Chapter 4

SEMANTICS OF OBJECT-ORIENTED LANGUAGES

A. WAGNER
University of Paderborn,
Department for Mathmatics and Computer Science,
Warburger Str. 100, D-33098 Paderborn
E-mail: awa@uni-paderborn.de

M. GOGOLLA
University of Bremen,
Department for Mathmatics and Computer Science,
PO Box 330440, D-28334 Bremen,
E-mail: gogolla@informatik.uni-bremen.de

In this contribution we relate the textual object description language TROLL *light*, the graphical object modeling language UML and an algebraic representation of object communities serving as semantic domain. Our focus is on the relationship between TROLL *light* and the algebraic representation providing an operational semantics for the textual specification language. The graphical notation of the UML is used to visualize the algebraic representation in an intuitive way.

Contents

4.1 Introduction **182**
4.2 Drafting TROLL light Concepts **183**
4.3 Other Representations of TROLL light Descriptions **185**
4.4 Abstract Syntax of TROLL light **190**
4.5 Operational Semantics for TROLL light . . . **194**
4.6 Conclusion **201**
References **202**
Appendix **204**
4.A Complete Specification **204**
4.B Basic Graph Transformation Definitions . . . **209**

4.1 Introduction

Graph transformations can be applied everywhere, where graphs are used as a natural and intuitive description of complex phenomena. One field where this is the case is object modeling. The important role graphs play during object-oriented analysis and design [2,16,3] shows their usefulness in the context of object orientation. A main characteristic of graph transformations is its direct operational executability. Combining both observations, it seems to be natural to use graph transformations to specify the operational behavior of objects.

In this contribution we follow this idea by providing an operational semantics for the textual object description language TROLL *light* [5,10,11] by means of graph transformations. But this is not the only possible application of graph transformations in the object-oriented field because what we are doing can be used also in combination with the graphical object modeling language UML [3].

The very basic idea of TROLL *light* is the uniform and coherent specification of both structure and behavior of objects. The underlying semantic model of TROLL *light* is a transition system or (as we call it) an object community. In the single states, information about objects existing in the current state is represented, and state transitions are caused by occurrence of finite sets of events. The aim of TROLL *light* is to model information systems and not systems in general. Therefore we do not take into account what is called in the literature true concurrency but we are satisfied with a simpler semantic domain in favor of ease of concepts. We concentrate on TROLL *light* but the results achieved are applicable to other related object description languages like OBLOG [17], TROLL [12], or CMSL [21] and even to other languages also following the object paradigm like MONDEL [18], ALBERT [7], or Π [9].

Numerous approaches of graph transformations have been put forward, among them the logical approach [6], the set theoretic approach [14], and the algebraic approach [8]. In this paper we follow the algebraic approach, mainly for two reasons: First of all, the algebraic approach allows for a stepwise rule construction and synchronization of rules using the concept of amalgamation [1]. Second, the same underlying concepts as known for the specification of abstract data types [4,15] serve as a means for a smooth integration of data type aspects into our model. In particular we use an approach to the single pushout transformation [13] of partial algebras, where graphs as well as data structures are encoded into the same partial algebra [20].

The structure of this contribution is as follows. First we sketch basic TROLL *light* concepts by means of an example. In the next section we consider other possible representations of TROLL *light* descriptions. The first is an object-oriented model using UML, the second is an algebraic representation. We show the close relationship between both, which enables us to use the UML notation later to visualize the algebraic representation. The forth section introduces the abstract syntax of TROLL *light*, which is used in the fifth section to define its operational semantics. The last section contains some concluding remarks.

4.2 Drafting TROLL light Concepts

We explain central TROLL light concepts by means of an example describing a car rentals application scenario. One of the central object types is the booking template as given in the appendix. Booking objects use the data types String, Int, and Bool and the object types Customer and Car. They have object-valued attributes TheCustomer and TheCar of object sort customer and car, respectively. The convention is that (data and object) types start with upper-case letters whereas corresponding sorts begin with lower-case ones. Furthermore, there are data-valued attributes IsClosed, BookingCat, StartDay and EndDay giving a switch for whether there is still something to do with the booking or not, a booking category describing the car's desired equipment, and the required start and end day of the car usage. There are derived attributes (whose values are not stored but are calculated from other items) IsOpen and IsCurrent. The things to possibly change the state of a booking are the events (or operations), in this particular case, a birth event createBooking, an event makeCurrent for actually assigning a car to a booking and an event to close the current booking.

The possible object states are restricted in the constraints section. For demonstration purposes, we have modeled the booking category and the dates as integer values. A more sophisticated choice would be to introduce appropriate data types for these items. But nevertheless in both cases we need to require somewhere that the booking's start day lies before (or is equal to) the end day. One place to do this is the constraints section described here, another choice would be a precondition in the behavior pattern to be described below. The valuations determine how the events (if they occur) effect the attribute values. The derivation section gives rules for computing the derived attribute values from other given things. In contrast to the simple derivations here, quite complex rules are also allowed in this place. For example, the Company template as given in the appendix computes in the DeliverCarsMenu derived attribute of sort bag(tuple(int,int,bag(tuple(string,int)))) all open booking numbers and

booking categories together with the collection of all free car numbers and car categories which fit to the booking. Last, as again shown in the booking template, the possible occurrence of events is described in a CSP-like notation in the behavior patterns. There is always one process with the name of the template to be defined which is the starting process, so here a booking's life begins with a createBooking event; after that the makeCurrent event is allowed (provided that the category of the given car fits the category of the booking); at last the event close may appear leading to a state where no further events are possible, but observations on the attributes can still be made.

The Booking template example, however, does not mention the sub-objects and the interaction section of a template. In the sub-objects section it is possible to introduce non-sharable components of the current object (sharing is possible via object-valued attributes). For example, the Company template as given in the appendix describes the structure of the overall system object (in database terms the schema) containing all other objects. Therefore, this template mentions the customers, cars and bookings as its sub-objects. Regarding object identification, from the Company point of view a booking is identified by a booking number (given as an integer number parameter for the sub-object symbol Bookings). From a conceptual point of view, the sub-object symbol Bookings gives a means for accessing booking objects from company objects. From a technical point of view, the symbol Bookings can be looked at and used in three different ways: It can be used as (1) an infinite array (notation: `Bookings(BookingNo : int): booking`) with an integer index giving for each integer a booking object (in each state there is only a finite number of booking objects different from the undefined booking), (2) a map (notation: `Bookings : map(BookingNo : int, Booking : booking)`) from integers to bookings, or (3) a set of tuples (notation: `Bookings : set (tuple (BookingNo : int, Booking : booking))`) with two components of sort integer and booking and with the additional constraint that the tuple set also establishes a function (`FORALL (B1, B2 IN Bookings) BookingNo (B1) = BookingNo(B2) IMPLIES Booking (B1) = Booking(B2)`).

In the interaction section, communication patterns between objects are specified. For instance, it is possible to require the simultaneous occurrence of an event in a component sub-object and an event in the enclosing aggregating object. In the Company template, the deliverCar event is, for instance, synchronized by interaction rules with the pickUp event of a corresponding car object and a makeCurrent event of a corresponding Booking object.

4.3 Other Representations of TROLL light Descriptions

In this section we consider other possible representations of TROLL *light* descriptions. The first is an object-oriented model using UML. The second is an algebraic representation. We show the close relation between both, which enables us to use UML notation later in the paper to visualize the algebraic representation.

4.3.1 Representing TROLL light Descriptions with UML

In Fig. 4.1 and Fig. 4.2 we have displayed the UML representation of a structure model corresponding to the attributes, sub-objects, and events sections of appropriate TROLL light templates for our scenario. Templates together with their data-valued attributes and events are pictured as rectangles. Associations (object-valued attributes and sub-object relationships) between object types are denoted as lines. As displayed by the diamond, all Customer, Booking, and Car objects are aggregates (sub-objects) of an object belonging to template Company[1]. Multiplicity is denoted as adornments at association edges. For example, the stars * close to Customer, Booking and Car denote that a company can consist of many such objects whereas the adornments 1 close to the diamonds near company expresses that each customer, booking, and car belongs to exactly one company. Also the references of a booking to its customer and car are given. In general, multiplicity is denoted by sets of integer intervals of the form low..high. The adornments * and 1 are abbreviations for 0..* and 1..1 where the star * expresses an unbounded cardinality.

The dynamic aspect is visualized by the statechart-like diagram in Fig. 4.13. Here, the behavior patterns are presented. For example, the process description of Company is given by the upper left side of the diagram. States correspond to process names and transitions to event occurrences. For reasons of simplicity, the rather involved event preconditions are not displayed. The interaction rules could be shown as additional adornments of edges like deliverCar / { pickUp, makeCurrent } because the event deliverCar in the DeliverCars state calls for event pickUp in Car and for event makeCurrent in Booking. The general possibility of concurrent event occurrences is represented as an orthogonal composition of and-states.

[1]This object may be viewed as "the" system object in the sense that all other objects appear as components respectively sub-objects of this distinguished object. In the TROLL light specification, however, the company template is just an ordinary template. But its role as the system object template becomes evident when the TROLL light animation system is invoked by telling it that the session should start with exactly one object of sort company.

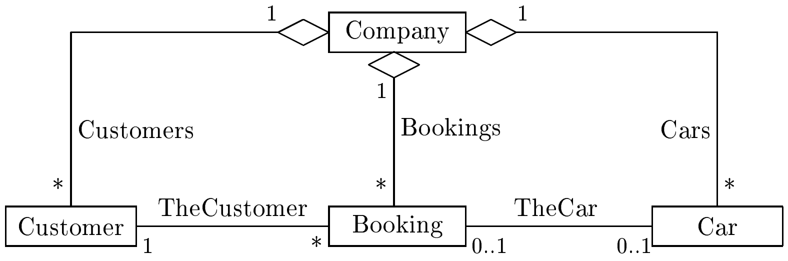

Figure 4.1: Description of templates, sub-objects and attributes

Booking
BookingCat : int StartDay, EndDay : int IsClosed, IsOpen, IsCurrent : bool
createBooking(aCustomer,aCat,aStartDay,anEndDay) makeCurrent(aCar) close

Figure 4.2: Description of TROLL light attributes and events

4.3.2 An Algebraic Version of TROLL light Descriptions

An algebraic version of TROLL *light* descriptions is based on the notions of a signature and an algebra. A signature consists of sort and operation symbols. It describes static structural information like templates, attributes and sub-objects. For a formal definition of these basic notions we refer to the appendix. Templates as well as data types, attributes and sub-objects become sorts of the signature. As a naming convention we use $attribute(template)$ as sort symbol for an attribute of a template and analogously for sub-objects. Moreover for each attribute we have an additional value sort. Elements of such a value sort can be thought of as being symbolic values. Sub-objects do not possess such value sorts. This difference will later model the fact that an attribute value may change, whereas a sub-object belongs to the same object for all its life. Operation symbols are used to represent the parameter sorts and the value sorts of the attributes. For each attribute and sub-object a domain operation (short d) assigns the attribute or sub-object to its template. Range operations (short r) get the attribute or sub-object and the parameters as parameters themselves. In case of a sub-object they are mapped to the sub-object itself. In case of an attribute they are mapped to a symbolic value. Finally we have value operations (short v) mapping symbolic values to actual ones.

In the following we give part of the signature for the company template.

sorts: Company,
String, Int,
Customer, Car, Booking,
Customers(Company), Cars(Company), Bookings(Company),
Today(Company), NextCustomerNo(Company), NextBookingNo(Company), ...
Today(Company)Val, NextCustomerNo(Company)Val, NextBookingNo(Company)Val, ...
opns: d: Company $\rightarrow$ Customers(Company),
d: Company $\rightarrow$ NextCustomerNo(Company), ...
r: Today(Company) $\rightarrow$ Today(Company)Val,
r: Cars(Company), String $\rightarrow$ Car,
v: Today(Company)Val $\rightarrow$ Int, ...

A partial algebra w.r.t. this signature models a state of a TROLL *light* object community. Moreover morphisms between partial algebras can be used to express transformation rules. See the appendix for a definition. The general idea is to partially map the pattern to be replaced to the structure replacing it by the morphism. In our context these rules are used to describe the effects of events. The following rule for example describes the combined effect of the two events `Company.addCustomer(aName, anAddress)` and `Customer.create(aName, anAddress)`. A new customer sub-object for the company is created, which can be accessed via the parameter having the old value of the attribute `NextCustomerNo`. Name and address attributes for the new object are set. The attribute `NextCustomerNo` of a company is increased by one. The behavior aspects of the events are neglected. An algebraic version of this rule is depicted in Fig. 4.3. Boxes denote elements of the according sorts. Numbers stand for the identity of elements, i.e., elements with the same number on the left and right hand side of the rule are preserved. Variables of data types are identified by a name instead of a number. Arrows denote the definedness of operations, where the source of the arrow is the domain of the operation and the target its value.

If this rule will be applied in order to represent a change of the state of the object community, an occurrence of the left hand side has to be found in the algebra representing the current state. Such an occurrence is called redex. The application of the rule follows exactly the intuition of other graph transformation approaches, with the only exception that there is a built-in application condition, stating that an operation can only be defined for an argument, if it is not already defined for this argument. We make use of this application condition in order to ensure that parameters uniquely identify attribute values or

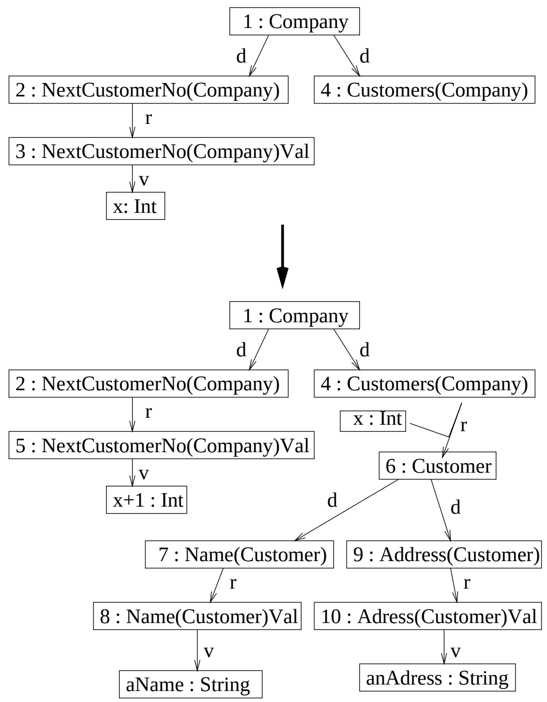

Figure 4.3: Formal version of the rule for adding a customer

sub-objects for parameterized attributes or sub-object relationships. The rule in Fig. 4.3 for example is only applicable, if the Company does not already have a customer identified by the number x. For a formal definition of rule application we refer to the appendix.

4.3.3 Relating TROLL light, UML and the Algebraic Version

By now we sketched three different versions of the same case study. The TROLL *light* version is a complete specification, whereas the UML version as well as the algebraic version are not complete. Partially this is because of a lack of space. So we could add some collaboration diagrams (or other diagrams) to the UML specification in order to describe the interaction between different objects. This would also include object creation and deletion. But the question which new values have to be assigned to the attributes would be still open.

Looking at the algebraic version we have shown that it can also deal with the static structure. Moreover, exactly changing the values of attributes can easily be described in this version by rules. Interaction between different objects is also unproblematic. It will be described by the composition of smaller rules to larger ones. What remains open here are behavior aspects. This is not due to general limitations of graph transformations. The basic ideas for the integration of the behavior description will be given later in this paper.

This situation is illustrated by Fig. 4.4. In the rest of this paper we will concentrate on the relationship between TROLL *light* and the transformation of partial algebras, since we want to sketch how algebra transformations or graph transformations in general may be used to define the semantics of object-oriented languages. For this purpose we introduce the abstract syntax of TROLL *light* in the next section.

Nevertheless also the UML version and the algebraic version are closely related. Signatures may be also seen as formalizations of static structure diagrams. Because of this correspondence algebraic rules can be depicted more clearer as rules for the transformation of static structure diagrams. Fig. 4.5 shows this better understandable version of the rule depicted in Fig. 4.3. Because of the better understandability we will use this way of presenting our rules in the rest of this paper and omit the more technical one. For a deeper understanding of the integration of graph transformation concepts in object modeling in order to be able to describe functional aspects of operations (such as the computation of new attribute values) we refer to [19].

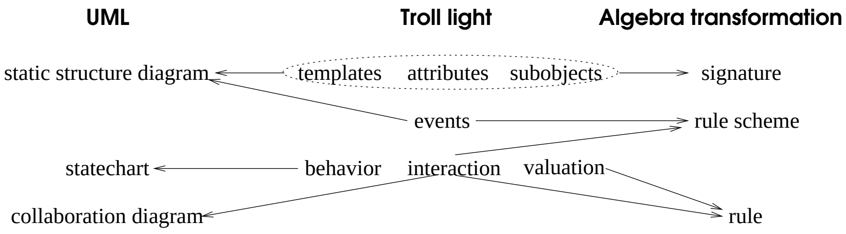

Figure 4.4: Relationship between different descriptions

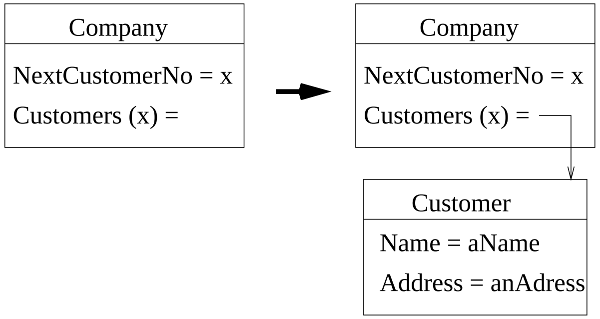

Figure 4.5: Intuitive version of the rule for adding a customer

4.4 Abstract Syntax of TROLL light

In this section we introduce the abstract syntax of TROLL *light* as needed for explaining the operational semantics of the language by means of graph transformations.

Sorts

We assume we have a global data signature $\Sigma_D = (S_D, \Omega_D, \Pi_D)$ with data sorts, data operations, and data predicates. Furthermore, we have a fixed set T of object descriptions (templates) where each template induces an object sort (surrogate sort). S_O contains the set of object sorts, and we have mappings $\mathrm{osort} : T \to S_O$ and $\mathrm{tpl} : S_O \to T$, each inverse to the other.

Example 4.1 (Sorts)

In our example we have:
$S_D = \{int, string\}$, $S_O = \{company, customer, booking, car\}$. □

Object interface

The object interface determines the signatures used. It is divided into a data signature (as indicated above) and an object signature.

The abstract syntax of an object description is (preliminarily) determined by the following sets:

- a set of indexed sub-object symbols $\textsc{Sub}(t) = \{\textsc{Sub}(t)_{[s,]o}\}_{s\in S,\ o\in S_O}$,
- a set of indexed attribute symbols $\textsc{Att}(t) = \{\textsc{Att}(t)_s\}_{s\in S}$,

- a set of indexed event symbols
  $\textsc{Evt}(t) = \{\textsc{Evt}(t)_{k[,s]}\}_{k \in \{\text{birth},\text{life},\text{death}\},\ s \in S}$.

Example 4.2 (Object interface)
For the company template we have the following sub-object, attribute, and event symbols.

$$\begin{aligned}\textsc{Sub}(\text{Company}) = \{\ & \text{Customers}_{\text{int},\text{customer}},\\ & \text{Bookings}_{\text{int},\text{booking}},\\ & \text{Cars}_{\text{string},\text{car}}\ \}\end{aligned}$$

$$\begin{aligned}\textsc{Att}(\text{Company}) = \{\ & \text{Today}_{\text{int}}\\ & \text{NextCustomerNo}_{\text{int}},\\ & \text{NextBookingNo}_{\text{int}},\\ & \text{CurrentCustomersAndCars}_{\text{bag(tuple(string,string))}},\\ & \text{ReturnCarsMenu}_{\text{bag(string)}},\\ & \text{DeliverCarsMenu}_{\text{bag(tuple(int,int,bag(tuple(int,int))))}}\end{aligned}$$

$$\begin{aligned}\textsc{Evt}(\text{Company}) = \{\ & \text{createCompany}_{\text{birth},\text{int}},\\ & \text{addCustomer}_{\text{life},\text{tuple(string,string)}},\\ & \text{addBooking}_{\text{life},\text{tuple(customer,int,int,int)}},\\ & \text{incTime}_{\text{life},\text{int}},\\ & \text{returnCar}_{\text{life},\text{string}},\\ & \text{closeReturnCars}_{\text{life}},\\ & \text{addCar}_{\text{life},\text{tuple(string,int,string)}},\\ & \text{deliverCar}_{\text{life},\text{tuple(int,string)}},\\ & \text{closeDeliverCars}_{\text{life}}\ \}\end{aligned}$$

□

Terms and formulas
The abstract syntax also allows to build terms and formulas. Terms stand for arbitrary data- or object-valued units. Due to the presence of sort expression constructors like `Set`, `List`, `Bag`, or `Tuple` and corresponding operations, the terms can have complex result sorts. Formulas denote truth values and their syntax is close to first-order calculus. Formulas are built from (1) atomic formulas, (2) logical connectives like negation, conjunction and disjunction, and (3) existential and universal quantification. We do not go into the details of terms and formulas because the rules used here for building them should be clear from the context.

Example 4.3 (Terms and formulas)
We mention two examples from the booking and company template.

```
(1) BookingCat(B)
(2) TheCustomer(B)
(3) aDay < Today
    AND
    IsOpen(B) IMPLIES aDay <= StartDay(B)
    AND
    IsCurrent(B) IMPLIES aDay <= EndDay(B)
```

(1) The first term given computes for a booking `B` the booking category `BookingCat(B)`, a term of data sort `int`. (2) The second term `TheCustomer(B)` gives for a booking its customer, a term of object sort `customer`. (3) The stated formula with free variable `aDay` expresses that `aDay` lies before `Today` but `aDay` does not lie after the start day of an open booking or after the end day of a current booking. □

S-terms
S-terms (sub-object terms) are needed for referencing objects which currently do not live. S-terms can only occur on the right-hand side of interaction rules where they denote unborn object whose birth is called for by the left-hand side of the interaction rule.

Let an object description $t \in T$ ($\text{osort}(t) = o$) be given. The syntax of t-local S-terms is defined by a set of sort-indexed terms $S\text{-}Term(t) = \{S\text{-}Term(t)_o\}_{o \in S_O}$ and a function free : $S\text{-}Term(t) \to \mathcal{F}(Var)$, determined as follows:

If $u \in \textsc{Sub}(t')_{s,o''}$ ($\text{osort}(t') = o'$), $\tau_1 \in Term(t)_{o'}$ and $\tau_2 \in Term(t)_s$, then $\boldsymbol{u(\tau_1, \tau_2)} \in S\text{-}Term(t)_{o''}$ with $\text{free}(u(\tau_1,\tau_2)) := \text{free}(\tau_1) \cup \text{free}(\tau_2)$.

Example 4.4 (S-terms)
Consider the following two interaction rules of the company template.

```
addCustomer(aName,anAddress) >>
  Customers(NextCustomerNo).createCustomer
                                 (aName,anAddress);
addBooking(aCustomer,aCat,aStartDay,anEndDay) >>
  Bookings(NextBookingNo).createBooking
             (aCustomer, aCat,aStartDay,anEndDay);
addCar(aCarNo,aCarCat,aBrand) >>
  Cars(aCarNo).createCar(aCarNo,aCarCat,aBrand);
```

Here, the terms `Customers(NextCustomerNo)`, `Bookings(NextBookingNo)`, and `Cars(aCarNo)` induce the S-terms $Customers(self, NextCustomerNo)$, $Bookings(self, NextBookingNo)$, and $Cars(self, aCarNo)$. □

E-terms

E-terms are evaluated to events. In general, they consist of (1) an object where an event takes place and (2) an event with appropriate parameters.

Let an object description $t \in T$ ($\mathrm{osort}(t) = o$) be given. The syntax of t-local E-terms is given by a set $E\text{-}Term(t)$ and a function $\mathrm{free} : E\text{-}Term(t) \to \mathcal{F}(Var)$, determined as follows:

> If $e \in \mathrm{EVT}(t')_{k,s}$ (where $\mathrm{osort}(t') = o'$), $\tau_1 \in Term(t)_{o'}$ for $k \in \{\mathrm{life}, \mathrm{death}\}$, $\tau_1 \in S\text{-}Term(t)_{o'}$ for $k = \mathrm{birth}$, and $\tau_2 \in Term(t)_s$, then $\boldsymbol{\tau_1.e(\tau_2)} \in E\text{-}Term(t)$ with $\mathrm{free}(\tau_1.e(\tau_2)) := \mathrm{free}(\tau_1) \cup \mathrm{free}(\tau_2)$.

Example 4.5 (E-terms)

Consider again the interaction rules of the company template which we have repeated above in the context of S-terms. All left and right-hand sides of the interaction rules induce E-terms. For example, the first rule induces the E-terms

$$self.addCustomer(aName, anAddress)$$

and

$$Customers(self, NextCustomerNo).createCustomer(aName, anAddress).$$

□

Object dynamics

Object dynamics concerns valuation rules, interaction rules and state transitions. In the concrete syntax a valuation rule $(form, evt, att, term)$ is denoted as `{ form } [ evt ] att=term`. An interaction rule $(form, term, evt, e\text{-}term)$ is given by

```
{form} term.evt >> e-term.
```

A transition $(form, evt, start, end)$ is represented as

```
PROCESS start = ({form} evt -> end).
```

For an event symbol e of an object description $t \in T$ the set $V(e)$ is defined as follows:

$$V(e) := \begin{cases} \{\mathrm{V}_s\} & \text{if } e \text{ is an event symbol with parameter } s \\ \emptyset & \text{otherwise} \end{cases},$$

where V_s is a variable of sort s.

The abstract syntax of an object description $t \in T$ is (in addition to the above entities) given by the following sets:

- a set of valuation rules $\mathrm{VAL}(t) \subset Form(t) \times \mathrm{EVT}(t) \times \mathrm{ATT}(t) \times Term(t)$, where for all $(\varphi, e, a, \tau) \in \mathrm{VAL}(t)$:
  - $e \in \mathrm{EVT}(t)_{\mathrm{birth}[,s]} \cup \mathrm{EVT}(t)_{\mathrm{life}[,s]}$
  - $a \in \mathrm{ATT}(t)_s$ implies $\tau \in Term(t)_s$,
  - $\mathrm{free}(\varphi) \cup \mathrm{free}(\tau) \subseteq V(e)$,
- a set of interaction rules $\mathrm{IAC}(t) \subset Form(t) \times \mathrm{EVT}(t) \times E\text{-}Term(t)$, where for all $(\varphi, e, \tau) \in \mathrm{IAC}(t)$:
  - $e \in \mathrm{EVT}(t)_{\mathrm{life}[,s]} \cup \mathrm{EVT}(t)_{\mathrm{death}[,s]}$,
  - $\mathrm{free}(\varphi) \cup \mathrm{free}(\tau) \subseteq V(e)$,
- a set of state transitions $\mathrm{TRA}(t) \subset Form(t) \times \mathrm{EVT}(t) \times \mathrm{N}_0 \times \mathrm{N}_0$, where for all $(\varphi, e, n_1, n_2) \in \mathrm{TRA}(t)$:
  - $(\varphi, e, 0, n_2) \in \mathrm{TRA}(t)$ iff $e \in \mathrm{EVT}(t)_{\mathrm{birth}[,s]}$,
  - $(\varphi, e, n_1, 0) \in \mathrm{TRA}(t)$ iff $e \in \mathrm{EVT}(t)_{\mathrm{death}[,s]}$ and
  - $\mathrm{free}(\varphi) \subseteq V(e)$.

Example 4.6 (Object dynamics)

We show three examples for valuation rules, interaction rules, and state transitions.

```
{TRUE} [PickUp] Available = FALSE
{TRUE} deliverCar >> Cars(aCarNo).pickUp
PROCESS Customer = ({TRUE} createCustomer -> CustomerUpdate)
```

□

4.5 Operational Semantics for TROLL light

In this section we sketch the operational semantics of TROLL light defined in terms of graph transformations. It is defined using the abstract syntax introduced in the previous section. We have already shown how the signature for a TROLL light description is constructed in Section 4.3.2. This can be straight forward formalized using the sets of sub-object symbols $SUB(t)$ and the sets of attribute symbols $ATT(t)$ for all templates t of the description. We will concentrate on dynamic aspects here. In the following we define rules associated with certain elements of the abstract syntax. These rules will later be used to define the operational semantics.

Basic Rule
For each event symbol $e \in EVT(t)$ we define a basic graph transformation rule b_e, where b_e is given by an extension of

- a morphism preserving one element of sort $osort(t)$, if $e \in EVT(t)_{life}$ or
- a morphism deleting one element of sort $osort(t)$ and one element of each sort for a sub-object symbol $s \in SUB(t)$ as well as one element of each sort for an attribute symbol $a \in ATT(t)$, where all operations d are totally defined, if $e \in EVT(t)_{death}$ or
- a morphism inserting one element of sort $osort(t)$ and one element of each sort for a sub-object symbol $s \in SUB(t)$ as well as one element of each sort for an attribute symbol $a \in ATT(t)$, where all operations d are totally defined, if $e \in EVT(t)_{birth}$

by an according preservation, deletion or insertion of elements representing the parameters in $V(e)$.

Example 4.7 (Basic Rule)
Figure 4.6 shows the basic graph transformation rules for the events `pickUp` and `createCar` of template Car. b_{pickUp} is trivially preserving an entity of sort car, whereas $b_{createCar}$ not only creates a new entity of sort car, but also creates entities of its attribute sorts (as indicated by the numbers). Moreover the parameters of event `createCar` are represented in the rule. □

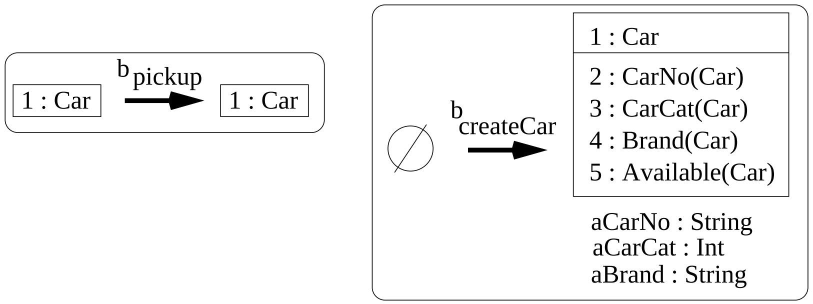

Figure 4.6: Examples of basic graph transformation rules

Valuation Rule
For each valuation rule $(\varphi, e, a, \tau) \in \text{VAL}(t)$ we define a valuation graph transformation rule $v_{(\varphi,e,a,\tau)}$ as an extension of the underlying basic graph transformation rule b_e by

- preserving additional elements associated with sub-terms of τ and
- preserving an entity of the attribute sort associated with a
- adding a symbolic value for each element of a sort associated with an attribute symbol to the left and right hand side of the rule, where all v operations are defined accordingly and the rule morphism deletes these values of the left hand side and
- adding an actual value for each symbolic value, where the actual value may be a variable, if it occurs on the rule's left hand side and is of a data type, and the actual value is induced by τ, if it occurs on the rule's right hand side.

Example 4.8 (Valuation Rule)
We resume the example of a valuation rule from the previous section.

```
{TRUE} [pickUp] Available = FALSE
```

In Fig. 4.7 we extend the basic rule b_{pickup} from Fig. 4.6. There is no sub-term of $\tau = FALSE$ which has to be preserved by the rule. Hence we only add an entity of sort $Available(Car)$ and the according symbolic value (represented as box) to the left and right hand side of the rule. The attribute is preserved, whereas the symbolic value is deleted and a new one is inserted. This fact is indicated by the use of the numbers in the figure. □

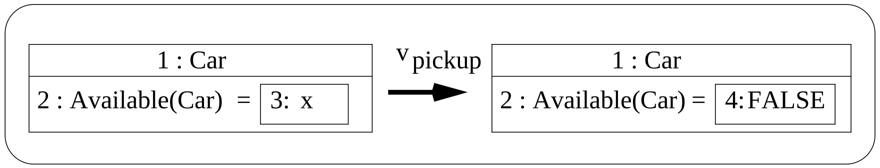

Figure 4.7: Example of a valuation graph transformation rule

Interaction Rule
For each interaction rule $(\varphi, e, \tau) \in IAC(t)$ we define an interaction graph transformation rule $i_{(\varphi,e,\tau)}$ as an extension of the disjoint union of the underlying basic graph transformation rules b_e and b_{e1} with $\tau = \tau 1.e1(\tau 2)$, where the disjoint union of two rules is constructed by the disjoint union of their right and left hand sides. The result is extended by

- dealing with additional elements associated with sub-terms of $\tau 1$ analogously to elements of the basic rule b_e, with the only exception that
- $e \in EVT(t)_{life}$ and $e1 \in EVT(t)_{birth}$ or $e1 \in EVT(t)_{death}$ and $\tau 1 \notin S\text{-}Term(t)$ implies that x of sort $Attr(t)Val$ with $v(x) \in b_{e1}$ is handled according to the elements of b_{e1}.

The main idea of this extension is that the description of the relation between the synchronizing object and the synchronized object belongs to the context of the synchronizing object and is hence handled accordingly. The exception is concerned with the symbolic value of the synchronized object. In the case that the synchronizing object is preserved, the symbolic value has to be handled analogously to the actual value.

Example 4.9 (Interaction Rule)
We resume the example of an interaction rule from the previous section.

```
{TRUE} deliverCar >> Cars(aCarNo).pickUp
```

In Fig. 4.8 we extend the disjoint union of the basic rules $b_{deliverCar}$ and b_{pickUp}, which preserve a company and a car respectively, by the elements associated with the term `Cars(aCarNo)`. This term denotes a sub-object relationship between the company and a car, which is preserved since the company is preserved. □

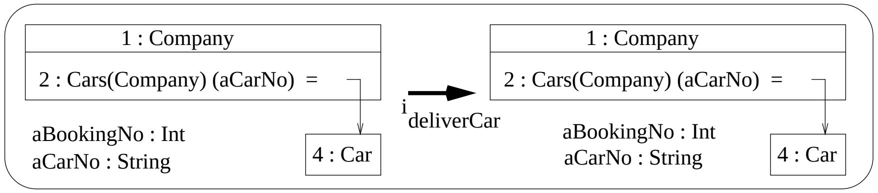

Figure 4.8: Example of an interaction graph transformation rule

Rule Scheme
The rule scheme shows how the different types of graph transformation rules derived from the different types of rules in the abstract syntax play together in order to describe the operational semantics of TROLL *light*. Hence the rule scheme describes the necessary synchronization and the according interfaces. Interfaces are formed by rules themselves. In order to serve as an interface, a rule must be part of the rule where it is the interface for. First of all, valuation rules for different attributes for the same event have to be synchronized. Their common interface is the basic rule for the event, which is part of all valuation rules by construction. Moreover, interaction rules have to be synchronized with valuation rules, if they describe the same event. Again the basic rule serves as an interface. But interaction rules have a second interface: the basic rule for the synchronized event. Figure 4.9 shows the general schema for putting rules into a rule scheme. Arrows denote a part of relationship. Each basic rule b_e is part of all its valuation and interaction rule. Moreover a basic rule can be part of an interaction rule for another event.

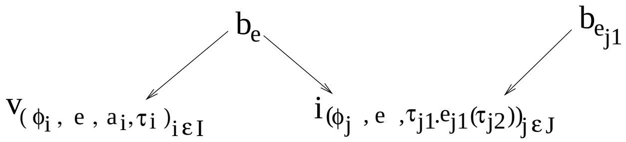

Figure 4.9: General case of a rule scheme for a TROLL *light* description

Example: Rule Scheme
Figure 4.10 shows the part of the rule scheme for our TROLL *light* description, which is concerned with event `deliverCar`. It contains the basic rules for events `makeCurrent, deliverCar` and `pickUp`. Event `deliverCar` is synchronized with the events `makeCurrent` and `pickUp` via the according interaction rules. Moreover there exist valuation rules for the latter two events.

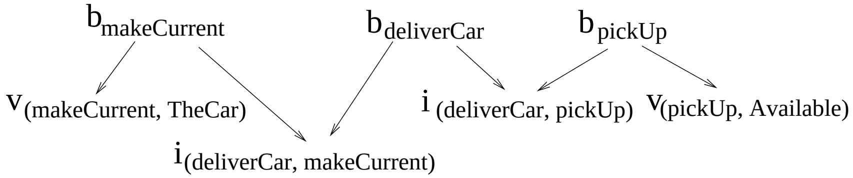

Figure 4.10: Rule scheme concerned with event `deliverCar`

Amalgamated Rule
If two rules are applied synchronously as described in the rule scheme, this is done by combining their effects in a newly constructed rule, called the amalgamated rule. Constructing this rule is done by gluing together the left hand sides of the rules to the left hand side of the amalgamated rule and their right hand sides to the new right hand side. For this gluing we use the same construction as for applying a rule. The embedding of the left and right hand side of the basic rule in the left and right hand sides of the rules to be synchronized can be viewed as rule and redex. Figure 4.11 visualizes this idea. The rule morphism of the amalgamated rule is uniquely defined for the elements of the left hand side of the amalgamated rule by the morphisms for the pre-images of these elements. Hence an element is deleted, if its pre-image in one of the rules is deleted. Otherwise it is mapped to this element of the right hand side of the amalgamated rule, which coincides with the image of its pre-image in one of the right hand sides of the rules to be synchronized.

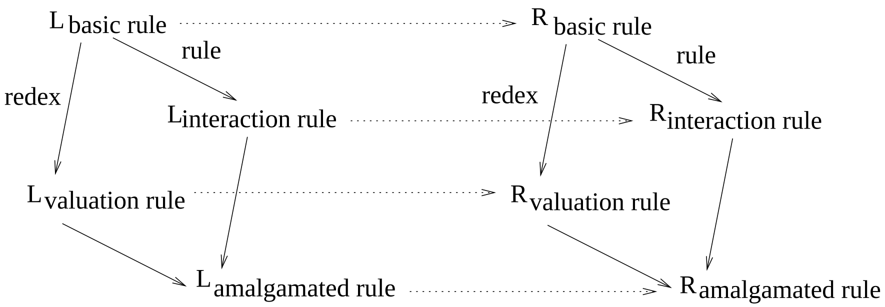

Figure 4.11: Synchronization as construction of the amalgamated rule

Example 4.10 (Amalgamated Rule)
Figure 4.12 shows an example of an amalgamated rule. It is the rule constructed for the rule scheme depicted in Fig. 4.10 concerned with the event `deliverCar` synchronizing the events `pickUp` and `makeCurrent`. Hence it contains the valuation rule for event `pickUp` setting the attribute Available of the car to False. Moreover, the attribute TheCar of the booking is set to the car object as done by the valuation rule for event `makeCurrent`. The car as a parameter object is passed from the interaction rule to the valuation rule, since it is part of the basic rule for event `makeCurrent`. □

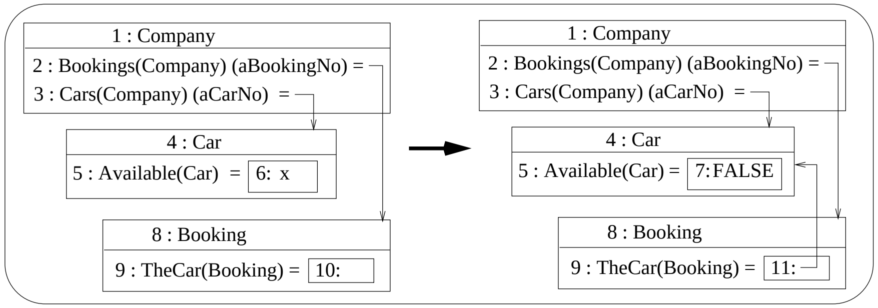

Figure 4.12: Amalgamated Rule for event `deliverCar`

Operational TROLL *light* Semantics
The rule scheme for a TROLL *light* description generally consists of many unconnected synchronization parts. The rules in one part are connected via interfaces, i.e., they have common basic rules. For each synchronization part we construct an amalgamated rule by iterated amalgamation of two of the involved rules. This is possible since the interface in terms of the basic rule is still part of the amalgamated rule. The operational semantics of a TROLL *light* description is then given by all possible derivations using the amalgamated rules for the synchronization parts of the rule scheme.

Remarks:
The following remarks are concerned with restrictions we made for this paper or which we have to make in general, since our semantics is not able to deal with the full range of concepts TROLL *light* provides.

- The behavior of TROLL *light* objects described by state transitions was not discussed during the construction of graph transformation rules above. Nevertheless state transitions can easily be added to the semantics by extending the signature by attributes modeling the current state of the objects, i.e., each object sort gets such a state attribute with according sorts and operations. The value sort of the attribute are the natural numbers, since states are described by natural numbers in the abstract syntax. Each state transition then becomes a valuation graph transformation rule for the state attribute constructed as all other valuation graph transformation rules.
- We did not show the integration of the data type specification explicitly here. If we presuppose an algebraic specification to be given, we could

formally deal with computations on data valued attributes by slightly changing the category on which the transformation is based [20]. This does not influence the way how the rules are applied without such computations.

- The representation of formulas as they occur in the abstract TROLL *light* syntax has also been omitted here. We have to state that we are not able to deal with the full range of possible formulas within our semantics. But in a restricted way formulas can be translated into additional application conditions for the rules, which restrict their applicability.
- In order to include derived attributes in our semantics we could divide between two types of rules: those described above induced by events and a second type used to compute new values for derived attributes. Within the semantics, a rule of the first type would only be allowed to be applied, if no rule of the second type is applicable any more. This idea is not yet worked out further. □

4.6 Conclusion

We have shown how graph transformations can be used to define semantics for object-oriented languages. In particular we have used the textual object description language TROLL *light* for this purpose. Moreover we have sketched the relationship of our approach with other graphical notations like the Unified Modeling Language. By discussing this relationship we were able to visualize our formalization using parts of this notation, which has become a standard in the area of object modeling in the last years.

As the underlying graph transformation approach we used a generalization of the single pushout approach to partial algebras. The main theoretical concept we have been using is that of amalgamation of rules in order to express synchronization.

Nevertheless, not all features of object specification languages were considered in detail here. Some aspects such as the integration of data types and the specification of the behavior have been omitted because of a lack of space. Some other aspects such as the integration of logical formulas as application conditions for rules or the use of rules in order to compute derived attributes are topics of ongoing research. Apart from treating TROLL *light* it would be interesting to draw the attention to other similar languages, in particular

CMSL [21] seems to be a stimulating candidate for such considerations.

References

1. P. Böhm, H.-R. Fonio, and A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *Journal of Computer and System Science*, 34:377–408, 1987.
2. G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.
3. G. Booch, I. Jacobsen, and J. Rumbaugh. *Unified Modeling Language. Notation Guide and Semantics*. Rational, 1997.
4. P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*, volume 32 of *Mathematical Research — Mathematische Forschung*. Akademie-Verlag, Berlin, 1986.
5. S. Conrad, M. Gogolla, and R. Herzig. TROLL light: A core language for specifying objects. Technical Report 92-02, Technical University of Braunschweig, 1992.
6. B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 193–242. North Holland, Amsterdam, 1990.
7. E. Dubois, P. Du Bois, and M. Petit. O-o requierements analysis: An agent perspektive. In O.M. Nierstrasz, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'93)*, pages 458–481, Berlin, 1993. Springer, LNCS 707.
8. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. 1st Graph Grammar Workshop*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer Verlag, 1979.
9. P. Gabriel. The object-based specification language π: Concepts, syntax, and semantics. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification - Proc. 8th Workshop on Specification of Abstract Data Types*, pages 254 –270, Berlin, 1993. Springer, LNCS 655.
10. M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL light. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, pages 17–32. Springer, LNCS 722, 1993.
11. R. Herzig, S. Conrad, and M. Gogolla. Compositional description of object communities with troll light. In C. Christment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94): Information Systems Design and Hypermedia*, Toulouse, 1994. Cepadues-Editions.

12. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The troll language. Technical Report 91-04, TU Braunschweig, 1991.
13. M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.
14. M. Nagl. Set-theoretic approaches to graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 41–54. Springer Verlag, 1987.
15. H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford University Press, Oxford, 1987.
16. J. Rumbaugh, M. Blaha, W. Premerlani, E. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.
17. A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-oriented specification of databases: An algebraic approach. In P. M. Stocker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB)*, pages 107–116, Palo Alto, 1987. Morgan-Kaufmann.
18. G. v. Bochmann, M. Barbeau, M. Erradi, P. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An object-oriented specification language. Technical Report Publication 748, Department d'Informatique et de Recherche Operationelle, Universite de Montreal, 1990.
19. A. Wagner. A formal object specification technique integrating object and functional model. *Int. journal of software engineering and knowledge engineering*, 7(4):503–524, 1997.
20. A. Wagner. *A Formal Object Specification Technique Using Rule-Based Transformation of Partial Algebras*. PhD thesis, TU Berlin, 1997.
21. R. Wieringa. Equational specification of dynamic objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990*, pages 415–438. North-Holland, 1991.

Appendix

4.A Complete Specification

4.A.1 Customer Template

```
TEMPLATE Customer
DATA TYPES
  String;
ATTRIBUTES
  Name : string;
  Address : string;
EVENTS
  BIRTH createCustomer(aName : string, anAddress : string);
  updateAddress(anAddress : string);
VALUATION
  [ createCustomer ]
      Name = aName, Address = anAddress;
  [ updateAddress ]
      Address = anAddress;
BEHAVIOR
  PROCESS Customer =
    ( createCustomer -> CustomerUpdate );
  PROCESS CustomerUpdate =
    ( { Address <> anAddress }
      updateAddress -> CustomerUpdate );
END TEMPLATE
```

4.A.2 Car Template

```
TEMPLATE Car
DATA TYPES
  String, Int;
ATTRIBUTES
  CarNo : string;
  CarCat : int;     -- 1..5
  Brand : string;
  Available : bool;
EVENTS
  BIRTH createCar
          (aCarNo : string, aCarCat : int, aBrand : string);
  pickUp;
  return;
CONSTRAINTS
  1 <= CarCat AND CarCat <= 5;
VALUATION
  [ createCar ]
      CarNo = aCarNo, CarCat = aCarCat,
```

```
      Brand = aBrand, Available = TRUE;
  [ pickUp ]
      Available = FALSE;
  [ return ]
      Available = TRUE;
BEHAVIOR
  PROCESS Car =
    ( createCar -> CarAvailable );
  PROCESS CarAvailable =
    ( { Available = TRUE }
      pickUp -> CarAway );
  PROCESS CarAway =
    ( { Available = FALSE }
      return -> CarAvailable );
END TEMPLATE
```

4.A.3 Booking Template

```
TEMPLATE Booking
DATA TYPES
  String, Int;
TEMPLATES
  Customer, Car;
ATTRIBUTES
  TheCustomer : customer;
  TheCar : car;
  BookingCat : int;   -- 1..5
  StartDay : int;     -- YYYYMMDD
  EndDay : int;       -- YYYYMMDD
  IsClosed : bool;
  DERIVED IsOpen : bool;
  DERIVED IsCurrent : bool;
EVENTS
  BIRTH createBooking(aCustomer : customer, aCat : int,
                      aStartDay : int, anEndDay : int);
  makeCurrent(aCar : car);
  close;
CONSTRAINTS
  1 <= BookingCat AND BookingCat <= 5;
  19900101 <= StartDay AND StartDay <= EndDay
                       AND EndDay <= 20001231;
VALUATION
  [ createBooking ]
      TheCustomer = aCustomer, BookingCat = aCat,
      StartDay = aStartDay, EndDay = anEndDay, IsClosed = FALSE;
  [ makeCurrent ]
      TheCar = aCar;
  [ close ]
      TheCar = NIL, IsClosed = TRUE;
DERIVATION
```



```
  IsOpen = UNDEF(TheCar) AND NOT(IsClosed);
  IsCurrent = DEF(TheCar) AND NOT(IsClosed);
BEHAVIOR
  PROCESS Booking =
    ( createBooking -> BookingOpen );
  PROCESS BookingOpen =
    ( { BookingCat <= CarCat(aCar) }
        makeCurrent -> BookingCurrent );
  PROCESS BookingCurrent =
    ( close -> BookingClosed );
  PROCESS BookingClosed = ( );
END TEMPLATE
```

4.A.4 Company Template

```
TEMPLATE Company
DATA TYPES
  String, Int;
TEMPLATES
  Customer, Car, Booking;
SUBOBJECTS
  Customers(CustomerNo : int) : customer;
  Cars(CarNo : string) : car;
  Bookings(BookingNo : int) : booking;
ATTRIBUTES
  Today : int;   -- YYYYMMDD
  NextCustomerNo, NextBookingNo : int;
  DERIVED CurrentCustomersAndCars :
    BAG(TUPLE(Name : string, CarNo : string));
  DERIVED ReturnCarsMenu : BAG(string);
  DERIVED DeliverCarsMenu : BAG(TUPLE(
    BookingNo : int, BookingCat : int,
    CarSelection : BAG(
      TUPLE(CarNo : int, CarCat : int))));
EVENTS
  BIRTH createCompany(aDay : int);
  addCustomer(aName : string, anAddress : string);
  addBooking(aCustomer : customer, aCat : int,
             aStartDay : int, anEndDay : int);
  incTime(aDay : int);
  returnCar(aCarNo : string);
  closeReturnCars;
  addCar(aCarNo : string, aCarCat : int, aBrand : string);
  deliverCar(aBookingNo : int, aCarNo : string);
  closeDeliverCars;
CONSTRAINTS
  NextCustomerNo >= 0;
  NextBookingNo >= 0;
  -- make sure each car is assigned
  -- to at most one booking
```

```
  FORALL (C IN Cars)
    (CNT(SELECT B
         FROM   B IN Bookings
         WHERE  TheCar(Booking(B)) = Car(C))
     <= 1);
VALUATION
  [ createCompany ]
      Today = aDay, NextCustomerNo = 0, NextBookingNo = 0;
  [ addCustomer ]
      NextCustomerNo = NextCustomerNo + 1;
  [ addBooking ]
      NextBookingNo = NextBookingNo + 1;
  [ incTime ]
      Today = aDay;
DERIVATION
  CurrentCustomersAndCars =
    SELECT Name(Customer(CUS)), CarNo(Car(CAR))
    FROM   CUS IN Customers, CAR IN Cars
    WHERE  EXISTS ( B IN Bookings )
             ( TheCustomer(Booking(B)) = Customer(CUS)
               AND TheCar(Booking(B)) = Car(CAR) );
  ReturnCarsMenu =
    SELECT CarNo(TheCar(Booking(B)))
    FROM   B IN Bookings
    WHERE  EndDay(Booking(B)) < Today AND
           IsCurrent(Booking(B));
  DeliverCarsMenu =
    SELECT BookingNo(B), BookingCat(Booking(B)),
      ( SELECT CarNo(Car(C)), CarCat(Car(C))
        FROM   C IN Cars
        WHERE  Available(Car(C)) AND
               CarCat(Car(C)) >=
               BookingCat(Booking(B)) )
    FROM   B IN Bookings
    WHERE  StartDay(Booking(B)) = Today AND
           IsOpen(Booking(B));
INTERACTION
  addCustomer >>
    Customers(NextCustomerNo).createCustomer(aName, anAddress);
  addBooking >>
    Bookings(NextBookingNo).createBooking(
      aCustomer, aCat, aStartDay, anEndDay);
  returnCar >>
    Cars(aCarNo).return,
    SINGLE(SELECT Booking(B)
           FROM   B IN Bookings
           WHERE  CarNo(TheCar(Booking(B))) =
                  aCarNo).close;
  addCar >>
    Cars(aCarNo).createCar(aCarNo, aCarCat, aBrand);
```



```
  deliverCar >>
    Cars(aCarNo).pickUp,
    Bookings(aBookingNo).makeCurrent(Cars(aCarNo));
BEHAVIOR
  PROCESS Company =
    ( createCompany -> AddCustomerOrBooking );
  PROCESS AddCustomerOrBooking =
    ( addCustomer -> AddCustomerOrBooking
    |
      { Today < aStartDay }
      addBooking -> AddCustomerOrBooking
    |
      { Today < aDay AND
        FORALL (B IN Bookings)
          (IsOpen(Booking(B)) IMPLIES
             aDay <= StartDay(Booking(B)) AND
          IsCurrent(Booking(B)) IMPLIES
             aDay <= EndDay(Booking(B))) }
      incTime -> ReturnCars );
  PROCESS ReturnCars =
    ( { aCarNo IN ReturnCarsMenu }
      returnCar -> ReturnCars
    |
      { -- expired bookings
        FORALL (B IN Bookings)
          ( EndDay(Booking(B)) < Today IMPLIES
              IsClosed(Booking(B))) }
      closeReturnCars -> DeliverCars );
  PROCESS DeliverCars =
    ( addCar -> DeliverCars
    |
      { aBookingNo IN
          ( SELECT BookingNo(Booking(B))
            FROM   B IN Bookings
            WHERE  StartDay(Booking(B)) = Today )
        AND Available(Cars(aCarNo)) }
      deliverCar -> DeliverCars
    |
      { FORALL (B IN Bookings)
          ( StartDay(Booking(B)) = Today IMPLIES
              IsCurrent(Booking(B)) ) }
      closeDeliverCars
          -> AddCustomerOrBooking );
END TEMPLATE
```

4.B Basic Graph Transformation Definitions

Definition 4.B.1 (S-Set, S-Mapping, S-Cartesian Products)
For any set S an *S-set* $\mathcal{A}_S$ is given by a family $\mathcal{A}_S = (A_s|s \in S)$ of sets. An S-set $\mathcal{A}_S$ is an *S-subset* of another S-set $\mathcal{B}_S$, if $A_s \subseteq B_s$ for all $s \in S$. An *S-mapping* $f : \mathcal{A}_S \to \mathcal{B}_S$ between two S-sets $\mathcal{A}_S$ and $\mathcal{B}_S$ is given by a family $f = (f_s : A_s \to B_s|s \in S)$ of mappings. Furthermore we define the *S-cartesian product* of sets and mappings by

- $A_w = A_{s_1} \times \cdots \times A_{s_n}$
- $f_w = f_{s_1} \times \cdots \times f_{s_n}$,

where $w = s_1 ... s_n \in S^*$. For $w = \lambda$ the product over an empty set of indices is given by the terminal object, which is the set containing only one element and the constant mapping. An S-relation $\mathcal{R}_s$ on the S-set $\mathcal{A}_s$ is an S-subset of the S-cartesian product A_w with $w = s...s$.

Definition 4.B.2 ((Sub-) Signature, (Sub-) Algebra)
A *signature* $SIG = (S, OP)$ consists of a set of sorts S and an $S^* \times S$ - indexed family $OP = (OP_{w,s})_{w \in S^*, s \in S}$ of operations. We use the notation $op : w \to s$ for $op \in OP_{w,s}$. A signature $SIG' = (S', OP')$ is a *sub-signature* of SIG, if $S' \subseteq S$ and $OP'_{w,s} \subseteq OP_{w,s}$ for all $w \in S'^*, s \in S'$.

A *partial SIG-algebra* A consists of an S-set $\mathcal{A}_S$ of carrier sets and a family of partial mappings A_{OP} such that $op^A : A_w \multimap A_s \in A_{OP}$ if $op : w \to s$. The domain of op^A is denoted by $dom(op^A)$. If all operations are defined for all elements in the domain, A is a *total SIG-algebra.*

A partial SIG-algebra B is a *sub-algebra* of another partial algebra A w.r.t. the same signature, written $B \subseteq A$ if $\mathcal{B}_S \subseteq \mathcal{A}_S$ and op^B is the domain and codomain restriction of op^A, i.e., $graph(op^B) = graph(op^A) \cap (B_w \times B_s)$. [2]

Definition 4.B.3 ((Partial) Morphism)
A *total morphism* $f : A \to B$ between two partial SIG-algebras A and B is an S-mapping $f : \mathcal{A}_S \to \mathcal{B}_S$ such that op^A defined for $x \in A_w$ implies op^B defined for $f_w(x)$ and $op^B(f_w(x)) = f_w(op^A(x))$.

A *partial morphism* $f : A \multimap B$ between partial SIG-algebras A and B is a total morphism $f\# : dom(f) \to B$ from a sub-algebra $dom(f) \subseteq A$ to B. [3]

[2] Note that this notion of a sub-algebra is a relative sub-algebra in the sense of [4].
[3] Note that this notion of a partial morphism is a quo-morphism in the sense of [4].

Definition 4.B.4 (Rule, Redex, Derivation)
A *rule* r is a partial injective morphism $r : L \multimap R$ from the rule's left-hand side L to its right-hand side R. A *redex* for r in an algebra G is a total morphism $m : L \to G$, such that the following construction of the derived algebra H has a well-defined result, i.e., the operations op^H are well-defined. Let the carrier sets of H be given by $H = (G - m(L)) + R_{/\equiv}$, where $\equiv$ is the image of the congruence induced by m on L under r. For $op : w \to s$ define $op^H(x) = op^R(x)$, if $x \in R_w$, and $op^H(x) = op^G(f_w(x))$, if $x \in (G + r(L))_w$ with

$$f_s ::= f_s(x) = \begin{cases} x & \text{, if } \quad x \in G \\ m(r^{-1}(x)) & \text{, if } \quad x \in r(L) \end{cases}$$

for all $s \in S$. For all other arguments op^H is left undefined.

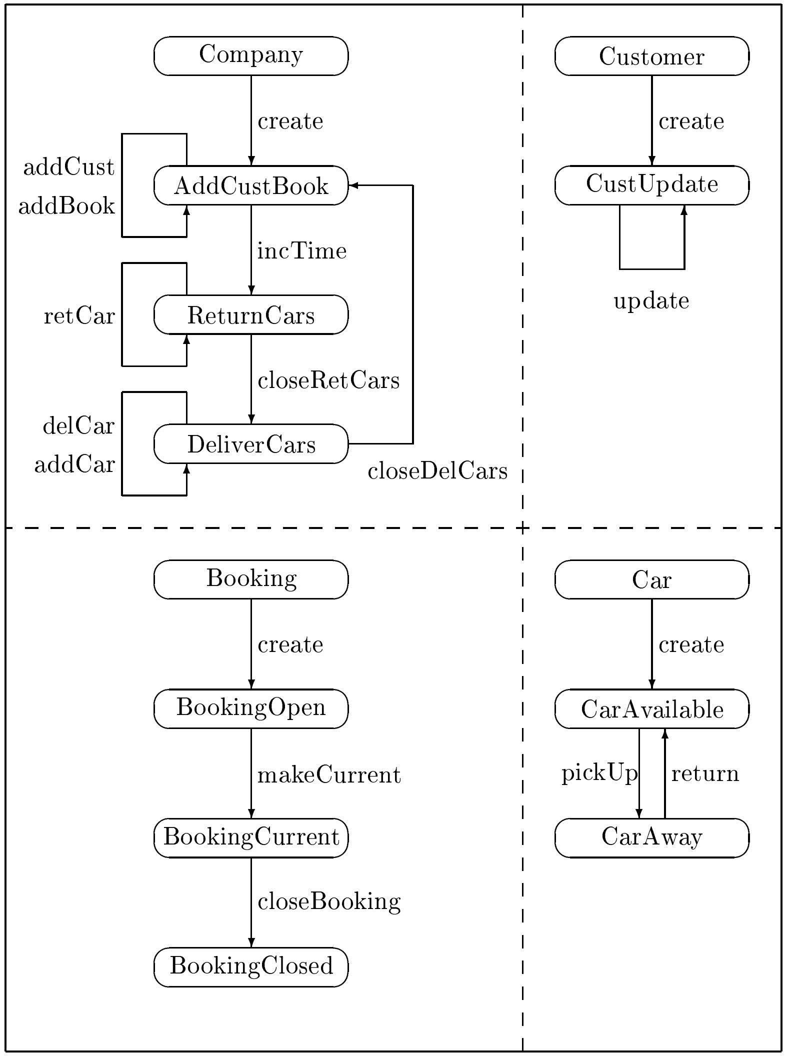

Figure 4.13: Description of TROLL light behavior patterns