omTroll – Object Modeling in TROLL[‡]

R.J. Wieringa[*]
R. Jungclaus[§]
P. Hartel[§]
T. Hartmann[§]
G. Saake[§]

Abstract

We make an attempt to use concepts of the OMT analysis stage to develop formal object-oriented specifications in the TROLL language. The purpose is twofold: on the one hand, ambiguities, vaguenesses, etc. in OMT (and other OOA approaches) can be discovered and eliminated easier; furthermore, clear semantics can be given to modeling constructs. On the other hand, a popular notation on top of a formal OO specification language helps in making such a language more usable in practice. After introducing briefly TROLL concepts, we analyze the OMT models and identify corresponding TROLL concepts. Finally, we introduce a modified graphical notation based on this analysis.

1 Introduction

In the past few years, there has been considerable activity in the area of object-oriented (OO) analysis and of OO formal specification. OO analysis methods like OMT [RBP+91], life cycle analysis [SM92] and Objectory [JCJÖ92] have been proposed mainly with a background in software development practice. On the other hand, formal OO specification languages like Oblog [CSS89], FOOPS [GM87], TROLL [JSHS91], CMSL [FW93], TROLL*light* [CGH92], object-oriented extensions to Z [SBC92] are being developed mainly with a background in mathematical theory. These formal languages have a background in logic, algebra, set theory or category theory and provide a rigorous framework within which to specify OO systems. The OO analysis methods have a background in system development practice and do not reach the level of formality achieved by formal specification languages, but the formal languages are based on academic research and are only now beginning to be applied to a few practical software development projects. This suggests that a marriage between the two approaches to modeling and development could be fruitful.

There are a number of advantages to be gained by such a marriage:

- It could help in discovering and eliminating ambiguity, vagueness, etc. in the informal representation techniques of OO analysis methods.

[1]Faculty of Mathematics and Computer Science, Free University, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Email: roelw@cs.vu.nl.

[4]Abt. Datenbanken, TU Braunschweig, Postfach 3329, D–38023 Braunschweig, Germany. Email: {jungclau|hartel|hartmann|saake}@idb.cs.tu-bs.de

[‡]Partially supported by ESPRIT BRA WG 6071 IS-CORE and ESDI, S.A., Lisboa.

- Formalizing an OO model forces us to make clear what must be implemented and does not leave modeling decisions to be made by the implementors.
- Combining a practical analysis method with a formal specification language helps in making the formal specification language fit for use.
- Formal specification languages usually come with logical inference systems, and combining this with a practical analysis method makes powerful techniques like automated theorem proving and animation available for CASE tools used in analysis.

In this paper, we combine the analysis stage of Object Modeling Technique (OMT) [RBP+91] with the `TROLL` language [JSHS91, HSJ+93]. Analysis is also called *conceptual modeling* in this paper. The result of analysis is a *conceptual model* of the system to be implemented. In section 2, we give a brief introduction to the `TROLL` language. In section 3, we discuss OMT and point out a number of problems revealed by formal analysis of the techniques used in the conceptual modeling stage of OMT. We also suggest improvements for these shortcomings. The improved version of OMT is then illustrated in section 4 by means of the Automated Teller Machine (ATM) example known from the OMT book [RBP+91]. Section 5 winds up the paper with conclusions and topics for further research.

2 Object-orientation and `TROLL`

`TROLL` is a formal language for the specification of object systems. An object is considered to evolve in a discrete manner by the occurrence of events. Objects come into existence by the occurrence of birth events and may cease to exist by the occurrence of a death event. Attributes are used to represent observation about the current state of an object. Objects may interact by synchronizing on events with the possibility of exchanging informations while interact (similar to process models). Objects are related in various ways. A role describes a temporal specialization of an object having additional properties and/or restricted behavior. A specialization is considered to be a permanent role. Objects may be components of other objects which itself are called composite objects. In `TROLL`, the composition of composite objects may change dynamically. Separately defined objects may be connected through global interactions. Over objects, we may define views that contain only derived information over the current states of objects. As an example let us specify a simple account. Please note that we are using a new version of `TROLL` [HSJ+93] that has a new syntax but relies on the same concepts than the previous version [JSHS91].

```
object class Account
  identification
    ACctNo: (No);
  template
    attributes
      No:nat constant.
      Balance:money.
      Owner:|Customer|.
      Limit:money
        initialised 0.
    events
      open birth.
      close death
```

```
      enabled Balance = 0.
    deposit(Amount:money)
      changing Balance := Balance + Amount.
    withdraw(Amount:money)
      enabled (Balance - Limit) >= Amount;
      changing Balance := Balance - Amount.
    set_Limit(New:money)
      enabled Balance >= New;
      changing Limit := New.
  constraints
    (Balance + Limit) >= 0;
end object class Account;
```

The local properties of account instances are specified in the template. Attributes are typed and may be further characterized—here, the value of the attribute No is not allowed to change and the value of the attribute Limit is set to the value 0 upon creation of an instance. The data type |Customer| is the type of internal identifiers for objects of class Customer. It is, however, not possible to access the properties of a Customer instance when only the internal identifier is available.

Events are the possible state transitions. They may have parameters of certain types. In the events section, we also describe properties that are connected with the event. These properties are

- enabling conditions—only if the condition is fulfilled in the current state, the event can occur;
- effects of event occurrences on attribute values—these effects are specified under the keyword changing ; and
- local calls in terms of other events that are called when the event occurs.

We may specify constraints that determine the possible values of attributes and the admissible evolutions of attribute values over time. Furthermore, obligations to specify events that are obliged to occur (when certain conditions hold) and a restricted form of process specifications (to describe behavior patterns) can be given in a template specification.

The TROLL approach advocates a strict notion of encapsulation. By default, all information is local to an object and not accessible to other objects. By using explicit visibility specifications (called context in TROLL) we may import signatures of other objects and have then the right to refer to current properties of those objects.

A class specification implicitly defines a class object, for which class constraints, class attributes and class events may be defined if desired.

In TROLL, we may define *composite objects* which incorporate other objects as parts. In our approach, the composition of objects may change dynamically by the occurrence of special events. As an example consider a bank having a manager and a number of automatic teller machines (ATMs) as components. The following specification fragment gives the signature of the corresponding template:

```
object class Bank
  identification: BankName: (Name);
  template
    context Manager, ATM;
    components
```

```
      Mgr:  Manager;
      ATMs:  SET(ATM);
    attributes
      Name:string constant.
    events
      ...
end object class Bank;
```

The composition is controlled through implicitely defined events which denote e.g. the assignment of a manager (`Mgr_Insert(...)`), the insertion (`ATMs_Insert(...)`) or removal (`ATMs_Remove(...)` of an ATM components.

A very important mechanism to reflect taxonomic relationships is the *role concept*. A role defines a possible temporary specialization of an object (which is then said to play the role). A role may reveal more properties in addition to the already known ones or may restrict the possible behavior further. As an example take the role manager played by a person. We assume a specification of the object class `Person` to be given. The `Manager` role would be specified as follows:

```
object class Manager
  role of Person
  template
    ...
end object class Manager;
```

A specialization (*is-a*) is regarded to be a special role, played for the whole lifetime of an object. The classification depends on a constant property. An example is the object class `SavAccount` of which the instances are specializations of `Accounts`.

```
object class SavAccount
  role of Account where ...;
  template
    attributes
      InterestRate:real.
    events
      accumulate_Interest.
      ...
end object class SavAccount;
```

The final important modeling concept in `TROLL` is a relationship. A relationship comprises interactions between and constraints over separately defined objects. It is specified for classes, but defines relationships between instances. A fragment of a communication relationship between a customer and an ATM is the following one:

```
relationship Cust-ATM between ATM-User C, ATM A
  interaction
    ...
    C.enter_amount(m) >> A.amount_entered(m);
    ...
```

Please note that we use 'object variables' to refer to instances of a class.

3 OMT models and TROLL specifications

An OMT model of a system consists of an object model, a dynamic model, and a functional model. The object model represents the structure of the objects in the system, the dynamic model describes aspects of control in the system, and the functional model represents the operations performed by the system. We look at these three kinds of models in turn. Page references in the following, unless indicated otherwise, are to *Object-Oriented Modeling and Design* [RBP+91].

3.1 Object model

The object model represents the structure of the objects in the system. The diagram technique used is a variation and extension of ER representation techniques, called *object diagrams*. An object diagram is an undirected graph in which the nodes represent object *classes* and the edges represent object *associations*. Class instances are called *objects* and association instances are called *links*. The class nodes are labeled by the class name, and by the names and signatures of the *attributes* and *operations* applicable to class instances. In TROLL terms, the operations are events, but we will use OMT terminology in this section.

3.1.1 Encapsulation in OMT and TROLL

In our opinion, encapsulation consists of a number of independent statements. First, it says that classes are specified in an **implementation-independent** way, because implementation is not playing any role in a conceptual model.

A second element of encapsulation is that attributes and operations are **local** to an object. This means that we can always talk of the attributes and operations *of* an object. OMT and TROLL, like all other object-oriented modeling approaches, use this kind of localization.

The third meaning component of encapsulation is that there is a restricted **visibility** of the attribute values and operations of an object.

The meaning of encapsulation is not defined for the analysis stage of OMT—only when it comes to implementation and to review basic notions of object-orientedness, it is mentioned. OMT basically does not restrict the visibility of properties. Operation descriptions (which are part of the Functional Model in OMT) are allowed to access properties of other objects and even side-effects are not forbidden (pages 130ff). Constraints can refer to every element of an object model, including other objects and their attributes. Only when it comes to design, OMT states restrictions: Operations work on local *attributes*, that should be hidden from other objects. Operations are implemented by *methods*. A method can only access the attributes of the object that executes it. Attributes of other objects should be accessed by calling operations of those objects, and those other objects must be related to the current object through links. A method is not allowed to traverse a link from a neighbor to a third object, because then object access becomes too dependent upon the overal structure of the model (page 249f). A method that implements an operation can only access the attribute values of the object that executes the method, and can only call the operations of objects directly linked to it. Stated in terms of specification instead of implementation, this restriction says that the axioms that define the meaning of an operation should only refer to attributes and operations declared in the class specification itself and/or to operations declared in a class specification reachable by traveling at most one association in the object model. This is different from the object-level description above, which describes the restriction in terms of links rather than associations. The difference is that we may travel many links and still travel only one association.

In TROLL, a much more restricted view of encapsulation is followed. Visibility has to be explicitly specified. This is due to the fact that an identifier in TROLL does not permit to access the local properties of a referenced object. Only a visibility relationship makes accessible the properties of other objects. We may navigate along visibility relationships transitively. We achieve a higher level of locality through these visibility rules in each stage TROLL is used. All axioms that define the structure and behavior of an object refer only to local properties. Since only axioms are used to define objects in TROLL, structure and behavior of objects is defined in an abstract way.

In a combined OMT/TROLL approach, we will use the TROLL rules for visibility of attributes and operations.

3.1.2 Mapping OMT concepts to TROLL concepts

Table 1 shows how the concepts in the OMT object model are mapped to TROLL concepts.

OMT	OMTROLL
Class	Class
Class attributes	Class attributes
Generalization	Generalization
Multiple inheritance	Multiple inheritance
—	Disjoint and exhaustive specializations
—	Roles
Object	Object
Association	(Relationship) Composite object
Ordered associations	Use list constructors
Constraints between associations	Constraints inside composite objects
Qualifier	Key constraint
Multiplicity	Composite Object plus a constraint
Aggregation	Composite Object
Link attribute	Composite Object
Constraints on attributes	Constraints on attributes
Attribute	Attribute
Derived attribute	Derived attribute
Derived class	Interface declarations
Derived associations	If non-recursive: Hierarchical interface
Operation	Event
Propagation of operations	Object interaction/event calling

Table 1: Mapping the object model to TROLL.

Let us give here some small examples for translating OMT-Diagrams into TROLL specification fragments. The resulting TROLL specifications are not complete since the OMT object diagrams only cover the structural aspects of the specification.

A first example is the specialization hierarchy for accounts depicted in Figure 1.

The resulting TROLL specification fragment for the specialization SavAccount is the following:

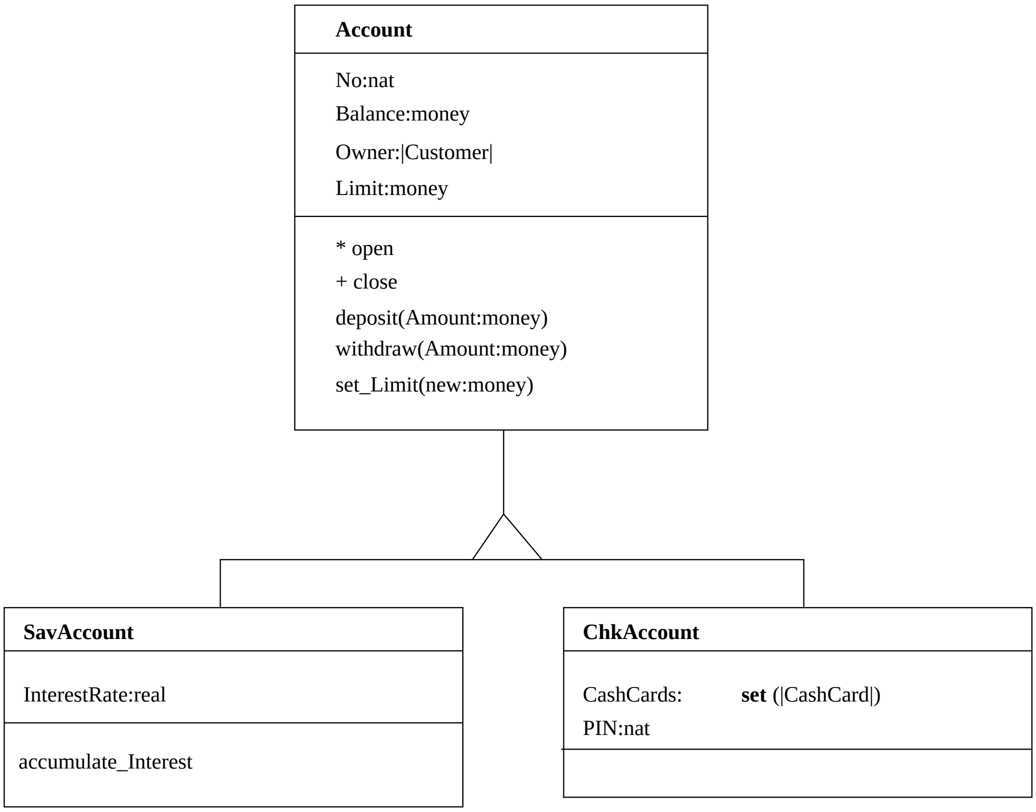

Figure 1: Account-Hierarchy

```
object class SavAccount
  role of Account where ...;
  template
    attributes
      InterestRate:real.
    events
      accumulate_Interest.
      ...
end object class SavAccount;
```

A second example is the diagram for the composite object **Bank**. A bank is modeled to have as components the accounts it maintains and the ATMs it owns. A multiplicity constraint is that at least one account must be part of the bank. The diagram is given in Figure 2.

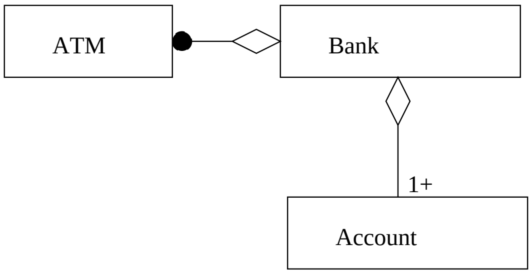

Figure 2: Composite Object Bank

The diagram is translated into the following TROLL fragment of a composite object specification:

```
object class Bank
  identification
    BankName:  (Name);
  template
    context Account,ATM;
    components
      Accts:  SET(Account).
      ATMs:  SET(ATM).
    attributes
      Name:  string constant.
    events
      ...
    constraints
      Accts_Card >= 1;
    ...
end object class Bank;
```

3.2 Dynamic model

The dynamic model represents aspects of the system concerned with control, including time, sequencing of operations, and interaction of objects. The representation technique used is the state chart [Har87, Har88]. For each class whose instances have interesting behavior, a state chart is made that describes all or part of the behavior of one object of a given class (page 110).

3.2.1 Events, guards, actions and activities

A state chart shows how events may change the state of a system. An *event* is something that happens instantaneously at a point in time. An event may have a *guard*, which is a Boolean function on attribute values of objects in the system that acts as a necessary precondition for the state transition associated with the event (page 91). If the guard evaluates to false, the state transition associated with the event cannot occur. A guard may contain conditions on attribute values of any object, not just the object in whose life it is evaluated (page 99).

It is *not* stated that an event is a local state change of an object. Rather, events seem to be instantaneous happenings that are outside the control of the object, which is an idea from Structured Analysis for real-time systems [WM85]. An event occurs in the life of *another* object or possibly outside the system. This explains the manner of speaking in "a guarded transition fires when its event occurs, but only if its guard condition is true" (page 91). In other words, an event can occur when the guard is false, but then the transition does not occur. In TROLL terms, an event is an attempted communication with the object that originates from another object or from outside the system. However, we will follow OMT terminology for the moment.

Guards are written in square brackets following the event name in a state chart. To bring out their use a preconditions, we will write them in curly brackets preceding the event name (figure 3).

Guards may also be used to label state transitions with which no event is associated (page 96). These guards, however, are no preconditions but *tests* to determine the next state after something happened. Since these tests are not connected to some external event, they are really *internal* events that make the behavior of the object appear from the outside as if it were nondeterministic. We propose to proceed in this way in TROLL—

we have to declare additional events and then can control state transitions by enabling conditions. This results in improved clarity of specifications.

When an event occurs, it can trigger an action, which is an instantaneous operation. After arrival in the next state, an activity can be triggered, which is an operation that has duration. This activity will terminate prematurely if an event causes a transition to another state. Activities represent "real-world accomplishments" (page 454), and actions represent the response of the system to a real-world event. However, actions may also perform an internal, control function of the system and then do not correspond to real-world events at all (page 93).

In the combination of OMT with `TROLL`, we will drop the concept of activity from the dynamic model, for any activity can always be represented as a process consisting of atomic, instantaneous operations, itself represented by a state chart (page 95). We return to this when we discuss roles below. Premature termination of an activity by the arrival of an event can also be specified explicitly. One reason for this simplification is that this is the way activities are specified in `TROLL` anyway; another reason is that it makes the dynamic model easier to read if all operations in it are atomic.

A second simplification that we will apply is that we will ignore the distinction between events and actions. What is an event in an object o_1 is just an action in another object o_2, that attempts to communicate with o_1 by triggering an action in o_1. Object interaction is discussed in detail below. The graphical representation of transitions can now be simplified to the one in figure 3.

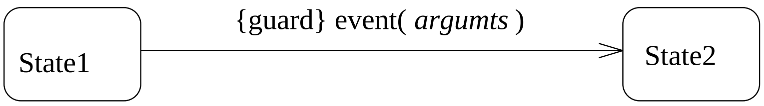

Figure 3: Transitions in a combined OMT/`TROLL` model.

Our adaptation of state charts is a simplification that does not sacrifice expressive power. This contrasts with objectcharts [BACH90], which extend statecharts by showing the attributes that are in scope in each state.

3.2.2 Object interaction in OMT

There are several communication mechanisms in OMT:

- The guard of a transition in one object may depend upon state of another object (page 99).

- An action of one object can consist of sending an event to one or more other objects (page 103); this is multicasting. It is indicated in the dynamic model by means of a dotted arrow from sender to receiver, or simply by the fact that the action name of one transition equals the event name on one or more other transitions. These other transitions must appear in other state charts.

- An operation can trigger operations in other objects, which is called *propagation* (page 60). It is shown in the object model by means of an arrow from the sender to the receiver.

The last two forms of object interaction duplicate one another, and we reduce them to one by identifying them both with event calling in `TROLL`. Event calling will not be represented in the object model, nor in the dynamic model, but in a third model, which we call the *communication model*, that shows what the communication actions taken by an event are. Figure 4 shows the general form of an interaction diagram. An event executed by

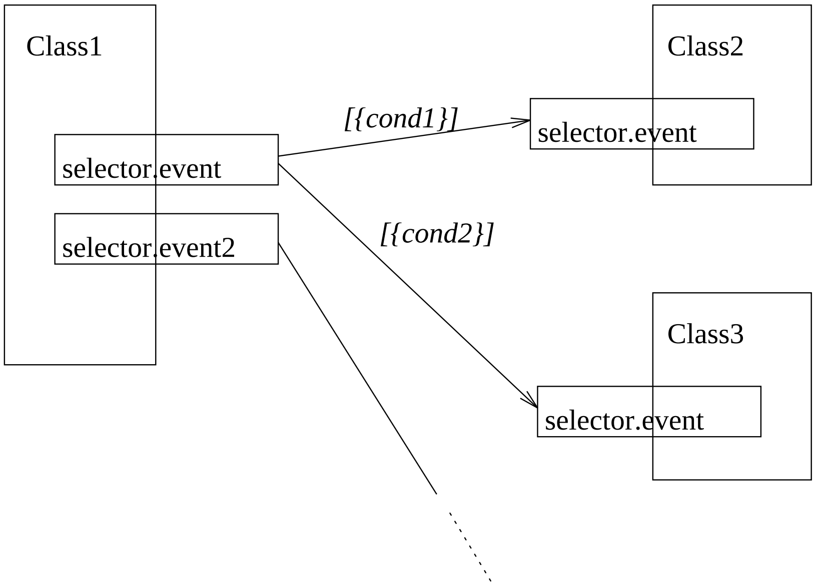

Figure 4: General Form of Interaction Diagrams

an instance of Class1 can call events of instances of other classes. This event calling is subject to the precondition $\{cond\}$.

Communication diagrams resemble event flow diagrams (page 175). OMT however uses event flow diagrams only as a heuristic to find the dynamic model and does not use it as "official notation" to represent communication.

In OMT, concurrent receivings of events by one object may cause serious problems if the effects conflict (*race condition*, page 103). In `TROLL`, this is not a problem since concurrent events in an object are possible only if the effects do not conflict. Again, we will use the `TROLL` approach here to remove another source of impreciseness.

3.2.3 Inheritance and roles

The taxonomic hierarchy is represented in the object model and allows single and multiple inheritance. This can be carried over unchanged to a combined OMT/`TROLL` model. As shown in Figure 5, we add the possibility to define pairwise disjointness of subclasses (represented by a d in the triangle connecting subclasses with superclass) and exhaustive subdivision of a superclass in (overlapping or disjoint) subclasses (represented by an e). A *de* subdivision is a partition of the superclass into subclasses.

There is a taxonomic hierarchy of events (page 98), that is independent from the taxonomic hierarchy of classes (page 111). We think this independence is not desirable and tie the event specialization closer to class specialization.

In `TROLL`, we may add properties to a template specification—we may restrict enabling conditions for events, we may add further effects of events and interactions when specializing a template specification as long as *we do not violate the properties of the base template*. Thus, we may only have monotonic inheritance in `TROLL`. In OMT, only weak statements are made that suggest the use of monotonic inheritance.

OMT does some hand-waving in the direction of the inheritance of a state chart by a subclass. The generalized state chart is required to be some kind of projection of the specialized state chart (page 111), but it is not made precise what this means. We plan to give a formal inheritance condition for object behavior in a forthcoming paper.

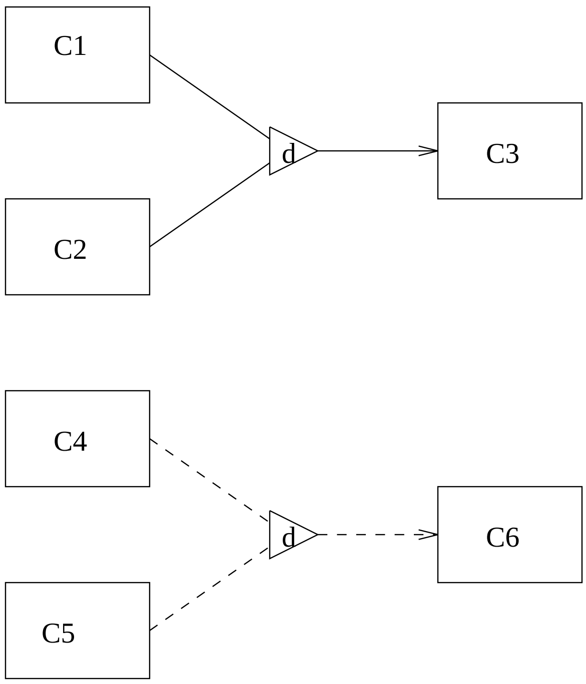

Figure 5: Representation of roles and specializations

In addition to the static taxonomic structure of figure 5, there is a dynamic taxonomic structure of *roles* in TROLL. Class C_1 is a role of class C_2 if each instance of C_1 is an instance of C_2, and for each instance of C_1 there are possible states where it is a C_2 but not a C_1. Less cryptically, $STUDENT$ is a role of $PERSON$ because, if John is a $STUDENT$, then John is a $PERSON$, and there are possible states John is not a $STUDENT$ but still a $PERSON$. Roles in TROLL are the basic concept for taxonomic hierarchies. Static specializations are modeled as permanent roles—thus, the specialization concept is a derived concept. Roles are not in OMT; instead, it is suggested to model temporary differences between objects by states (page 111) which is not considered sufficient by us.

Roles in TROLL provide the possibility of class change. Class change is not possible in OMT, but there is a facility that resembles it: An activity associated with a state may be itself specified as a state chart (page 95). This is the reason why we could drop activities from the dynamic model. We now add the ability to represent roles in the dynamic model by means of dashed lines, as shown in figure 5. Static and dynamic subclassing must not be mixed in one subclassification, i.e. we cannot have one triangle in the diagram connected to dashed and solid lines. However, we may specify several alternative subclassifications of one class, some of which are static, and others of which are dynamic. Note that the representation of taxonomy and roles is part of the object model, not of the dynamic model.

3.2.4 Mapping the dynamic model to TROLL

Table 2 gives correspondences between OMT and TROLL concepts that we will use for mapping graphical notations to TROLL.

As a small example consider the specification of the dymanics of an account. An account has quite liberal behavior of which the state diagram is shown in Figure 6.

A state diagram mainly introduces enabling conditions. The fragment of the corresponding TROLL specification which can be derived from this diagram shall be repeated here:

OMT	TROLL
Event	Event
State	State
Action	Event with event calling
Sending an event to another object	Event calling
Propagation of operations	Event calling
Guard	Permission
Activity	Role
State generalization (nesting)	Role
Concurrent subdiagrams	Parallel composition
Splitting of control	Parallel composition
—	Obligations
—	Active/Passive Events

Table 2: Mapping the dynamic model to TROLL.

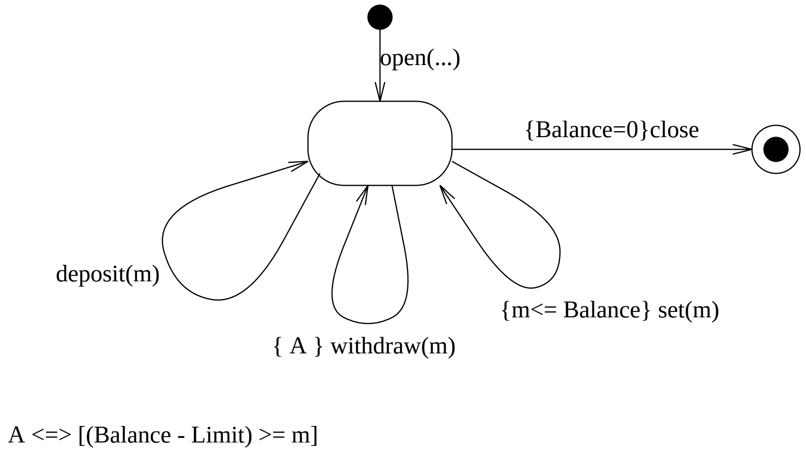

Figure 6: Account-Dynamics

```
events
  open birth.
  close death
    enabled Balance = 0.
  deposit(Amount:money)
    ...
  withdraw(Amount:money)
    enabled (Balance - Limit) >= Amount;
    ....
  set_Limit(New:money)
    enabled New <= Balance;
    ...
```

3.3 Functional model

The functional model shows the meaning of operations by showing how values are transformed by the system. The representation technique used is the data flow diagram (DFD).

A DFD is a directed graph whose nodes are *processes*, *data stores* or external entities, called *actors*, and whose arcs are *data flows*. Processes in a DFD may be specified themselves by DFD's, which creates a hierarchical structure of DFD's. A process not specified by another DFD must be specified by an informal description, like a decision tree or structured English. Each process may correspond to one or more operations from the object model, and each operation may correspond to one or more processes.

Each data store is a set of records that contains some or all of the attribute values of objects of a class. There need not be a correspondence between classes and data stores, and in general there is not.

Each actor corresponds to an object that can generate events to which the system must respond; the response of the system consists of processing the data flow generated by the actor until data stores are updated and/or an output data flow is produced.

The functional model is used to specify the meaning of operations by saying what processing a system does in response to external events. The functional model has been rightly criticized by Hayes and Coleman [HC91] as being too operational and imprecise. The specification of the meaning of an operation should be a declarative, formal specification of the effect of the event. In TROLL we have such a declarative specification mechanism of event effects, and consequently *we will drop the functional model.*

A second argument for dropping the functional model, in addition to it being vague and imprecise, is that it takes a completely different perspective from the other two models. The object model and the dynamic model partition the system into objects, which have local state and behavior. The functional model, by contrast, separates state and behavior by partitioning the system into data stores and processes. A data store may contain a fragment of the state of all instances of a class, and a process may perform part or all of the processing of an operation. This difference in perspective makes the relationship between the functional model and the other two models hard to understand.

4 Improved Notation – An Example

In this section we want to introduce the notation of OMTROLL using the by now well known Bank example. The section is divided into subsections introducing the structural model, the dynamic model, and the interaction model.

4.1 Strutural Modeling

The structural model of OMT basically is an extended Entity-Relationship diagram—therefore, it matches the structural abstruction mechanisms of TROLL quite well. The differences between OMT and OMTROLL are the following:

- Roles are the basic mechanism to express a taxonomy between objects. Specializations are derived and are represented by dashed subclass symbols.
- Taxonomic relationships can be constrained to be exhaustive or disjoint. By default, they are not disjoint (in contrast to OMT).
- For associations, we dropped ordering and qualification.
- Communication relationships are depicted by a flash between the related class boxes.

The following example models the structure of our ATM-Bank example (Figure 7). Any constraints on the structure have to be stated in terms of TROLL constraints.

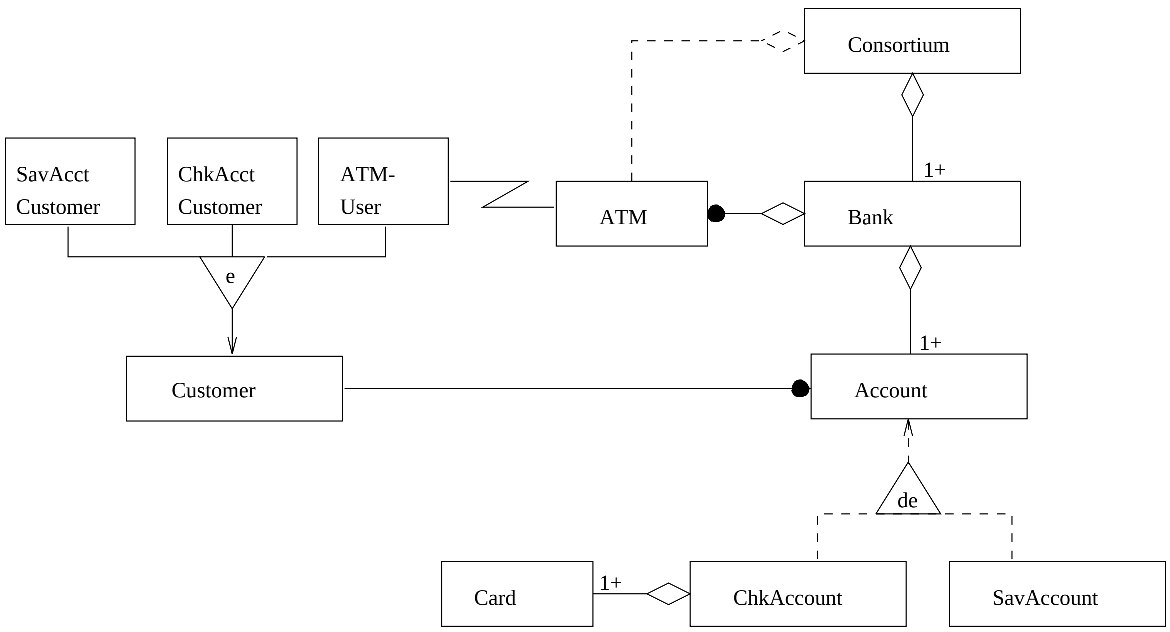

Figure 7: Object Model

4.2 Dynamic Modeling

For the dynamic model, the notation of OMT is kept without major changes. The following figures show further examples in addititon to the specification of the behavior of accounts (Figure 6). Figure 8 describes the more complex behavior of an ATM.

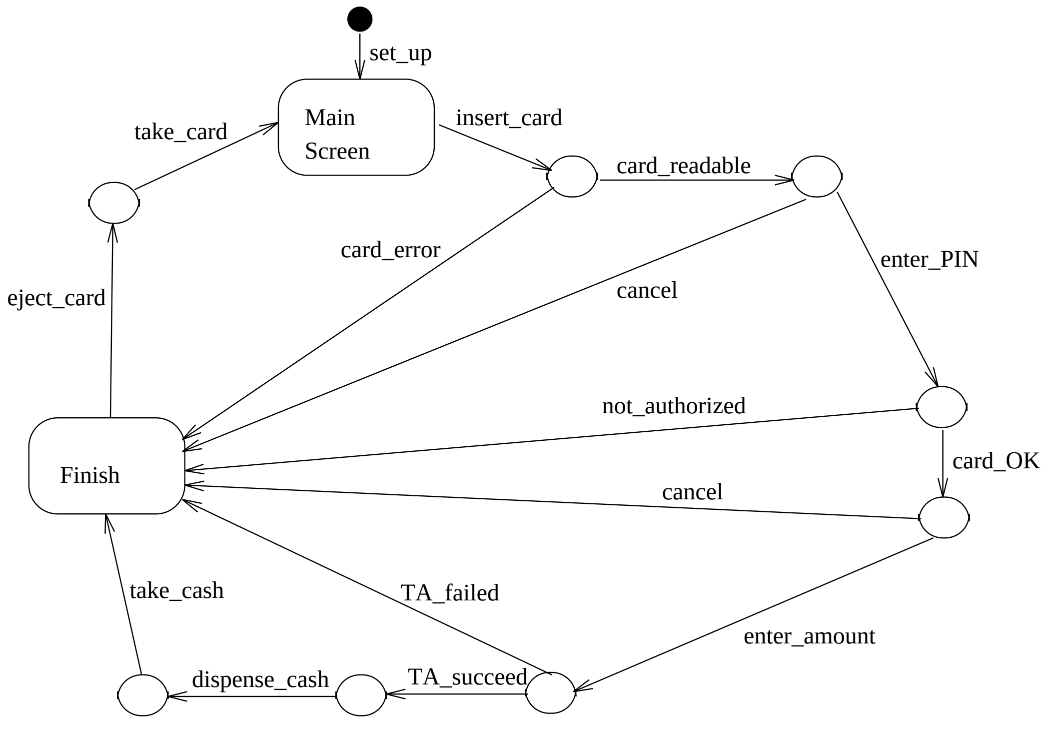

Figure 8: State Diagram for class ATM

In Figure 9 we depict the behavior of composite objects of class Bank. Here we explicitly have concurrency in terms of two processes in the same object that are not synchronized.

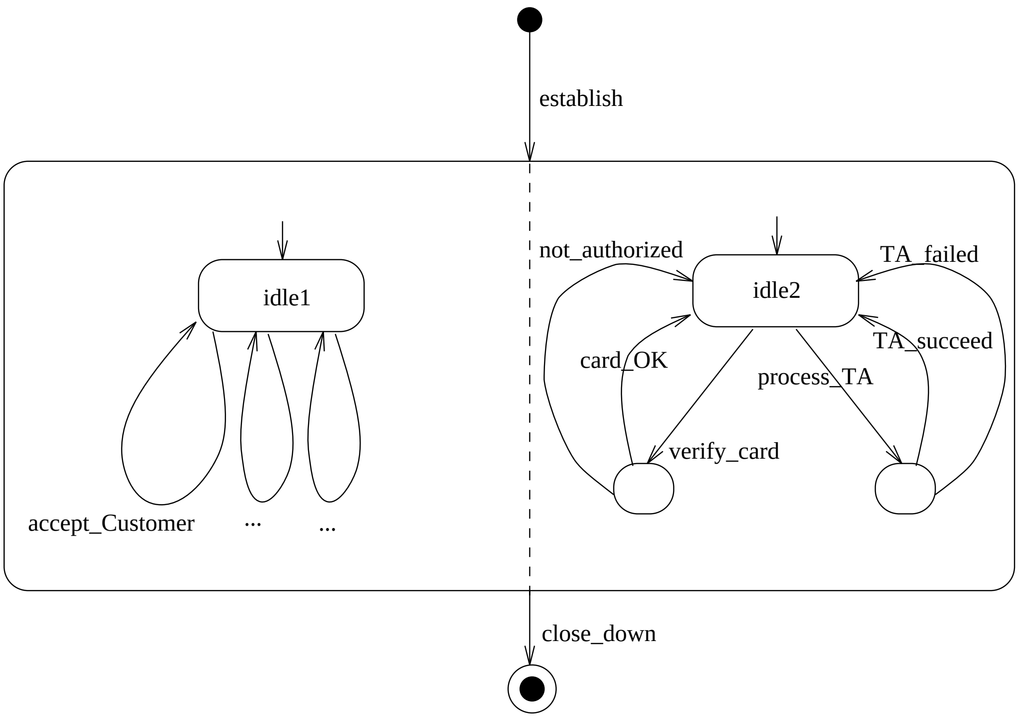

Figure 9: State Diagram for class BANK

4.3 Interaction Modeling

Interaction diagrams are not part of OMT although simplified versions are mentioned as helpful auxiliary diagrams (Figure 8.19 on page 175). The general notation of interaction diagrams in the OMTROLL notation is given in Figure 4. We may have multicasting (one event calls several others in several other objects) and interactions may be guarded by conditions ($\{cond\}$)—when the condition does not hold in the current state, the interaction does not take place.

As an example consider the diagram specifying the interaction between objects in our example (Figure 10).

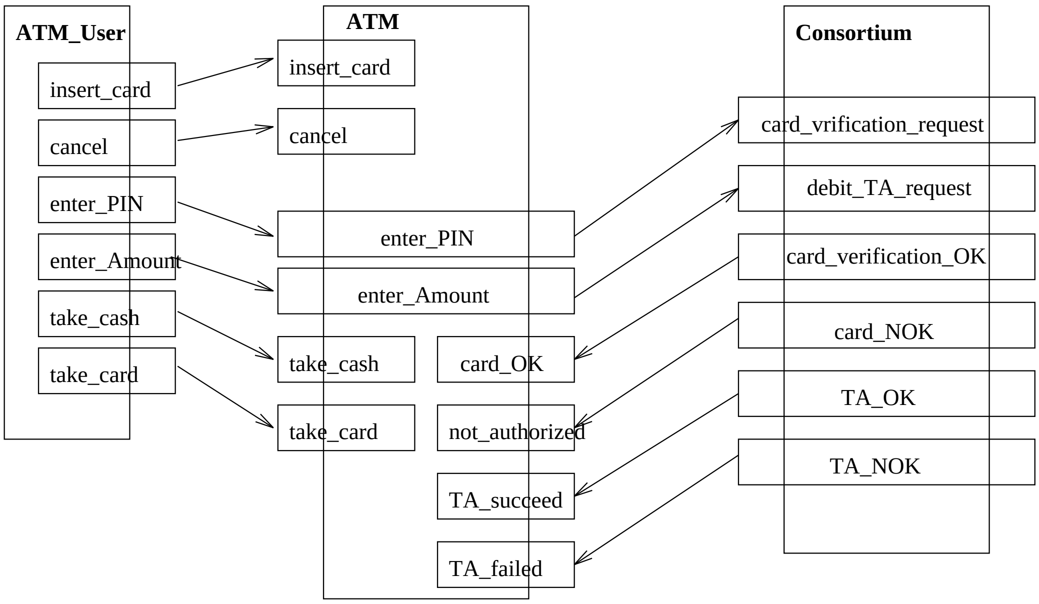

Figure 10: Interaction Diagram

Separation of the representation of communication from the representation of behavior simplifies the model. The distinction between communication and behavior corresponds to Milner's distinction between static and dynamic operators in CCS, which represent the way systems are connected and the behavior of systems, respectively [Mil89].

5 Conclusions

We simplified OMT graphical techniques in a number of respects:

- The functional model is dropped, because it is imprecise and has a different philosophy from the other two models. The effect of events on objects is specified declaratively in TROLL.
- We added interaction diagrams, which make the communication structure of the model explicit and allows one to specify multicasts as well as preconditions for communications.
- We dropped the representation of propagation of operations from the object model, and we dropped the representation of event calling from the dynamic model. This does not reduce expressive power, because interaction can be represented by interaction diagrams.
- Activities have been dropped, since they violate a basic atomicity principle for system transactions. They can be replaced by statecharts that consist of atomic events only.
- We added the concept of a role, which is a process that an object can perform only when it is in a certain state. This is a more systematic concept than that of activity and does not violate the atomicity principle.

Some minor adaptations of OMT techniques are:

- Visibility rules for attributes have been replaced by TROLL visibility rules.
- We dropped qualification and ordering of associations.
- We separated guards from tests.
- We dropped the distinction between actions and events.
- We added the possibility to specify disjointness or exhaustiveness of specializations graphically.

Returning to our starting point, this paper illustrates that combining a formal specification technique with a practical analysis method leads to an improvement of both. We have been able to simplify OMT techniques while at the same time adding expressive power, such as the ability to specify roles and to represent mutlicasts explicitly. We think the increased simplicity of the method makes it easier to use.

In the future, we plan to look at other analysis methods, such as life cycle analysis [SM92] and Objectory [JCJÖ92].

References

[BACH90] S. Bear, P. Allen, D. Coleman, and F. Hayes. Graphical specification of object-oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming*, pages 28–37, October 1990.

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

[CSS89] J.F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 User's Manual, version 2.3.* Instituto Superior Técnico, Lisbon, May 1989.

[FW93] R.B. Feenstra and R.J. Wieringa. CMSL 3.0: a language for describing conceptuel models. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[GM87] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, 1988.

[HC91] F. Hayes and D. Coleman. Coherent models for object-oriented analysis. In A. Paepcke, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 171–183, 1991. Sigplan Notices, 26(11).

[HSJ+93] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. `TROLL`–2 report. Informatik-bericht, TU Braunschweig, 1993. *In preparation.*

[JCJÖ92] I. Jacobson, M. Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Prentice-Hall, 1992.

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

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[SBC92] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Springer, 1992.

[SM92] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States.* Prentice-Hall, 1992.

[WM85] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Three volumes.