A Web-based Animator for Validating Object Specifications

Mark Richters and Martin Gogolla
Universität Bremen
FB 3 Mathematik und Informatik
Arbeitsgruppe Datenbanksysteme
Postfach 330 440, D-28334 Bremen
e-mail: {mr|gogolla}@informatik.uni-bremen.de

Abstract

One of the central tasks in developing information systems is the specification of desired system properties. We use the object specification language TROLL light to formalize the conceptual model of a system. A TROLL light specification describes structural as well as dynamic characteristics of objects representing real-world entities. For validating specifications, we have developed an animation tool allowing us to reflect structural properties and dynamic behavior. The main concepts of the animator are illustrated by a working example. We present a simple specification of a car rental company and describe some of the steps performed during a typical animation session. The animation allows to check whether desired properties are fulfilled by the given object descriptions. Finally, some design and implementation issues regarding the utilized persistent programming environment and the Web-based user interface are discussed.

1. Introduction

The information system development process can be split up into two dominant phases: the specification and the implementation phase. The goal of the specification (or requirements engineering) phase is to achieve a first formalized description of the system to be developed. The resulting formal specification should abstract from implementation details, and therefore it is usually called a conceptual schema. Based on this schema and by considering further non-functional requirements, a working system is developed in the implementation (or design engineering) phase [15].

Here, we concentrate on the specification phase which involves at least the following important tasks: (1) Find user demands on the system in mind (requirements detection), (2) describe a conceptual model of the system in mind (modeling), (3) test whether the conceptual model satisfies formally described quality criteria (analysis), and (4) test whether the conceptual model meets the informal user requirements (validation).

For the first formalized description of real-world entities we use the object specification language TROLL *light* [7, 10, 12], a dialect of OBLOG [21] and TROLL [13]. Other similar approaches are TELOS [14], MONDEL [3], CMSL [22], PI [9], ALBERT [8] or RML [11]. The language TROLL *light* is especially well-suited for information system design because it embodies ideas from structural and dynamic modeling paradigms, namely semantic data models, and data type specification for structural, and process theory for dynamic aspects.

Thus, emphasis is put on the specification phase in the software life cycle process, and in order to support modeling static and dynamic aspects of information systems in an integrated way, we adopt the object-oriented approach to specifying information system. Therefore, TROLL *light* serves as the basis for a corresponding specification environment including tools like an animator allowing the prototypical execution of specifications for validation purposes.

Validation means that specifications of formal conceptual schemas should be checked against informal system requirements given by users in order to meet the validation task from above. This is known as the informal correctness problem. One of the most prominent ways to assure informal correctness of conceptual schemas consists in rapid prototyping which means to construct an experimental version of the system on a quick and low-cost basis. The prototype will often illustrate only the very important aspects of a required behavior, namely its functionality, thereby neglecting other aspects like performance or security. Nevertheless, by observing the behavior of a prototype, clients of a system can better judge on the usefulness of a conceptual schema than by reading specification texts only. Typical tools supporting prototyping may be screen painters, report generators, program generators, or animation systems. The

TROLL *light* animator is designed to simulate the behavior of the specified information system as an object community. By this the informal view of the real-world fragment to be modeled is checked against the current specification.

The structure of the paper is as follows. Section 2 introduces the example specification of a car rental company. In Section 3 this example is used to demonstrate a typical animation session. Some issues regarding the animator's design and implementation are described in Section 4. Section 5 focuses on the implementation of the user interface, and Section 6 gives some final remarks.

2. Object Specification with TROLL *light*

In this section, we introduce a specification that serves as an example for the animation session described in section 3. The specification is also used to demonstrate basic concepts of TROLL *light*. Instead of going into details we present an informal description of TROLL *light* features where needed.

The domain of our example is a car rental company. For demonstration purposes, our model is a rather simple one. The main entities are customers, cars and bookings. Figure 1 shows a diagram in ER notation [6] with the corresponding entity types drawn as rectangles.

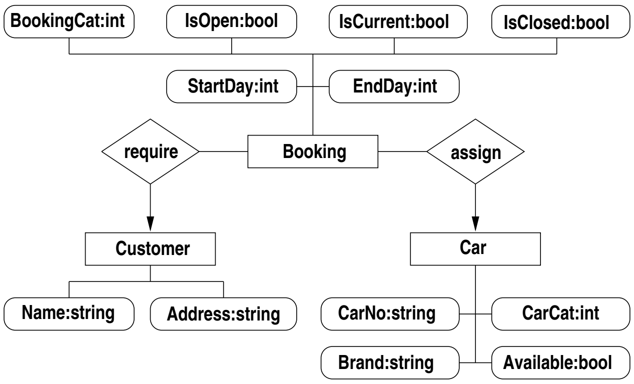


Figure 1. ER diagram for the car rental company.

A customer has attributes for the name and address. Before a car can be obtained by a customer, a booking entry is required. A booking is used to make a reservation for a certain car category and period of time. For reasons of simplicity, we represent date values by integers. If a car is available, it can be assigned to the booking, changing the booking's status from "IsOpen" to "IsCurrent". Finally, a booking is getting closed when the car is returned. An entity of type Car is uniquely identified by the attribute "CarNo" and carries information about category and brand. The attribute "Available" indicates, whether a car is available for assignment to a booking. As an additional constraint for the selection of cars, we require that a car provides at least the category desired by a booking.

The ER diagram properly captures the structural part of the model. But we would also like to formally describe the behavioral properties of entities as well as additional constraints. TROLL *light* allows us to specify the car rental company as a set of object descriptions, called templates which have the following general structure.

```
TEMPLATE name of the template
  DATA TYPES   data types used in current template
  TEMPLATES    other templates used in current template
  SUBOBJECTS   slots for subobjects
  ATTRIBUTES   slots for attributes
  EVENTS       event generators
  CONSTRAINTS  restricting conditions on object states
  DERIVATION   rules for derived attributes
  VALUATION    effect of event occurrences on attributes
  INTERACTION  synchronization of events in different objects
  BEHAVIOR     description of object behavior by a CSP-like process
END TEMPLATE
```

Speaking in rough terms, the `DATA TYPES` and `TEMPLATES` sections are the interfaces to other templates, the `SUBOBJECTS`, `ATTRIBUTES`, and `EVENTS` sections constitute the template signature, and in the remaining sections axioms concerning static (`CONSTRAINTS` and `DERIVATION`) and dynamic (`VALUATION`, `INTERACTION`, and `BEHAVIOR`) properties are specified.

The mapping of entity types as shown in Fig. 1 to templates in TROLL *light* is straightforward (refer to the appendix for the full specification). Each entity type is modeled by a template with corresponding attributes. The functional relationship types "require" and "assign" are mapped to attributes in the booking template each referencing a customer object and car object, respectively. For keeping instances of customers, cars, and bookings we need an additional object, the company. A company object serves as the root of an object hierarchy and has set-valued subobjects for customers, cars, and bookings. This root object can be considered as a kind of schema object. Most of the events for manipulating objects are specified in context of the company object.

2.1. Dynamic properties

Beside the structural part, each template contains rules that determine the behavior of an object in case of certain events. A simple case, for example, can be found in the behavior section of the customer template (cf. Appendix A). After creation of a customer object, its address can be changed repeatedly, provided that the new address differs from the old one. With similar behavior specifications we want to guarantee that bookings change state from "open", via "current" to "closed" in correct order.

The main processes for handling cars, bookings, and customers are specified in the company template. Process states and transitions are depicted in Fig. 2. States are labeled with names and numbers. Named directed edges indicate possible events for a transition. By creating a company object we reach state 1 where it is possible to add customers and bookings. Incrementing the time to the next weekday changes state to *ReturnCars*. Leaving this state is possible only if all currently booked cars with an end day less than the new current date are returned. In the specification, this constraint is enforced by a precondition which is attached to the event `closeReturnCars` in the process description of *ReturnCars*. Finally, the purpose of state *DeliverCars* is to satisfy all open bookings with a start day matching the current date by delivering appropriate cars. After closing delivery the whole sequence can be repeated.

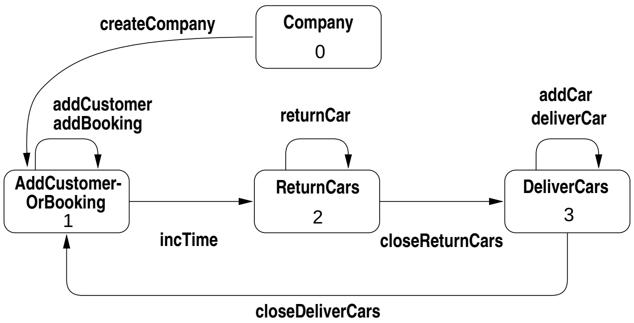


Figure 2. Automaton for the car rental company.

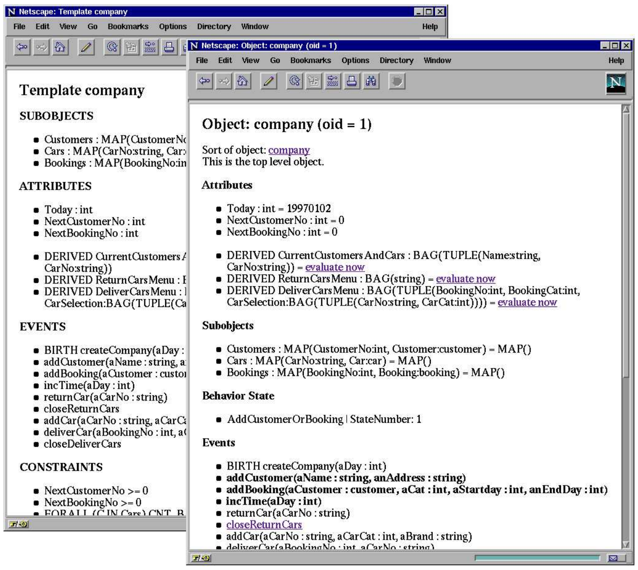
The specification is designed to capture the following goals:

- a booking will be correctly processed, i.e., it will become a "current" booking when reaching its start day, and it will get closed when passing its end day.
- none of the bookings are accidentally skipped. Incrementing time never skips beyond a start day of an open booking.
- a car has at least the desired category if assigned to a booking.
- users should be supported in recognizing feasible events and processes.

3. An Animation Session

In this section, we show how the animation tool can be used for validating specifications. We will animate the previously described specification of a car rental company.

The animator is controlled by a Web browser and presents its results as dynamically generated hypertext documents. An animation session begins with a static document providing two links. The first link points to a list of available templates. These templates can be considered the schema information of all objects manipulated during a session. By following the second link, a user requests a document showing the state of the root object. The root object is the entry point for all further operations. Part of the state of the initial company object and its template is shown in Figure 3.



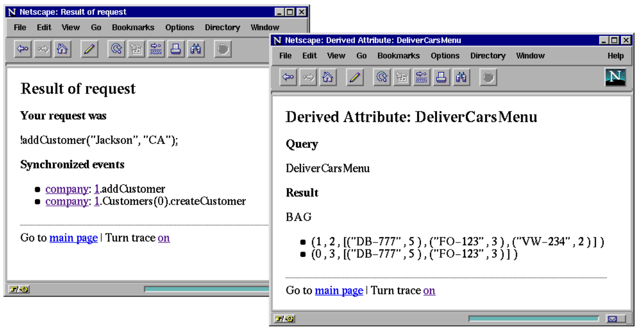
(a) (b)

Figure 3. (a) Company template. (b) Company object.

For every object, there is a link to its template. Thus, a user can easily switch between a concrete object and its definition by simply following links. A document for object instances visualizes attribute values, subobjects, and the current behavior state of an object. The current state of the company object, as shown in Fig. 3b, has been reached after executing the initial birth event `createCompany` by the system. At this point, the attribute values have been calculated by the valuation rules for the birth event. You can also see that derived attributes are only calculated on request by selecting the corresponding link. The subobject maps for customers, cars, and bookings are still empty.

As specified in the behavior section of the company template the state *AddCustomerOrBooking* allows us to add customers or bookings, and to increment the time (see also Fig. 2). These events are also being highlighted in the list of possible events. However, the highlighting is solely based upon determining leaving edges from the current state in the state diagram without considering further constraints like preconditions. Hence, this feature does not prevent one from choosing events that are later rejected. It only excludes events that are definitely not possible.

Simple events without parameters can be triggered directly by selecting the hypertext link. Other events require an explicit event specification. Part of every document showing an object is a form field at the bottom (refer to Fig. 5a). We can enter either query terms or event specifications and submit them to the animator for evaluation in context of the currently selected object. For example, after submitting the event `addCustomer("Jackson", "CA")` the input is being parsed and evaluated. The result of the event request is shown in Fig. 4a. Interaction rules in templates determine which events do occur simultaneously. The animator displays a list of synchronized events together with the affected objects.



(a) (b)

Figure 4. (a) Result of an event. (b) Value of a derived attribute

In order to populate the company, we perform the following additional events:

```
!addCustomer("Warren", "NY");
!addBooking(Customers(0), 3, 19970203, 19970205);
!addBooking(Customers(1), 2, 19970203, 19970206);
!incTime(19970203);

-- no cars to return
!closeReturnCars;
```

The parameters of `addBooking` events denote a customer object, a desired category, and start and end day of the booking. In the example, a booking is created for each of the two customers, asking for cars of category 3 and 2, respectively. We increment the time to the start day of both bookings. Since there are currently no cars delivered, we immediately skip the state *ReturnCars* by triggering an event `closeReturnCars` and reach state *DeliverCars*.

If you take a look at the company template, you will notice that we have designed the derived attributes to aid in selecting feasible events. The attribute `DeliverCarsMenu`, e.g., uses the following nested select-term to determine for each booking the set of cars that would satisfy the booking.

```
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))
```

In the current state of the animation session, evaluation of the attribute `DeliverCarsMenu` yields the bag $\{\!\{ (1, 2, \{\!\{\ \}\!\}), (0, 3, \{\!\{\ \}\!\}) \}\!\}$ as a result. Each tuple contains a booking number, a desired category, and a bag of suitable cars, which is currently empty since we have not added any cars yet. Now, consider the following events.

```
!addCar("FO-123", 3, "Ford");
!addCar("VW-234", 2, "VW");
!addCar("DB-777", 5, "Mercedes");
```

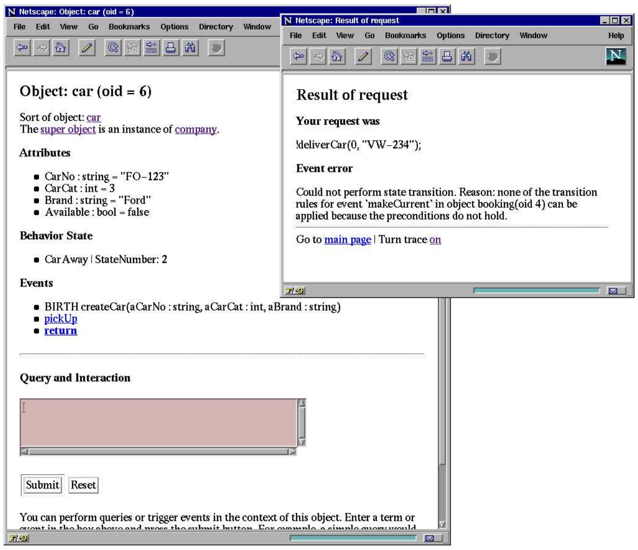
After executing the events, the attribute `DeliverCarsMenu` evaluates to

$$\{\!\{ \quad (1, 2, \{\!\{ (\text{"DB-777"}, 5), (\text{"FO-123"}, 3), (\text{"VW-234"}, 2) \}\!\}), \\ (0, 3, \{\!\{ (\text{"DB-777"}, 5), (\text{"FO-123"}, 3) \}\!\}) \quad \}\!\} \ .$$

The animator displays this value as shown in Fig. 4b. Note that the second tuple for booking number 0 does not contain the VW because its category is not sufficient. Thus, an attempt to deliver that car results in an error message shown in Fig. 5b. Instead, we proceed with the event `deliverCar(0, "FO-123")`. The state of the car object with car number “FO-123” after this event is shown in Fig. 5a. You can see that its attribute “Available” is false now. The behavior specification restricts the possible set of next events to a `return` event. Since car objects only exist as subobjects of company objects, there is also a link to its super object at the top of the document.

4. Animator design issues

The previous section focused on application aspects of the animation tool. In the following, we describe some as-



(a) (b)

Figure 5. (a) Car object. (b) Error message after event.

pects of the underlying system and its implementation. Basically, the animation system consists of the following components:

- persistent representations of object specifications, object states and complex values,
- evaluation of expressions as part of a calculus of complex values,
- execution of state transitions (object creation, change of object attributes and behavior states, object destruction), and
- a user interface for visualization of object states and accepting user requests for events and ad-hoc-queries.

Our animation tool requires that specifications and objects can be accessed and manipulated in several successive sessions. Users should be able to suspend and restart a session at any time. Hence, session data must be made persistent. The situation is complicated by the fact that the complex structure of templates and objects cannot easily be captured by a conventional data model such as, e.g., the relational model.

We chose the system Tycoon [17] and its persistent programming language TL as the basis for an implementation of the animation system. Tycoon is a programming environment that considerably improves the construction, maintenance and operation of "persistent application systems" (PAS) [16]. A PAS is a software system that gives its users a flexible, problem oriented and safe access to large sets of long-lived objects of application-specific types [2]. In our application domain, TROLL *light* object specifications and object instances have to be considered "long-lived objects" of arbitrary size and number.

The scalable Tycoon architecture integrates persistent data, programs and threads. It strictly separates tasks of storage, manipulation, modeling and representation into well-defined system layers [18]. The underlying programming language TL has a rich type system and allows for generic programming and external communication. It provides orthogonal persistence, type completeness, and higher order functions. The language TL is strictly typed and is neutral with respect to the data model.

Our animation tool is completely written in TL. The implementation effort was significantly reduced by applying Tycoon's advanced concepts of expressive orthogonal language constructs and persistence. Persistence as an orthogonal property of data has early been recognized as a desirable feature of database programming languages [1]. By implementing the animation tool in TL, we do not have to distinguish between transient data, like intermediate results from calculations, and persistent data, like template descriptions. Persistent objects are identified by reachability from a top level binding. Thus, an animation database containing all the necessary information describing a session can be stored without further effort. Unreferenced data will be automatically garbage collected. Data type orthogonality ensures that any TL-type can be made persistent. As a consequence, the full set of basic TL-types and type constructors can be used for data modeling.

5. The Web-based User Interface

The primary purpose of the user interface is to visualize the description and current state of objects. The user interface provides a uniform way to trigger events and to specify queries for further exploration. Some more issues to consider are:

- Objects are structured into trees by subobject relationships. A user must be able to follow these relationships and navigate the hierarchy. Moreover, attributes containing object references extend the tree structure into a lattice.
- Values of attributes can be arbitrarily complex.
- User-defined data sorts describing, e.g., audio or image data may require special treatment for presentation.

We decided to use a conventional Web browser as the user interface for several reasons. First, a hypertext system provides basic means suitable for navigating an object hierarchy. Each HTML-document visualizes information about a single object. By employing a hypertext system, we can model relationships among objects in a natural way by providing hypertext links. Thus, the user can explore the object hierarchy by simply following the corresponding hypertext links.

Documents are always generated dynamically on request, since the content of a document depends on the current state of the related object. A simple request may result from the selection of a hypertext link in a document, for example, to explore a different object that is referenced by an attribute in the currently selected object.

The request mechanism for the dynamic generation of documents is realized by using the Common Gateway Interface (CGI) on the Web server side. Requests are embedded in an HTML-anchor by augmenting the URL with necessary information, i.e. the kind of request and any arguments like oid's of related objects or names of specifications. To provide an entry point to an animation session, it is sufficient to keep a single static document, a file, with two links, the first one pointing to the set of available specifications, the second one leading to the root object of a hierarchy of instances.

6. Conclusion

This paper presented a sketch of our object description language TROLL *light* and an animator for the language as an important ingredient of an accompanying specification environment. The animation system supports the development of reliable information systems in validating the informal view of the real-world fragment to be modeled against the current conceptual schema and in checking whether desired properties are fulfilled by the specified object descriptions.

However, the implemented tools are far from being perfect. For example, arbitrary user-defined data types, graphical visualization of subobject relationships or enhanced visualization of object behavior are not supported well. Furthermore, the current system lacks an undo feature to reverse the effects of events. Persistent checkpoints could be used to mark each consistent state after a state modification, thus allowing an unlimited number of undo steps.

On a methodological level, we plan to integrate TROLL *light* with well-known techniques for object-oriented analysis and design. Methods like Booch [4], OMT [20] or, more recently, UML [19] are very popular among software engineers. However, a solid formal foundation of the models introduced by these methods is still missing. By combining these methods with a specification language with well-defined semantics, we avoid ambiguities and enable formal reasoning about conceptual models.

If we consider the implementation of database applications, a formal TROLL *light* specification is a good starting point for an automatic translation to schema definition languages like ODL, the object definition language proposed by the ODMG [5].

Up to now, only the first steps towards implementation of specifications on the basis of object-oriented programming languages and databases have been taken, and much more work should be done here.

References

[1] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[2] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[3] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An Object-Oriented Specification Language. Publication 748, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1990.

[4] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

[5] R. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan-Kaufmann, 1996.

[6] P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

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

[8] E. Dubois, P. Du Bois, and M. Petit. O-O Requirements Analysis: an Agent Perspective. In O. Nierstrasz, editor, *ECOOP'93 — Object-Oriented Programming*, pages 458–481. Springer, Berlin, LNCS 707, 1993.

[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 (ADT'91)*, pages 254–270. Springer, Berlin, LNCS 655, 1993.

[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)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

[11] S. Greenspan, J. Mylopoulos, and A. Borgida. On Formal Requirements Modeling Languages: RML Revisited. In B. Fadini, editor, *Proc. 16th Int. Conf. on Software Engineering (ICSE'94)*, pages 135–148. IEEE Computer Society Press, 1994.

[12] R. Herzig, S. Conrad, and M. Gogolla. Compositional Description of Object Communities with TROLL light. In C. Chrisment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94): Information Systems Design and Hypermedia*, pages 183–194. Cépaduès-Éditions, Toulouse, 1994.

[13] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91–04, Technische Universität Braunschweig, 1991.
[14] M. Koubarakis, J. Mylopoulos, M. Stanley, and M. Jarke. TELOS: A Knowledge Representation Language for Requirements Modelling. Technical Report CSRI-222, University of Toronto, 1988.
[15] P. Loucopoulos. Conceptual Modeling. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, pages 1–26. John Wiley & Sons, New York, 1992.
[16] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993.
[17] F. Matthes. The Tycoon Project, `http://www.sts.tu-harburg.de/projects/Tycoon/entry.html`, 1997.
[18] F. Matthes, G. Schröder, and J. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1995.
[19] Rational Software Corporation, Unified Modeling Language (UML) version 1.0, `http://www.rational.com`, 1997.
[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (NJ), 1991.
[21] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P. Stocker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB)*, pages 107–116. Morgan-Kaufmann, Los Altos (CA), 1987.
[22] R. Wieringa. Equational Specification of Dynamic Objects. In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG2.6 Working Conference, Windermere (UK) 1990*, pages 415–438. North-Holland, Amsterdam, 1991.

A. 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
```

B. 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
```

C. 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
```

D. 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
```