Aggregation in a Behavior Oriented Object Model*†

Thorsten Hartmann
Ralf Jungclaus
Gunter Saake

Abt. Datenbanken, TU Braunschweig
Postfach 3329, W-3300 Braunschweig
e-mail {hartmann|jungclau|saake}@idb.cs.tu-bs.de

Abstract

TROLL is a language to specify information systems with dynamic behavior. Here, we elaborate on the specification of *object aggregation* in TROLL. We distinguish between two kinds of aggregation, static and dynamic aggregation. Static aggregation means that the composition of objects is described using predicates over constant properties. Dynamic aggregation means that we may alter the composition of objects by invoking special operations (*events*) that are implicitly defined for each dynamic complex object. Additionally, we describe the specification of disjoint complex as a means for structuring a specification. We introduce language features to describe object aggregation and give some hints towards their semantics.

1 Introduction

The first steps in an object-oriented approach to information system development concentrate on the abstract description of the relevant static and dynamic aspects of real-world objects (the Universe of Discourse, UoD) [Gri82, RBP+90, Boo90] (*conceptual modeling*). Very often, such objects are *composed* from other objects or have other objects as parts (*complex objects*). This is common e.g. in engineering [BB84, DGL87], in office systems [NT89] or in software development. In order to facilitate a natural description of these real-world objects, a formalism for object-oriented conceptual modeling should support the *aggregation* of objects.

Up to now, object aggregation has been neglected in the context of object-oriented programming (OOP) [Mey88, Weg90]. Aggregation in OOP has to be simulated using instance variables that refer to objects. This way, no clean separation between aggregation and association is possible [Kin89, RBP+90]. This is also the case in models for object-oriented databases that have their roots in object-oriented programming languages (like GemStone [BMO+89]).

Other data models support *structural* aggregation. Structural aggregation here means that complex types are constructed from simpler types – the elements of complex types are complex data structures [RS87, LRV88, SS90]. Complex types define the structure of complex objects.

Logical approaches to complex objects [Mai86, KL89] concentrate on structural aggregation. In some approaches, updates are addressed but the semantics are not completely defined yet.

Semantic data models [UD86, HK87, PM88] are particularly suited to describe the structures in a UoD in an abstract way. Semantic modeling and knowledge representation approaches in AI

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

†In: Proc. 6th European Conf. on Object Oriented Programming, O. Lehrman Madsen (editor), Springer, Berlin, LNCS 615, 1992, pages 57-77

grew out of similar motivations [BM86, ST89]. An essential feature of many semantic modeling approaches is the support of aggregation. Like (object-oriented) data models, semantic data models have a major drawback: the dynamic or behavioral issues are often left undefined [Kin89].

In *object-oriented analysis*, the problem of representing real-world systems composed from parts has been addressed [RBP+90, Boo90]. The cited approaches also address the modeling of behavior. The description of structural and behavioral aspects, however, is *separated*. Moreover, object aggregation is mostly addressed on the structural level only.

Our approach to conceptual modeling tries to overcome the mentioned drawbacks. The TROLL-language we introduce in this paper is a conceptual modeling language that integrates the description of structural properties of objects with the description of the *behavior* and *evolvement* of objects over time. TROLL is based on a formal model of objects. The language itself is logic-based in that it supports the description of relevant aspects of objects using assertions. TROLL supports abstraction mechanisms known from semantic data modeling like specialization, generalization, roles and aggregation. The main ideas of our approach are similar to the ones in [Wie90] and [KS91] although the solutions are quite different.

In this paper, we focus on the specification of complex objects in TROLL. Difficulties compared to structural aggregation arise because of the integration of behavioral aspects in TROLL. Special attention has to be paid to the behavior over time of complex objects and to the synchronization of the part's behavior. Other issues to be addressed are the encapsulation of the parts and behavioral autonomy of parts. TROLL supports three kinds of object aggregation: static aggregation, dynamic aggregation, and disjoined aggregation. Static aggregation is used when the composition is static and can be described declaratively in the conceptual model. Dynamic aggregation supports the dynamic composition of objects driven by actions. Both kinds allow to describe *shared* subobjects. Disjoint aggregation supports local components that are not shared.

In the next section, we introduce the basic characteristics of object-oriented conceptual modeling approaches in general and the TROLL-approach in particular. Furthermore, we formalize our concept of object and object inclusion. In Section 3, we introduce briefly the main features of the language TROLL. Section 4 then describes the different kinds of aggregation in TROLL in detail. Finally, we give some conclusions.

2 Basic Concepts

The basic concept of object-oriented design is the concept of *objects as units of structure and behavior* [SE91]. Objects integrate the structural and behavioral aspects of some 'system entities' into inseparable design units. An object has an *encapsulated internal state* which can be observed and manipulated exclusively through an object 'interface'. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e., methods), object-oriented database approaches put emphasis on the observable structure of objects (through attributes). We propose to support both views in an equal manner for the design of objects, i.e. object structure may be observed through attributes and object behavior may be manipulated through events, which are abstractions of methods.

Dynamic objects somehow *communicate* with each other. In object-oriented programming, this may be realized by method calling (i.e., procedure call) or by process communication mechanisms. For conceptual modeling, the concepts of *event sharing* and *event calling* serve as abstractions of those implementation-related techniques.

Objects are *classified* into *object classes and class types*. The term object class denotes a collection of existing objects which are conceptually treated as objects of the same kind (extensional classification), whereas an object type describes the possible instances of an object description (intensional description). Both concepts are commonly used in an integrated way, i.e. an object type defines the structure of a corresponding object class and vice versa.

Objects (and object classes) are often embedded into a *class hierarchy with inheritance*. Even if the concept of inheritance in object-oriented techniques seems to be relevant for most researchers, there seems to be still no agreement which kinds of inheritance should be supported by an object-oriented approach. We distinguish two sorts of inheritance relations, *syntactic inheritance* denotes inheritance of *structure or method definitions* and is therefore related to reuse of code (and to overriding of code for inherited methods). *Semantic inheritance* denotes inheritance of *object semantics*, i.e. of the objects themselves. This kind of inheritance is known from semantic data models, where it is used to model one object that appears in several roles in an application.

A concept which has attracted a lot of attention in object-oriented data models is the *composition of complex objects from objects.* In object-oriented design, the composition has to obey both structural *and* behavioral aspects. Along with the strict concept of object encapsulation, object composition can be modeled by *safe object import* (which in fact is semantic inheritance).

2.1 Formalization of Object Concept

An abstract object can be seen as an observable communicating process encapsulated by an access interface. An object may be observed by reading the current values of its attributes, whereas the object evolution is driven by local event occurrences — maybe by the initiative of the object itself (active objects) or as a result of a communication with some other objects (event calling). Both the values of attributes and the possible parameters of events are data values in the sense of abstract data types.

To formalize such an object concept, we start with giving a formal semantics to data values. Data values are elements of the carrier set of some abstract data type (ADT) [EM85]. An ADT is defined by a carrier set for data values along with data type specific functions on this set. We will not consider the specification of abstract data types in this paper.

The next step is to formalize the notion of object interface. Again we can take the notion of a *signature* from the ADT framework. An object signature describes the attributes as nullary functions into a certain domain (a data type), and describes events as functions with several parameters with a special event sort as domain. Such an object signature fixes the language symbols we can use if we talk in some formal logic about an object.

Let us now start with the formalization of the process component of an object description. Let X be a set of events.

Definition 2.1 *A set of events X is said to be an* OBJECT EVENT ALPHABET *iff $X = B_X \cup D_X \cup U_X$ such that $B_X \neq \emptyset$ and B_X, D_X, U_X are pairwise disjoint.* ■

An object event alphabet is composed of *birth, death* and *update* events. We require the set of birth events to include at least one event.

Within our framework, we do not want to have concurrency inside simple objects. Nevertheless, we also want to model composite objects containing other simple objects. For this reason, we must use the general concept of event snapshots. Event snapshots are sets of events concurrently occurring in *different* part objects.

Definition 2.2 *Let X be an object event alphabet. A subset $s \in 2^X$ is called a* SNAPSHOT *over X.* ■

Snapshots can be classified according to birth or death events of the composite object they contain. Birth snapshots contain at least one birth event and no death events, death snapshots contain at least one death event and no birth events.

Definition 2.3 *The set of* BIRTH ($\mathcal{B}_X$) *and* DEATH SNAPSHOTS ($\mathcal{D}_X$) *are defined as*

$$\mathcal{B}_X = \{s \in 2^X \mid \exists b \in B_X : b \in s, \forall e \in s : e \notin D_X\}$$

$$\mathcal{D}_X = \{s \in 2^X \mid \exists d \in D_X : d \in s, \forall e \in s : e \notin B_X\}$$

■

A snapshot – occurring in a complex object – may therefore contain several events occurring in *different* component objects. In this framework we use linear processes because we do not want to have intra-object parallelism in non-complex objects. One possible choice is to describe an object process as a collection of sequences of event snapshots, the so-called *object life cycles.* For the definition of life cycles we need the notion of finite and infinite life cycles.

Definition 2.4 *The set of* FINITE ($\mathcal{L}_X^*$) *and* INFINITE LIFE CYCLES ($\mathcal{L}_X^\omega$) *over* X *are defined as*

$$\mathcal{L}_X^* = \{s_1 \dots s_n \mid n \in I\!N,\ s_1 \in \mathcal{B}_X, s_n \in \mathcal{D}_X,\ \forall i \in I\!N, 2 \leq i \leq n-1 : s_i \notin \mathcal{B}_X \cup \mathcal{D}_X\}$$

$$\mathcal{L}_X^\omega = \{s_1 s_2 \dots \mid s_1 \in \mathcal{B}_X,\ \forall i \in I\!N, i \geq 2 : s_i \notin \mathcal{B}_X \cup \mathcal{D}_X\}$$

■

The set of life cycles over X is then defined as the union of the empty, finite and infinite life cycles. Infinite life cycles describe objects that will never be destroyed. The empty life cycle ϵ describes the fact that an object has not been created yet.

Definition 2.5 *The set of* LIFE CYCLES *over* X *is defined as* $\mathcal{L}_X = \{\epsilon\} \cup \mathcal{L}_X^* \cup \mathcal{L}_X^\omega$. ■

Now, we are able to define the behavioral component of an object model, determining it's *possible states*.

Definition 2.6 *A* PROCESS P *is a pair* (X, Λ) *where* X *is an object event alphabet and* $\Lambda \subseteq \mathcal{L}_X$ *is the set of permitted life cycles over* X *such that* $\epsilon \in \Lambda$. ■

Next we have to formalize the *observation of actual object properties.* We can do this by introducing an observation structure fixing the attribute values at each possible state of an object evolution. A possible formalization defines this observation structure as a mapping from prefixes of object life cycles to attribute value mappings. Let A be a set of attributes. Associated with each attribute $a \in A$ is a data type $\texttt{type}(a)$ that determines the range of the attribute. Let $\Pi(\Lambda)$ be the set of all finite prefixes of life cycles in Λ and $\texttt{carrier}(\texttt{type}(a))$ be the carrier set for the data type $\texttt{type}(a)$.

Definition 2.7 *An* OBSERVATION STRUCTURE V *over a process* P *is a pair* (A, α) *where* A *is a set of attributes and* α *is an observation mapping :*

$$\alpha \colon \Pi(\Lambda) \rightarrow obs(A) \subseteq \{(a, d) \mid a \in A \wedge d \in \texttt{carrier}(\texttt{type}(a))\}$$

A given prefix $\pi \in \Pi(\lambda)$ *is called* POSSIBLE STATE *of an object.* ■

An object model is defined by putting together a process and an observation structure over that process yielding a complete description of possible behavior and evolution over time.

Definition 2.8 *An* OBJECT MODEL $ob = (P, V)$ *is composed of a process* P *and an observation structure* V *over* P. ■

After having a semantics framework for single objects, we have to give semantics to object combinations and object synchronization. The semantics of putting objects together can be handled similar to process combination of process theory. The combination of objects is modeled by *structure-preserving* mappings between objects, the so-called *object morphisms* [EGS90, ES91]. A special case of object morphism is the *object embedding morphism.* It describes an embedding of an object ob_2 as an encapsulated entity into another object ob_1. This implies that the set of life cycles of ob_2 must be preserved and ob_1-events must employ ob_2-events to alter ob_2's state. The latter requirement is captured by the concept of *calling.* If an event e_1 calls an event e_2, then whenever e_1 occurs, e_2 occurs simultaneously.

Let us now define an object embedding morphism between objects ob_1 and ob_2. Assume, ob_2 should be embedded into object ob_1. Firstly the events of ob_2 are included in the set of events of ob_1. For the life cycle of the (composite) object ob_1 we state that if we constrain a life cycle to the events of the embedded (or part) object, we have to obtain a valid life cycle of the embedded object. Secondly the attributes of ob_2 are included in the set of attributes of ob_1. For observations of ob_1 projected to the attributes of the part object, we must obtain the same observation as for applying the observation mapping of part object ob_2 to a life cycle of ob_1 restricted to the events of ob_2.

Formally, this is expressed in the following definition:

Definition 2.9 *Let $ob_1 = (P_1, V_1)$ and $ob_2 = (P_2, V_2)$ be object models. Then ob_2 is embedded into ob_1, $ob_2 \hookrightarrow ob_1$, iff*

$$\begin{array}{lllll} (1) & X_2 \subseteq X_1 & \wedge & \forall \lambda_1 \in \Lambda_1 \exists \lambda_2 \in \Lambda_2 \colon \lambda_1 \downarrow\!\downarrow X_2 = \lambda_2 \\ (2) & A_2 \subseteq A_1 & \wedge & \forall \tau_1 \in \Pi(\Lambda_1) \colon \alpha_1(\tau_1) \downarrow A_2 = \alpha_2(\tau_1 \downarrow\!\downarrow X_2) \end{array}$$

■

$\lambda_1 \downarrow\!\downarrow X_2$ denotes life cycle restriction. All events that are not in X_2 are eliminated from the snapshots in λ_1 and only the sequence of *non-empty snapshots* is taken into account. A single arrow on observations like in $\alpha_1 \downarrow A_2$ only takes into account those attributes that are in A_2. Object embedding is the semantic basis for *static* and *dynamic aggregation* described in Section 4.

Static aggregation is also the semantic basis for other high-level language constructs describing specialization/generalization hierarchies, temporary roles of objects, interfaces, and relationships between objects. For a detailed description of these issues see [SJE91, JSS91a, JSH91, JHSS91, JSHS91, HJ91, SJ92].

With the concepts of snapshots and object embedding, we may now formalize calling and sharing of events, the basic constructions for modeling the communication between objects, especially the synchronization of the parts of complex objects.

Definition 2.10 *Let $ob_1 = (P_1, V_1)$ and $ob_2 = (P_2, V_2)$ be object models. Let ob_2 be embedded into ob_1, $ob_2 \hookrightarrow ob_1$, $e_1 \in X_1$ and $e_2 \in X_2$. Then e_1* CALLS FOR *e_2 iff*

$$\forall \lambda_1 \in \Lambda_1 \forall s \dot{\in} \lambda_1 : e_1 \in s \Rightarrow e_2 \in s$$

e_1 IS SHARED WITH *e_2 iff*

$$\forall \lambda_1 \in \Lambda_1 \forall s \dot{\in} \lambda_1 : e_1 \in s \Leftrightarrow e_2 \in s.$$

■

The notation $s \dot{\in} \lambda$ denotes the occurrences of snapshots s in the life cycle sequence λ. Event occurrences of e_1 are only possible, if the corresponding event e_2 occurs in the object ob_2. Event calling can be characterized as synchronous, asymmetric communication whereas event sharing denotes symmetric communication.

One may look at calling and sharing as an additional restriction to the life cycles of communicating objects. According to the definition of embedding, the set of life cycles of the communicating objects may be restricted, that is, objects embedded in an environment of other objects may *not* be able to behave 'freely', i.e. there may be life cycles of the components that are not allowed in the context of the composition. Calling and sharing is the semantic foundation for synchronization between object processes inside complex objects.

The step from single objects to object classes is done by introducing an *object identification* mechanism. Again, object identifiers are modeled as elements of some abstract data type. A class type defines a template for objects together with an identification space. An *object class* is a mapping from a set of object identifiers to actual objects.

Definition 2.11 *A* CLASS TYPE *$ct = (I, ob)$ consists of a set I of object identifiers and a (prototype) object model ob.* ■

The set I is the carrier set of an arbitrary abstract data type. Following this definition, each object of a class has the same object model as structure definition.

Object classes in our setting are sets of objects of a corresponding class type. Thus, with each object class we associate one corresponding class type. A class type describes possible extensions of a class definition in terms of sets of objects identified by identifier values $i \in I$. Therefore, possible class populations are subsets of $\{\langle i, ob \rangle \mid i \in ct.I \wedge ob = ct.ob\}$. Usually, the term *object class* is connected with the current state of a class variable, i.e. a concrete current object population with current states for included objects.

Definition 2.12 *A state of an* OBJECT CLASS $oc = (ct, O)$ *consists of a class type* ct *and a set of instances* O. *Instances* $\langle i, ob, \pi \rangle \in O$ *are described by an object identifier* $i \in ct.I$, *an object model* $ob = ct.ob$ *and a current state* $\pi \in \Pi(ct.ob.\Lambda)$. *The identifier* i *identifies instances in* O *uniquely.*
■

In this way, an object class definition is extensional. Note that we use the "dot notation" to refer to components of structures.

3 Basic Language Features for Object Specification

In this section, let us consider the specification of single objects in the language TROLL. The structure and behavior of an object is specified in a *template* which does, however, not describe the object itself. Let us, for example, give the template describing a bank account :

```
template account
  data types |BankCustomer|,money,bool,UpdateType,date;
  attributes
    constant Holder:|BankCustomer|;
    Balance:money;
    CreditLimit:money;
  events
    birth open(Holder:|BankCustomer|);
    death close;
    new_credit_limit(Amount:money);
    accept_TA(Type:UpdateType, Amount:money, Time:date);
    accept_update(Type:UpdateType, Amount:money);
    update_failed;
    withdrawal(Amount:money);
    deposit(Amount:money);
  constraints
    initial CreditLimit = 0;
  valuation
    variables m:money;
    [new_credit_limit(m)]CreditLimit = m;
    [withdrawal(m)]Balance = Balance - m;
    [deposit(m)]Balance = Balance + m;
    ...
  behavior
    permissions
      variables t,t1:UpdateType; m,m1,m2:money;
      { Balance = 0 } close;
      { not sometime(after(accept_update(t1,m1)))
          since last(after(update_failed) or
                     after(deposit(m2)) or
                     after(withdrawal(m2)))} accept_update(t,m);
      { sometime(after(accept_update(deposit,m)))
          since last after(accept_update(t1,m1)) } deposit(m);
```

```
{ sometime(after(accept_update(withdraw,m)))
    since last after(accept_update(t1,m1)) and
    (m <= Balance + CreditLimit)} withdrawal(m);
  ...
end template account
```

Attributes and events make up the *local signature* of a template. The signature of a template includes the data type signatures, the signatures of embedded objects (see below) and the local signature defined for the template and is the alphabet underlying the specification of objects.

In the `account`-template, we have the attributes `Holder` (constant), `Balance`, and `CreditLimit` which represent the relevant observable properties of accounts. Note that the codomain of the attribute `Holder` consists of *identifiers* of instances of class `BankCustomer` i.e. *references* not the object itself. The attribute `Holder` furthermore is constant, i.e. it is set at creation time of an instance and may never be altered in the whole life of the instance. The data type `money` may be defined by a language for the equational specification of abstract data types like ACT ONE [EM85].

The possible observable states may be restricted using integrity constraints. Such constraints are specified in the **constraints**-section of a template. After the keyword **initial**, we may state conditions to be fulfilled right after the creation of an instance. All other constraints are formulae of a future tense temporal logic, i.e. we allow the specification of dynamic integrity constraints in the sense of [Saa91, Lip90].

The events describe the state transitions of the object. The event `open` is declared to be the *birth-event* of instances. The occurrence of the `open`-event denotes the creation of an instance that is described by an `account`-template. Dually, the event `close` denotes the destruction of an instance. All other events are update-events that alter the state of an instance. Event parameters allow for data to be exchanged during communication between objects and to define the effects of event occurrences on attribute values.

These effects are specified in the **valuation** section of a template. The formulae in this section are formulae of a *modal* logic similar to the one described in [FS90]. Take for example the formula :

```
[new_credit_limit(m)]CreditLimit = m;
```

It denotes that immediately after the occurrence of the event `new_credit_limit` instantiated with a value `m`, the attribute `CreditLimit` has the value `m`. Note that we use an implicit frame rule stating that attribute values not being altered by valuation rules remain unchanged after the occurrence of an event.

The set of possible life cycles may be specified using several different formalisms. This is due to the fact that objects may represent a wide variety of UoD concepts which include *passive* objects like accounts and *active* objects like a clock or even objects performing query processing [JSS91b]. In TROLL we support the following formalisms for process specification:

- *Permissions* state enabling conditions for event occurrences. That is, an event may occur only if its enabling condition evaluates to true in the current state. Permissions are weak preconditions that take into account the actual parameters of events.
- *Obligations* state long-term goals that have to be fulfilled by each instance eventually. Obligations are completeness conditions for life cycles.
- *Patterns* describe chunks of life cycles explicitly using a (restricted) CSP-like notation [Hoa85]. Patterns are a more concise notation for parts of a life cycle that can be transformed into permissions and obligations.

Obligations and patterns are not described in this paper. In the `account`-template, we have only permissions. The rule

```
{ sometime(after(accept_update(withdraw,m)))
    since last after(accept_update(t1,m1))
    and (m <= Balance + CreditLimit)} withdrawal(m);
```

states that a `withdrawal`-event instantiated with an appropriate instance of the data type `money` may only occur if there has been occurring an `accept_update`-event for withdrawal since the last `accept_update` occurrence and the amount to be withdrawn is less or equal to the maximum amount that may be withdrawn in the current state. Note that the predicate **after**`(e)` evaluates to **true** in the state immediately after the occurrence of the event `e` only.

A template makes up the body of a *class type definition*. A class then can be seen as a container for instances of the associated type. In a class type specification we define an *identification mechanism* along with the template. An identification mechanism describes the set of possible external identifiers of instances. Each external identifier will be mapped to an immutable internal surrogate.

Let us give the specification of the object class `Account` as an example:

```
object class Account
  identification
    Number:nat;
  template account
end object class Account
```

A detailed description of the language features of TROLL may be found in [JSHS91]. We now want to discuss the various ways to specify aggregated objects in TROLL.

4 Complex Object Structures

Complex objects are a means to describe aggregation of subobjects to structured objects. This construction – known from semantic data models – is especially useful for the description of non standard applications.

Based on [BB84], we may distinguish two kinds of complex objects : *non-disjoint* complex objects which may have components being members of several aggregations and *disjoint* complex objects which are private to one aggregation.

Non-disjoint complex objects may share components. Thus, components are autonomous objects. In TROLL, we further distinguish two kinds of non-disjoint complex objects: dynamic complex objects and static complex objects. The former describing objects which change their composition at runtime, the latter describing objects with a composition determined at specification time. The introduction of two different constructions for dynamic complex objects is motivated by the possibility of checking the specification of static aggregation at compile time.

Disjoint complex objects do not share any components. This implies that components cannot exist outside the complex object. The components are *local* to the complex object. The composition is always static.

For disjoint and non-disjoint complex 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. Coordination and synchronization between the complex object and its components or between the components must be performed by communication. The semantic foundation for the specification of complex objects is object inclusion, i.e. safe object import together with communication via event calling and event sharing.

In the next three (sub)sections we describe the aggregation of simple objects to complex objects according to the above mentioned categories. We start with static aggregation which serves as the basic construct used to explain the semantics of dynamic and disjoint complex objects.

4.1 Static Aggregation of Objects

In section 2.1, we introduced the concept of object embedding. Object embedding is directly reflected in the TROLL-language. Embedding denotes an object model included in another object without violating any of its properties. Repeating the formal definition of object inclusion, this means that :

- Attribute values of the embedded object may only be altered by events local to it (encapsulation of update semantics).
- Interactions do not cause any violation of the life cycle specification of both objects (synchronization and encapsulation of object processes).

In other words: interaction between the components of a complex object may only be achieved through explicit communication.

The structure of static complex objects never changes. It is described using predicates over identifiers and constants. The composition therefore consists of all objects which satisfy the constraints given by the aggregation predicate (see the following example). By structure of a complex object we denote the syntactic structure, i.e., the signature of the complex object. Possibly there are components belonging to the aggregation that are not yet born. In the underlying semantics, these objects have an empty life cycle.

As mentioned above, TROLL supports a primitive language construct to define explicitly embedded (sets of) objects. This construct may be applied to single object specifications as well as to class type specifications.

As an example let us model a bank object including a clock object (with natural properties like `Time`, `Date` etc.) and a set of account objects with identification numbers from 100000 to 999999 :

```
object Bank
  template
    data types set(|Acct|),nat,UpdateType,money,date,|BankCustomer|;
    including
      instance Clock renaming Date by Today;
      A in Account where A.Number>=100000 and A.Number<=999999 as Acct;
    attributes
      TheAccounts:set(|Acct|);
    events
      birth establish;
      death close_down;
      open_account(Acc:nat,BC:|BankCustomer|);
      close_account(Acc:nat);
      process_TA(Acc:nat,Type:UpdateType,Amount:money, Time:date);
    valuation
      variables n:nat;
      [open_account(n)]TheAccounts = insert(Acct(n),TheAccounts);
      [close_account(n)]TheAccounts = remove(Acct(n),TheAccounts);
    behavior
      permissions
        variables n:nat; m:money;t:UpdateType;d:date;c:|BankCustomer|;
        { not in(TheAccounts) } open_account(n,c);
        { sometime after(open_account(n)) } close_account(n);
        ...
      interactions
        variables t:UpdateType; m:money; n:nat; d:date; c:|BankCustomer|;
        open_account(n,c) >> Acct(n).open(c);
        close_account(n) >> Acct(n).close;
        process_TA(n,t,m,d) >> Acct(n).accept_TA(t,m,d);
        ...
end object Bank
```

The including section consists of two including clauses. The first clause denotes the inclusion of a single object `Clock`. This object may be used to timestamp bank transactions. Inside the bank template the current date is referred to with the attribute symbol `Today`.

The second including clause denotes the inclusion of a proper subset of possible `Account` objects. The object variable `A` can be used to refer to objects of the object class `Account`. The component selector `Number` used in the selection condition is a *constant* property of account objects and may therefore be used in the selection condition. A reference to non-constant object properties like e.g. attribute values is neither allowed nor possible in the context of static aggregation.

The example has been extended to show possible interaction between the bank object and the accounts. Since the aggregated object is constructed from a set of accounts, we must use an operation $\texttt{Acct} : \texttt{nat} \rightarrow \texttt{|Acct|}$ to generate identifiers for included objects. The name `Acct` may be used inside the bank object as if it were defined as a global name. Note that $\texttt{|Acct|} \subseteq \texttt{|Account|}$, i.e. for all $X \in \texttt{|Account|}$ the aggregation predicate holds iff $X \in \texttt{|Acct|}$.

Identifiers are only *references* to objects, thus a second operation taking an object identifier and yielding the object itself is needed. Since there is no ambiguity – the accounts are *subobjects* of the bank – we could leave out the second operation, assuming that `Acct(n)` delivers an object instance. The current balance of the account number 123456 may be referred to with the expression: `Acct(123456).balance`.

As an example of communication between the complex object and one of its parts see the following calling expression :

```
open_account(n,c) >> Acct(n).open(c);
```

In this case for example the `Bank` event `open_account(n,c)` calls the `Account` event `open(c)` in component object `Acct(n)`. Thus the creation and destruction of `Account` instances is triggered by the `Bank` object. Nevertheless, the syntactic structure of the bank object is static. Static aggregation describes the *potential* object composition at specification time. Possibly there are component objects that have an empty life cycle.

The semantics of static object aggregation is obtained by embedding the subobjects into the complex object in the sense of definition 2.9 and 2.10. Constraints on the life cycles and observations of the components are reflected in this definitions. Event snapshots are atomic, that is either all events contained in a snapshot are allowed in the current state of the participating objects or none of them may occur. This way calling and sharing expressions imply a further restriction on the behavior of the component objects: some of their life cycles may not be possible in the context of the enclosing complex object.

4.2 Dynamic Complex Objects

The use of dynamic complex objects allows for high level description of object composition. Components may be specified as *single components* as well as *sets* or *lists* of components. As for static aggregation, the components of the dynamic complex object have a life of their own and may be shared by other objects as well.

With the specification of a dynamic complex object denoted with the keyword **components** we have implicitly defined events to update the composition. Additionally we have implicit attributes to observe the composite object. For example for a set of component objects we have events to insert and delete objects and an attribute to observe set-membership. For lists of objects we have events to append and to remove objects and for single objects there are events to assign and remove them as well as an attribute to test if the component is assigned.

This view of complex objects is operational instead of declarative. The composition is determined at runtime. We can say that static aggregation is *value based* whereas dynamic aggregation is *event driven*. Before we introduce how dynamic aggregation fits in our semantic framework of embedding, we will give an example of another bank specification, including the accounts as a set of components and additionally a single instance of the class `Person` denoting the manager of the bank. Some parts of the bank specification are omitted since we want to deal with the important parts of a component specification :

```
object Bank
  template
    data types nat,|BankCustomer|;
```

```
  components
    Manager:Person;
    Accounts:SET(Account);
  events
    birth establish;
    death close_down;
    open_account(Acc:nat,BC:|BankCustomer|);
    close_account(Acc:nat);
    ...
  behavior
    permissions
      variables n:nat; c:|BankCustomer|;
      { not Accounts.IN(Account(n)) } open_account(n,c);
      { sometime after(open_account(n,c)) } close_account(n);
      ...
    interaction
      variables n:nat; c:|BankCustomer|;
      open_account(n,c) >> Accounts(Account(n)).open(c);
      open_account(n,c) >> Accounts.INSERT(Account(n));
      close_account(n) >> Accounts(Account(n)).close;
      close_account(n) >> Accounts.REMOVE(Account(n));
      ...
end object Bank;
```

Initially, the `Bank` has no component. To manipulate the composition, the events `Accounts.INSERT(|Account|)` and `Accounts.REMOVE(|Account|)` are implicitly declared with the component specification and 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. For single object components, in this case `Manager`, events to assign and remove a manager (`Manager.ASSIGN(|Person|)`, `Manager.REMOVE(|Person|)`) are implicitly included in the signature of `Bank`.

For the behavior of the complex object the communication between the complex object and its components or between the component objects must be specified. See for example the clauses :

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

which state, that every time an account identified by the natural number `n` is opened, the event `open` is called in the corresponding object `Account(n)` and it becomes a member of the set of components. Note that the event `open_account` and therefore also the events `Accounts(Account(n)).open` and `Accounts.INSERT(Account(n))` in the `Bank` object may only occur if the expression **not** `Accounts.IN(|Account|)` instantiated with the appropriate `Account` identifier evaluates to true.

Dynamic complex 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 complex objects [JSHS91].

On the semantic level this is done in the following way: for each possible composition the corresponding semantic template is obtained by including the object signatures of all objects that are part of this complex object. Whenever the composition changes, a new semantic object is constructed in the same way and the old one can be forgotten. The observable properties of the unchanged components remain the same in the new semantic instance.

Consider for example the `Bank` object containing a set of `Accounts`. Just to explain the idea, we may construct a *semantic object class* `Bank_Semantics` identified by data type instances of `|Person|` and **set**`(|Account|)` :

```
object class Bank_Semantics
```

```
  identification
    Manager_ID:|Person|;
    Accounts_ID:set(|Account|);
  template
    including
      instance Person(Manager_ID) as Manager;
      A in Account where in(A.id,Accounts_ID) as Accounts;
    ...
end object class Bank_Semantics
```

The first including clause denotes the inclusion of the instance of `Person` associated with the value of the key attribute `Manager_ID`. The second clause includes the set of object instances from class `Account` for all object identifiers in the set `Accounts_ID`. The expression `A.id` denotes the identification of the object instance `A`. The operation **in** is defined for the (parameterized) data type **set**(`|Account|`).

For each semantic object of this class its structure is defined by embedding the `Person` object and all `Account`s that are elements of its (external) identifier, i.e. static aggregation. Changing the composition thus yields a new (semantic) object instance with a new internal identifier. We require this new instance to have the same observable state as the old instance. On the conceptual level the identity of the complex object remains unchanged.

The rest of the specification, especially the translation of the generated attributes and events is straightforward and therefore omitted. Note, that this construction is only used to describe the semantics of dynamic complex objects in terms of static aggregation as the basic mechanism to describe object composition.

4.3 Disjoint Complex Objects

Disjoint complex objects may not share components with other objects. The parts of disjoint object structures are strongly dependent on the composition. They cannot exist independently from the aggregation. Access to the subobjects is only possible via the enclosing object.

In TROLL, disjoint complex objects are described using *subtemplates*. Subtemplates are a means for structuring a specification. Conceptually, subtemplates describe local subobjects not usable outside the complex object. A subtemplate is a complete object description, specified either in the complex object or imported from a previously specified template.

The subobjects are specified together with attributes and events of the complex object and they have a local name. The name can optionally be supplied with one ore more parameters describing several subobjects with the same structure.

The structure (or signature) of a disjoint complex object is fixed, i.e. there is no way to change it as in the case of dynamic complex objects. There are close relationships to static object aggregation but in this case the components may not be used by other objects.

However – like for static aggregation – not all of the subobjects have to exist at a given point in time. Only the (sub)objects that have a non empty life cycle are existing. Thus, it is possible to describe *recursive object structures* within the framework described below. For example we can specify books with `section` subtemplates that in turn contain subtemplates for sections. Only (sub^n)sections where a birth event occurred are in the current set of existing subobjects.

Let us now turn to our example bank world. In this context we specify a class of banks each containing a set of divisions. The first step is to model the `Division` template :

```
template division
  data types |Person|;
  attributes
    Manager:|Person|;
    ...
  events
```

```
    birth openDivision;
    death closeDivision;
    setManager(|Person|);
    ...
  valuation
    variables P:|Person|;
    [setManager(P)] Manager = P;
    ...
end template division;
```

This specification fragment shows a division of a bank having an attribute `Manager` which can be changed using the event `setManager(|Person|)`. Divisions can be created with the birth event `openDivision` and destroyed with the death event `closeDivision`. In the second step the template specification of divisions can be used as subtemplates in a possible bank object :

```
object class Bank
  identification
    name:string;
  template
    data types string,|Person|;
    subtemplates
      Divisions(string):division;
    ...
    events
      newDivisionManager(string,|Person|);
      ...
    interaction
      variables D:string, P:|Person|;
      newDivisionManager(D,P) >> Division(D).setManager(P);
      ...
end object class Bank;
```

Inside the `Bank` template we have a possible `Division` template for each value of the internal name space, in this case for each possible string. We can use the components specified with the subtemplate by supplying the subtemplate name and its internal identification. For example, we may refer to the event `setManager` (of the subtemplate) writing :

```
  Division('dString').setManager(Person('pString'))
```

We go one step further and assume, that we have internally usable *template identifier spaces* as in the global case for objects. Then we have the data type `|Division|` defined as the tuple :

```
  |Division| = tuple(string)
```

We can refer to the component supplying an (internal) identifier of the data type, e.g. `|Division|` to the operation `Division` yielding the subobject itself. Again, the first form mentioned above : `Division('dString')` can be seen as an abbreviated notation since we must first generate an identifier and may then refer to the actual subobject. The special case of non-parameterized subtemplate names is implicitly included in this definition, assuming, that the identification space reduces to the simple name – an analogy to the reference to single objects that are uniquely identified using the object name.

As an example for the use of an internal identifier for divisions see the following clause (slightly modified, see above):

```
  interaction
    variables D:|Division|, P:|Person|;
    newDivisionManager(D,P) >> Division(D).setManager(P);
```

states, that an event denoting the advancement of a person P to be the new division manager for division D calls for the event setManager(P) in the subobject Division(D), assuming that the variables P and D are bound to appropriate object identifiers of |Person| and |Division|.

Note that different Bank instances can have the same names for their divisions. The names of the subobjects are local to the enclosing object as are the objects themselves. The (sub)object identifier data type is only visible inside the bank object.

Some comments to the proposed constructor for disjoint complex objects are in order. For non-disjoint complex objects the occurrence of birth and death events may be induced by several objects possibly sharing the same components. In the case of disjoint complex objects, the birth of a subobject may only be induced by the parent i.e. the enclosing object. The parent object can be seen as the root of the disjoint object structure.

The natural way to explain the semantics is static aggregation. Objects specified using subtemplates may be transformed to simply structured objects. Therefore we may generate an object class description employing the identification space of the original object together with the internal template identification. The template of this new object class is obtained from the subtemplate specification. We do not want to go into details of this construction but give a simple example. The Bank object may be transformed to the object classes Division :

```
object class Division
  identification
    Bank:|Bank|;
    SubObjectID:string;
  template division
end object class Division;
```

and a slightly modified object class Bank :

```
object class Bank
  identification
    name:string;
  template
    including D in Division where D.id.Bank = SELF.id;
  ...
end object class Bank;
```

Each object instance of the class Division may be shared by exactly one Bank class object only. This property is guaranteed with the including condition in the modified Bank object. The condition is another example for using predicates to describe the aggregation of static complex objects, i.e. using the basic mechanism of static aggregation to describe the semantics of high level language constructs.

5 Conclusions

In this paper, we described language features of TROLL to specify information systems. TROLL tries to integrate the description of structural and behavioral properties of objects. In particular we have elaborated on different kinds of object aggregation.

Static aggregation means that we describe the composition of objects by predicates over the static parts of templates meaning that aggregation is described declaratively. This approach directly bases upon object embedding, thus the specification of static aggregation can be analyzed at compile time.

Dynamic aggregation, however, emphasizes the operational description of object aggregation. The composition of aggregations may be altered by special events that are implicitly declared with a composite object. For the semantics, we have to employ a two-level translation using a different template for every possible composition. Whenever the composition changes, a new semantic object is created which then represents the complex object.

The aggregation of disjoint complex objects can be looked at as a structuring primitive of our language. Disjoint complex objects can be used to describe private parts of objects which can be manipulated only in the context of the aggregation. Like static aggregation, disjoint complex objects are directly based upon object embedding.

We think that it is possible to describe a wide variety of object compositions in our approach. Thus, TROLL may be employed for different types of information systems. In particular, our approach takes into account the temporal evolvement of objects. This issue has been neglected in the structural approaches towards object aggregation so far.

The motivation for providing a large number of (sometimes redundant) features in TROLL is convenience – we do not want to force the specifier to translate or transform a natural way of looking at things the way required by the specification language (s)he uses.

Acknowledgements

We are grateful to Cristina Sernadas who developed the basic features of TROLL with us. For many fruitful discussions on the language we are grateful to all other members of IS-CORE, especially to Hans-Dieter Ehrich, Amílcar Sernadas, José Fiadeiro, and Egon Verharen.

References

[BB84] Batory, B.; Buchmann, A.: Molecular Objects, Abstract Data Types and Data Models: A Framework. In: *Proc. VLDB'84*, Singapore, 1984. pp. 172–184.

[BM86] Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Management Systems.* Springer-Verlag, Berlin, 1986.

[BMO+89] Bretl, R.; Maier, D.; Otis, A.; Penney, J.; Schuchardt, B.; Stein, J.; Williams, E. H.; Williams, M.: The GemStone Data Management System. In: Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989, pp. 283–308.

[Boo90] Booch, G.: *Object-Oriented Design.* Benjamin/Cummings, Menlo Park, CA, 1990.

[DGL87] Dittrich, K.; Gotthard, W.; Lockemann, P.: Complex Entities for Engineering Applications. In: *Proc. Int. Conf. on the ER-Approach.* North-Holland, Amsterdam, 1987, 1987, pp. 421–440.

[EGS90] Ehrich, H.-D.; Goguen, J. A.; Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: deBakker, J.W.; deRoever, W.P.; Rozenberg, G. (eds.): *Proc. REX/FOOL Workshop*, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin, pp. 203–228.

[EM85] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (eds.): *Proc. ISCORE Workshop WS'91.* TU Braunschweig, Informatik Berichte 91-03, 1991, pp. 1–24.

[FS90] Fiadeiro, J.; Sernadas, A.: Logics of Modal Terms for System Specification. *Journal of Logic and Computation*, Vol. 1, No. 2, 1990, pp. 187–227.

[Gri82] Griethuysen, J.J. van: Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.

[HJ91] Hartmann, T.; Jungclaus, R.: Abstract Description of Distributed Object Systems. In: Tokoro, M.; Nierstrasz, O.; Wegner, P.; Yonezawa, A. (eds.): *Proc. ECOOP'91 Workshop on Object Based Concurrent Computing. Geneva (CH), 1991.* Springer, LNCS Series, 1991. *To appear.*

[HK87] Hull, R.; King, R.: Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Vol. 19, No. 3, 1987, pp. 201–260.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[JHSS91] Jungclaus, R.; Hartmann, T.; Saake, G.; Sernadas, C.: Introduction to TROLL – A Language for Object-Oriented Specification of Information Systems. In: Saake, G.; Sernadas, A. (eds.): *Proc. ISCORE Workshop WS'91.* TU Braunschweig, Informatik Berichte 91-03, 1991, pp. 97–128.

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.: Language Features for Object-Oriented Conceptual Modeling. In: Teory, T.J. (ed.): *Proc. 10th Int. Conf. on the ER-approach*, San Mateo, 1991. pp. 309–324.

[JSHS91] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Technical Report 91-04, TU Braunschweig, 1991.

[JSS91a] Jungclaus, R.; Saake, G.; Sernadas, C.: Formal Specification of Object Systems. In: Abramsky, S.; Maibaum, T. (eds.): *Proc. TAPSOFT'91, Brighton.* Springer, Berlin, LNCS 494, 1991, pp. 60–82.

[JSS91b] Jungclaus, R.; Saake, G.; Sernadas, C.: Using Active Objects for Query Processing. In: Meersman, R.; Kent, W.; Khosla, S. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK), 1990)*, Amsterdam, 1991. North-Holland, pp. 285–304.

[Kin89] King, R.: My Cat Is Object-Oriented. In: Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989, pp. 23–30.

[KL89] Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989.

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

[Lip90] Lipeck, U.W.: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, Vol. 76, 1990, pp. 115–142.

[LRV88] Lecluse, C.; Richard, P.; Velez, F.: O_2, an Object-Oriented Data Model. In: *Proc. ACM SIGMOD Conf. on Management of Data.* ACM, New York, 1988, 1988.

[Mai86] Maier, D.: A Logic for Objects. Technical Report CS/E–86–012, Oregon Graduate Center, 1986.

[Mey88] Meyer, B.: *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[NT89] Nierstrasz, O.; Tsichritzis, D. C.: Integrated Office Systems. In: Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989, pp. 199–215.

[PM88] Peckham, J.; Maryanski, F.: Semantic Data Models. *ACM Computing Surveys*, Vol. 20, No. 3, 1988, pp. 153–189.

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

[RS87] Rowe, L. A.; Stonebraker, M. R.: The POSTGRES Data Model. In: *Proc. 13th VLDB*, 1987, pp. 83–96.

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 6, No. 1, 1991, pp. 47–74. North-Holland.

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

[SJ92] Saake, G.; Jungclaus, R.: Specification of Database Dynamics in the TROLL-Language. In: Harper, D.; Norrie, M. (eds.): *Proc. Int. Workshop Specification of Database Systems, Glasgow, July 1991.* Springer, London, 1992, pp. 228–245.

[SJE91] Saake, G.; Jungclaus, R.; Ehrich, H.-D.: Object-Oriented Specification and Stepwise Refinement. In: *Proc. Int. IFIP Workshop on Open Distributed Processing, Berlin (D).* North-Holland, 1991. *To appear.*

[SS90] Scholl, M. H.; Schek, H.-J.: A Synthesis of Complex Objects and Object-Orientation. In: Meersman, R.; Kent, W. (eds.): *Proc. IFIP WG 2.6 Working Conference on Object-Oriented Databases (DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam, pp. 349–372.

[ST89] Schmidt, J. W.; Thanos, C. (eds.): *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989.

[UD86] Urban, S. D.; Delcambre, L.: An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models. In: *Proc. Int. Conf. on Data Engineering*, Los Angeles, 1986. ACM, New York, 1986, pp. 382–387.

[Weg90] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOP Messenger*, Vol. 1, No. 1, 1990, pp. 7–87.

[Wie90] Wieringa, R.J.: Equational Specification of Dynamic Objects. In: Meersman, R.; Kent, W.; Khosla, S. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam, 1991, pp. 415–438.