TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

INFORMATIK-BERICHTE

93–08

Abstract Specification of Object Interaction

Thorsten Hartmann
Gunter Saake

Abt. Datenbanken
Techn. Universität Braunschweig
Postfach 3329
D–38023 Braunschweig, Germany
(E-mail: {hartmann|saake}@idb.cs.tu-bs.de)

Braunschweig
December 93

Abstract Specification of Object Interaction[1]

Thorsten Hartmann
Gunter Saake

[1] This work was partially supported by CEC under ESPRIT-III Basic Research Action Working Group No. 6071 IS-CORE II (Information Systems – COrrectness and REusability). The work of Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3.

Abstract

As a consequence of the growing interest in active databases the need for abstract activity modelling concepts arises. The object-oriented approach provides a natural view of evolving systems as collections of objects having both structure as well as behaviour. Based on this view we dicuss *event calling structures* as an abstract model for synchronous communication of objects. This basic model defines a powerful activity concept on a higher description level abstracting from operational procedure calling.

Event calling structures have to be analyzed for consistency in terms of conflicting updates. We discuss possible restrictions that make such analyzation feasible. Since event calling structures are used to model triggering of a set of events, finiteness of calling sets is a desirable feature. Again we identify restrictions that make it possible to guarantee finiteness.

For the execution of event sets, an operational description of the execution semantics is given that takes into account data dependencies between different events in a calling set and derives ordering information for event execution. We sketch a possible formalization that draws on attribute grammars.

Acknowledgements

For many fruitful discussions we are grateful to all members of IS-CORE, especially to Hans-Dieter Ehrich, Amílcar Sernadas, Cristina Sernadas who developed the basic ideas of the object model underlying Troll. For many discussions on the language Troll2 we thank Peter Hartel, Ralf Jungclaus, Jan Kusch, and all other colleagues in our group.

Contents

Chapter 1

Introduction

The modelling of dynamic behaviour has long been neglected in data modelling [Kin89, Bee90]. Object-oriented data models partly resolve the problem by providing local events (or operations or methods) that may alter an object's state. Methods are invoked by message passing. Messages either invoke methods inside one object or request the execution of local methods in other objects, i.e. they describe also *interaction* between objects.

Activity by combining operations for complex updates or for maintaining consistency of a database has been an active field of research for several years now. Systems that are capable of reacting upon events and capable of maintaining consistency after updates are called active databases [Day88]. Active databases provide a means to *implement* more complex operations on a database, but they do not provide a means to *design* or *specify* such operations or behaviour on an abstract level early in the development process.

Recently, there have been attempts to integrate an object-oriented data model with concepts of active database models in terms of implementation models based on nested transactions and execution models [BM91]. Although no syntactic means was proposed, the model goes some way into the direction of developing applications involving interactions on (object-oriented) databases, but still lacks specification and analysis of cooperative behaviour.

In another thread of research, the design of active behaviour using rules and Petri net-like formalisms was investigated for conceptual modelling. In [TNCK91], situation-action rules were proposed that were integrated in an ER-model as first-class concepts. Thus, behaviour was not designed locally for objects as it is e.g. in [DPG91]. A similar statement can be made for other attempts to specify activity and interaction rule-based [DBM88, WF90]. Object/Behaviour Diagrams use a form of Petri nets for activity design where value tokens serve as triggers [KS91].

In the research context of this paper, work has been done towards the precise modelling of behaviour over time of objects [SSE87, SFSE89, FSMS91, EDS93]. The object model introduced in this paper is based on concepts developed in that research context. It goes one step further (compared to object-oriented data models) in that *sequences* of event occurrences are considered along with concepts to model activity and interaction between objects on a high level of abstraction. One focus is on the *specification* of temporal object behaviour [SJ92, JSHS91, SJH94].

Here we focus on the abstract modelling of object interaction. This requires the support of the *analysis* of activity specifications, both syntactically and on the basis of a truly abstract operational model that takes into account both control flow and data flow in complex activities. The presented activity model defines the semantics of object interaction in TROLL [JSHS91, Saa93, HSJ$^+$94]. The language TROLL *light* [GCH93] supports a restricted version forbidding synchronous events inside elementary objects.

During the design of TROLL2, a communication model for object interaction had to be chosen fitting to the abstraction level appropriate for conceptual modelling of information systems. To justify the choice of event calling as basic interaction model, we will shortly discuss some alternatives for object interaction:

- *Method calling* of some object-oriented programming languages [Str86, Mey88] is directly derived from procedure calling of imperative languages. Method calling fixes the order of method execution, and termination is undecidable. Concurrent method execution is not directly supported. As result we have only few possibilities for analyzing method execution chains towards optimization.
- *Messages* between objects are used in ACTOR based languages [Hew77, Agh86, SY87, YSTH87]. Objects interact via sending messages which are queued (asynchronous communication) and executed in some order. This model allows parallel execution of methods in independent objects. However, formalization of this model is not easy, and variants of execution orders tend to be used for an operational programming style.
- *Triggers of active databases* are based on an execution model not anticipating an execution order [DBM88, BM91, DPG91]. The intuitive execution model is set-based, i.e., two triggers are intuitively activated concurrently. The *effect* of a trigger chain, however, depends on the execution order leading to unexpected side effects if trigger declarations are optimized. Termination of trigger chains is undecidable, too.
- *Event calling* bases on a simple synchronous communication model easy to formalize on a logical level — a synchronization of two events corresponds to an implication between their occurrences (see following chapters). It can be characterized by following properties:
  - The execution model is set-based (a prerequisite for some optimization methods) and abstracts from operational aspects.
  - The effect of an event calling does not depend on the execution order — which enables optimizing event calling towards operational code.
  - Since the execution model is set-based – an event occurs at most once in an event calling closure – simple conditions can guarantee termination (provided we include some simple restrictions into the model).
  - Event calling supports concurrent events.

- The logic-based formalization is straightforward.

Of course, such an execution model abstracts away the execution order and other operational aspects. The aim of this report is to show that optimization and consistency checking is possible for such an abstract model.

The paper is structured as follows: In the next section we briefly present the object model we are using. Section 3 deals with the modelling of object interaction. In Section 4, we are investigating the syntactical analysis of specifications of activity, before we are presenting an abstract execution model in Section 5 which allows further analysis of activity specifications on an operational level. Finally, we draw some conclusions and point out issues for further research.

Chapter 2

Abstract Dynamic Objects

2.1 Motivating Example

An object is constructed from a possible behaviour and observations over that behaviour associated with local states. Using a specification language like TROLL2 [HSJ+94], we can specify such objects in the following way:

```
object Account
  attributes
    Balance:money .        /* amount of money             */
    CreditLimit:money .    /* maximal credit              */
  events
    establish              /* creation event              */
      birth .
    withdraw(Amount:money) /* withdraw money from account */
      enabled
        (Amount>0.00) and ((CreditLimit+Balance>=Amount)
      changing
        Balance := Balance - Amount .
    deposit(Amount:money)  /* deposit money on account    */
      ....
end object Account ;
```

Here we modelled an account object with a balance, a credit limit and several other properties (attributes) together with a behavioural part (events) that introduce operations to create an account and to withdraw and deposit money together with conditions restricting the occurrence of such events in the *life cycle* of such an object (enabled, the condition models that a withdraw can occur if there is enough money or the `CreditLimit` is not exeeded). We only sketch the modelling of concrete state changes induced by events here in this simple example by means of changing rules described in Section 3.2.

In the next sections we will provide a model that formalizes events, attributes, state changes etc.

2.2 Formalization: Behaviour of Objects

In this section, we are concentrating on the model level rather than on the specification of such models in terms of a language. Some language features will be used for presentation purposes. The interested reader may refer to [JSHS91] for a detailed description of TROLL and to [HSJ+94] for TROLL2. Work related to this report can also be found in [CGH92] that introduces a simplified version of TROLL (TROLL *light*) tailored for e.g. verification purposes [Con94]. Languages like CMSL [Wie90a, Wie90b] and Albert [DDP93] are based on similar ideas. A more detailed overview over related approaches can be found in [Jun93, Saa93].

The basic ideas for understanding the object model underlying TROLL, TROLL2 and also OBLOG [CSS89, SSG+91] will be introduced in the sequel. In [SSE87] first ideas for such a formalization of objects can be found.

Each object has a set of *events* EVT. An event can be regarded as an abstraction of a *possible atomic state transition*. An event is strictly *local to an object*. The declaration

```
deposit(Amount:money)
```

declares a set of events, one for each possible value in the carrier set of the data sort `money`. A particular event is described by an *event term*, e.g.

`deposit(m)` or `deposit(100.00)`

where `m` is a variable instantiated by a value of type `money` and `100.00` a constant of type money. Some events may be marked as being *active*, i.e. in a closed world of a specification they may occur by own initiative (see Section 3.1).

Over a set of events we define *snapshots*:

Definition 2.2.1 (*Snapshot*)
Let EVT be a set of events and 2^{EVT} the power set over EVT. Then a subset $s \in 2^{EVT}$ is called a *snapshot* over EVT. ■

A snapshot is a set of simultaneous occurrences of events, thus an event cannot occur several times in the same snapshot. Events are the *basic* state transitions observable in an object system whereas *snapshots* define composite state changes. We will elaborate this issue in the next sections.

Definition 2.2.2 (*Object Life Cycle*)
Let EVT be a set of events. A sequence $\hat{s} = s_0 s_1 s_2 ...$ such that all $s_i\,(i \in \mathbb{N})$ are snapshots over EVT, is called an *object life cycle*. A finite prefix of length i $\sigma = s_0 s_1 ... s_i$ for $i \in \mathbb{N}$ is called the *i-th state*. The last snapshot s_i in a state σ is denoted by $last(\sigma)$. ■

Thus, the evolution of an object over time is described by a sequence of snapshots, i.e. the sequence of sets of local events. Many occurrences of the same event can occur in several snapshots in an object life cycle and we do not exclude the presence of empty snapshots in object life cycles. Empty snapshots represent states where no event can be *observed*. This way, we may "stretch" life cycles by inserting empty snapshots and we may restrict ourselves to infinite life cycles since finite ones can always be expanded by appending empty snapshots.

Definition 2.2.3 (*Object Behaviour*)
Let EVT be a set of local events. A set B of object life cycles over EVT is called *(possible) object behaviour*. ■

Events may not occur in arbitrary states within an object life cycle. In each state, we have a set of *enabled events*. An event `evt` may occur in a snapshot s_i only if it is enabled in state σ where $last(\sigma) = s_i$. This is written as a predicate $\texttt{enabled}_\sigma(\texttt{evt})$. In this paper, we will not enter into conditions telling us whether an event is enabled in a certain state or not – the reader is referred to [JSHS91, EDS93, Jun93].

2.3 Objects as Observed Processes

A set of *attributes* denotes the state-dependent observable properties of objects. We assume a fixed universe of data types and sorts $s \in S$ to be given. Each attribute $\texttt{att} \in ATT$ is *typed*, i.e. it has an associated sort $sort(\texttt{att}) \in S$. The interpretation of a sort is a suitable set of values and is denoted by $\mathcal{U}(s)$ for a sort $s \in S$.

Like events, attributes are declared as follows. For example:

```
IncomeInYear(Year:nat):money
```

declares a set of attributes of type `money`, one for each possible value in the carrier set of the data sort `nat`. Particular attributes are described by *attribute terms*, e.g.

`IncomeInYear(y)` or `IncomeInYear(1993)`

where `y` is a variable instantiated by a value of sort `nat` and `1993` a constant of sort `nat`. With each state σ of an object life cycle we associate a *current observation* giving us the current *values* of the attributes.

Definition 2.3.1 (*Observations*)
Let B be an object behaviour, $\mathcal{S}(B)$ be the set of states over B, and ATT a set of attributes. The set $O(ATT) = \{(\texttt{att}, d) \mid \texttt{att} \in ATT, d \in \mathcal{U}(sort(\texttt{att}))\}$ denotes the possible observations over ATT. A mapping

$$obs : \mathcal{S}(B) \rightarrow 2^{O(ATT)}$$

is called an *observation structure over B* iff for each state σ of an object life cycle $\hat{s}$ the observation

$$obs(\sigma) \in 2^{O(ATT)}$$

fulfills the following condition:

$$[(\mathtt{att}, d) \in obs(\sigma) \ \wedge \ (\mathtt{att}, d') \in obs(\sigma)] \Rightarrow d = d'$$

That is, attributes cannot have more than one value in each state, but they are allowed to be undefined in a state. ■

In a more general setting, we may drop this condition and will be able to also talk about *non-deterministic* attribute values (see for example [SE91]). An object now is composed from an object behaviour and suitable observations over the possible object life cycles.

Definition 2.3.2 (*Object Model*)
An *object model* OB $= (EVT, B, ATT, obs)$ consists of a set of events EVT, an object behaviour B over EVT, a set of attributes ATT, and an observation structure obs such that for each pair of subsequent states $\sigma, \sigma' \in \mathcal{S}(B)$ where $\sigma' = \sigma\, s$ the following condition holds:

$$(s = \emptyset) \ \Rightarrow \ [obs(\sigma') = obs(\sigma)]$$

s is the current snapshot in state σ'. ■

Only the occurrence of events may alter an observation, i.e. empty snapshots do not have any effect on observations over an object life cycle. This gives rise to the definition of *equivalence classes* of object life cycles: Two life cycles are considered equivalent iff they are equal when we eliminate all empty snapshots.

2.4 Object Composition

When we put objects together to form composite objects, we are interested in the behaviour of the composite object as a whole. Now, the events local to all components are local to the composite object, i.e. the components are *embedded* into the composite object. Each snapshot in the object life cycle of the composite object may include events of components. A composition is admissible if the restriction of each snapshot of each object life cycle of the composite object to the events local to a component yields an object life cycle equivalent to an object life cycle of the component. In other words: the aggregation of objects into a composition does not change the behaviour of the parts but the set of (possible) object behaviour (cf. Def. 2.2.3) may be smaller.

Composition of objects thus means the following:

- Snapshots over components are merged, i.e. we are constructing unions of component snapshots (please recall that snapshots may be empty, i.e. we may have stretched life cycles);
- observations over components are merged according to the merging of life cycles.

The composition of snapshots and observations includes the *synchronization of events* since all events in a snapshot are occurring simultaneously. Synchronization along with parameter passing models *communication*.

In our model, communication is only possible *inside composite objects*. Certain composite objects, however, can be regarded as being virtual, i.e. a system of communicating objects may be regarded and modelled as a composite object. The modeling of consistent interaction between components is the topic of the next section.

Chapter 3

Object Behaviour Specification

In this section we will introduce some basic features of TROLL2 related to specification of object state changes together with their formalization. Without giving a detailed description of the TROLL2 syntax, we will provide some intuitive examples. The interested reader may refer to [HSJ+94] for more details.

3.1 Specification of Snapshots

As mentioned in the previous section, communication is achieved by synchronization. We have, however, to introduce a notation that allows us to describe communication. Furthermore, communication almost always involves control flow, i.e. there are objects that initiate communications and there are objects that are requested to communicate by others. Besides control flow we also have to talk about *data flow* if we want to implement snapshots. For the time being we will put emphasis on the model level however and we will only sketch language and implementation issues in this section.

The basic primitive in our approach to describe communication is *event calling*. Specificationwise, it is defined as a *state dependent relation* over events.

Definition 3.1.1 (*Calling Relation*)
Let EVT be a set of events. The *calling relation* $\gg_\sigma \subseteq EVT \times EVT$ relates events wrt. a state σ of an object life cycle $\hat{s}$.

The calling relation is *transitive*, i.e. the following condition always holds:

$$(\mathtt{evt}_1 \gg_\sigma \mathtt{evt}_2) \wedge (\mathtt{evt}_2 \gg_\sigma \mathtt{evt}_3) \;\Rightarrow\; (\mathtt{evt}_1 \gg_\sigma \mathtt{evt}_3)$$

■

Intuitively, calling should be understood as *asymmetric, synchronous communication.* Calling could be enlarged to *interaction* if we talk about synchronizing different objects by means of an enclosing object [HJ92]. Notationwise, (conditional) calling is written as:

```
{ condition } calleeEvent_term1 >> event_term2, event_term3, ...;
```

or in the context of an event specification in TROLL2 as:

```
calleeEvent(...formal parameters...)
  ...
  calling
    { condition } calledEvent_term2, calledEvent_term3, ...;
  ...
```

This means that the calling relation only holds between events denoted by the event terms if the condition holds in the current state. `Condition` is a formula constructed over observable properties of objects, constants of data types etc. The commalist for event terms is just an abbreviation of several calling rules. This notation can also be interpreted as some kind of *ECA-rule* [DBM88]: an "*Event*," denoted by `event_term1` causes (an)other "*Event(s)*" `event_term2` etc. depending on certain "*Conditions*."

Definition 3.1.2 (*Closure of Calling*)
Let `evt` be an event occurring in state σ. The *closure of* **`evt`** *wrt. calling in state* σ is defined as

$$\mathtt{cl}_\sigma(\mathtt{evt}) = \{\mathtt{evt}' \mid \mathtt{evt} \gg_\sigma \mathtt{evt}'\}$$

■

As mentioned earlier we distinguish between active and passive events, the former defining the events that can occur by own initiative. Snapshots are called *complete* if the closures of all events in a snapshot are subsets of the snapshot and there exists an active event for each closure:

Definition 3.1.3 (*Complete Snapshots, Isolated Closures*)
Let σ be a state. A snapshot $s_i = last(\sigma)$ is *complete* iff

(1) $\forall \mathtt{evt} \in s_i$: $\mathtt{cl}_\sigma(\mathtt{evt}) \subseteq s_i$

(2) $\forall \mathtt{evt} \in s_i\ \exists \mathtt{evt}' \in s_i$: $(\mathrm{active}(\mathtt{evt}') \wedge \mathtt{evt} \in \mathtt{cl}_\sigma(\mathtt{evt}'))$

The closures in a snapshot are called *isolated* if all closures of active events are mutually disjoint (there exists exactly *one* active event $\mathtt{evt}_\mathtt{a}$ for each passive event $\mathtt{evt}_\mathtt{p}$, such that $\mathtt{evt}_\mathtt{p}$ is in the closure of $\mathtt{evt}_\mathtt{a}$). ■

Thus, the calling relation $\gg_\sigma$ in some state σ defines a necessary condition that must be fulfilled by the corresponding snapshot s_i. Calling closures of active events must not interfere, i.e. they have to be isolated. Active events can occur by own initiative, that is, on the specification level we cannot always avoid overlappig snapshots. During runtime of an object system the isolation property must be fulfilled however (corresponding to the isolation property of (acid) transactions).

For a more restrictive but similar notion of event closures compare also [GCH93].

3.2 Specification of Attribute Changes

Recall that the occurrence of an event may have effects on the observable properties of an attribute, i.e. it may affect the values of certain attributes (see Section 2). In order to define *consistent snapshots*, we will define a state dependent relation **affects** telling us which attributes are affected by the occurrence of an event.

Definition 3.2.1 (*Affects-Relation*)
Let EVT be a set of events and ATT be a set of attributes. The *affects-relation* is defined as

$$\mathbf{affects}_\sigma \subseteq EVT \times ATT$$

`evt` **affects**$_\sigma$ `att` iff the occurrence of `evt` alters the value of the attribute `att`. ∎

Notationwise, event *effects* on attributes are written as follows:

```
anEvent(... formal parameters ...)
  ...
  changing
    { condition_1 } attribute_term_1 := data_term_1,
          ....
    { condition_n } attribute_term_n := data_term_n,
  ...
```

if the condition evaluates to true then *after* occurrence of an event denoted by `anEvent` the value of the attribute denoted by `attribute_term_`i will be determined by `data_term_`i. In TROLL2 it is possible to refer to *parameters* of an event by their names.

As an example we will refine the account specification from Section 2 with additional changing rules. This time, we introduce an attribute `Red` denoting a possibly negative balance, the `Balance` attribute may only be positive now:

```
object Account
  attributes
    Balance:money .        /* amount of money             */
    CreditLimit:money .    /* maximal credit              */
    Red:bool .             /* overdrawn ?                 */
  events
    establish              /* creation event              */
      birth .
    withdraw(Amount:money) /* withdraw money from account */
      enabled
        (Amount>=0) and
        if Red
          then (CreditLimit-Balance) > Amount
          else (CreditLimit+Balance) > Amount
        fi ;
```

```
      changing
        {not Red and Amount<=Balance}
                            Balance := Balance - Amount ;
        {not Red and Amount>Balance}
                            Balance := Amount - Balance ,
                            Red := true ;
        {Red} Balance := Balance + Amount .
    deposit(Amount:money)  /* deposit money on account    */
        {not Red} Balance := Balance + Amount ;
        {Red and Amount<Balance}
                            Balance := Balance - Amount ;
        {Red and Amount>=Balance}
                            Balance := Amount - Balance ,
                            Red := false .
end object Account ;
```

Now the withdraw and deposit events are specified changing the observable state of an object according to the previous state and actual parameters of the events. Depending on the account condition (`not Red` or `Red`) the `Balance` and `Red` attributes are changed, i.e. decreased or increased respectively for `Balance` and toggled for `Red`.

In TROLL2, we can alternatively specify the changing rules in an if then else form. If then else is an operation for all data types in TROLL2. An example can be seen in the enabling condition for event `withdraw`. For details please refer to [HSJ+94].

Now let us define consistent snapshots. A snapshot is consistent iff the effects of event occurrences in a snapshot are not conflicting.

Definition 3.2.2 (*Consistent Snapshots*)
Let σ be a state such that $s_i = last(\sigma)$ is a complete snapshot. s_i is *consistent* iff the following condition holds:

$$\forall \mathsf{e}_1 \in s_i \; \forall \mathsf{e}_2 \in s_i \; \forall att \in ATT\colon \; [(\mathsf{e}_1 \; \mathbf{affects}_\sigma \; \mathtt{att}) \wedge (\mathsf{e}_2 \; \mathbf{affects}_\sigma \; \mathtt{att})] \Rightarrow \mathsf{e}_1 = \mathsf{e}_2$$

■

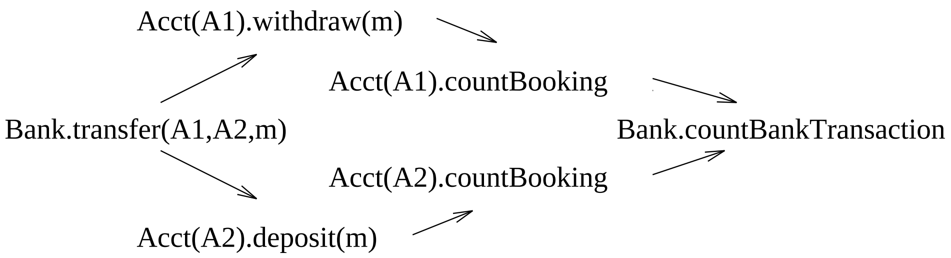
In contrast to procedure calls and triggers in active databases – where we may have calls resulting in *several* invocations of each operation – see the following example. In account objects, booking transactions are counted locally (we provide only the calling specification and omit the rest of the example here):

```
withdraw(Amount:money)
              calling countBooking ;
deposit(Amount:money)
              calling countBooking ;
```

Suppose now we want to specify an event `transfer`, transferring an amount of money between accounts. In a surrounding bank object (denoted as `Bank`) the transfer events are counted be means of calling rules depending on the `countBooking` events of the account subobjects (denoted as `Acct(A`*i*`)`):

```
object Bank
  ...
  events
    Bank.transfer(A1,A2,m) >> Acct(A1).withdraw(m),
                              Acct(A2).deposit(m);
    Acct(A).countBooking >> countBankTransaction;
  ...
end object Bank ;
```

now using the `>>` syntactical representation.[1] Graphically the situation is depicted as follows:



The event `countBankTransaction` – although called twice by two account subobjects – is only contained *once* in a calling closure.[2]

In the following chapters we will introduce techniques for analyzing calling structures (consistency and finiteness) and then define an abstract execution model for event snapshots.

[1] The difference of these notations comes into consideration because of the clustering of event properties around the event specification as introduced in TROLL2 compared with *object interaction* specified in composite objects [HSJ+94]. Conceptually both notations refer to the abstract model introduced in Section 2.

[2] In the next chapter we will see that *isolated closures* (cf. Def. 3.1.3) are necessary for a correct counting of bank transactions. At the specification level we cannot guarantee this property.

Chapter 4

Analyzing Calling Structures

Event calling is a very powerful communication mechanism. General construction of event snapshots is a computational complete formalism [Saa93] — arbitrary calling with complex parameters is equivalent to recursive functions. For using such a mechanism as activity model for object *specifications*, we need techniques to analyze calling structures to guarantee *consistency* and *termination* — or at least give warnings if these properties can not be decided during analysis. Additionally, we have to state sufficient conditions for these properties as a help for application developers to write 'good' specifications.

As mentioned above, we are mainly interested in three properties of descriptions of calling structures:

- Are all described snapshots *consistent*, i.e. is it possible that attribute modification conflicts occur?
- Are all snapshots isolated? If not, suitable runtime checks have to be generated.
- Are all described snapshots *finite* or may we have infinite calling sequences?

These properties and corresponding techniques to analyze event calling structures are discussed in the following sections.

4.1 Consistent Snapshots

Like changing rules, calling rules can be specified with a *condition* modelling rules that are only applied in specific object states. Since such a *condition* may refer to current parameters as well as to attributes, both the calling relation $\gg_\sigma$ and the $\mathbf{affects}_\sigma$ relation are dependent on observations.

What can be done for checking consistency of calling structures? During runtime of a system, a check can be performed as part of a general check: conflicts on attribute valuations can be detected as write conflicts during transaction processing. For specifications, an additional consistency check during compile-time is surely desirable.

Based on the calling and changing specification, we can analyze calling structures based on the following observation. We define the relation **possibly-affects** as the union over all possible state-dependent **affects** relations:

$$\textbf{possibly-affects} = \bigcup_{\sigma \in \mathcal{S}(B)} (\textbf{affects}_\sigma)$$

Now we can prove the property *maybe inconsistent* in some state based on this relation. However, this semantic characterization of **possibly-affects** is still not practical. We use a weaker relation **may-affect** based on the *syntactic specification* of attribute changing rules: the relation **may-affect** holds between an event `evt` and an attribute `att` iff a corresponding valuation rule appears in the specification document regardless of any state dependent conditions. Similarly, we define relations **possibly-calls** and **may-call** based on calling rules. Of course, we have

$$\forall(\sigma \in \mathcal{S}(B)) \;\; (\textbf{affects}_\sigma \subseteq \textbf{possibly-affects} \subseteq \textbf{may-affect})$$

and

$$\forall(\sigma \in \mathcal{S}(B)) \;\; (\ggg_\sigma \subseteq \textbf{possibly-calls} \subseteq \textbf{may-call}).$$

For practical checking, we compute the relations **may-affect** and **may-call** over the (finite) set of *relevant event terms* rather than over the set of events EVT (in general infinite). For a given event term `evt` first the set of *may be called* event terms is computed. Then the elements of this set are tested pairwise on conflicts using the **may-affect** relation. Taking also activity into account, we may test also the isolation property. In this sense an event term `evt` may occur if it is enabled and if it is being called from an active event (see next section).

This checking procedure tests if a conflict situation inside a calling structure may occur based on the specification clauses only, i.e. without looking at state-dependent influences. Therefore, the result of this task is *sufficient* to guarantee consistent snapshots — however, as we will see in Sections 4.2 and 4.3, there are specifications leading always to consistent snapshots being rejected by this syntax-based check.

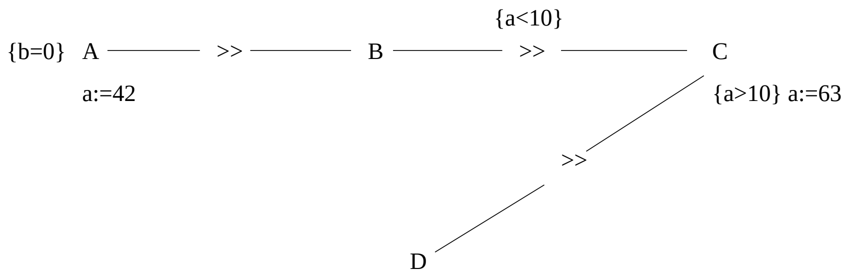
4.2 Conflict Detection

Events are possibly conflicting if they are elements of one calling closure and alter the same attribute. For detecting inconsistent snapshots we also have to take into account *activity* of events and calling of events. That is, the isolation property (see Definition 3.1.3) must not be violated for concrete snapshot executions.

As an example for an object specification (using Troll2 [HSJ+94]) with possibly incosistent snapshots we introduce an example as follows:

```
object Example
  attributes
    a:int .
    b:int .
  events
    ...
    A
     active ;
     enabled b=0 ;
     changing a := 42 ;
     calling B .
    B calling { a<10 } C .
    C changing { a>10 } a := 63 .
    D
     active ;
     calling C .
    ...
end object Example ;
```

Graphically the situation is depicted as follows:



Syntactically the specification of events `A` and `C` leads to conflicts because both events alter the attribute `a` and are members of the same calling chain `A>>B>>C`. The conflict is 'virtual' however, since `C` changes `a` only if `a` is greater than ten and then `C` is not called by `A` (transitively via `B`). A syntax based check as depicted in the last section is more restrictive than necessary in this case here.

The specification of object `Example` does not always respect the *isolation property* for calling closures: the active events `A` and `D` do not define isolated calling closures (see Definition 3.1.3):

$$\{ \texttt{A, B, C} \} \cap \{ \texttt{D, C} \} = \{ \texttt{C} \}$$

Conflicts can only be precluded if we forbid `A` and `D` occurring in one snapshot (see below).

Forgetting about the several state dependent conditions (used in this specification) to describe a syntactical checking procedure, we can generate the following Prolog rules, identifying events and attributes with Prolog *objects*, event `A` with `evtA` etc. The specification parts are coded into prolog facts as follows:

```
active(evtA).       /* two events are active                 */
active(evtD).
enabled(evtA).      /* all events enabled (syntax based check) */
enabled(evtB).
enabled(evtC).
enabled(evtD).
calls(evtA,evtB).   /* unconditional calling     - " -        */
calls(evtB,evtC).
calls(evtD,evtC).
affects(evtA,attA). /* unconditional changing    - " -        */
affects(evtC,attA).
```

denoting the active events that are necessary for snapsots, the enabling conditions are not concidered so far, the calling relation, and the affects relation. Additionally the following rules are provided as an algorithm for detecting conflicts:

```
happens(X) :- enabled(X), active(X).
happens(X) :- enabled(X), calls(Y,X), happens(Y).
conflict(X,Y) :- happens(X), happens(Y),
                 affects(X,Z), affects(Y,Z), X≠Y.
```

These rules model the two possible conflict situations with the rule `conflict`. `Conflict` is fulfilled for a given pair (`evtX`,`evtY`) if they both happen and affect the same attribute. An event happens if it is enabled and active or enabled and called by another event that happens. A Prolog query:

```
conflict(evtA,_X).
```

thus yields `_X = evtC`. This goal is fulfilled because of the conflicting chains `A>>B>>C` and `D>>C` (both, `A` and `D` are active). The goal is fulfilled twice: In the first case the closure of active event `A` contains two events changing attribute `a` (syntax based check only). In the second case the isolation property is violated for active events `A` and `D`.

Up to now we forgot about the conditions involved in the actual changing and calling rules. We may also introduce concrete attribute values by means of predicates that model the binding of attributes to values and also the conditions in the following way:

```
hasValue(attA,7).
hasValue(attB,0).
active(evtA).
active(evtD).
enabled(evtA) :- hasValue(attB,B), B=0.
enabled(evtB).
enabled(evtC).
```

```
enabled(evtD).
calls(evtA,evtB).
calls(evtB,evtC) :- hasValue(attA,A), A<10.
calls(evtD,evtC).
affects(evtA,attA).
affects(evtC,attA) :- hasValue(attA,A), A>10.
```

which describes a binding of `a` to `7` and `b` to `0`. This translation now makes it possible to check for conflicts *in a given state of an object*. Again the above mentioned query will detect a conflicting pair (`A`,`C`). Now the goal is only fulfilled for the violation of the isolation property.

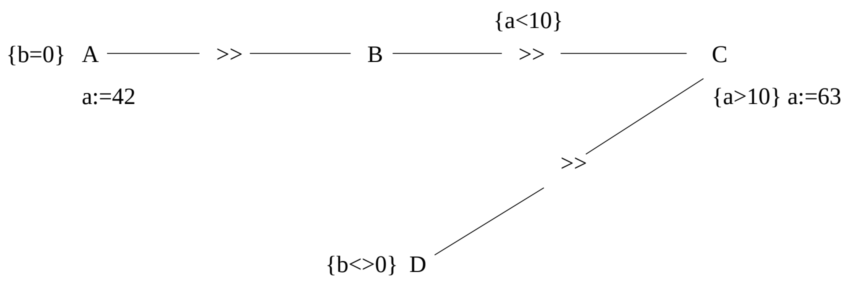
The conflict disappears if we alter the specification of event `D` in the following way:

```
D
  active
  enabled b<>0
  calling C .
```

The PROLOG translation thus is extended with the following rule:

```
enabled(evtD) :- hasValue(attB,B), B≠0.
```

Again, graphically the situation is depicted as follows:



Now events `A` and `D` are enabled for `b=0` respectively `b<>0`, that is, they cannot be in *one* snapshot any more assumed we have deterministic attributes as defined in Def. 2.3.1. The property of *isolated* snapshots must be checked during runtime of the system because a restriction as defined here is not always realistic in real applications. During specification and analysis time the possible conflicts should be flagged by a suitable tool to enable runtime checks to be inserted for an implementation.

Although we can introduce and check several bindings of attributes to values by means of different facts `hasValue` iteratively, we have no means to *decide in general* about conflicting snapshots this way. We can only check for a finite set of bindings, i.e. object states. The modeled translation shows however, that a checking is possible and straightforward for the syntactical part (i.e. the **may-affect** relation) and for special cases (states).

4.3 Finiteness of Snapshots

As a motivating example, we start with the specification of a (decimal) counter object [Saa93]. This object represents the digits of a decimal counter as attributes. Upon occurrence of the (active) counting event `Count` the digits are incremented starting with the leftmost digit `Digit(1)` and successively higher digits if necessary:

```
object Counter
 attributes
   Digit(n:nat):{ 0...9 }
     initialised Digit(n) = 0 .
 events
   CreateCounter birth .
   Count
     active
     calling IncDigit(1) .
   IncDigit(n:nat)
     changing
       { Digit(n) < 9 } Digit(n) := Digit(n) + 1 ;
       { Digit(n) = 9 } Digit(n) := 0
     calling
       { Digit(n) = 9 } IncDigit(n+1) .
end object Counter ;
```

As observable properties we introduce an infinite set of attributes, one for each natural number or equivalently an infinite array. The attributes are *specified* to be zero upon creation of the object. Note here that the formal parameter specification `n:nat` refers to a universal quantification for `n` [HSJ+94].

Digits are changed by corresponding `IncDigit` events. As usual, an overflow of a digit increments the next higher digit (by calling the corresponding `IncDigit` event). Therefore, several `IncDigit` events (for different parameter values) can occur simultaneously (as known from clocks with digital display). There can be no attribute modification conflicts, because different instances of `IncDigit` change different attributes `Digit(n)` — but there is no upper bound on the number of simultaneous `IncDigit` events in a snapshot (an arbitrary number of `Digits` with current value `9` maybe changed to `0`). This observation motivates the search for checking algorithms and sufficient conditions to guarantee *finite snapshots*.

Finiteness of snapshots is in principle undecidable, if we use a general formalism to describe state dependent calling structures like, for example, in the TROLL language [JSHS91, HSJ+94]. It can be shown that a test on finiteness is equivalent to the halting problem by coding, for example, a Turing machine by one calling chain of an object. Computation steps of this machine can be coded as parameters of one event calling structure where an event calls another instantiation of the same event name with new parameters (similar to the calling in the `Counter` object). Halting of this machine is equivalent to having a finite snapshot.

In formalisms for database languages, we are sometimes more interested in decidable sublanguages than in computational completeness — we would like to characterize syntactically a subset of all calling structure specifications where finite snapshots (meaning terminating computations) are always guaranteed. The following conditions are sufficient to guarantee finiteness of snapshots and can often be checked based on the syntax:

1. Object creation events do not call (directly or transitively) other creation events.

2. An event directly calls a finite number of creation events only.

   The first two rules guarantee that only finitely many objects are created in a snapshot.

3. For each object instance, for each event name only a finite number of instantiations are allowed. For practical checks, we can reduce this in most cases to allowing at most one event instance.

On the first glance these conditions look very restrictive. For example, the `Counter` object does not satisfy these conditions. The `Counter` object guarantees always finite snapshots, but this property can only be proved using induction over all possible reachable states. We have to prove that each reachable observation can only contain a finite number of digits showing a `9`. Thus a syntactical check as proposed in Sections 4.1 and 4.2 is easily implemented but may fail for specifications as shown above.

Our sufficient conditions still allow, however, a wide range of meaningful activity specifications. They are much more liberal than a condition allowing only one event at a time per object (like for example in TROLL *light*), or one event occurrence of an event generator (like in OSL [SSC92, Jun93]), and they can be checked during analysis of a specification document.

The counter object for example may be rewritten in the following way. The `Digit` attributes can be transformed to separate local objects. Then we have only one `IncDigit` event per `Digit`-object (without parameters) and birth events of `Digit`-objects called by need only – leading to an accepted calling structure. Even if we need more general calling structures, the check of these conditions can serve as a warning and as hint to perform a finiteness check by hand or to build in an explicit termination condition.

Chapter 5

An Abstract Execution Model

In the previous sections we introduced *calling closures* as sets of *event occurrences* (called *events* from now on) generated by the state dependent calling relation $\gg_\sigma$. An initial event thus determines a *set* of related events that conceptually occur in parallel and are embedded into a *snapshot*. If we want to *execute* calling closures, this *conceptual* view of calling closures as sets must be translated into an *operational* view taking into account an *execution order* of events where abstractly specified state transitions are *implemented* by means of more concrete transitions and transition sequences.

Up to now we mainly talked about the formalization of objects and composite state transitions (snapshots). The execution of snapshots however needs *implementation techniques* not mentioned so far. Implementation here means implementing state transitions defined by snapshots by determining an execution order of *conceptually parallel events* and is not meant in the sense of *reification* in the sense of [FM93, EDS93] where *event sequences* are considered as *refinements* or *reifications* of *abstract events*.

5.1 Structuring Snapshots

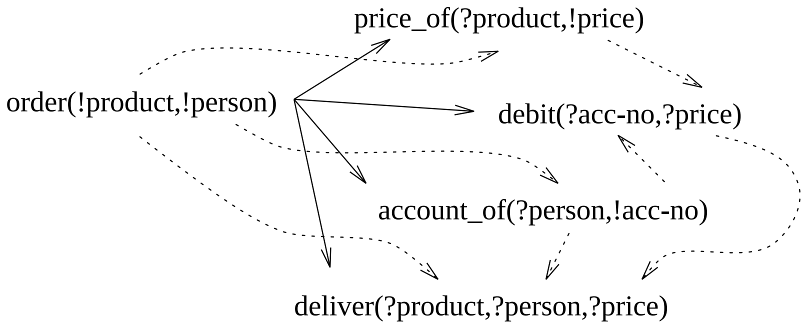
On the one hand conceptually all events in a snapshot are executed in parallel. On the other hand the calling relation implicitly models a *causality* that is reflected in the notation for event calling. An initial event *induces* a set of dependent events transitively. Starting from this initial event an execution order is then restricted by the *partial order* induced by the calling relation, i.e. the *control flow*.

Besides control flow defined by event calling we must also take into account *data flow*. As can be seen in the decimal counter specification, event calling may be specified with formal parameters denoted by variable symbols. In a single calling clause same variables denote same values but the actual value can be *set* at an arbitrary position, i.e. by some event in the closure.

As an example see the following specification fragment from a warehouse specification dealing with persons, products, prices etc. We assume that this interaction is specified in an object that uses several services, i.e. other events by calling. In this example we abstract from the distributed nature of such an 'application.' Part of the order processing may be specified – using the interaction form of calling – as:

```
interaction
  variables prod:string, person:string, price:money, acc-no:nat;
  order( !prod, !person ) >>
                  price-of( ?prod, !price ),
                  account-of( ?person, !acc-no ),
                  debit( ?acc-no, ?price ),
                  deliver ( ?prod, ?person, ?price ),
                  ...;
```

That is, the order processing event calls for the price lookup event in some price list object, the account number lookup event for the given person, a debit event for the amount of money to be withdrawn from the customer account, and causes the delivery event for the product to the customer. The control and data flow dependencies are depicted in the following figure:



The ! denotes *output-parameters* whereas the ? denotes *input-parameters*, ie. !-prefixed variables must be known in the *context* where the event is executed respectively be computed in further calling steps starting in this context. For example the event `price-of` must occur in an environment where a 'product to price' table is available. We only assume that the `price-of` event *delivers* a value for `price` – may be using further event calls not specified here. The syntax used here is not exactly TROLL2-Syntax [HSJ+94]. The data flow tags !,? are only used for clarifying things here. In TROLL2 only the ! tag is used in event parameter specifications to denote which parameters must be bound by a specific event.

Conceptually the closure $\mathtt{cl}_\sigma$(`order(prod,person)`) contains at least the five events mentioned and may be part of a bigger snapshot. From now on we will talk about execution of *closures*.

5.2 Execution of Event Closures

Possible execution orders of events are restricted by the partial order defined by the control flow. An ordering of the events mentioned on the right hand side of the calling clause is not specified explicitly. Thus the next task is to find a *refined execution order* for the events not fixed by the control flow order. For the time being we pose the following conditions on events and on the binding of parameter variables to values:

1. We allow only closures of an instance of an active event.
2. The active event must be fully instantiated, ie. only contain ! prefixed variables.
3. The active event always is the first event to be executed (this is implied by the partial order defined by the control flow).
4. All variables must be set *exactly once*.

Furthermore, if an event should be executed

5. its actual parameters must either be known (eg. previously set during the course of execution) *or* must be set by the event to be executed, e.g. known in its context.

The stated rules guarantee that all parameter values of a particular event are known when it comes to execution as part of a calling closure described by event calling. Thus the data-flow directives ! and ? introduced in the above example are used to *refine* the control flow order according to the above conditions. See for example the following refined (sequential) execution orders (1) and (2):

(1) *correct execution order*	(2) *incorrect execution order*
`order(!prod,!person)`	`order(!prod,!person)`
↓	↓
`price-of(?prod,!price)`	`deliver (?prod,?person,?price)`
↓	↓
`account-of(?person,!acc-no)`	`account-of(?person,!acc-no)`
↓	↓
`debit(?acc-no,?price)`	`debit(?acc-no,?price)`
↓	↓
`deliver (?prod,?person,?price)`	`price-of(?prod,!price)`

Both executions start with the *active* event `order`. (1) on the left side is *correct* according to the above mentioned restrictions on execution orders. (2) on the right side violates the condition of parameters being *"set before used"* as is for example not the case for the `price` parameter (`deliver`-event). This reflects the intuitive semantics of such calling construction: we cannot debit an unknown amount of money from a given account. Generally we are not restricted to sequential execution. The refined *partial order* still may contain events that can be executed in parallel provided there are no data flow dependencies.

As we can see from this discussion the control flow information restricted by the partial order defined with event calling plus the data flow information can be used to determine *possible* executions, ie. imposes a structure on the calling closure. Based on these information, a concrete execution order may be derived from further information inferred from the whole specification of an object and is not discussed here. For example unrelated events can be executed in parallel if the underlying system structure allows this, e.g. if the participating events take place in objects residing on different machines.

5.3 Enhanced Execution Model

With the conditions enumerated in the previous section we may find possible orders of events inside a calling closure. Firstly, employing the dependencies between settings of parameters, these rules do not allow data flow against control flow as is for example needed for query results. Secondly, it is possible to specify calling closures that have *no* possible execution sequence according to the model described in the previous section (mutually set variables on the right hand side of calling rules). As an example from a bank world see an event that transfers `p` percent of balance of account `acc_s` to `acc_d` (source to destination) and “returns” an `ok` status:

```
interaction
  variables acc_s:nat, acc_d:nat,p:real,x:money,ok:bool;
  trans_p_percent(!acc_s,!acc_d,!p,?ok) >>
                              per_cent_of_balance( ?p, !x ),
                              debit( ?acc_s, ?x, !ok ),
                              credit( ?acc_d, ?x );
```

As one can easily see the parameter `ok` of the active event transfer is instantiated with the `debit` event, e.g. it is set to true only if the amount of money can be provided by `acc_s`. Up to now this situation is forbidden due to rule no. 2. A suitable solution for this problem is to insert *system generated* events:

```
interaction
  variables acc_s:nat, acc_d:nat,p:real,x:money,ok:bool;
  pre_trans_p_percent(!acc_s,!acc_d,!p) >>
                           per_cent_of_balance( ?p, !x ),
                           debit( ?acc_s, ?x, !ok ),
                           credit( ?acc_d, ?x ),
                           trans_p_percent( ?acc_s, ?acc_d, ?p, ?ok );
```

With this example the active event is rewritten to an event containing only instantiated variables and the original event is added with rewritten parameters. Furthermore this technique can also provide a solution for events with mutually set parameters. Trying to execute such kinds of dependent events would lead to *deadlock*, so an analyzer tool based on this refined execution model has to detect such dependencies and has to insert suitable `pre_` events.

Similar techniques as depicted in Section 4 must be used to detect such conflicts. The simple execution model introduced in the previous section seems to be restrictive with respect to variable bindings. It should be noted however, that we do not have to introduce additional 'machinery' to execute partially instantiated events.

As we can guess from this simple example these transformations are not always trivial. In general it is necessary to analyze the *dependencies* between setting and reading variable values. Often there will be several alternatives and it may be a *design decision* which one is to be chosen.

In the next section we will introduce a correspondence between the execution model depicted informally here and *attributed grammars*.

5.4 Attribute Grammar Based Model

The intuitively depicted execution model of the last sections has to be formalized in a suitable way. A promising candidate for such a formalization are *attribute grammars* [Knu68, AM91]. An attribute grammar is a context free grammar enhanced with *attributes*[1] for production rules carrying information about the concrete occurrence of a rule in a derivation tree.

Attribute grammars are used to specify the context sensitive parts and the semantics of programming languages. Besides these applications – that have their roots mainly in compiler construction and compiler generators – nowadays attribute grammars are used for a variety of related applications and have proven to be useful as implementation and optimization aids [AM91] not only for compilers. For example [Tsa89] describes the use of attribute grammars to *optimize* logic programs using similarities between *grammar productions* and *rules* of prolog programs (*horn clauses*). Similarly we can also compare function application in functional programs with grammar productions which is close to calling rules as depicted in the last sections. In [Hoh93] attribute grammars are used to define the translation of a query formalism for an extended Entity-Relationship model to a suitable query calculus.

In our context similarities to calling structures amount to calling rules/production rules, event parameters/attributes, application conditions/semantic conditions for calling. Data flow in calling structures can be compared to attribute evaluation top down and bottom up in a derivation tree called *inherited* and *synthesized* attributes in attribute grammars.

Calling structures are defined upon sets of events which is a major difference to attribute grammars where several occurrences of nonterminals may have the same attribute bindings at different positions in a tree however. As another difference we observe that calling rules have no predefined order in contrast to grammar productions where the order of symbols on the right side defines the structure of the generated sentence. But, as

[1] Attributes in the context of attribute grammars are different from attributes used in object specifications.

we have seen in the last section, an order has to be found if we want to execute calling structures. A possible way to analyze and execute calling sets may be a transformation into an attribute grammar that respects specific attribute dependencies, i.e. belongs to a certain class of grammars.

Without giving a detailed formalization of attribute grammars in this report we will only introduce the basic features of attribute grammars and provide a simple example drawn from the previous section but formulated as an attributed grammar. Here we can only show possible future work to further analyze calling structures as depicted in the last sections.

An context free grammar G is defined as a tuple $G = (V, N, S, P)$ where V is the (finite) underlying alphabet of symbols, $N \subseteq V$ the nonterminal symbols, $S \in N$ the start symbol, and P a set of production rules. Since we do not need terminal symbols in this context we assume $V = N$.

To define an attribute grammar, a finite set of attributes $A(X)$ is associated to every symbol $X \in V$. $A(X)$ is partitioned into two disjoint sets $A(X) = S(X) \cup I(X)$ modelling *synthesized* attributes and *inherited* attributes respectively. Furthermore, to each rule we may associate a *semantic condition* $C(X)$ restricting the possible trees generated by G (compare application conditions for calling rules). We will not use semantic conditions in the sequel, they are mentioned to show the similarities with calling rules. For grammar productions there exist evaluation rules for the symbols used in a rule. Classes of attributed grammars are usually defined according to dependencies among attributes. For a general and more detailed definition of attribute grammars see [Alb91], for classes of grammars see [DJL86].

Often the start symbol S is restricted to have only *synthesized* attributes, that is, $I(X) = \emptyset$. In our context inherited attributes of the start symbol are used to model *input* parameters of the active event generating the calling closure whereas the *synthesized attributes* of all nodes and especially the leaves of a calling tree also model the binding of parameters (attributes) to values caculated from the current object state. To clarify these ideas we provide the following example based on the one from Section 5.3:

```
interaction
  variables a_s:nat, a_d:nat,p:real,x:money,ok:bool;
  trans_p_percent(!a_s,!a_d,!p,?ok) >>
                              per_cent_of_bal( ?p, !x ),
                              debit( ?a_s, ?x, !ok ),
                              credit( ?a_d, ?x );
```

This specification of a calling structure can be modeled as an attributed grammar in the following way.

```
Grammar G:
  V = { trans_p_percent, per_cent_of_bal, debit, credit},
  S = { trans_p_percent },
```

```
P = { p_0:trans_p_percent → per_cent_of_bal debit credit }

Attributes (for symbols in V):
  I(trans_p_percent)={a_s,a_d,p},    S(trans_p_percent)={ok},
  I(per_cent_of_bal)={p},            S(per_cent_of_bal)={x},
  I(debit)={a_s,x},                  S(debit)={ok},
  I(credit)={a_d,x},                 S(credit)=∅

Evaluation rules (for p_0):
  trans_p_percent.ok := debit.ok,
  trans_p_percent.a_s := ‘‘to be provided from environment’’,
  trans_p_percent.a_d := ‘‘to be provided from environment’’,
  trans_p_percent.p := ‘‘to be provided from environment’’,
  per_cent_of_bal.x := ‘‘calculated from current object state’’,
  per_cent_of_bal.p := trans_p_percent.p,
  debit.ok := ‘‘calculated from current object state’’,
  debit.a_s := trans_p_percent.a_s,
  debit.x := per_cent_of_bal.x,
  credit.a_d := trans_p_percent.a_d,
  credit.x := per_cent_of_bal.x
```

We assume the notation for the evaluation rules as being intuitive (dot notation). Synthesized attributes of the right hand side of a rule may be calculated using the state of the object, for example how `debit.ok` is derived is not shown here. Inherited attributes of the root event `trans_p_percent` are delivered from the environment where the event is executed.

Pictorially we can represent this grammar production in Figure 5.1, Page 32 where synthesized attributes are written on the left hand side of symbols and inherited attributes on the right hand side. Attribute dependencies are noted with dotted lines whereas control dependencies are noted with arrowed lines.

In the figure we can easily see which dependencies exist between symbols (events) of the right hand side of a production. Changing of the rule:

```
p_0: trans_p_percent → per_cent_of_bal debit credit
```

to

```
p_0: trans_p_percent → debit credit per_cent_of_bal
```

yields a different attribute grammar where we have dependencies of attributes in a different direction (right to left). As we have seen in the last section such an (execution) order is not executable in our sense of calling set execution. The property of an attribute grammar to have only dependencies from left to right is called *L-attributed* [Wil79, EF81]. In this sense L-attributed grammars define possible execution orders for event closures.

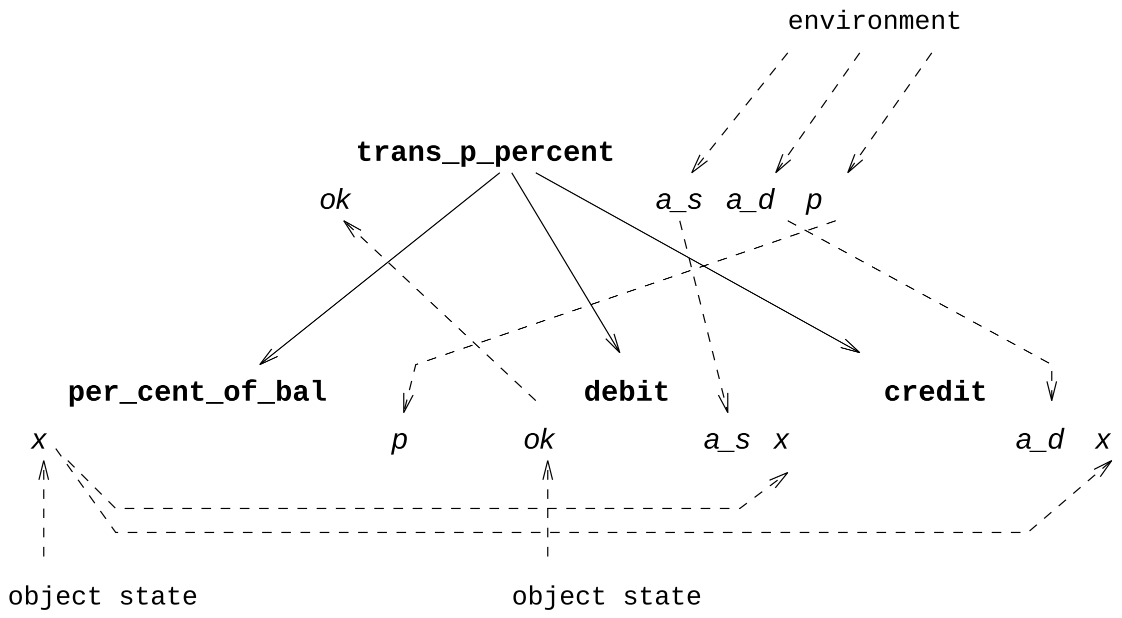


Figure 5.1: Grammar Production P_0

Investigating the possible applications of attribute grammars to find execution plans for event closures will be a further direction of our work. As we have seen in this section, attribute evaluation algorithms may be used to describe calling closure executions not based on the simple model introduced in Section 5.1.

Chapter 6

Conclusions and Outlook

In this report we have introduced a basic model for synchronous communication of evolving abstract objects. A model of objects integrating sequences of sets of simultaneous events and observations associating finite prefixes of such sequences with values for local attributes was defined. Object interaction is modeled on an abstract level by relating event occurrences through a state-dependent, asymmetric relation, the *calling relation*. We have defined the notions of complete event snapshots and consistent snapshots that do not impose conflicting updates of attribute values.

Based on this basic model we have proposed techniques for the syntactic analysis of specifications in order to be able to detect non-executable interaction specifications. We have presented necessary conditions that guarantee consistency of snapshots and finite snapshots, i.e. finite sets of synchronized events.

The emphasis of the approach described here is laid on *specifying* and *modelling* complex interaction structures and object behaviour, whereas active databases aim at *implementing* behaviour. Nevertheless the different concepts have similarities and active database technology seems to be a suitable implementation platform.

In order to be able to transform specifications of interacting objects into executable systems we proposed an abstract execution model. Snapshots were conceptionally regarded as ACID transactions. By imposing a structure on snapshots possible execution orders are defined. Here data flow had to be obeyed in addition to the control flow specified in interaction specifications. The use of attribute grammars for analysis and execution purposes must be further worked out.

The approach described in this paper is embedded into a project that aims at prototyping and implementing object-oriented specifications. To develop specifications in TROLL2, we are currently implementing a prototypical software development environment that integrates tools for editing specifications (graphically and syntax directed), check consistency of specifications, and for generating executable code (*C++*,PROLOG). The graphical interface (concepts are introduced in [WJH+93]) is based on OMT [RBP+90].

The language and formalisms introduced here are used for a *core language* of TROLL2. On top of this core language TROLL2 contains abstraction principles like classes, roles, inheritance, composite objects, and relationships between objects (not introduced here, the interested reader may refer to [HSJ+94]) that have to be mapped onto the core language. The development environment will integrate also tools that automatically executes these transformations towards a final compilation to executable (prototyping) code.

Since TROLL2 is a logic based language, relationships to proposals for updates on deductive object models (e.g. [Abi88, KLS92]) have to be further investigated.

Additionally we have to work on optimizations of interaction executions and on mapping the abstract execution model to (active) (object-oriented) database models. Composite state transitions introduced in this report have to be integrated with suitable transaction models e.g. for refinement of events [EDS93, FM93] and long term activities. Candidates for implementation are nested transaction models and ConTracts [WR92].

Bibliography

[Abi88] Abiteboul, S.: Updates: a New Frontier. In: M. Gyssens, D. Van Gucht, J. Paredaens (ed.): *Int. Conf. on Database Theory (ICDT'88).* Springer LNCS 326, 1988, pp. 1–18.

[Agh86] Agha, G.A.: *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

[Alb91] Alblas, H.: Introduction to Attribute Grammars. In: Alblas, H.; Melichar, B. (eds.): *Attribute Grammars, Applications and Systems.* Springer, LNCS 545, 1991, pp. 1–15.

[AM91] Alblas, H.; Melichar, B. (eds.): *Attribute Grammars, Applications and Systems.* Springer, LNCS 545, 1991.

[Bee90] Beeri, C.: A Formal Approach to Object Oriented Databases. *Data & Knowledge Engineering*, Vol. 5, No. 4, 1990, pp. 353–382.

[BM91] Beeri, C.; Milo, T.: A Model for Active Object Oriented Database. In: Lohman, G. M.; Sernadas, A.; Camps, R. (eds.): *VLDB'91.* Morgan Kaufman, 1991, pp. 337–349.

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

[Con94] Conrad, S.: On Certification of Specifications for TROLL *light* Objects. In: Orejas, F. (ed.): *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (WADT/Compass'92.* Springer, LNCS, 1994.

[CSS89] Costa, J.-F.; Sernadas, A.; Sernadas, C.: OBL–89 User's Manual (Version 2.3). Internal report, INESC, Lisbon, 1989.

[Day88] Dayal, U.: Active Database Management Systems. In: *Proc. Conf. on Data and Knowledge Bases*, Jerusalem, 1988. pp. 150–169.

[DBM88] Dayal, U.; Buchmann, A.; McCarthy, D.R.: Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In: Dittrich, K. R. (ed.): *Advances in Object-Oriented Database Systems (Proc. 2nd Int.*

Workshop on Object-Oriented Database Systems), Bad Münster am Stein, 1988. LNCS 334, Springer, Berlin, 1988.

[DDP93] Dubois, E.; Du Bois, P.; Petit, M.: O-O Requirements Analysis: An Agent Perspective. In: Nierstrasz, O. (ed.): *ECOOP'93—Object-Oriented Programming (Proc. 7th European Conference)*, Kaiserslautern, 1993. LNCS 707, Springer-Verlag, Berlin, 1993, pp. 458–481.

[DJL86] Deransart, P.; Jourdan, M.; Lorho, B.: A Survey on Attribute Grammars, Part I: Main Results on Attribute Grammars. Rapports de Recherche 485, Institut National de Recherche en Informatique et en Automatique (INRIA), 1986.

[DPG91] Diaz, O.; Paton, N.; Gray, P.: Rule Management in Object-Oriented Databases: A Uniform Approach. In: *Proc. 17th Int. Conf. on Very Large Databases VLDB'91*, 1991.

[EDS93] Ehrich, H.-D.; Denker, G.; Sernadas, A.: Constructing Systems as Object Communities. In: Gaudel, M.-C.; Jouannaud, J.-P. (eds.): *Proc. TAPSOFT'93: Theory and Practice of Software Development.* LNCS 668, Springer, Berlin, 1993, pp. 453–467.

[EF81] Engelfriet, J.; Filè, G.: The formal Power of One-Visit Attribute Grammars. *Acta Informatica*, Vol. 16, 1981, pp. 275–302.

[FM93] Fiadeiro, J.L.; Maibaum, T.: Actions are Objects: Refinement in the temporal Logic of Objects. In: Lipeck, U.W.; Koschorreck, G. (eds.): *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, 1993, pp. 213–226.

[FSMS91] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. 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. 243–284.

[GCH93] Gogolla, M.; Conrad, S.; Herzig, R.: Sketching Concepts and Computational Model of TROLL *light.* In: Miola, A. (ed.): *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93).* Springer, Berlin, LNCS 722, 1993, pp. 17–32.

[Hew77] Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, Vol. 8, 1977, pp. 323–364.

[HJ92] Hartmann, T.; Jungclaus, R.: Abstract Description of Distributed Object Systems. In: Tokoro, M.; Nierstrasz, O.; Wegner, P. (eds.): *Proc. ECOOP'91 Workshop on Object-Based Concurrent Computing. Geneva (CH), 1991.* Springer, LNCS 612, Berlin, 1992, pp. 227–244.

[Hoh93] Hohenstein, U.: *Formale Semantik eines erweiterten Entity-Relationship-Modells*, volume 4 of *Teubner-Texte zur Informatik*. Teubner., 1993.

[HSJ+94] Hartmann, T.; Saake, G.; Jungclaus, R.; Hartel, P.; Kusch, J.: TROLL2: Revised Version of the Modelling Language TROLL . Informatik-Bericht, TU Braunschweig, 1994. *In preparation.*

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

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

[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.

[KLS92] Kramer, M.; Lausen, G.; Saake, G.: Updates in a Rule-Based Language for Objects. In: Yuan, Li-Yan (ed.): *Proc. 18th Int. Conf. on Very Large Databases, Vancouver*, 1992, pp. 251—262.

[Knu68] Knuth, D.E.: Semantics of Context Free Languages. *Math. Systems Theory*, Vol. 2, 1968, pp. 127–145.

[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.

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

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

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

[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.

[SFSE89] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (eds.): *Information System Concepts: An In-Depth Analysis*, Namur (B), 1989. North-Holland, Amsterdam, 1989, pp. 225–246.

[SJ92] Saake, G.; Jungclaus, R.: Specification of Database Applications 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.

[SJH94] Saake, G.; Jungclaus, R.; Hartmann, T.: Application Modelling in Heterogeneous Environments using an Object Specification Language. *Int. Journal of Intelligent and Cooperative Information Systems*, 1994. *To appear.*

[SSC92] Sernadas, A.; Sernadas, C.; Costa, J. F.: Object Specification Logic. Research report, INESC/DMIST, Lisbon (P), 1992. *To appear in Journal of Logic and Computation.*

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Stoecker, P.M.; Kent, W. (eds.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87.* VLDB Endowment Press, Saratoga (CA), 1987, pp. 107–116.

[SSG+91] Sernadas, A.; Sernadas, C.; Gouveia, P.; Resende, P.; Gouveia, J.: OBLOG – Object-Oriented Logic: An Informal Introduction. Internal report, INESC, Lisbon, 1991.

[Str86] Stroustrup, B.: *The C++ Programming Language.* Addison-Wesley, Reading, 1986.

[SY87] Shibayama, E.; Yonezawa, A.: Distributed Computing in ABCL/1. In: Yonezawa, A.; Tokoro, M. (eds.): *Object-Oriented Concurrent Programming*, Cambridge, MA, 1987. The MIT Press, pp. 91–128.

[TNCK91] Tanaka, A. K.; Navathe, S. B.; Chakravarthy, S.; Karlapalem, K.: ER-R: An Enhanced ER Model with Situation-Action Rules to Capture Application Semantics. In: Teory, T. (ed.): *Proc. 10th Int. Conf. an the ER-Approach*, San Mateo, 1991. pp. 59–75.

[Tsa89] Tsakpinis, A.: *Werkzeuge zur Optimierung von Logischen Programmen: Erkenntnisse aus einem Vergleich mit Attributierten Grammatiken.* PhD thesis, Univ. Erlangen, 1989.

[WF90] Widom, J.; Finkelstein, S.J.: Set-Oriented Production Rules in Relational Database Systems. In: *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, 1990. pp. 259–270.

[Wie90a] Wieringa, R. J.: *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[Wie90b] 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.

[Wil79] Wilhelm, R.: Attributierte Grammatiken. *Informatik-Spektrum*, Vol. 2, 1979, pp. 139–131.

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

[WR92] Wächter, H.; Reuter, A.: The ConTract Model. In: Elmagarmid, A. K (ed.): *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, Palo Alto, 1992, pp. 219–263.

[YSTH87] Yonezawa, A.; Shobayama, E.; Takada, T.; Honda, Y.: Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In: Yonezawa, A.; Tokoro, M. (eds.): *Object-Oriented Concurrent Programming*, Cambridge, MA, 1987. The MIT Press, pp. 55–90.