Chapter 5

The Formal Semantics of Form Charts

In this section we give a formal semantics for the final artifacts of Form-Oriented Analysis, form charts and the data model. The focus is herein naturally on the semantics of form charts, later the semantics of the data model and the data dictionary are introduced in UML.

5.1 Form Charts in UML

Form charts have been defined as bipartite state transition diagrams. The formal semantics in this chapter is motivated by the goal to define state transition diagrams in such a way that they easily support the dialogue constraint language, DCL. The main problem is here to give semantics for path expressions. This is achieved in this chapter by two contributions. Path expressions are generalized in Section 5.5 to arbitrary UML class diagrams. The semantics for form charts are defined in such a way that the path expressions in DCL map exactly with our general definition of path expressions. In other words, DCL with its path expressions can be seen as a subset of our OCL extension.

In this section we give an operational semantics for general state transition diagrams — not only for bipartite STDs as they are needed for form charts — in order to clarify the general character of the introduced semantics. Our operational semantics are based solely on the semantics of the most important UML diagram type, namely core class diagrams. Within the UML specification operational semantics are known under different terms, they are called semantics, detailed semantics or dynamic semantics. The term behavioral semantics can be found, too, but refers typically only to semantics of behavioral features. We use the term operational semantics. We will give operational semanics for state transition diagrams by introducing so called state history diagrams, or SHDs for short, our own version of state transition diagrams. The main idea behind SHDs will be, that they are defined as a restriction of class diagrams. In

the concluding discussion of this section we will compare our approach with the definition of state machines in UML. As we will see, our semantics are based only on the fundamental semantic concept of core class diagrams, which serve as the semantic foundation in the modeling universe in which UML is located. As a consequence SHDs allow the specification of temporal constraints on finite state automata without any need for further temporal formalisms, solely through the combination of the already defined concepts. Form charts, the key diagram of Form-Oriented Analysis will then be a simple application of SHDs, though we will support it with an own semantic framework. We then also give precise semantics to the dialogue constraints introduced in DCL.

One may allow us to modestly point out that the approach chosen here achieves a sound formal basis for all the special constructs introduced in Form-Oriented Analysis in a rather short and lightweight way. This is achieved through maximal reuse, mainly because we were able to fully reuse the semantics of class diagrams for our new artifacts.

In the subsections 5.1.1 to 5.1.3 we explain state history diagrams. In subSection 5.1.4 we model form charts as bipartite SHDs. In subSection 5.1.5 we introduce the dialogue constraint language, DCL.

5.1.1 State History Diagrams

If we model a complex system as a finite state machine, this finite state machine is typically only a part of the model. The behavior may depend further on a classical data model. But in many cases the system behavior may especially depend on the history of state transitions. State history diagrams give a convenient general modeling tool for such systems. Submit/response style interfaces will be a special application case.

State history diagrams, SHDs for short, are a semantic unification of class diagrams and state transition diagrams. If a system is modeled with an SHD, the history or trace of the finite state machine is a part of the actual system state. In other words, for each state visit an instance of some class is kept in the actual system state, so that these instances together form a log of the state transition process so far. On this object net we can define constraints concerning the history of state transitions. Hence we can define certain temporal constraints without having to introduce a temporal extension into the constraint language.

Form charts are a specialization of SHDs used in Form-Oriented Analysis. Form charts are bipartite SHDs with the Dialogue Constraint Language DCL, which is mapped on OCL.

In state history diagrams the log of the state visits is kept as a linked list of object instances. The links represent the transitions between state visits. SHDs are based on the idea to choose the class diagram for this log in such a way that it is isomorphic to the state transition diagram.

This is possible since state transition diagrams and core class diagrams can be modeled by a similar metamodel. The basic metaclasses are nodes and connectors. In STDs the nodes are states and the connectors are directed transitions. In core class diagrams the nodes are classes and the connectors are

binary associations.

A state history diagram is now a special class diagram which can be read as an STD at the same time. The consequence of this approach is that a single diagram describes the state machine on the one hand and on the other hand serves at the same time as class diagram for the aforementioned history of state transitions. Such a class diagram must however adhere to rigorous restrictions that will be given in due course.

We adopt the following rules for speaking about SHDs. The diagram can be addressed as SHD or as STD or as class diagram, emphasizing the respective aspect. The nodes are called state classes, the connectors are called transitions and they are associations, their instances are called state changes. A run of the state machine represented by the STD is called a process. The visit of a state during a process over the STD is identified with an instance of the state class and is called a visit. Hence a process is the object net over the SHD. This object net is a path from the start visit of the process to the current visit. This path is seen as being directed from the start visit. Each prefix of the path is a part of the whole current path. This matches the semantics of the aggregation. Hence all transitions are aggregations. The aggregation diamond points to the later state. In SHD however, the transitions are not drawn with diamonds but with single arrows. The associated state classes are called source and target. We will use the SHD for the modeling of form charts. If a system is used by several clients, each client lives in an own object space. Therefore the singleton property is local to the client' object space. If one would like to model all clients accessing a system in a single object space, this could be done by using several SHD's in a single object space. In that case one has to use a slightly modified framework in which the `StartState` is not a singleton, but there is one `StartState` instance for each run of the finite automaton.

5.1.2 Modeling SHDs as Class Diagrams in UML

In the previous subsection we have defined SHDs as a restriction of a class diagram, not as a new diagram type. In this subsection we give a formal treatment of this approach in the context of UML. We achieve the definition of SHD's as restricted class diagrams in the following way: SHDs are class diagrams in which all elements are derived from a special semantic modeling framework, the shdframework. Note that we use model level inheritance here. In the UML context a specification alternative would be a package with stereotypes in which the SHD is defined by metamodel instantiation, instead of model inheritance. But first we adhere strictly to the economy principle, and argue that modeling is more lightweight than metamodeling, therefore we use modeling wherever possible. Secondly our approach allows for a quite elegant formulation of the central SHD semantics, namely that the object nets are paths. In Section 5.2 we will discuss the benefits of our modeling approach.

However, we want to use the stereotype notation for pure notational convenience. For that purpose we introduce an auxiliary stereotype package in which for each public element of our framework a stereotype of identical name is in-

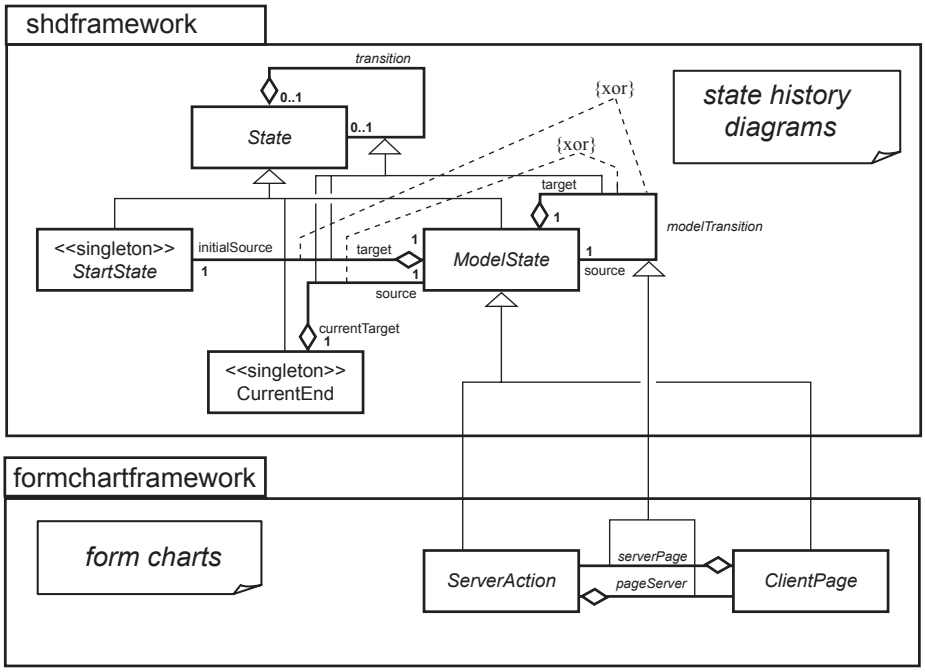

Figure 5.1: Frameworks for state history diagrams and form charts

troduced which has the metalevel constraint that its instances must inherit the framework class of same name.

The shdframework is depicted in Figure 5.1 together with the form chart framework, which extends the shdframework.

In the shdframework we define a hierarchy for classes as well as for associations as shown in Fig. 5.1. Indeed we make intensive use of the concept of association inheritance. The basic class is `State` and it has an aggregation to itself called `transition`. The ends of `transition` have roles `source` and `target`.

Before we explain, how the framework introduces the desired semantics to SHDs, we explain, how it should be used in creating SHDs. As mentioned earlier, all elements of an SHD must be derived from the following public elements in the shdframework. All state classes in the SHD must be derived from `ModelState` and all transitions must be derived from `modelTransition`. The marker for the start of the SHD is derived from `StartState`, and its transition to the first state is derived from the unnamed transition to `ModelState`. Only these four elements of SHDs are public. All elements except the class CurrentEnd are abstract, only the classes derived by the Modeler can be instantiated. The singleton stereotype of `StartState` requires that the whole class hierarchy derived from `StartState` has only one instance.

In a concrete SHD of course the generalization dependencies of SHD ele-

ments to the framework are not depicted. Instead we make use of the auxiliary stereotypes mentioned earlier. Therefore we are entitled to change the graphical appearance of stereotyped classes in the SHD as we will do especially in form charts.

5.1.3 Constraints on the Object Net

We now explain, how the shdframework in Figure 5.1 formalizes our constraints on the process as the object net over an SHD. We show that the shdframework enforces that the object net is a path.

For this purpose it is necessary that every visit but the start visit must have exactly one predecessor, and every visit but the last visit must have exactly one successor. The formalization of this demand poses two separate problems: on the one hand the formalization of the constraint that each inner node of the path must have exactly one predecessor and successor, and on the other hand the exemption of the start and the end visit in exactly the correct manner. The important problem of guaranteeing, that the object net is cycle-free is already achieved by the use of aggregations since aggregations are defined to be cycle-free on the object net level.

We first address the problem of exempting the terminal nodes.

There are two flavors of formalization: first one could exempt the start visit from the general rule. The second way is to use a technique similar to the sentinel technique in algorithms: an artificial predecessor to the start node is introduced. This artificial visit is of a `StartState` class, which cannot be revisited. We choose this second method, since it has the advantage, that it delegates the semantic details to auxiliary classes. Both classes `StartState` and `ModelState` are derived from `State`. In the same way, the current visit has always an artificial successor from the class `CurrentEnd`. All states created by the modeler in the SHD shall be derived from `ModelState`. Each time a new state A is visited, a new instance of A must be created. This new visit gets the old current state as a predecessor and the current end as a successor.

We now discuss, how the cardinalities expressed in the class diagram in Figure 5.1 give the desired semantics and address the second problem, namely the formalization of the constraint that each inner node of the path must have exactly one predecessor and successor.

We sum up these deliberations in the following

Theorem 1 *The object net over a state history diagram is a directed path.*

Proof. If two classes A and B are connected with an association r, then in OCL for an object $a : A$ the group of associated objects $a.r$ is an OCL collection, i.e. a multiset or bag. The cardinality on the association end specifies the number of elements of the collection, i.e. multiple occurences counted, not ignored. Objects associated by a derived asociation $q < r$ are always elements of the collection concerning r, hence $a.q \subset a.r$. This relation is the key for the following cardinality discussion; if an association end of an abstract association

has a cardinality 1, then exactly one derived association must have exactly one link.

As we have said about SHDs, each transition in the SHD must be derived from `ModelTransition` in the diagram, and each model state must be derived from `ModelState`. This implies that if a model state has several outgoing transitions, they are all derived from `modelTransition`. Hence for an instance of this state only one of these transitions can be instantiated at the same time due to the cardinalities for `modelTransition`. In this way the shdframework introduces an implicit Xor constraint on all outgoing transitions, and vice versa on all ingoing transitions of a model state. Hence each instance of a model state derived from `modelState` must have exactly one predecessor and one successor. The Xor constraints that are explicitly modeled in the `shdframework` take care, that an instance of a model state can have one of the terminal nodes as a predecessor or successor. We will discuss this for the `StartState`. The Xor constraint responsible for the treatment of the start node is the Xor constraint that connects the `modelTransition` with the transition from `StartState` to `ModelState`. This Xor constraint allows that an instance of a model state can have either another model state instance or a `StartState` instance as a predecessor. In the same way an instance of a model state can have either another model state instance or a `CurrentEnd` instance as a successor. The instances of `StartState` and `CurrentEnd` have only one transition with cardinality 1, so the instances of these states must have exactly one link attached. Since both classes are singletons, the object net over a form chart contains exactly only one object with a single outgoing link, exactly one object with a single ingoing link, and otherwise only objects with exactly one in- and one outgoing link. Together with the fact, that all associations are aggregations and therefore the object net must not contain cycles, it follows, that the object net must be a single directed path. □

5.1.4 Form Charts as SHDs

Form charts are bipartite state transition diagrams, which we will model now as SHDs. For this purpose we introduce the formchartframework, which introduces specializations of the `ModelState` and `ModelTransition` elements of the shdframework, as shown in Figure 5.1. Form chart model elements must be derived from the formchartframework elements, except the rather technical start elements, which are still derived from the shdframework directly. Elements of form charts are therefore also derived from elements of the shdframework, though indirectly. Hence form charts are SHDs.

The form chart framework enforces in itself that form charts are bipartite and introduces the known names for form chart elements.

We introduce two subclasses to `State`, namely `ServerAction` and `ClientPage`. All states in the form chart have to be derived from these classes. We derive aggregations `pageServer` and `serverPage` between them from the transition aggregation in order to enforce that form charts are bipartite: In the form chart all transitions must be derived from either `pageServer` or `serverPage`. The

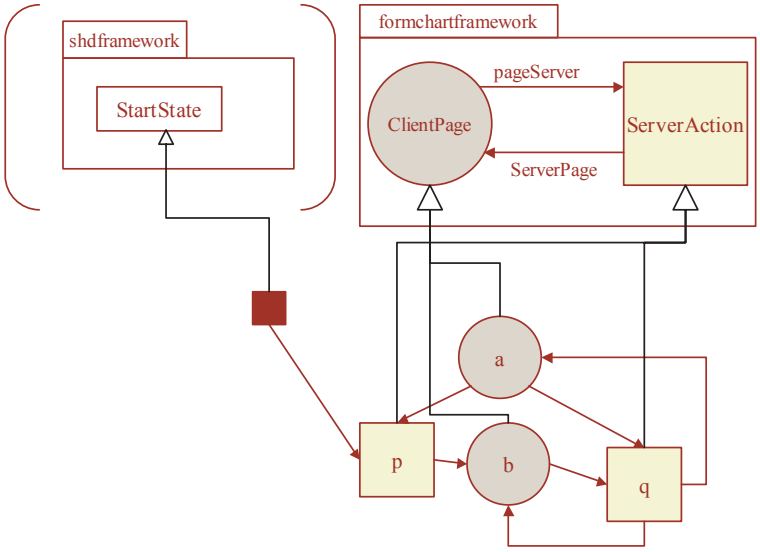

Figure 5.2: A form chart is derived from the semantic framework

usage of the framework for modeling frameworks is shown in Figure 5.2. Only the derivations of the states are shown, the derivations of the transitions are omitted.

We assume again that each form chart framework element is accompanied by a stereotype of same name. Each stereotype introduces its own graphical representation. The << ClientPage >> stereotype is depicted by a bubble, the << ServerAction >> is depicted by a square. In the form chart, bubbles and squares contain only their names.
The << serverPage >> and << pagePerver >> associations are depicted as arrows, even though they are aggregations.

Figure 5.3 shows a form chart and an example object net over this form chart, depicted actually below the form chart. The start state is omitted. The object net is a path alternating between client states and server actions. If a `ModelState` in the form chart has no outgoing transition, this state is a terminal state for the dialogue; the dialogue is completed, once the state is entered.

5.1.5 Dialogue Constraint Language

Form charts have new constraint stereotypes. We call OCL plus the new constraint stereotypes the dialogue constraint language DCL. The important well-formedness rules concerning the bipartite structure of form charts are already specified by the class diagram in Fig. 5.1.

In this section we define the well-formedness rules of the different constraint stereotypes introduced in DCL. The placement of the DCL constraints is depicted in Figure 4.1.

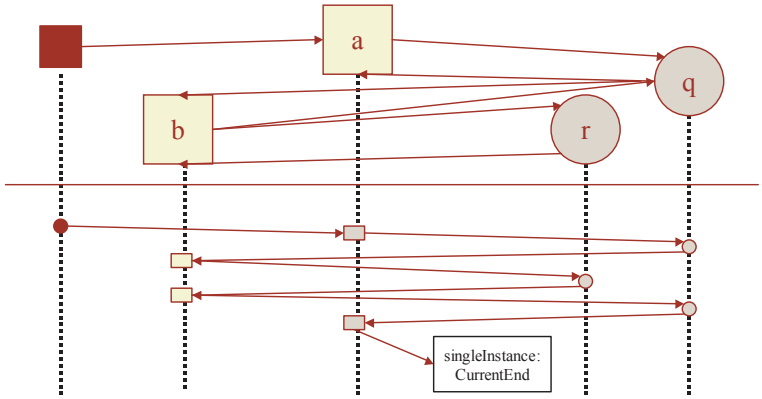

Figure 5.3: The object net over a form chart is a path.

5.1.6 Well-Formedness Rules

In this section we give the formal rules where the new DCL constraints are allowed. These rules are well-formedness rules for form charts. They are OCL metamodel constraints assigned to the newly introduced stereotypes.

Two kinds of DCL constraints are placed on classes, namely client input constraints and server input constraints. The other kinds of constraints are placed at the ends of transitions.

Formally, we define stereotypes for constraints, similar to the stereotypes << invariant >> , << precondition >> and << postcondition >> for OCL. The new stereotypes apply to constraints at transition ends and are called dialogue constraints. They are derived from the << dialogueconstraint >> stereotype. They are all contained in the formchartframework so that they can be used in a form chart. They conclude the allowed elements within the form chart.

Two of the dialogue constraints, namely client output and server output constraints share important semantic properties, as we will see in the section on the semantics of dialogue constraints. This leads to the intuitive approach to make them specializations of a single constraint type, which we will call << stateoutputconstraint >> . This constraint type can now be understood as a natural constraint already on the SHD level. So the two types of output constraint in the form chart can be understood as the natural dichotomy of this single SHD constraint due to the bipartite structure of the form chart.

The dialogue constraint stereotypes have metamodel constraints as presented in the following. They primarily argue over the constrained element, which can be accessed as `constrainedElement` in the metamodel.

```
enablingcondition
constrainedElement→instanceOf(AssociationEnd) and
constrainedElement.association→instanceOf(pageServer) and
constrainedElement.aggregation=none
```

```
clientoutputconstraint
constrainedElement→instanceOf(AssociationEnd) and
constrainedElement.association→instanceOf(pageServer) and
constrainedElement.aggregation=aggregate

serveroutputconstraint
constrainedElement→instanceOf(AssociationEnd) and
constrainedElement.association→instanceOf(serverPage) and
constrainedElement.aggregation=aggregate

flowcondition
constrainedElement→instanceOf(AssociationEnd) and
constrainedElement.association→instanceOf(serverPage) and
constrainedElement.aggregation=none

serverinputconstraint
constrainedElement.stereotype=ServerAction

clientinputconstraint
constrainedElement.stereotype=ClientPage
```

In the metamodel constraints for << serverinputconstraint >> and << clientinputconstraint >> we use the fact that we have for each framework class a corresponding stereotype which must be assigned to each subclass of the framework class.

Informal metamodel constraints are: Only one constraint of the same stereotype is allowed for the same context. The numbers of flow conditions must be unique. They must not be strictly ascending in order to facilitate feature decomposition. For each ServerAction there may be only one flow condition that is not numbered.

5.2 Discussion

We have modeled form charts by frameworks, the shdframework and the formchartframework. The presented semantics is directly motivated by our application domain, however, it is also is a novel operational semantics for state transition diagrams in general, independent from the presented motivation. We outline and discuss modeling alternatives in the following.

5.2.1 SHD's and form charts by strict metamodeling

As we mentioned earlier, a modeling alternative would be a pure metamodel for form charts. The most general method of introducing new diagrams in the

UML is given by the metamodeling technique. We therefore could define the form chart elements by strict metamodeling [109], i.e. purely as a framework of stereotypes for metamodel elements of class diagrams. The intention of this approach would be still to define form charts (or more generally SHDs) as a variant of class diagrams in such a way that it is guaranteed that the object net is a path. The constraints expressed in our framework modeling approach would have to be expressed within the stereotype description. A related topic is the use of the composition notation. One could think of expressing certain cardinalities in the framework implicitly by using composition instead of aggregation. However, composition would be able to express only one direction of the cardiality at best, furthermore there are different opinions about the exact meaning of composition. Therefore it seems more convenient to stick to aggregation, and to use explicit cardinalities, as it was done in the framework.

The diagram in Figure 5.3 resembles a message sequence chart. However, the diagram is completely different. Above the horizontal line are classes, not instances. Below the line are instances, not method invocations.

5.2.2 Modeling form charts with state machines

UML has its own diagram type for state automata called state machines. State machines are based on David Harel's statecharts [46]. However, in UML state machines have no operational semantics defined within UML, indeed no formal operational semantics is part of the specification. Instead, operational semantics is given by reference to a state machine that is described verbally. A semantics of statecharts based on pseudocode is given in [137]. Our definition of state history diagrams has an operational semantics based solely on core class diagrams, the semantic core of UML.

One could model form charts with UML state machines. For this approach one has to drop the SHD semantics and hence loses support for temporal path expressions. Nevertheles we present the alternatives in the following for reasons of completeness. UML state machines models the behavior of objects by modeling the life cycle of a single object as a finite automaton. There are again two alternatives for us to employ them, namely modeling the form chart as a bipartite state machine or transforming the server actions. The latter alternative means that each page server transition will be transformed to as many statecharts transitions, as the targeted server action has outgoing transitions. In that case the flow conditions would become guard conditions directly on the page, the server actions do not appear. In the example of the delete page, the page would have two outgoing transitions with event submit, one of them with the guard condition "password valid" the other with guard condition "password invalid", as shown in Figure 5.4. The guard conditions have to be made exclusive in order to prevent nondeterministic behavior. A variant would be to omit the self transition with guard condition "invalid password". However, there is a subtle semantic difference between both variants. Namely in the first variant, the history of state visits could have repeated adjacent visits of the deletePage state, while in the later alternative there would be only one instance of this

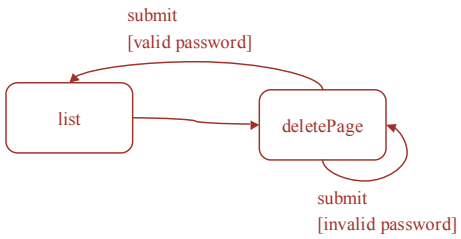

Figure 5.4: The deletion subdialogue as UML state machine

state instead.

The second major alternative would be to model the form chart as a bipartite UML state machine. We need no compound states for form charts, but we would have to enforce bipartite structure. Server actions would be required to have completion transitions that are triggered automatically upon completion as it is known from activities in activity diagrams. The flow conditions in form charts can be mapped to guard conditions of the server action. However, the flow conditions in our form chart semantics are exclusive since they are mapped to elsif branches of an OCL condition. For the state machine modeling one has to make the guard conditions exclusive, otherwise the semantics of state machines would prescribe nondeterministic behavior.

Except for flow conditions and enabling conditions, DCL constraints cannot be mapped to guard conditions. This is due to the fact that the other conditions like server output and client output constraints are in principle design by contract annotations. These DCL conditions would have to be mapped to pre- or postconditions of actions assigned with the transitions in the state machine.

The fact that Form Charts use only flat state transition diagrams in contrast to hierarchical statecharts does not restrict the expressibility of form charts due to the fact that form charts are coupled with a semantic data model. Form charts can be easily combined with statechart notation if the statechart notation is interpreted as visualization of parts of the session objects of the semantic data model.

A proposal for representing states by classes can also be found in the State design pattern [6], which describes, how an object can delegate its state change to a state object. The object appears to change its state within a finite set of states. The set of states is finite, since each state is represented by a class. The pattern does not prescribe how to model a finite automaton over the state set, but discusses procedural implementations. First it proposes implementing the transitions in the so called Context class using the pattern. An alternative proposal is a table lookup. Both are nonequivalent to the SHD model using associations to model state transitions.

5.2.3 Petri Nets

Petri Nets are a state transition formalism based on bipartite graphs. Form charts resemble Petri nets due to the fact that server actions resemble Petri net transitions. Petri nets as a finite state machine model are classically conceived as being never in a state corresponding to a transition. The main difference between Petri nets and bipartite state diagrams is therefore that the state of a Petri net is not necessarily a single bubble, but possibly a compound state, depending on the type of Petri net, e.g. a function from bubbles to numbers for place/transition nets.

It is possible to give form charts a Petri net semantics. We have outlined, how to define the semantics of form charts based on non-bipartite state machines. A Petri net semantics can be given in a similar way by defining only client pages as places and introducing a Petri net transition for every path of length two to another client page. Such Petri net semantics however involves only trivial transitions i.e. transitions with one ingoing and one outgoing edge.

5.3 Data Modeling in Form-Oriented Analysis

The whole analysis data model is divided into two main layers, data dictionary and semantic data model. Both parts may share records of business data, which are then packed into a business signature repository. This is a part of the data dictionary, which is open for reuse in the semantic data model.

The data dictionary contains the signatures of server actions and client pages, together with their parts.

5.3.1 Semantic Data Model

The semantic data model represents the system state between user interactions. Remember, that OCL itself has no fixed rule, when OCL invariants given for a class in the semantic data model have to hold [32]. For Form-Oriented Analysis therefore we fix this ambiguity and prescribe, that these class invariants have to hold every time a server action is left. In defining the semantic data model we are not concerned with technical transactionality. Instead access to the semantic data model is governed by the same analysis-level atomicity as it is used in other analysis techniques, namely Modern Structured Analysis. Analysis-level atomicity has in Form-Oriented Analysis the following semantics: The system modeled by FOA is at a single point in time only with regard to one user in a server state.

5.3.2 Data Dictionary Metapackage

The data dictionary is the model of the message signature types appearing in the FOA Model. Data dictionary types are also called object-by-value types, or messages. They represent immutable values. Such immutable types are familiar

also from OCL, where results of OCL expressions are conceived as immutable types, they are called value types in OCL. [32].

The concept of a data dictionary plays a key role in Structured Analysis. We use a reconstruction of the data dictionary concept in the context of UML. In this context the data dictionary is obviously a class diagram with restricted features. Data dictionary types are simple and straightforwardly hierarchical in their structure. In Structured Analysis data dictionaries are constructed by sum and product operations and additional list constructs. In the context of UML all these mechanisms can be modeled as composition, and the distinction between sum, product and lists is expressed by cardinality.

Formally we introduce two stereotypes for data dictionary types, namely << messagePart >> and << message >> . Messages represent data dictionary types that are used as super parameters. As said above, messages are constructed by composition from message parts. << message >> is derived from << messagePart >> . Hence they may only have message parts as composite parts. They may use recursive composition. They can contain special opaque references to objects of the data model, which we discuss in the next section.

5.3.3 References to the Semantic Data Model

Since our semantic data model corresponds within Structured Analysis to a single consolidated store [78], references to the semantic data model could be given as a foreign key, referencing to artificial primary keys in the data model. This foreign key would not be protected by referential integrity, since the data dictionary object must remain the same even if the object is deleted. Furthermore we make mandatory that artificial primary keys are not reused. This however is not the elegant solution we would like to see in our analysis model. In our high level considerations we clearly prefer to replace primary key mechanisms by associations since a foreign key is a low level realization of an association. However, UML associations unfortunately behave semantically different from foreign keys. Namely foreign keys can be compared after the referenced object is deleted, UML links on the other hand are deleted as soon as one of the corresponding objects is deleted.

We call associations that represent foreign key based references as opaque references. We use for such references a stereotype << opaque >> in our data dictionary. Opaque references are the only kind of references to the data model that are allowed within the data dictionary. The term opaque refers to the fact that opaque references behave semantically like foreign keys, but their value is guaranteed to be hidden by the mechanism. Opaque references must not be explicitly declared opaque within the Data dictionary, instead we use the convention that references to the data model must be denoted as attributes of the data dictionary class. This notation is conceived as specification that the reference is opaque. Opaque references may remind weak references types in imperative garbage-collecting languages, i.e. references without referential integrity. However, in conceptual modeling the term weak reference is used for a completely different concept, namely references to semantically weak objects, i.e. for com-

position style relationships [145]. We therefore recommend speaking of opaque references, motivated by the fact, that they correspond to opaque references known from the programming language MODULA-2, since the semantic data model can be seen as a single data abstraction unit. Within Modern Structured Analysis [66], the key fields correspond to our opaque references. Key fields are denoted by a sign.

Summing up the requirements for messages given so far, we conclude that a data type that is part of the data dictionary can have the following components:

- Primitive valued attributes.
- messageParts associated by composition.
- References to objects from the semantic data model. They are stereotyped as opaque references.

Each message object represents with its object ID a single event of message passing. For that reason, if the same information is sent twice, we model this information still as two distinct message objects. A naive modeler may view this as waste of space, therefore it is important to be aware of the purpose of an analysis model once more: The analysis model is designed for clear semantics, not for efficient implementation. The actual design and implementation is not required to chose the same representation, it is not even required to actually keep a log of all messages sent in the same way as it is done in the analysis model. Together with a state visit of a model state, a message of a data dictionary type forms a superparameter. We recall that this terminology refers to the fact, that a message is a single object, but it has to be seen as the analogue of a whole list of actual parameters provided for that state visit.

5.4 Semantics of Dialogue Constraints

Dialogue Constraints have been introduced as stereotypes for constraints. In this section we give the formal semantics of dialogue constraints by giving for each dialogue constraint a translation into parts of standard OCL constraints. As we will see, all DCL constraints are related to OCL method constraints in their semantics. Accordingly, it is a reasonable choice to map them onto parts of OCL preconditions.

For that purpose we have to make a formal construction that makes explicit and utilizes the aforementioned unification of structural and behavioral features of form charts. We have argued that SHDs and therefore form charts achieve a unification of structural and behavioral semantics, since the relevant temporal evolution of SHDs is already represented in the evolution of its object net, and not only in the evolution of method calls. Now, in order to integrate this view with standard behavioral modeling, especially with OCL method constraints, we have to construct a behavioral semantics which is a perfect mirror of the structural features of SHDs. Hence behavioral semantics of SHDs has the task

to introduce methods which are called in a defined mandatory way in connection with state changes. These methods serve as contexts for standard OCL constraints, i.e. preconditions and postconditions. The DCL constraints are then mapped onto parts of these constraints.

As we have done in the case of the form chart framework, we first give a behavioral model of state history diagrams and then give a more restrictive model for form charts. The behavioral model has the purpose to prescribe, which method calls have to be performed in the course of state changes. Partly these method calls can be enforced by the specification tools of our UML notation. Partly however, we again reach the limits of UML semantics, this time for the behavioral models. As it was observed in [118] message sequence charts are unsuited for giving behavioral semantics because they even lack the expressibility to specify mandatory behavior. Message sequence charts just specify example interaction sequences. Harel and Damm proposed Live sequence charts with richer semantics, which also allows for specification of conditions. We do not need this expressibility for our purposes since in the semantics of form charts mandatory behavior will be unconditional, as we will see. Therefore we simply specify mandatory behavior as pseudocode, where necessary.

We define in the abstract class `ModelState` three methods, enterState(), makeASuperParam() and changeState(), which perform the handshake between subsequent state visits. The central method is the enterState() method for each state, which has to be called on each newly inserted visit. For that purpose the enterState() method is declared abstract in the `ModelState` class. Each modeler defined derived state must overwrite the enterState method. The new visit has to be seen as being the conceptual parameter of its own enterState() method. `ModelState` has an attribute `signature`, which has to be from the corresponding data dictionary class and which replaces the parameter list, therefore we have assigned the name superparameter to the `ModelState` instance. Each state has a makeASuperParam() method, which must be called when the state is left, and which constructs the superparameter. The superparameter is passed to the enterState() method in sigma calculus style. This means that the enterState() method is called on the superparameter without method parameters. State changes are performed by a single method changeState() in the old state. The changeState() method of one state calls its own makeASuperParam() and the enterState() of the next state. makeASuperParam() and enterState() must not be called from any other method. changeState() is defined final in `ModelState`. In Java-like pseudocode:

```
abstract class ModelState extends State {
  // ....
  abstract ModelState makeASuperParam();
  abstract void enterState();
  final void changeState(){
     ModelState aSuperParam = makeASuperParam();
     aSuperParam.enterState();
  }
}
```

The control logic that invokes changeState() of the current visit is not prescribed. However, the only way to change the state is by calling changeState() of the current visit.

State Output Constraints

When we introduced the dialogue constraints, we characterized two of them as the specializations of StateOutputConstraint, the single new constraint type already available on the SHD level. The formal reason for SHDs having a new constraint context is that this new context is conceptually placed on the edge between two states, and it is therefore called transition context. In this newly introduced transition context there is no `self` keyword, but the role names of the transition ends can be used, especially the role names `source` and `target` from the general transition. The role names refer both to the `ModelState` as well as to the corresponding data dictionary object (the attribute name `signature` can be omitted. The central added value of the state output constraint is the fact that source as well as target are properly typed, i.e. they have as type the `ModelState` at the respective end.

From the final specification of changeState() as given above it is known that the precondition of enterState() is immediately executed after the postcondition of makeASuperParam(), hence the system state is the same in both constraint contexts. The state output constraint is intended to be executed in that point in time, which can be seen as a single instant, therefore in principle one could use one of the mentioned contexts instead. Semantically therefore we map the state output constraint either onto the post condition of makeASuperParam() or onto the precondition of enterState(). However, in both context there is less typing information available than in the state output constraint. The state output constraint makes use of the fact that in SHDs for each enterState() method the possible predecessor are known from the diagram, and for each changeState() method the possible successors are known. This is captured by the properly typing of source and target.

The more specific dialogue constraints defined for form charts pay tribute to the more elaborate model of form charts. In the case of form charts, the two classes of states have more specializes semantics than in the case of general SHDs as we will explain in the following.

It is important to realize that state output constraints must not be mistaken for guard conditions known from state machines. Guard conditions are in state machines specify behavior. State output constraints are on the other hand design by contract constraints. They constrain the possible implementations.

5.4.1 Semantics for Form Chart Dialogue Constraints

ClientPage Visits

ClientPage visits are the superparameters computed by the preceding ServerAction makeASuperParam() and offered to enterState(). The ClientPage methods

however have to be seen as provided by a Browser. enterState() and makeASuperParam() of a ClientPage are therefore not individually modeled, but conceived as being interpreted by a generic browser concept. This concept is called the abstract browser. The browser therefore is a parametric polymorphic concept. The named ClientPage methods are not implemented, but interpreted by using type reflection on the ClientPage class.

The ClientPage class contains the information that has to be shown to the user together with interaction possibilities, links and forms. Since form charts are used in the analysis phase, the ClientPage superparameter is assumed to be a pure content object. The ClientPage superparameter is a hierarchical constant data type constructed with aggregations. As explained earlier, in Form-Oriented Analysis we consider an abstract browser as given. The analyst's view of the browser is a black box taking the content object and delivering a state change to a ServerAction later. The presentation of the content to the user and the construction of the method calls to the allowed server actions according to the SHD is the task of the abstract browser. The analyst assumes that the page offers a form for each outgoing transition of the ClientPage. However in a current ClientPage visit certain forms may be disabled. For this purpose the ClientPage is assumed to have for each outgoing transition A a flag formAenabled which specifies whether the transition is enabled. They are specified by the enabling conditions.

Generation of ServerAction Visits

ServerAction visits are the actual superparameters that are given in a state change to a ServerAction. The objects are created whenever the user triggers a state change in the dialogue. The ServerAction superparameter is constructed by the browser by using the ClientPage visit as a page description. Since form charts are in contrast to concrete technologies like HTML a strong typed concept, the type description of the serverAction has not to be contained in the ClientPage visit, but the default parameters and the enabled flags have to be provided. The abstract browser constructs the new ServerAction visit from the user input.

The semantic data model is updated by a sideeffect of the server action, for which Form-Oriented Analysis makes explicitly no strict specification requirements. OCL offers means for specifying updates with the postcondition constraint stereotype, which uses a rudimentary temporal calculus given by the modal operator `pre`. However, this calculus may lead the average modeler easily to nonoperational specifications without any need. Instead we use in our example an SQL like update pseudocode notation as an extension for OCL. This pseudocode is using three types of directives: insert, delete, update. Note however that from the standpoint of Form-Oriented Analysis these operations are deliberately informal in contrast to the DCL constraints. A typical specification of the side effect would be to specify a single method call in the side effect. Business operations can be decomposed; the top layer is formed by the side effects of server actions. Server actions are executed atomic and are therefore

multi-user safe operations on the semantic data model.

Enabling Conditions

The outgoing transitions in the class diagram for each ClientPage depict the statically allowed page changes. Often a certain form shall be offered only if certain conditions hold, e.g. a bid in an auction is possible only if the auction is still running. Since the page shown to the user is not updated unless the user triggers a page change, the decision whether to show a form or not has to be taken in the changeState() leading to the current ClientPage visit. The enabling condition is mapped to a part of a precondition of enterState().

```
enterState()
pre:  formAEnabled = enablingConditionA
pre:  formBEnabled = enablingConditionB
```

Alternatively each enabling condition can be seen as a query that produces the Boolean value that is assigned to formXenabled. Typically, the same constraint has to be re-evaluated after the user interaction. In the example above, the auction may end while the user has the form on the page. Then the same OCL expression is also part of another constraint stereotype, especially
<< serverinputconstraint >> or << flowcondition >> .

Server Input Constraint

These constraints appear only in incomplete models or models labeled as TBD, to be defined [48]. A server input constraint expresses that the ServerAction is assumed to work correctly only if the server input constraint holds. In a late refinement step the server input constraint has to be replaced by transitions from the ServerAction to error handlers. Context of the server input constraint is the ServerAction visit. Server input constraints are not preconditions in a design by contract view, since server input constraint violations are not exceptions but known special cases.

Flow Conditions

Flow conditions are constraints on the outgoing transitions of a ServerAction. Context of flow conditions is the ServerAction visit. The semantics of flow conditions can be given by mapping all flow conditions of a state onto parts of a complex postcondition on this.makeASuperParam(). This postcondition has an `elsif` structure. In the `if` or `elsif` conditions the flow conditions appear in the sequence of their numbering. The flow condition must be evaluated on the system state before executing the state change, hence the flow condition has to be transformed by adding the modal operator pre to all state accesses.

In the `then` block after a flow condition, it is assured that a visit of the targeted ClientPage is the new Current State. In the final `then` block the same

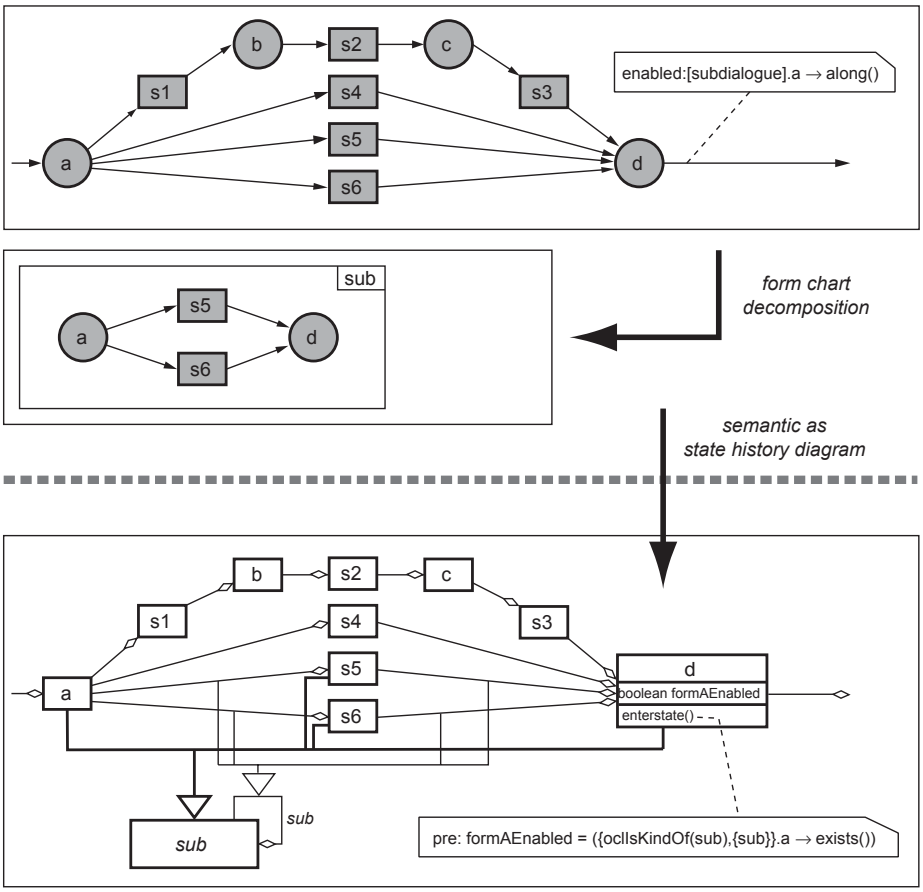

Figure 5.5: Semantics of path expressions in DCL

check is performed for the target of the serverPage transition without a flow condition.

Client Output Constraints and Server Output Constraints

Client output constraints and server output constraints are specializations of state output constraints and live in the new transition context.

5.4.2 Path Expressions in DCl

As an introduction to the general concept of path expressions in OCL we explain path expressions in DCL. Path expressions allow expressing a condition about the path that was taken up to now by the dialogue within the state diagram. In form charts a test on whether the dialogue has chosen a fixed single path can be tested with the new along OCL feature. The path is written backwards

in time. The along feature simply test whether the chosen object exists. More generally constraints are important in which it is tested whether the path has certain properties as long as it remained in a subdialogue. Hence the path has to be restricted to a subdialogue. The concept of form chart features is viable to this approach. Form chart features must not be mistaken for OCL features. The word feature is derived in the context of form charts from the requirements engineering community.

Path expressions that are restricted to paths allowed in a feature are written in DCL by the feature name in square brackets. Formally this concept is a shorthand. DCL path expressions are mapped to general path expressions as introduced in Section 5.5. For this purpose, form chart features are not just diagrams, but come along with a class definition. For each feature diagram a `ModelState` with the name of the diagram is created with a transition to itself, again with the feature name. All model states in the feature as well as the transitions shown in the feature are implicitly derived from these two elements. This is made explicit in Fig. 5.5.

5.4.3 Discussion

We have given semantics for DCL constraints by mapping them to OCL pre- and postconditions. For that purpose we have introduced behavior into our semantics for state history diagrams. In this section we want to discuss briefly an alternative model. As it turns out, it would be possible to model DCL constraints as OCL constraints of the stereotype invariant. As the authors of OCL state explicitly in [32], OCL is ambiguous with respect to the question, at which point in time an invariant must hold. In hat respect it differs from Eiffel [105], where invariants are specified to hold at each instant at which the instance is observable to clients. We can specify for our model that all OCL class invariants must hold immediately after each state transition of our form chart. At this instant we assume the superparameter of the new state to be fully constructed. Astonishingly, we can now map all DCL constraints to invariants. Generally, pre- and postconditions cannot be mapped to invariants. Since we have seen that DCL constraints map easily to pre- and postconditions, it is at least remarkable that such a mapping of DCL constraints to invariants is possible. Indeed, this alternative semantics of DCL will be possible only due to the unified character of our STD semantics. The key argument is that we can give an OCL expression which can be used as part of an invariant, and as such specifies that this OCL invariant is executed only for the current visit and therefore only once for each visit. This condition is simply the check, whether the current visit is the currently last in the path. Due to the sentinel technique used in modeling SHD's, this can be recognized by checking, whether the succeeding state is the currentEnd:

```
self.target->isTypeOf(CurrentEnd)
```

The key technique by modeling a DCL constraint A as invariants is now the

conjunction of this expression to A. By doing so we achieve that A is evaluated only for the current visit.

The technique shown above may have the advantage that no behavioral model is needed for form charts, but has also disadvantages, namely in context with flow conditions. The result of flow conditions has to be stored in the state.

5.5 Path Expressions

We now define path expressions for collecting objects along the transitive closure of link paths, called gathering in the following. The notation is needed to give semantics to the "along" notation of the dialogue constraint language used e.g. in writing enabling conditions during Form-Oriented Analysis. However path expressions have a justification in their own right. We start with an unrestricted wildcard notation for expressing path navigations. Consider the following OCL expression.

```
A
⋆.C
```

For every arbitrary fixed object of the context type A the expression denotes the bag of objects of target type C that are reachable from the context type object along a path of links, i.e. not only directly connected objects, but all reachable objects are gathered. Consider the example given in Fig. 5.6. It shows the bag resulting from the application of the above expression to a concrete object net. An object that is reachable along several link paths occurs more than once in the bag, one time for each path. Only cycle-free link paths in the object net are considered, i.e. link paths, in which each object is visited only once. This ensures finiteness of the result bag.

Theorem 2 *A path expression specifies a finite bag.*

Proof. By definition of the path expession there is a one-to-one correspondence between the object occurrences in the result bag and the cycle-free paths. There can be only as many cycle-free paths in the object net, as there are permutations of the objects.Hence the number of cycle-free paths and therefore te number of bag elements is finite. □

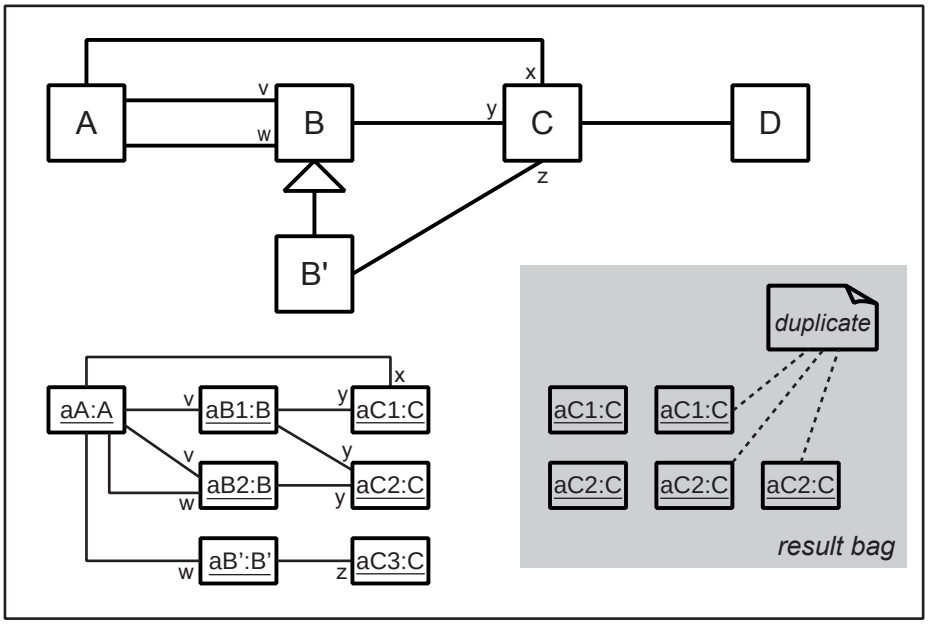

Figure 5.6: An example class and object diagram, together with the result bag of the path expression `*.C`, applied to the only instance of Class A.

With respect to a possible generalization hierarchy of the classes only such link paths are considered for which the links are instances of connected, but strictly interchanging associations in the class diagram. Therefore the object `aC3:C` in the current example does not belong to the result set of the above expression, because following the link path, from the viewpoint of the start object the connected object `aB':B'` if of type `B` and has no link to the object `aC3:C`. The above expression has the same meaning as the following OCL expression.

```
A
self.v.y→union(self.w.y)→union(self.x)
```

Recall from the latter expression that in OCL a multi-step navigation is a shorthand notation for the repeated application of collect and therefore yields a bag.

The wildcard notation may be used straightforwardly for writing constraints on cyclic class diagrams based on aggregations, too. The semantics remains the same, except that a path is only considered, if all aggregation links point in the same direction.

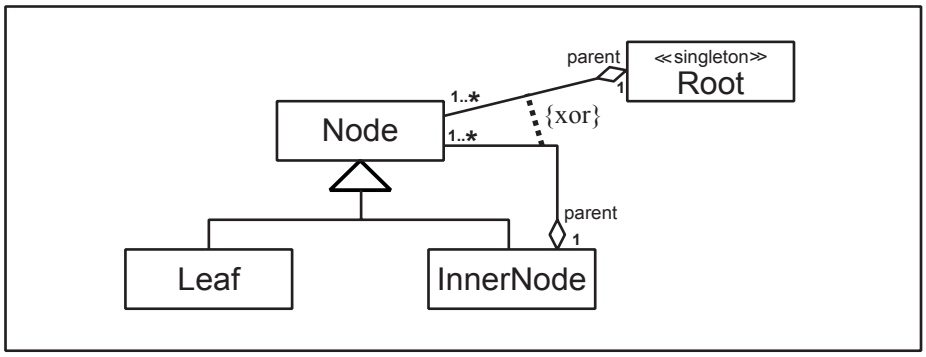

Figure 5.7: UML tree definition

Consider the example in Fig. 5.7. The following constraint yields the set of leaves for an object tree which is accessed through its root node.

```
Root
⋆.Leaf→asSet
```

This expression has only non-trivial counterparts in UML. A notation like path expressions is clearly needed. Cyclic class diagrams with aggregations are the backbone of proven object-oriented patterns, both from problem domains and solution domains, e.g. the structural kernels of both the composite design pattern and the organisation hierarchies analysis patterns [74] are trees.

We proceed with the general notation for path expressions, which is summarized in the following expression:

```
ContextType
{oclConstraint,{package.associationName,..}}.GatheringType
```

The path expression consists of a structured wildcard and the type of the objects that are to be gathered. The structured wildcard is a constraint on the link paths that may be followed to gather objects. The wildcard consist of an OCL constraint and an association constraint which is a set of association specifications. In a valid link path, every object must fulfill the given OCL constraint. There are subtle typing aspects of this mechanism. The OCL constraint of a path expression is not tight to a single context; it must be evaluated with respect to objects of possibly different types. This is not a problem if all classes of the underlying class association path have a common supertype and the OCL constraint is written in terms of this type. Otherwise the expression must be made general enough by first questioning the type of the object.

Furthermore in a valid link path every link must adhere to the association constraint. This constraint is a set of association specifications. A link in a valid link path must be an instance of an association which is a generalization

of one of the association specified in the association constraint. The modeler specifies an association by giving its qualified name consisting of a package name and the association name. If the model is not structured by packages, only an association name suffices. Association names are unique in packages. If the association specification is an empty set, the link path is not constrained with respect to the links.

The structured wildcard is a powerful narrowing mechanism, e.g. to exclude object net cycles from constraints involving path expressions. It is exploited in section to give semantics to path expressions of dialogue constraint language used in form charts.

At least consider the first path expression in unrestricted wildcard notation in the preceding section. It is a shorthand notation for the following verbose path expression.

A
{true,∅}.C

5.5.1 Multiple State occurrences, Enabling Conditions, Path Expressions

In Figure 4.2 we have shown, how an enabling condition based on path expressions can be shown graphically by multiple occurrences of a state in the form chart. Now we have to give precise semantics to this method in the following way: If a state occurs more then once in a form chart diagram, these occurrences represent distinct anonymous subclasses of the state, called substates. If the same system is specified with a form chart with only one occurrence of the state, then whenever this state is visited an instance of the according substate is instantiated. The parent state is effectively abstract since it cannot be instantiated through the form chart operational semantics. Therefore the state named s5 has to get path dependent enabling conditions because for each of its substates can proceed only through one of the outgoing edges.

The menu feature can be understood by the above semantics for multiple state occurrences, too. In a menu, an edge between two state sets represents the complete bipartite graph of edges between the elements of the page set. We can either see the menu feature as a graphical abbreviation for the complete graph. We can interpret the elements of the state set as multiple state occurrences of the same state. In this way we yield semantics for the menu feature equivalent to the interpretation as abbreviation.