n) >> Accounts.INSERT(Account(n));
open_account(n) >> Accounts(Account(n)).open;
```

which state, that every time an account identified by the natural number n is opened, it becomes a member of the set of components, and the event open is called in the corresponding object Account(n). The event open_account in the Bank object *may only occur if the expression* (**not** Accounts.IN(|Account|)) evaluates to true. Calling in the opposite direction—from the component to the composite object—is also supported in this framework.

As mentioned above, with the definition of components, implicitly generated events and attributes are added to the signature of the base object. These special events are used to update the composition, i.e., to alter the structure of the composite object. Similarly the attributes are used to observe this structure. In case of single-object components we have attributes to test if a component is assigned (DEFINED), to retrieve the object identifier of the component (ID) and events to assign and remove components (ASSIGN, REMOVE). Similarly for set- and list-valued components there are attributes like EMPTY and CARD, and events like INSERT and REMOVE, etc. For list components we provide means to access the component objects, for example FIRST (the first element of the list). For a detailed list and description of the generated attributes and events see Jungclaus et al. [1991] and Saake [1993].

The semantics of dynamic composite objects may not be described with safe object inclusion directly, since this concept only allows for static object

composition. Nevertheless, we may introduce a two-level translation to describe the semantics of dynamic composite objects regarding the *possible* compositions induced by a specification of dynamic components [Jungclaus 1993; Jungclaus et al. 1991].

7. SPECIFICATION OF SYSTEMS

When it comes to describing systems of interaction objects, it is not sufficient to provide only the structuring mechanisms described in the previous section. In system specification, we have to deal with *relationships* among objects, with *interfaces* to objects and with *object societies*. In other words we have to describe the *interconnections* among relatively independent objects that only synchronize sometimes in their life. Interfaces together with relationships define these interconnections.

Object descriptions are the units of design corresponding to real-world entities. In the case of complex real-world structures, these design units may be aggregated to composed object. The parts of a complex object may communicate, interchange data values, and show restricted behavior.

TROLL supports the specification of communication and integrity relations among objects via the *relationship construct*. Relationships on the one hand are used to depict the interconnection among separately defined objects, defining the communication patterns among these objects. On the other hand, relationships can be used to describe integrity constraints among instances of (possibly different) classes.

Another important topic in structuring complex systems is the declaration of *interfaces* or *abstractions* to objects hiding details of object classes. TROLL provides the *interface* concept to define object interfaces offering construction principles similar to those for views in relational databases.

7.1 Relationships

Relationships connect objects that are specified independently. Basically, relationships are language constructs to describe how system components are connected in order to describe the whole system.

Global Interactions. Global interactions describe communication among objects. We use the syntax for interactions depicted for complex objects. Global interactions along with the specification of the connected objects describe patterns of communication among objects.

Global interactions and constraints are specified with a special language construct, the **relationship**. First, the relationship is given a proper name. Second, the participating objects are specified giving their object names or class names. In the third part of a relationship specification, the possible *calling* relations are given.

For an example for a relationship describing interactions, consider the following relationship between banks and ATMs:

```
relationship RemoteTransaction between Bank, ATM;
  data types |ATM|,nat,money,UpdateType;
  interaction
```

```
    variables atm:|ATM|; n,p:nat; m:money;
    ATM(atm).check_card_w_bank(n,p) ➤ Bank.verify_card(n,p,atm);
    Bank.no_such_account(atm) ➤ ATM(atm).bad_account_msg;
    ...
end relationship RemoteTransaction;
```

From a process point of view, such a relationship describes how the involved processes *synchronize*. The event check_card_w_bank occurring in an ATM denotes a request to the bank to verify the inserted cash card. The other clause concerns the result of the card checking at the bank which must be transmitted to the corresponding ATM.

In interaction specifications, we may want to refer to the history of events in the connected objects. Consider the interaction between an ATM customer (of which the specification may become the description of a user interface later) and an ATM. Here, we must put precedence rules into conditions for interactions to model the process of communication:

```
relationship UseATM between ATMCustomer, ATM;
  data types |ATMCustomer|, |ATM|, |CashCard|, nat, money;
  interaction
    variables C: |ATMCustomer|; atm: |ATM|; CC: |CashCard|; p:nat;
    Customer(C).insert_card(CC, atm) ➤ ATM(atm).read_card(CC);
    { sometime after(Customer(C).insert_card(CC, atm)) ⇒
      Customer(C).enter_PIN(p, atm) ➤
        ATM(atm).check_card_w_bank(CashCard(CC).ForAccount, p);
    ...
end relationship UseATM;
```

For precedence rules, we may use the **after** predicate. The above-mentioned rule states that once a particular ATM customer inserted a cash card, the input of the PIN calls for the check_card_w_bank event in the *same* ATM.

We use a simple execution model where a chain of calls may only be carried out if all called events are permitted to occur (atomicity principle). More details on this model, for example, data transfer among events of a calling chain, can be found in Hartmann and Saake [1993]. We are aware of the limitations of this approach with respect to exceptions and long transactions.

Global Constraints. When we model systems by putting together objects, we sometimes have to state constraints that are to be fulfilled by related but independently specified objects. Such *global constraints* set up another kind of relationship among objects.

To specify global constraints, we need to know the participating objects. It is also possible that we want to describe constraints among different objects of the same class. The two points mentioned lead to the decision to use the same construct for the specification of global constraints as for global interactions. The close relationship between constraints and interactions (in fact, global interactions are a special kind of global constraints on the life cycles of participating objects) leads to two blocks: an **interaction** and a **constraints** block inside a **relationship** construct.

Consider the following example. When modeling our banking world, there may be a regulation that one particular bank customer may only be holder of at most one checking account. In TROLL, this would be specified as follows:

```
relationship CheckingHolder between CheckingAccount C1, CheckingAccount
C2;
  data types |BankCustomer|, nat;
  constraints (C1.Holder = C2.Holder) ⇒ (C1.No = C2.No);
end relationship CheckingHolder;
```

This constraint is an example for a global relationship since it cannot be specified to be local to *one* instance of the class CheckingAccount. Here we also have an example using two different *object variables* for instances of the same class.

7.2 Interfaces

Objects in TROLL such as those introduced up to now consist of a behavior and an observation component. The former is specified as the possible evolution of objects in terms of allowed events in some given object state, the latter corresponds to attribute values for a given state and their change according to the occurrences of events. Both components may be seen from outside the object; that is, *all* attribute values may be read, and event occurrences may be observed or triggered. Specifying a system of objects, we sometimes want to hide details of objects. It may be necessary to encapsulate some attributes and events explicitly since they must not be known outside the object if we specify interconnection channels with other objects that only have to observe a special facet needed for communication. This is particularly true for implementation and refinement issues, where the user must not know about complicated data structures or event traces used to described internal activities of the system. In TROLL such restricted interfaces may be defined as *projection interfaces.*

When specifying object (class) types, we describe possible extensions in terms of object instances with a structure and behavior according to the type specification. These objects are classified into object classes with the corresponding type. Object classes may be further structured when we select subpopulations of class instances. In TROLL this is achieved in defining *selection interfaces.*

Projection Interfaces. When specifying a system of objects, we sometimes want to hide or encapsulate some details. In the process of implementing objects, an event Bank_TA may be further specified in terms of other events occurring in sequence or parallel. We do not want to see these additional events on the level described so far. This way, we may specify encapsulated modules where the internal behavior is not relevant for the user of these parts.

In the case of our banking world, we need object interfaces, for example, if we want to describe ATMs for a specialized class of users (in this case customers). Here, customers must not know about all the functionality of ATMs:

```
interface class ATMToCustomer
  encapsulating ATM;
    data types bool, |CashCard|, nat, money;
    attributes
      dispensed:bool;
    events
      ready; read_card(in C:|CashCard|);
      cancel; card_accepted; bad_PIN_msg; bad_account_msg;
      issue_TA(in Acct:nat, in Amount:money); ...
end interface class ATMToCustomer;
```

The only observation for customers is the status of the ATM in terms of the bool-valued attribute dispensed. Information about the amount of money available inside the machine should (for obvious reason) not be visible.

An interface to dynamic objects defines also the possible events visible at this level of system description. Here a customer should only be able to talk to the ATM at "customer level," i.e., he or she must be able to insert cards, issue transactions, and so on. Customers must not be able to refill a machine or even remove it. Also they should not see details of the internal operations; for example, the event check_card_w_bank is not relevant at "customer level."

Additional to simple projections, TROLL allows the definition of derived attributes and events in interface declarations [Jungclaus 1991; Saake and Jungclaus 1992b].

Selection Interfaces. Another kind of access restriction is the selection interface. In contrast to projection interfaces that may be applied to single objects or object classes, selection interfaces may only be used for object classes. Here we can restrict the access to subsets of objects contained in object classes.

In this language version we support only selections based on constant properties of objects (see Saake and Jungclaus [1992b] and Saake et al. [1992] for selection based on arbitrary object properties). For example we may define an interface for another class of customers. Since ATMs only have the constant key attribute IdentNo, we can specify an (artificial) interface for a proper subset of ATMs with an identification number between 100 and 199:

```
interface class ATMToCustomer2
  encapsulating ATM;
  selection where IdentNo > = 100 and IdentNo < = 199;
  ...
end interface class ATMToCustomer2;
```

Here the actual visible population is limited using a predicate over the key attributes. Naturally, the selection interface can be combined with the projection interface.

The semantics of a selection interface definition is given by an object class specialization determining the proper subset of the encapsulated class. The condition in the **where** clause can directly be used as specialization predicate (see Section 5.3).

As mentioned above, interfaces resemble the well-known *view* construction in relational database systems. Views in relational systems may be defined for joins among different relations. In our approach, joins are described by aggregations. Therefore we support another kind of interface, the join interface which is based on selection interfaces. In the case of join interfaces, the encapsulation clause describes the joined class populations whereas the selection condition represents the join predicate. Naturally, this kind of interface can be combined with the projection interface.

8. RELATED APPROACHES

In this section, we give a brief overview of the key features of a number of languages (both graphical and textual) that follow a similar object paradigm. The list provided is by no means complete but sketches the main approaches that influenced the language TROLL. The order of the presentation is not an assessment, but tries to group languages according to similar concepts and features (as far as possible in a linear text).

OBLOG [Sernadas et al. 1987] was presented in 1987. It was the starting point for the development of a number of languages which were precursors of TROLL. The language was based on an algebraic framework for the dynamics of objects.

The basic unit of specification in OBLOG are *object types*. An object type defines a set of potential instances, their structure, and admissible behavior. The potential instances are specified by explicitly constructing the name space. The structure is defined by attributes. The behavior is defined by *events* and *trace specifications* where the admissible sequences of events are specified along with the effects of events on the values of attributes. Interaction among objects are defined by calling events (the event space is global).

Relationships of any kind among objects are specified explicitly by *links*. Links describe *morphisms* among object descriptions [Sernadas et al. 1987]. A link is always described in the source specification of a morphism and consists of a *surrogate map* defining the relationship among the surrogate spaces, and a *template map* where maps between event and attribute terms are defined which describe identical attributes and events.

Historically OBL-89 is the revised version of Sernadas et al. [1987] and is defined in Costa et al. [1989]. A graphical language based on the ideas of Sernadas et al. [1987] is GraphicalOblog [ESDI 1993]. A simplified version, TROLL *light*, is defined in Conrad et al. [1992] and Gogolla et al. [1993]. TROLL *light* emphasizes the *verification of object properties*.

CMSL [Wieringa 1990; 1991] is a language for the specification of conceptual models. CMSL combines approaches tc algebraic specification of data types, semantic data modeling, and algebraic specification of processes. An

object type is specified through attributes (which are modeled as functions mapping an object identifier to a data value), events (being the update operations on attributes), and life cycles or processes (which define the admitted sequences of events).

Events are modeled as functions where the parameter sorts are domains while a special event sort is the codomain. The effects of event occurrences on attribute values are specified by positional terms like in TROLL. Furthermore, events can be synchronized using an explicit synchronization operator and messages to define the local effects of (global) communication events. Life cycles of objects are specified as processes, i.e., similar to our pattern specifications.

CMSL Version 1 supports the concept of role besides static specialization. It has been one of the first proposals to define this concept formally. However, the latest version now called LCM [Feenestra and Wieringa 1993] does not support roles. The language is fully based on algebraic specification. The interpretation structures are *version algebras*: each possible state (in terms of a tuple of attribute values) is represented by an element of the algebra, and admissible state transitions are defined as functions among states. The algebra of all possible states and state transitions makes up the semantics of an object.

TLOOM (*t*emporal *l*ogic-based *o*bject-*o*riented *m*odel) [Arapis 1991] is a language for the modeling of dynamic aspects of database applications that combines approaches for the temporal logic-based specification of database behavior and object-oriented approaches. TLOOM emphasizes the description of the temporal evolution of object behavior and the temporal composition of object behavior. The fundamental notions in TLOOM are *Objects* which directly model real-world entities, *Messages* that can be sent and received from objects, and *Roles* to describe particular aspects or behavior that an object exhibits during a period of time. *Contexts* comprise roles and public constraints. Public constraints here describe the global relationships between the components and their local behaviors.

TEMPORA [Theodoulidis et al. 1990] is another approach putting emphasis on the modeling of temporal behavior. In TEMPORA, a conceptual model has three components: a structural model (called Entity-Relationship-Time model ERT), a Rule Model, and a Process Model. The structural model is an ER model extended with concepts to describe composite objects and specialization. Additionally, concepts to timestamp constructs are offered. The Process Model is based on data flow diagrams. The Rule Language is used for the declarative specification of constraints, derivations, and behavior and may constrain both the Process Model and the ERT model. The Rule Language is based on first-order temporal logic.

ALBERT (*a*gent-oriented *l*anguage for *b*uilding and *e*licitating *r*equirements for real-*t*ime systems) [Dubois et al. 1993] is a language that provides graphical features for structural descriptions and declarations and textural features for expressing various types of constraints. The main concepts behind ALBERT are those of *agent*, *action*, and *perception* where the notion

of agent is close to the notion of object in TROLL. An agent models an entity having *local contractual responsibilities*. An action models a discrete change occurring in a system. Perceptions allocated to agents model the knowledge an agent has about the behavior of other agents (its environment). An agent specification consists of a *declaration* part and a *constraints* part.

ERAE [Dubois et al. 1991] is a formal specification language for the conceptual modeling of dynamic systems. It is basically a many-sorted first-order real-time temporal logic and supports the declarative specification of systems composed from entities. Furthermore, ERAE provides the notion of event which is modeled as a special sort denoting (global) occurrences of events. It provides some structuring mechanisms which, however, do not rely on the notion of object.

Taxis and **TDL** [Borgida et al. 1993; Mylopoulos et al. 1980] as well as **RML** and **Telos** [Greenspan et al. 1986; Mylopoulos et al. 1993] support the description of *data classes*, and *functions* and *transactions* manipulating them. On top of these descriptions a grouping of such actions can be formulated with *scripts*. Scripts in TDL thus model long-term noninstantaneous behavior of sets of objects. A special feature of TDL scripts is the integration of a basic *exception-handling* mechanism that is employed for further structuring a design document according to normal and exceptional behavior. Scripts resemble transient composite objects in TROLL. TROLL lacks however an exception mechanism. Telos and TaxisDL are closely related to cover the requirements acquisition and design phase of information systems in a more or less uniform framework.

Object/Behavior Diagrams [Kappel and Schrefl 1990; 1991] use a graphical notion based on an extended Entity-Relationship model for the structural part and Petri nets for the behavioral part. A system specification consists of a number of diagrams that cover different aspects of the system to be described. *Object Diagrams* describe structural aspects of systems whereas *Life Cycle Diagrams* define sequences of state transitions in terms of a Petri net-based notation. *Activity Specification Diagrams* specify activities in terms of their signature and by pre- and postconditions. *Activity Realization Diagrams* specify the implementation of abstract activities. *Activity Invocation Diagrams* describe communications among objects in terms of message passing.

DisCo (*dis*tributed *co*operation) [Kurki-Suonio et al. 1991] is a language supporting the specification of distributed, reactive systems at a high level of abstraction. In DisCo, actions are not local to objects but shared among participating objects—thus, there are two fundamental concepts: *objects* and *actions*. DisCo supports modularity and transformation of specifications. It is based on an interleaving model. Compared to TROLL, DisCo does not support the integration of local state and local state transitions.

9. CONCLUSIONS

In this article we have presented the language TROLL, a specification language suited for the early stages of information systems design. A lan-

guage used for this purpose should offer a large number of abstraction concepts to enable convenient natural descriptions of UoD aspects. The concept of object grew out of a number of approaches toward UoD modeling like semantic data models, knowledge representation, process languages, and object-oriented languages. Object-oriented conceptual modeling supports the organization of a system description in terms of object descriptions that integrate the static and dynamic aspects local to an entity.

TROLL is based on a number of formal sublanguages like temporal logic, etc., which are the tools to specify properties and evolution of objects. The basic building blocks of specifications are template specifications. Templates are anonymous descriptions of instances of classes. A class is made up from an identification mechanism and a template specification.

Aspects are abstractions of an object that allow us to look at an object from different angles. The abstraction mechanisms provided by TROLL are specialization and roles. Specialization describes objects belonging to subclasses depending on observable properties (attributes) whereas roles describe objects belonging to subclasses depending on certain situations during an objects life (occurrence of events).

In TROLL, objects that are composed from other objects are called composite objects. The composition of composite objects may be altered by special events that are implicitly declared with a component specification. Life cycles of component objects may be synchronized by synchronous interaction.

A system is regarded to be a collection of interacting objects. TROLL offers language features to state relationships among separately defined objects. Relationships are interactions or dependencies and are used to put local specifications in a global system context. Interfaces for object are introduced to support cooperative design.

TROLL offers a large number of language features that are sometimes redundant. We think however that during UoD modeling the designer/specifier should not be forced to transform natural descriptions into a small number of nonredundant language features but should be able to describe objects in a natural way. To manage the complexity of specification documents we are currently developing an environment that will support a graphical interface for TROLL specifications based on OMT [Jungclaus et al. 1994; Wieringa 1993], syntax-directed editors for more detailed specifications, and analysis and prototying tools. All tools are clustered around a repository to store TROLL documents.

ACKNOWLEDGMENTS

Thanks to all members of the ISCORE project, especially to Amilcar Sernadas (who is the project leader), Hans-Dieter Ehrich, José-Luiz Fiadeiro, and Roel Wieringa. Thanks also to Alex Borgida and the other (anonymous) referees who made helpful suggestions to improve the quality of the article and to all those who have commented on earlier versions of the TROLL language.

REFERENCES

ARAPIS, C. 1991. Temporal specifications of object behavior. In *Proceedings 3rd. Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems MFDBS-91*, B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, Eds. Lecture Notes in Computer Science, vol. 495, Springer-Verlag, Berlin, 308–324.

ATKINSON, M., BANCILHON, F., DEWITT, D., DITTRICH, K. R., MAIER, D., AND ZDONIK, S. B. 1989. The object-oriented database system manifesto. In *Proceedings International Conference on Deductive and Object-Oriented Database Systems* (Kyoto, Japan, Dec.), W. Kim, J.-M. Nicolas, and S. Nishio, Eds. 40–57.

BEERI, C. 1990. A formal approach to object oriented databases. *Data Knowl. Eng. 5*, 4, 353–382.

BOOCH, G. 1990. *Object-Oriented Design*. Benjamin/Cummings, Menlo Park, Calif.

BORGIDA, A. 1985. Features of languages for the development of information systems at the conceptual level. *IEEE Softw. 2*, 1, 63–73.

BORGIDA, A., MYLOPOULOS, J., AND SCHMIDT, J. W. 1993. The TaxisDL software description language. In *Database Application Engineering with DAIDA*, M. Jarke, Ed. Springer, Berlin, 65–84.

CODD, E. F. 1979. Extending the relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4, 397–434.

COHEN, B., HARWOOD, W. T., AND JACKSON, M. I. 1986. *The Specification of Complex Systems*. Addison-Wesley, Reading, Mass.

CONRAD, S., GOGOLLA, M., AND HERZIG, R. 1992. TROLL *light*: A core language for specifying objects. Informatik-Bericht 92-02, TU Braunschweig.

COSTA, J. F., SERNADAS, A., AND SERNADAS, C. 1989. OBL-89. User's Manual. Version 2.3. Tech. Rep., Instituto Superior Técnico, Instituto de Engenharia de Sistemas e Computadores, Lisbon.

DATE, C. J. 1986. *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass.

DUBOIS, E., DU BOIS, P., AND PETIT, M. 1993. O-O requirements analysis: An agent perspective. In *ECOOP'93—Object-Oriented Programming*, O. Nierstrasz, Ed. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, Berlin, 458–481.

DUBOIS, E., DU BOIS, P., AND RIFAUT, A. 1992. Elaborating, structuring and expressing formal requirements of composite systems. In *Advanced Information Systems Engineering CAISE'92*, P. Loucopoulos, Ed. Lecture Notes in Computer Science, vol. 593. Springer-Verlag, Berlin.

DUBOIS, E., HAGELSTEIN, J., AND RIFAUT, A. 1991. A formal language for the requirements engineering of computer systems. In *From Natural Language Processing to Logic for Expert Systems*, A. Thayse, Ed. John Wiley & Sons, Chicester, U.K., 269–345.

EHRICH, H.-D., DROSTEN, K., AND GOGOLLA, M. 1988. Towards an algebraic semantics for database specification. In *Proceedings of the 2nd IFIP WG 2.6 Working Conference on Database Semantics "Data and Knowledge" (DS-2)*, R. A. Meersmann and A. Sernadas, Eds. North-Holland, Amsterdam, 119–135.

EHRICH, H.-D., GOGUEN, J. A., AND SERANDAS, A. 1990. A categorial theory of objects as observed processes. In *Proceedings REX/FOOL Workshop*, J. W. deBakker, W. P. deRoever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 489. Springer, Berlin, 203–228.

EHRICH, H.-D. AND SERNADAS, A. 1991. Fundamental object concepts and constructions. In *Information Systems—Correctness and Reusability*, G. Saake and A. Sernadas, Eds. TU Braunschweig, Informatik Bericht 91-03, 1–24.

EHRIG, H. AND MAHR, B. 1985. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer-Verlag, Berlin.

EMERSON, E. A. 1990. Temporal and modal logic. In *Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier Science Publishers B.V., Amsterdam, 995–1072.

ENGELS, G., GOGOLLA, M., HOHENSTEIN, U., HÜLSMANN, K., LÖHR-RICHTER, P., SAAKE, G., AND EHRICH, H.-D. 1992. Conceptual modelling of database applications using an extended ER model. *Data Knowl. Eng. 9*, 2, 157–204.

ESDI. 1993. *OBLOG CASE V1.0—The User's Guide*. ESDI Espirito Santo Data Informatica, S. A., Lisbon.

FEENSTRA, R. AND WIERINGA, R. 1993. LCM 3.0: A Language for describing conceptual models - Syntax definition. Report ir-344, Faculteit der Wiskunde en Informatica, Vrije Universiteit, Amsterdam.

FIADEIRO, J. AND SERNADAS, A. 1990. Logics of modal terms for system specification. *J. Logic Comput. 1*, 2, 187-227.

GOGOLLA, M., CONRAD, S., AND HERZIG, R. 1993. Sketching concepts and computational model of TROLL *light*. In *Proceedings of the 3rd International Conference on Design and Implementation of Symbolic Computation Systems (DISCO'93)*, A. Miola, Ed. Lecture Notes in Computer Science, vol. 722. Springer, Berlin, 17-32.

GREENSPAN, S., BORGIDA, A. T., AND MYLOPOULOS, J. 1986. A requirements modelling language and its logic. In *On Knowledge Base Management Systems*, M. L. Brodie and J. Mylopoulos, Eds. Springer-Verlag, Berlin, 471-502.

HARTMANN, T. AND SAAKE, G. 1993. Abstract specification of object interaction. Informatik-Bericht 93-08, Technische Universität Braunschweig.

HARTMANN, T., JUNGCLAUS, R., AND SAAKE, G. 1992. Aggregation in a behavior oriented object model. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, O. L. Madsen, Ed. Lecture Notes in Computer Science, vol. 615. Springer, Berlin, 57-77.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J.

HULL, R. AND KING, R. 1987. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv. 19*, 3, 201-260.

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

JUNGCLAUS, R., HARTMANN, T., AND SAAKE, G. 1993. Relationships between dynamic objects. In *Information Modelling and Knowledge Bases IV: Concepts, Methods and Systems*, H. Kangassalo, H. Jaakkola, K. Hori, and T. Kitahashi, Eds. IOS Press, Amsterdam, 425-438.

JUNGCLAUS, R., HARTMANN, T., SAAKE, G., AND SERNADAS, C. 1991. Introduction to TROLL—A language for object-oriented specification of information systems. In *Information Systems - Correctness and Reusability*, G. Saake and A. Sernadas, Eds. TU Braunschweig, Informatik Bericht 91-03, 97-128.

JUNGCLAUS, R., SAAKE, G., HARTMANN, T., AND SERNADAS, C. 1991. Object-oriented specification of information systems: The TROLL language. Informatik-Bericht 91-04, TU Braunschweig.

JUNGCLAUS, R., WIERINGA, R., HARTEL, P., SAAKE, G., AND HARTMANN, T. 1994. Combining TROLL with the object modeling technique. In *Innovationen bei Rechen—und Kommunikationssystemen*. Springer, Berlin.

KAPPEL, G. AND SCHREFL, M. 1990. Using an object-oriented diagram technique for the design of information systems. In *Dynamic Modelling of Information Systems (Proceedings of the International Working Conference)*, H. G. Sol and K. M. van Hee, Eds. North-Holland, Amsterdam, 121-164.

KAPPEL, G. AND SCHREFL, M. 1991. Object/behavior diagrams. In *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, Calif., 530-539.

KENT, W. 1978. *Data and Reality*. North-Holland, Amsterdam.

KURKI-SUONIO, R., SYSTÄ, K., AND VAIN, J. 1991. Real-time specification and modeling with joint actions. In *Proceedings of the 6th International Workshop on Software Specification and Design*. ACM Press, New York.

LEVESQUE, H. 1986. Knowledge representation and reasoning. *Ann. Rev. Comput. Sci. 1*, 255-287.

LIPECK, U. W. 1990. Transformation of dynamic integrity constraints into transaction specification. *Theor. Comput. Sci. 76*, 115-142.

MANNA, Z. AND PNUELI, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Vol. 1, *Specification*. Springer-Verlag, New York.

MILNER, R. 1980. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin.

MILNER, R. 1990. Operational and algebraic semantics of concurrent processes. In *Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier Science Publishers B.V., Amsterdam, 1201–1242.

MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. 1980. A language facility for designing interactive database-intensive applications. *ACM Trans. Database Syst. 5*, 2, 185–207.

MYLOPOULOS, J., BORGIDA, A., JARKE, M., AND KOUBARAKIS, M. 1993. Representing knowledge about information systems in Telos. In *Database Application Engineering with DAIDA*, M. Jarke, Ed. Springer, Berlin, 31–64.

PECKHAM, J. AND MARYANSKI, F. 1988. Semantic data models. *ACM Comput. Surv. 20* 3, 153–189.

PERNICI, B. 1990. Objects with Roles. In *Proceedings of the ACM/IEEE International Conference on Office Information Systems. SIGOIS Bull. 11*, 2–3, 205–215.

PNUELI, A. 1986. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*. J. de Bakker, W. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 224. Springer-Verlag, Berlin.

PNUELI, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*. IEEE, New York, 46–57.

REISIG, W. 1985. *Petri Nets*. Springer-Verlag, Berlin.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, N.J.

SAAKE, G. 1991. Descriptive specification of database object behavior. *Data Knowl. Eng. 6*, 1, 47–74.

SAAKE, G. 1993. *Objektorientierte Spezifikation von Informationssystemen*. Teubner, Stuttgart/ Leipzig.

SAAKE, G. AND JUNGCLAUS, R. 1992a. Specification of database applications in the TROLL-language. In *Proceedings of the International Workshop Specification of Database Systems*, D. Harper and M. Norrie, Eds. Springer, London, 228–245.

SAAKE, G. AND JUNGCLAUS, R. 1992b. Views and formal implementation in a three-level schema architecture for dynamic objects. In *Advanced Database Systems: Proceedings of the 10th British National Conference on Databases (BNCOD 10)*, P. M. D. Gray and R. J. Lucas, Eds. Lecture Notes in Computer Science, vol. 618. Springer, Berlin, 78–95.

SAAKE, G. AND LIPECK, U. W. 1989. Using finite-linear temporal logic for specifying database dynamics. In *Proceedings of the CSL '88 2nd Workshop Computer Science Logic*, E. Börger, H. Kleine Büning, and M. M. Richter, Eds. Springer, Berlin, 288–300.

SAAKE, G., JUNGCLAUS, R., AND EHRICH, H.-D. 1992. Object-oriented specification and stepwise refinement. In *Proceedings of the Open Distributed Processing*, J. de Meer, V. Heymer, and R. Roth, Eds. (*IFIP Transactions C: Commun. Syst. 1*), North-Holland, Amsterdam, 99–121.

SERNADAS, A. 1980. Temporal aspects of logical procedure definition. *Inf. Syst. 5*, 167–187.

SERNADAS, A., SERNADAS, C., AND EHRICH, H.-D. 1987. Object-oriented specification of databases: An algebraic approach. In *Proceedings of the 13th International Conference on Very Large Databases VLDB'87*, P. M. Stoecker and W. Kent, Eds. VLDB Endowment Press, Saratoga, Calif., 107–116.

THEODOULIDIS, C., WANGLER, B., BUBENKO, J. A., AND LOUCOPOULOS, P. 1990. A conceptual model for temporal database applications. SYSLAB Rep. 71, SYSLAB, Stockholm Univ., Stockholm.

VAN GRIETHUYSEN, J. J. 1982. Concepts and terminology for the conceptual schema and the information base. Rep. N695, ISO/TC97/SC5.

WIERINGA, R. J. 1990. Algebraic foundations for dynamic conceptual models. Ph.D. thesis, Vrije Universiteit, Amsterdam.

WIERINGA, R. J. 1991. A conceptual model specification language (CMSL Version 2). Tech. Rep. IR-248, Vrije Universiteit, Amsterdam.

WIERINGA, R. AND DE JONGE, W. 1991. The identification of objects and roles—Object identifiers revisited. Tech. Rep. IR-267, Vrije Universiteit, Amsterdam.

WIERINGA, R., JUNGCLAUS, R., HARTEL, P., HARTMANN, T., AND SAAKE, G. 1993. OMTROLL—Object modeling in TROLL. In *Proceedings of the International Workshop on Information Systems Correctness and Reusability IS-CORE '93*, U. W. Lipeck and G. Koschorreck, Eds. 267–283.

WING, J. M. 1990. A specifier's introduction to formal methods. *IEEE Comput. 23*, 9 (Sept.) 8–24.

Received May 1992; revised August 1993; accepted September 1994