An Introduction to State History Diagrams

Technical Report B-02-09

Dirk Draheim and Gerald Weber
Institute of Computer Science
Free University Berlin
email: {draheim,weber}@inf.fu-berlin.de

March 2002

Abstract

This paper provides two contributions to UML modeling, namely state history diagrams and path expressions. Both concepts are directly motivated by a new analysis technique, form oriented analysis, which is tailored for an important class of interactive applications including web applications.

Contents

1 Introduction

We introduce two contributions to modeling techniques which have applications in practical modeling tasks. First we define state history diagrams, which are a semantic unification of class diagrams and state transition diagrams. Secondly we introduce path expressions, an extension of OCL [8] [4] which decisively improves the navigational features of OCL. Both contributions are not only abstractly introduced, but shown to provide significant added value in the context of form based applications.

Form based applications are an appropriate abstraction from an important class of interactive systems including especially web applications. Form oriented analysis [1] [2] addresses the analysis phase of form based applications, in contrast to other approaches focusing on OO design of web application architectures [3]. Form based analysis makes use of both introduced concepts. Form based analysis has a special type of SHD called form charts. Form charts are bipartite state transition diagrams with an OCL extension DCL, which is mapped in this paper on OCL.

Form oriented analysis is also an intuitive application domain of the second innovation, OCL path expressions. However, path expressions are a much more general concept. One of the advantages of OCL as a declarative language is the dot notation for navigation e.g. Path expressions are a necessary extension of this concept, which can be intuitively conceived as a powerful structural wildcard notation for paths.

In section 2 we introduce state history diagrams. In section 3 we introduce path expressions. In section 4 we recall form oriented analysis and form charts. In section 6 we introduce the dialogue constraint language, DCL.

2 State History Diagrams

State transition diagrams, STDs, are the nonhierarchical version of state machines [4]. Our contributions in this paper are presented for this subclass of state machines, but are fully generalizable to state machines. However we introduce only the theory for STDs, since this part is sufficient for the application domain we discuss, namely form charts.

STDs 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. The motivation for state history diagrams, SHD, is the fact that in important applications of STDs to every STD a structurally identical, canonical class diagram has to be considered. The object net over this class diagram is the history of the visits to the states of the STD. The concept behind SHDs is the approach that one needs only a single diagram which is STD and class diagram at the same time. This must not be confused with the metamodeling approach, where a single diagram *type* is sufficient.

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 connectives are called transitions and they are associations, their instances are 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. Considering only the connections through transitions, this object net is a path from the start visit of the process to the current visit. This path is seen 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 define a type hierarchy for classes and associations, as shown in Fig. 1. The basic class is `State` and it has an aggregation to itself, called `transition`. All state classes shall be derived from `State` and all transitions are indirectly derived from `transition`. These derivations do not appear in the SHD. The ends of `transition` have roles `source` and `target`.

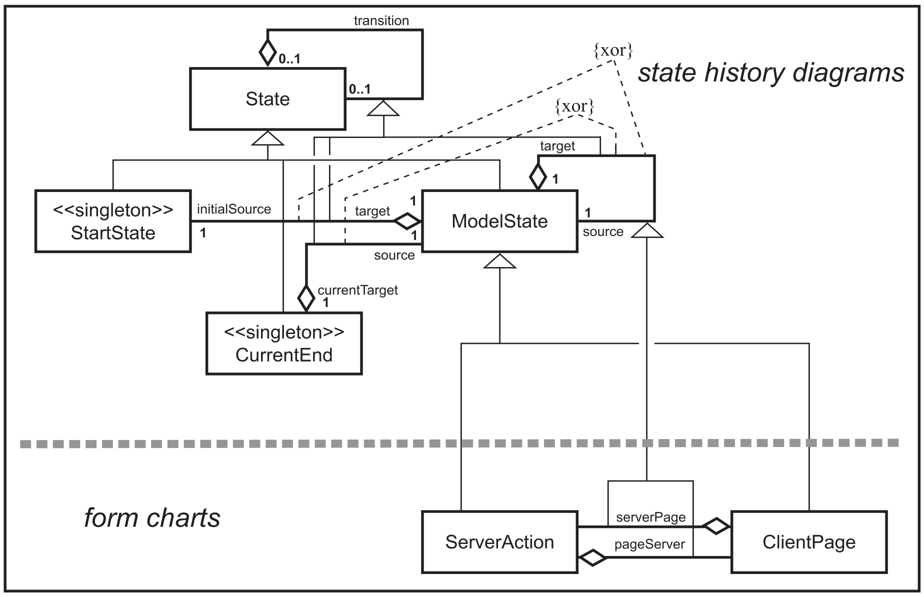

Figure 1: State history diagrams and form charts

2.1 Constraints on the Object Net

We now discuss the formalization of the constraints on the process as the object net over a SHD. Except for the start visit every visit must have exactly one predecessor. 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. Both `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 indirectly derived from ModelState. The cardinalities are expressed in the class diagram in Figure 1. 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.

2.2 Semantics of State History Diagrams

The model states in SHDs can have attributes and also parts defined by aggregations. Visits of model states are assumed to be deep immutable. Each state has an enterState() method, which has to be called on each newly inserted visit. The new visit has to be seen as being the conceptional parameter of its own enterState() method. The attribute list of the ModelState replaces the parameter list, therefore we have assigned the name superparameter to this concept of a single parameter. 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 which 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.

2.3 State Output Constraints

In SHDs for each enterState() method the possible predecessor are known from the diagram, and for each changeState() method the possible successors are known. SHDs have a new constraint context which is conceptually placed on the edge between two states. The constraint in this context is called state output constraint. This new context corresponds to the fact that from the implementation of changeState() it is known that the precondition of enterState() is immediately executed after the postcondition of makeASuperParam(). The transition constraint could be placed as post condition of makeASuperParam() or as precondition of enterState(). However, in each of the contexts the actual type of the respective other type is not known. This is overcome in the newly introduced transition context: there is no `self` keyword, but the rolenames of the transiotion ends can be used, especially the rolenames `source` and `target` from the general transition.

3 Path Expressions

We introduce 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 which is used in writing enabling conditions during form chart analysis. However path expression have a justification in their own right. We start with an unrestricted wildcard notation for expressing path navigations. Consider the following OCL expression.

<u>A</u>
⋆.B

For every arbitrary fixed object of the context type A the expression denotes the bag of objects of target type B that are linked to the context type object by a path of links, i.e. not only directly connected objects, but all reachable objects are gathered. Consider the example given in Fig. 2. 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. Only link paths in the object net which are acyclic are considered. This ensures finiteness of the result bag. With respect to a possible generalization structure only such link paths are considered that are instances of strictly interchanging class association paths 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.

<u>A</u>
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.

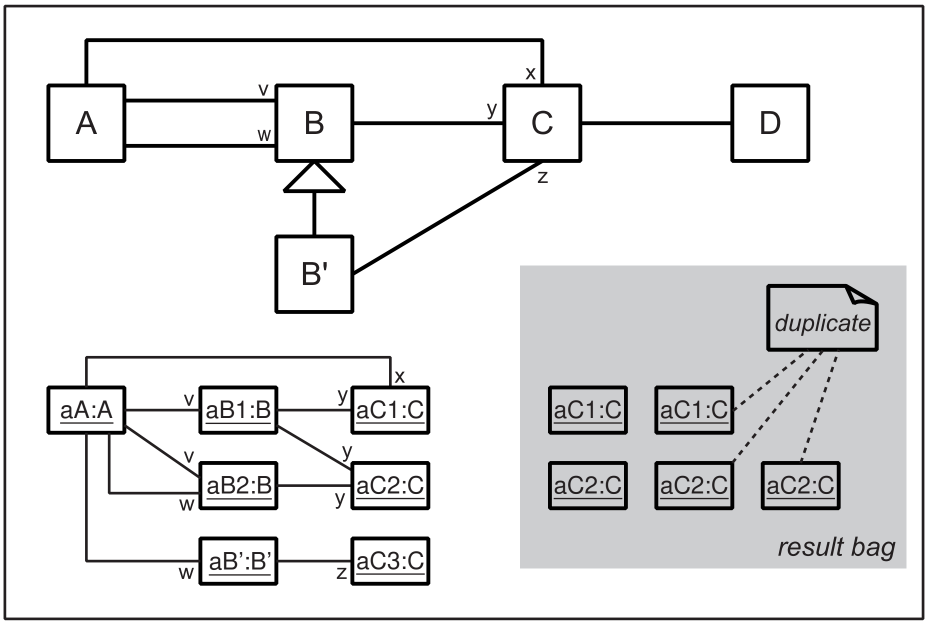

Figure 2: Example class and object diagram

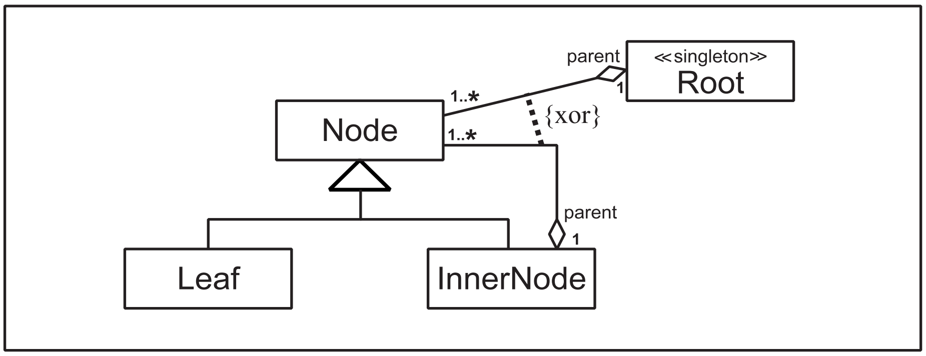

Figure 3: UML tree definition

The wildcard notation may be used straightforwardly for writing constraints on cyclic class diagrams, too. The semantics remains the same. Consider the example in Fig. 3. The following constraint yields for an object tree which is accessed through its root node, the set of its leafs.

```
Root
⋆.Leaf→asSet
```

This expression has only non-trivial counterparts in UML. A notation like path expressions is clearly needed. Cyclic class diagrams are the backbone of proven object oriented patterns, both from problem domains and solution domains, e.g. the structural reducts of both the composite design pattern and the organisation hierarchies analysis patterns [7] 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 which 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 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 preceeding section. It is a shorthand notation for the following verbose path expression.

$\underline{\mathtt{A}}$
$\{\mathtt{true},\emptyset\}.\mathtt{C}$

4 Form Oriented Analysis

The usefulness of SHDs and path expressions is exemplified in form oriented analysis. Form oriented analysis is tailored for form based systems like today's web interfaces. These systems have a specific interaction style, named form based, submit/response. This style allows for abstraction from finegrained user interaction on one page and allows for viewing user interaction as high level requests. From an analysis point of view the user virtually submits fully typed requests in the strong type system. In terms of the user interface community as laid down in the seminal Seeheim model [6] form based systems can be seen as using an application independent user interface management system, which is best exemplified by the standard web browser.

Using these insights, in form oriented analysis the interface is modeled by a visual artifact called form chart beside an analysis class diagram. A form chart models the states of the user interface. We consider in this paper only single page dialogues. In a further work we analyze multi window dialogues. However, single window dialogues are a useful abstraction of web applications.

4.1 Form Charts as SHDs

The form chart is a bipartite state transition diagram which we model in this paper as a SHD.

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 the bipartiteness: In the form chart, all transitions must be derived from either `pageServer` or `serverPage`. We explain now the special properties of form charts as SHDs. In section 5 we explain the semantics of form charts.

In form charts the ServerAction states are short lived and left automatically. Only the Client-Page states wait for a user input.

Form charts have new constraint stereotypes. In these stereotypes certain extensiond of OCL are allowed. We call the new constraint stereotypes together with these extension the dialogue constraint language DCL. The important well-formedness rules concerning the bipartiteness of form charts are already specified by the class diagram in Fig. 1.

5 Semantics of Form Charts

The semantics of form charts contains the following complexes: First the interpretation of ClientPage visits, secondly the rules for creating the ServerAction visits and thirdly as presented in the section 6 the semantics of the different constraint stereotypes.

5.1 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 which 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 datatype 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.

5.2 Generation of ServerAction Visits

ServerAction visits are the actual superparameters which 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.

6 Dialogue Constraint Language

In this section we define the well-formedness rules and semantics of the different constraint stereotypes introduced in DCL (Fig.4). Afterwards we explain special path expressions in DCL.

6.1 Well-Formedness Rules

Form charts introduce new kinds of constraints at the ends of transitions [2]. These new constraint stereotypes together with their semantics form the dialogue constraint language. The semantics

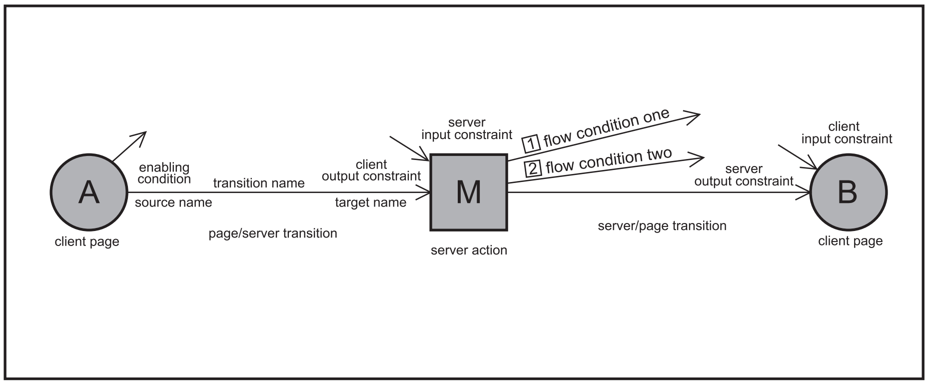

Figure 4: Form chart notational elements

of the new constraint stereotypes is given in section 5. In this section we give the formal rules where these new constraints are allowed. These rules are well-formedness rules for form charts.

Formally, we define stereotypes for constraints, similar to the stereotypes << invariant >> , << precondition >> and << postcondition >> . These stereotypes are constraints at transition ends, and are called dialogue constraints. They are derived from the stereotype << dialogueconstraint >> . The dialogue constraints appear as new labels in DCL, a variant of OCL described in section 6. The dialogue constraint stereotypes have the following metamodel constraints:

```
enablingcondition
self.stereotype→instanceOf(sourceend) and
self.association→instanceOf(pageserver)

clientoutputconstraint
self.stereotype→instanceOf(targeteend) and
self.association→instanceOf(pageserver)

serveroutputconstraint
self.stereotype→instanceOf(targeteend) and
self.association→instanceOf(serverpage)

flowcondition
self.stereotype→instanceOf(sourceend) and
self.association→instanceOf(serverpage)
```

Nonformal metamodel constraints are: Only one constraint of the same stereotype is allowed for the same context. The numbers of flow conditions must be ascending. For each ServerAction there may be only one flow condition which is not numbered.

6.2 Semantics for Constraint Stereotypes

6.2.1 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 which is assigned to formXenabled. Typically, the same constraint has to be reevaluated 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 >> .

6.2.2 Server Input Constraint

These constraints appear only in incomplete models, or models labeled as TBD, to be defined [5]. 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.

6.2.3 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 condition appear in the sequence of their numbering. 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 check is performed for the target of the serverPage transition without a flow condition.

6.2.4 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.

6.3 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 to express a condition about the path which 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 he 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 3. For this purpose, form chart features are not just diagrams, but come along with a class definition. For ech feature diagram, a modelState with the name of the diagram is created with a transition to itself, again with the feature name. All modelStates 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.

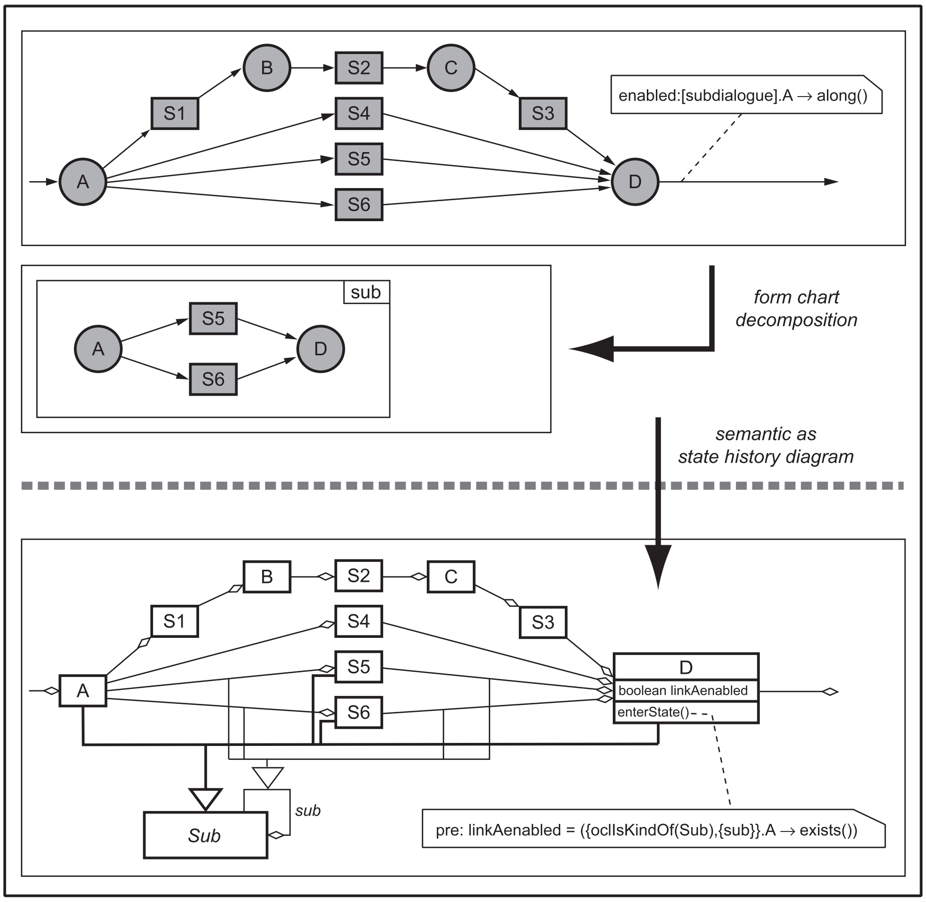

Figure 5: Semantics of path expressions in DCL

References

[1] Draheim, D., Weber, G.: An Introduction to Form Storyboarding. Technical Report B-02-06. Institute of Computer Science, Free University Berlin, March 2002.

[2] Draheim, D., Weber, G.: Form Charts and Dialogue Constraints. Technical Report B-02-08. Institute of Computer Science, Free University Berlin, March 2002.

[3] J. Conallen, "Modeling Web Application Architectures with UML", Communications of the ACM 42(10), 1999, pp.63–70.

[4] Object Management Group, "OMG Unified Modeling Language Specification", version 1.4, September 2001.

[5] IEEE Std 830-1993, "Recommended Practice for Software Requirements Specifications", Software Engineering Standards Committee of the IEEE Computer Society, New York, 1993.

[6] G. E. Pfaff, "User Interface Management Systems", Springer, Berlin, 1985.

[7] Martin Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997

[8] J. Warmer, A. Kleppe, "The Object Constraint Language", Addison Wesley, 1999