On the Quality of Navigation Models with Content-Modification Operations

Jordi Cabot[1], Jordi Ceballos[1], and Cristina Gómez[2]

[1] Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
{jcabot,jceballos}@uoc.edu
[2] Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
cristina@lsi.upc.edu

Abstract. Initially, web development methods focused on the generation of read-only web applications for browsing the data stored in relational database systems. Lately, many have evolved to include content-modification functionalities. As a consequence, we believe that existing quality properties for web model designs must be complemented with new property definitions. In particular, we propose two new quality properties that take the relationship between navigation models and the related data models into account. The properties check if navigation models include all necessary content-modification operations and whether all possible navigation paths modify the underlying data in a consistent way. In this paper, we show how to determine if a navigation model verifies both properties and also how to, given a data model, automatically generate a preliminary navigation model satisfying them.

1 Introduction

Many web development methods are evolving to cover the definition of full-fledged web applications, including data processing and manipulation functionalities. As a consequence, the models involved in the specification of a web application (that is, the data model to specify the data used by the application, the navigation model to describe the organization of the front-end interface and the presentation model to personalize its graphical aspect) have been extended with new modelling primitives.

One of the most relevant evolutions is the extension of navigation models with content-modification primitives that permit to modify the data managed by the web application. These primitives may be basic operations (insert/delete/update operations, as in WebML [4]) or references to more complex operations defined in the data model (as in OOWS [13]) or in a different model (as in the operational models proposed in [10]).

These new primitives complicate the definition of web model designs. Even for small web applications, data and navigation models can become huge and complex. Consequently, their definition is a time consuming and error prone process. This is a critical issue since their quality is very important, especially when web designs are used to automatically derive the implementation of the web application.

Up to now, quality properties for navigation models are based mainly on a purely syntax consistency analysis of the model structure (for instance, a common

L. Baresi, P. Fraternali, and G.-J. Houben (Eds.): ICWE 2007, LNCS 4607, pp. 59 – 73, 2007.

verification is to check the *reachability* of all pages or that there are no *broken* links). These properties do not consider quality issues involving the new content modification operations that may appear in the navigation models. Therefore, existing properties are not suited to assess the quality of such models.

The main goal of this paper is to complement the existing set of quality properties defined for navigation models with two new quality properties that focus on the relationship between the content-management operations appearing in a navigation model and the data model specified for the same web application. Through these properties we can check early in the development process the quality of these extended navigation models. Additionally, we show how these properties help in the automatic generation of a preliminary navigation model once the related data model has been specified. This way we speed up the web development process because designers need not define the navigation model from scratch.

The first property we propose is the *completeness* of a navigation model with respect to its data model. We say that a navigation model is complete if the user can manipulate all the data underlying the web application by means of the modification operations included in the model (except for those parts of the data that the designer defines as derived or read-only). Incomplete navigation models result in web applications with data that a user can never modify. As an example, consider the partial data model shown in Fig. 1. Assuming that all elements of the data model are modifiable, a navigation model with a single page to modify sale objects is incomplete since users are unable to enter or modify sale lines.

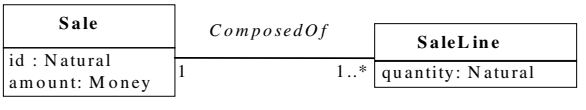

Fig. 1. Example of a data model

The second property is the *correctness* of a navigation model with respect to its data model. A navigation model is correct when all navigation paths admit at least one possible run (i.e. a run-time execution) that leaves the underlying data in a consistent state. Incorrect navigation models result in web applications with some paths that always end up in an inconsistent state. Every time the user interacts with the web application following such paths, an error arises and all user actions carried out until then must be rolled back (or repaired). Following the previous example, since sales must be composed of a minimum of one sale line, navigation models containing a path that permits to insert new sales but where no new sale lines can be created lead to a state where these sales always violate the minimum multiplicity of the *ComposedOf* relationship type. Therefore, such new sales must be discarded.

The rest of the paper is structured as follows. Section 2 reviews some basic concepts of data and navigation models. Section 3 formalizes our quality properties and characterizes the conditions that navigation models must satisfy in order to verify them. Then, Section 4 shows how these properties can be used to derive a preliminary complete and correct navigation model from an initial data model. Section 5 discusses the related work. Finally, Section 6 presents some conclusions and further work.

2 Basic Concepts of Data and Navigation Models

Web modelling languages provide several models to specify a web application. In this section, we review briefly the basic concepts and terminology of data and navigation models, which are the focus of this paper.

2.1 Data Model

A *data model* (also known as *content model*) defines the knowledge about the domain that a web application must have to perform its business functions. Fig. 2 shows an example data model, represented in UML, meant to (partially) model a simple e-commerce application. It contains information about sales, their sale lines and the products they contain. Sales may be associated with the customer purchasing them.

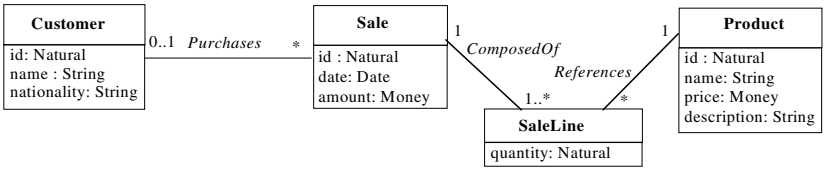

Fig. 2. The data model for the e-commerce application

The most basic constructors in data models are entity types (i.e. classes), relationship types (i.e. associations) and generalizations. Each *entity type ET* contains a set of *attributes*. For instance, in Fig. 2, *Sale* is an entity type with the attributes *id*, *date* and *amount*. Each binary *relationship type RT* has a name and two participants. A participant ET_i in *RT* may have a minimum and maximum cardinality, determining the minimum (resp. maximum) number of relationships (i.e. links) of *RT* in which ET_i may participate. We denote by $min(ET_i,RT)$ and $max(ET_i,RT)$ these cardinality constraints. For instance, *ComposedOf* states that each sale consists of at least one sale line so $min(Sale,ComposedOf)=1$. Each generalization, denoted by $Gens(ET,ET_1,\ldots,ET_n)$, relate a supertype *ET* with a set of subtypes $ET_1,\ldots,ET_n$. Generalizations may be disjoint and/or complete.

Additionally, the data model may include the definition of several operations to modify the state of the data. The basic operations (i.e., content-modification primitives) we consider are: $InsertET(x,v_1,..,v_n)$ (resp. $DeleteET(x)$) to perform the addition (removal) of the entity x into (from) entity type *ET* (optionally, attributes of x may be initialized with values $v_1,..,v_n$), $UpdateA_iET(v,x)$ to set v as the new value for the attribute A_i of entity x and $InsertRT(x_1,x_2,)$ (resp. $DeleteRT(x_1,x_2)$) to perform the addition (removal) of the fact that entities x_1,x_2 participate in an instance of *RT*. More complex operations can be defined as a sequence of these basic ones.

2.2 Navigation Model

A navigation model (also known as a *hypertext model*) specifies the organization of the front-end interfaces of a web application. Fig. 3 shows an excerpt of a possible

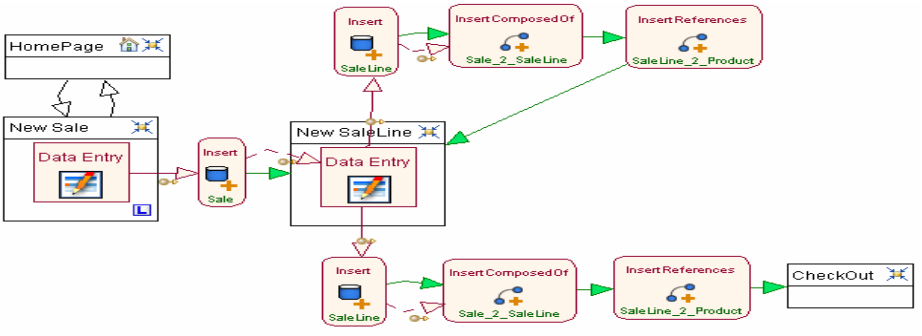

Fig. 3. A fragment of a navigational model for the e-commerce application

navigation model in WebML [4] for the e-commerce application presented in Fig. 2. The model shows the interface to create new sales and their related sale lines.

The most basic constructors of navigation models are pages and links. Pages may include several elements to specify the page contents. For practical purposes, our navigational models do not show the pages internal structure.

Many web modelling languages permit to define navigation models with content-modification operations that are executed as a result of browsing a link. As an example, Fig. 3 shows that when the user navigates to *NewSaleLine* from the *NewSale* page, the operation *InsertSale* is executed (using the parameters provided by the user in *NewSale*). In some languages these operations are simple create/update/delete operations equivalent to the basic operations defined above. Alternatively, other languages allow defining that browsing a link triggers the execution of a complex operation *op* defined in the data model (or in some other model). Then, the basic operations executed during the navigation are the ones specified in the definition of *op*.

Fig. 3 shows the process for creating a new sale. From the home page the user accesses the *NewSale* page. From here, the user may move to the *NewSaleLine* page or return to the *HomePage* again. During the navigation to *NewSaleLine* a new sale (empty or with the selected customer, not shown in the figure) is created because of the *InsertSale* operation attached to the link. In this page, the user selects the product to buy and indicates the quantity. Then, the user may either navigate to the *CheckOut* page (the new sale line and the connections between the line and the sale and between the line and the selected product will be created when browsing the link) or to buy additional products by following the link leading to the *NewSaleLine* page again.

3 Complete and Correct Navigation Models

In this section, a navigation model N is formalized as a graph G_N (section 3.1) in order to check whether N satisfies the completeness (section 3.2) and correctness (section 3.3) properties with respect to its corresponding data model D.

3.1 Graph Representation

Given a navigation model N, the graph $G_N = (V_N , A_N)$ is obtained by means of the following rules:

- Every page in N is a vertex in V_N.
- Every link in N from a page X to a page Y is an arc from X (i.e. from the vertex representing X in G_N) to Y in A_N.
- The label of an arc a stores the (possibly empty) ordered sequence of basic operations associated to the link l represented by a in G_N.

Note that G_N is a directed graph (digraph), since being able to navigate from a page X to a page Y does not imply that the navigation from Y to X is also possible. Sometimes G_N turns out to be a multigraph [3] since it may contain multiple arcs with the same orientation between a pair of vertices v_1 and v_2. This happens when the page corresponding to v_1 contains several links targeting the page represented by v_2.

Fig. 4 shows the graph corresponding to the navigation model of Fig. 3

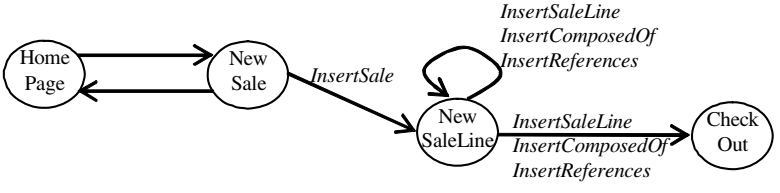

Fig. 4. Graph definition for the navigation model of Fig. 3

3.2 Completeness of a Navigation Model

Intuitively, a navigation model N is complete when the users of the web application may perform all basic operations[1] over the modifiable[2] elements of the corresponding data model D through interacting with the set of pages in N. Incomplete navigation models result in web applications with parts of the data that users cannot modify.

The set of basic operations that N must contain are the ones explicitly provided by the designer in D (or in some additional model [10]). If no operations are provided, this set of necessary basic operations may be automatically generated from D using a simple set of rules. For instance, we could generate an *InsertET* operation for each entity type *ET* in D, an *UpdateA*$_i$*ET* operation for each attribute A_i of *ET* in D and so forth. We denote by set_{op} the set of operations (either defined or generated) for D.

Definition 3.2.1. N is complete when, for each operation *op* in set_{op}, it exists, at least, an arc a in A_N where $op \in label(a)$[3].

[1] When different user groups or roles are defined, we require that at least one of them can perform such an operation.
[2] Designers can mark parts of the data model as *read-only* or *derived*.
[3] *Label(a)* returns the ordered sequence of operations associated to the arc a.

As an example, the graph of Fig. 4 is an incomplete navigation model regarding the data model of Fig. 2 since, for instance, basic operations to create customers and products are missing (no arc contains those operations).

Definition 3.2.2. *N* is minimal when it is complete and, for each operation *op* in set_{op}, there is a single arc *a* in A_N satisfying that $op \in label(a)$.

It is worth noting that non minimal navigation models may be useful. The designer may decide on purpose to offer several alternatives (i.e. several navigation paths) to execute the same kind of modification in the web application. However, we believe it is worth detecting these cases so that the designer can review and validate them.

Given the previous definitions, verification that a given navigation model *N* is complete (or minimal) is quite straightforward, we just need to generate the graph G_N for *N* and check if definition 3.2.1 (or definition 3.2.2) is satisfied by G_N.

3.3 Correctness of a Navigational Model

A navigation model defines the possible navigation paths permitted in the web application. Each navigation path admits several runs (i.e. run-time executions). Each run represents a possible interaction scenario between a user and the application. During a run several modification operations may be applied over the population of the entity and relationships types defined in the data model.

In some particular executions, these operations may turn the data into an inconsistent state (a state where some integrity constraint defined in the data model is not satisfied). This may happen, for instance, when the user does not enter appropriate values in the forms of the pages visited during the navigation. In such cases, all changes performed during the run must be discarded.

It may happen that all possible runs following the same navigation path fail (i.e. leave the data in an inconsistent state due to the execution of the basic operations included in the path). Clearly, such navigation path is completely useless and should be disabled in order to improve the performance and the usability of the web application.

As an example, consider the graph of Fig. 5 representing a simple navigation model consisting of a home page and a page for deleting existing sales. The user selects the sale to be deleted and then browses a link that deletes the sale and returns to the same page again so that additional sales can be deleted. According to the multiplicities of the relationship type *ComposedOf* (Fig. 2), all sales must be related with at least one sale line and all sales lines must be related to a sale. Therefore, when, after deleting a sale, we do not delete the associated sale lines as well (or assign those sale lines to a different sale) these multiplicity constraints will be violated. Hence, every single time the user tries to interact with this navigation model, an error will be raised[4], independently of the sale the user selects.

Intuitively, correct navigation models are those that do not include navigation paths that always (i.e. for all possible runs) lead to an inconsistent data state, regardless of the parameter values the users provide during the interaction with the pages in the path.

[4] Obviously, for this particular example we could define the database so that the sale deletion removes all related sale lines in cascade. However, since this information is not expressed in the model, this must be manually done after the initial code-generation.

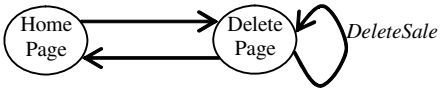

Fig. 5. Graph definition for deleting sales

Definition 3.3.1. A navigation model *N* is correct iff all navigation paths are correct.

This correctness definition relies on the computation of all navigation paths in *N* (section 3.3.1) and on the formalization of the correctness property for a given navigation path (section 3.3.2).

3.3.1 Determining the Possible Navigation Paths in a Navigation Model

Definition 3.3.2. Let S_{NavP} be the set of possible navigation paths in *N*. Then, $S_{NavP} = AllPaths(G_N)$, where $AllPaths(G_N)$ returns the set of all paths in G_N that do not include repeated arcs (these kinds of paths are also known as *trails* [3]).

S_{NavP} considers as valid navigation paths all possible paths. However, in general, not all possible navigation paths in *N* are valid, since a user cannot start browsing the web application choosing an arbitrary page but beginning in some predefined entry page. Similarly with the exit pages, users are expected to follow the navigation path until they arrive to some predefined exit page; after that they can quit or start from the beginning again. For this purpose, some web modelling languages support the concept of home page, the notion of transaction [4] or the concept of service [11].

When this information is available in the navigation model, we may discard from S_{NavP} those paths that either do not start in an entry page or do not finish in an exit page. As an example, given the navigation model of Fig. 3 we could define that *HomePage* is the entry page and that the exit pages are *HomePage* and *CheckOut* page. Then, the navigation sequence {*HomePage*, *NewSale*, *NewSaleLine*, *CheckOut*} would represent a valid path while *{HomePage, NewSale}* would not.

Clearly, the user may quit the web application before reaching an *exit* page. However, in that case the user is not properly interacting with the web application and thus the correctness of this partial interaction does not affect the correctness of the navigation model.

3.3.2 Correctness of a Navigation Path

Correctness of a navigation path depends on the operations associated to the arcs contained in the path. The basic idea is that some operations require the presence of other operations in a precedent or subsequent arc in the path in order to leave the data in a consistent state. For instance, a path including an *InsertSale* operation on an arc a_i requires that at least an operation *InsertComposedOf* appears further in the path (that is, it must exist an arc a_j, j>=i, where $InsertComposedOf \in label(a_j)$). Otherwise, every run on this navigation path will end up in an inconsistent state due to the insertion of a sale not related with any sale line, thus violating the minimum multiplicity constraint of *Sale* in *ComposedOf*.

When an operation op_1 requires the presence of another operation op_2 in the same path we say that op_1 depends on op_2. Dependencies for an operation depend on the

type of the operation (insert, update,…) and on the integrity constraints defined in the data model. We just consider graphical constraints (as the cardinality, disjoint and complete constraints) since most web modelling languages do not permit the definition of textual integrity constraints.

A navigation path must satisfy all dependencies of all operations included in the path to have a chance of finishing successfully. We denote by $SeqOp_{nav}$ the ordered sequence of all operations associated to the arcs contained in the path. Given a navigation path *nav* consisting of the sequence of arcs $a_1 \ldots a_n$, the first operation in $SeqOp_{nav}$ is the first operation in $label(a_1)$ and the last operation is the last operation in $label(a_n)$.

Note that the satisfaction of all dependencies is a necessary condition but not a sufficient one to ensure that all runs following the navigation path end successfully (this will depend on the exact parameter values provided by the user at run-time); this just guarantees that a successful run exists at least (i.e. a navigation path including the creation of a sale and the creation of a link between the sale and a sale line may fail if the parameters for the operations are not the appropriate ones; a navigation path not including the link creation after the sale creation will always fail).

Definition 3.3.3. A navigation path *nav* is correct when, for each operation op_i in $SeqOp_{nav}$, the set of dependencies dep_{opi} for op_i is satisfied in $SeqOp_{nav}$

It may happen that an operation op_i requires N ($N>1$) operations of type op_j. This dependency is satisfied in $SeqOp_{nav}$ when the N op_j operations explicitly appear in it. Alternatively, it is also satisfied if the navigation path *nav* consists of the arcs $a_1 \ldots a_n$, op_i is associated to an arc a_i, op_j to an arc a_j and there is a cycle in the graph including a_j but not a_i. Iterating through the cycle N times, the user could generate the required N op_j operations when running the application.

In the following we define how to compute the exact set of dependencies dep_{op} for an operation *op*. A dependency for an operation *op* is defined as a tuple <*direction, operation, number*> where *operation* is the name of the operation required by *op* and *direction* indicates if *operation* must be executed before *op* (symbol ←), after[5] *op* (symbol →) or if the exact position of *op* is irrelevant (symbol ↑). *Number* informs about how many operations of type *operation* are required by *op*. More complex dependencies are expressed as a sequence of simple ones joined with the logical AND and OR operators (as an example, *op* may require the existence of the operations op_1 and op_2 or, alternatively, the existence of the operation op_3).

Definition 3.3.4. Let *ET* be an entity type and *op* be an operation defined over *ET*. Dep_{op} is computed as follows:

- If *op* = *InsertET*, there is a dependency dep_{RT} = <→, *InsertRT, min(ET,RT)*> for each *RT* where $min(ET,RT) \geq 1$. Additionally, if *ET* is the supertype of a complete generalization $Gens(ET,ET_1,\ldots,ET_n)$ there is a dependency $dep_{GensSup}$ = <→, $InsertET_i$, *1*> for at least one ET_i. If *ET* is a subtype of a disjoint and complete generalization *Gens(ET',ET,…)* we need a dependency $dep_{GensSub}$ = <←, *InsertET', 1*>. Dep_{op} is the union of the dep_{RT}, $dep_{GensSup}$ and $dep_{GensSub}$ dependencies.
- If *op* = *DeleteET*, there is a dependency dep_{RT} = <←, *DeleteRT, min(ET,RT)*> for each *RT* where $min(ET,RT) \geq 1$. Additionally, if *ET* is a subtype of a disjoint

[5] But not necessarily *immediately* before or after.

and complete generalization *Gens(ET', ET,…)* there is a dependency $dep_{GensSub}$ = <→, *DeleteET', 1*>. If *ET* is the supertype of a complete generalization *Gens(ET,ET$_1$,…,ET$_n$)* there is a dependency $dep_{GensSup}$ = <→, *DeleteET$_i$, 1*> for at least an *ET$_i$*. Dep_{op} is the union of the $dep_{RT,}$ $dep_{GensSup}$ and $dep_{GensSub}$ dependencies.

Note that no dependencies are needed for the *UpdateA$_i$ET* operation since changes on attribute values do not either violate cardinality, complete or disjoint constraints.

Definition 3.3.5. Let *RT* be a relationship type with two participants *ET$_1$* and *ET$_2$*. Let *op* be an operation defined over *RT*. Dep_{op} is computed as follows[6]:

- If *op* = *InsertRT*, there is a dependency dep_{op} = <↑, *DeleteRT, 1*> (if *min(ET$_i$,RT)* = *max(ET$_i$,RT)* for just one participant *ET$_i$*) OR <←, *InsertET$_i$, 1*> for each *ET$_i$* such that *min(ET$_i$,RT)* = *max(ET$_i$,RT)* ≥1.
- If *op* = *DeleteRT*, there is a dependency dep_{op} = <↑, *InsertRT, 1*> (if *min(ET$_i$,RT)* = *max(ET$_i$,RT)* for just one participant *ET$_i$*) OR <←, *DeleteET$_i$, 1*> for each *ET$_i$* such that *min(ET$_i$,RT)* = *max(ET$_i$,RT)* ≥1.

Consider the navigation path *nav* for the graph of Fig. 4 consisting of the navigation sequence: {*HomePage*, *NewSale*, *NewSaleLine*, *CheckOut*}. For this path, $SeqOp_{nav}$ = {*InsertSale*, *InsertSaleLine*, *InsertComposedOf*, *InsertReferences*}. To check the correctness of *nav* we must consider the dependencies for all operations in $SeqOp_{nav}$. According to the previous rules, the dependencies are:

$dep_{InsertSale}$ = <→, *InsertComposedOf, 1*>
$dep_{InsertSaleLine}$ = <→, *InsertComposedOf, 1*> *AND* <→, *InsertReferences, 1*> $dep_{InsertComposedOf}$ = <←, *InsertSaleLine, 1*> *OR* <↑, *DeleteComposedOf, 1*>
$dep_{InsertReferences}$ = <←, *InsertSaleLine, 1*> *OR* <↑, *DeleteReferences, 1*>

$SeqOp_{nav}$ satisfies the dependency for *InsertSale* since there is an *InsertComposedOf* operation after the *InsertSale* operation in the sequence. Dependencies for *InsertSaleLine* are satisfied as well because $SeqOp_{nav}$ includes the *InsertComposedOf* and *InsertReferences* operations after *InsertSaleLine*. Dependencies for *InsertComposedOf* require that before this operation we find in $SeqOp_{nav}$ an *InsertSaleLine* operation or (anywhere) a *DeleteComposedOf*. Since $SeqOp_{nav}$ contains the *InsertSaleLine* operation before the *InsertComposedOf* operation, this OR-dependency is also satisfied. This is likewise with the dependencies for *InsertReferences*. Therefore, all dependencies for the operations in the path are satisfied and we may conclude that this navigation path is correct.

Given all these previous definitions, the verification that a navigation model *N* is correct can be summarized in the following steps: 1 – Generation of the graph G_N, 2– computation of the function *AllPaths*, 3 – Computation of $SeqOp_{nav}$ for each navigation path, 4 – Determining the dependencies for all operations in $SeqOp_{nav}$ with the rules presented above and 5 – Checking that all paths satisfy the correctness definition 3.3.3.

[6] For some (unusual) cardinality combinations additional (longer) sets of dependencies could be defined.

4 Generating a Complete and Correct Navigational Model

Given a data model *D*, we show in this section how to automatically derive a preliminary navigation model that is guaranteed to be complete and correct with respect to *D* (idea of *correctness-by-construction* [9]). Designers may complement this initial navigation model (for instance, adding all details on the internal page structure) to obtain the full navigation model for the web application.

Clearly, such automatic generation offers three main benefits. Firstly, designers save time and effort by not defining the navigation model from scratch. Secondly, it minimizes costly design errors since designers depart from an already complete and correct model. Finally, the generated model may present several alternative navigation designs that go beyond the one/s the designer had in mind and that may be worth exploiting in the final navigation model.

The construction of this initial model is split in several phases:

1. Generation of basic pages and links to ensure the completeness of the navigation model
2. Combining basic pages to create correct navigation paths
3. Refactorings to improve the structure of the generated navigation model

As a result of the process, we obtain a graph G_N representing the new navigation model. Then, this graph can be translated into the actual navigation model by means of reversing the rules introduced in section 3.1.

Each of the steps is described as follows.

4.1 Generation of a Complete Navigation Model

To be complete, the navigation model must contain all necessary data manipulation operations to modify the underlying web application data. Thus, for each required operation *op,* the graph G_N must include at least an arc *a* verifying that $op \in label(a)$.

Additionally, for each created arc *a* we define a new vertex v_1 in the graph representing the page where the user can select/enter all the parameters required by *op*. The link represented by *a* is anchored in that page, that is, v_1 is the origin vertex for *a*. At this stage of the process, we could use as a target page any other page of the model. The easiest option is to assume that either the target page is the same page (so that the user can apply again the same operation) or the home page.

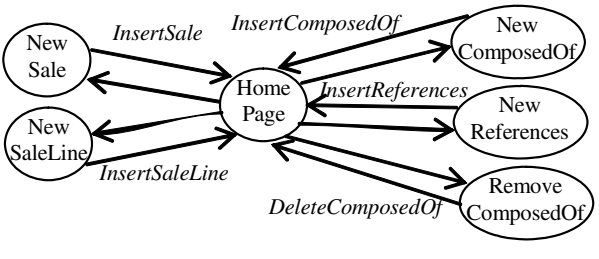

Fig. 6. Graph corresponding to a complete navigation model

Assuming that the only operations that can be executed over the data model of Fig. 2 are: *InsertSale*, *InsertSaleLine*, *InsertComposedOf*, *InsertReferences* and *DeleteComposedOf*, a complete navigation model could be the one corresponding to the graph shown in Fig. 6 From the home page, the user can access five different pages. Each page permits to enter the necessary information to execute the operation attached to the single exit link in the page. The exit link leads to the home page again.

4.2 Generation of a Complete and Correct Navigation Model

The previous graph corresponds to a complete navigation model but not a correct one since, for instance, the user can insert a new sale without inserting the corresponding sale lines thus leaving the data in an inconsistent state. In this section we extend this initial graph to ensure its correctness. Due to space limitations, we focus on the correctness of the subgraph including the home page and the new sale page.

As seen in section 3.3, correctness depends on the dependencies among the operations contained in the navigation paths of the graph. For instance, the occurrence of an *InsertSale* operation in a navigation path *nav* requires that, to be correct, *nav* includes an *InsertComposedOf* operation as well.

Therefore, a first step to ensure the correctness of the graph of Fig. 6 is to extend the graph by adding a new vertex and a new arc with the *InsertComposedOf* operation after the *NewSale* vertex (see Fig. 7). Now after inserting a new sale the user is redirected to a page to create a *ComposedOf* link.

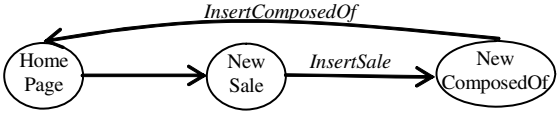

Fig. 7. Graph corresponding to a partially correct navigation model

However, this navigation path is not yet correct. The addition of the *InsertComposedOf* operation forces us to consider also the dependencies of this new operation. *InsertComposedOf* requires a previous operation *InsertSaleLine* or, alternatively, the existence of a *DeleteComposedOf* operation. None of them already appear in the path so we need to add one of them as well. Since we have two different alternatives to satisfy the correctness (either to add a new vertex and link for the *InsertSaleLine* or for the *DeleteComposedOf* operation), we duplicate the path created up to now. Each path takes one of the possible options so that we can cover all possible scenarios. After, we continue the process with each path separately.

In the path with the *InsertSaleLine* operation we need to further add a new *InsertReferences* operation. Dependencies for this latter operation are already satisfied by the path so we can stop the process for this path. The path with the *DeleteComposedOf* operation does not need new extensions (dependencies for *DeleteComposedOf* are satisfied if earlier in the path we find a *DeleteSale* or a *InsertComposedBy* as it is the case).

Fig. 8 shows the final aspect for this part of the graph. From the home page, we can insert a new sale in two different ways. We can either insert the sale and the related sale line or, alternatively, we can insert the sale and assign an existing sale line (previously related to a different sale) to the sale. The designer may decide if this second alternative makes sense in this particular domain. The decision cannot be automated.

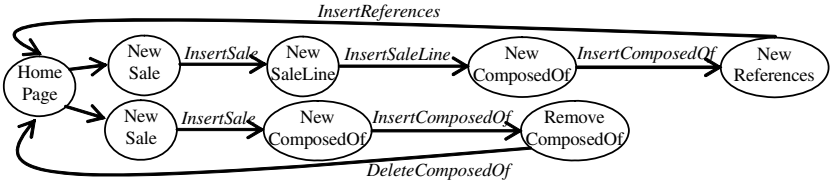

Fig. 8. Graph corresponding to a correct and complete navigation model

The generation process may add some extra arcs to create cycles in the graph. Cycles are created when the multiplicities of relationship types modified in the path are greater than one. In the previous example the graph should offer the option of adding several sale lines for the same sale as permitted by the multiplicities of *ComposedOf*.

More formally, the generation of a correct navigation model can be summarized in the following steps:

1. Compute the function *AllPaths* (see section 3.3.1) for the complete graph obtained in the previous section. *AllPaths* in this case just needs to consider the arcs between the home page and each specific page created for the required operations.
2. For each path compute the *transitive closure* of the dependencies of the operations in the path. The *transitive closure* can be computed by means of recursively applying the dependency rules of section 3.3.2 over the new operations added to the path until no more operations are added. When finding alternative dependencies (none of them already satisfied in the path) we create an additional path for each alternative.
3. Extending the graph with the new vertices and arcs that are necessary to satisfy all dependencies appearing in the transitive closure.

4.3 Refactorings for Navigation Models

Refactorings were initially proposed at the code level [8] as a disciplined technique for improving the structure of existing code (using simple transformations) without changing the external observable behavior. More recently, some work has been carried out to apply this technique on software models instead of on source code [12].

In this section we propose three simple refactorings to automatically improve the structure of the navigation model obtained at the end of the previous step. In our case, "without changing the external observable behavior" means without turning the model into an incomplete or incorrect model, that is, refactorings are transformations of the graph G_N representing a navigation model N that keep the completeness and correctness properties of N.

Refactoring 1. *Removal of redundant navigation paths*. Given two navigation paths n_1 and n_2, we say that paths n_1 and n_2 are equivalent when $SeqOp_{n1} = SeqOp_{n2}$, where $SeqOp_{ni}$ represents the ordered sequence of operations associated to the arcs of n_i. If two navigation paths are equivalent they are also redundant since the removal of one of them does not affect the completeness or the correctness of the navigation model. This refactoring removes all redundant navigation paths. The removal of a path consists in the removal of all arcs and nodes in the path not appearing in any other (non-redundant) path.

Refactoring 2. *Head-Merging of navigation paths*. Given two navigation paths $n_1=\{<v_1,a_1,v_2>,<v_2,a_2,v_3>,\ldots,<v_{x-1},a_{x-1},v_x>\}$ and $n_2=\{<v'_1,a'_1,v'_2>, <v'_2,a'_2,v'_3>, \ldots,<v'_{y-1},a'_{y-1},v'_y>\}$ we can merge the beginning of the two paths when there is an interval *1..i* where for all *k*, $1<=k<=i$, $label(a_k)=label(a'_k)$ (that is when the first part of the path coincides in n_1 and n_2). After applying the refactoring, n_2 becomes $n_2=\{<v_1,a_1,v_2>, <v_2,a_2,v_3>, \ldots, <v_{i-1},a_{i-1},v_i>,<v_i,a'_i,v'_{i+1}>, \ldots,<v'_{y-1},a'_{y-1},v'_y>\}$

Refactoring 3. *Tail-Merging of navigation paths*. Given two navigation paths $n_1=\{<v_1,a_1,v_2>, <v_2,a_2,v_3>,\ldots,<v_{x-1},a_{x-1},v_x>\}$ and $n_2=\{<v'_1,a'_1,v'_2>, <v'_2,a'_2,v'_3>, \ldots,<v'_{y-1},a'_{y-1},v'_y>\}$ we can merge the end of the two paths when there is an interval 1..*i* where for all *k*, $1<=k<=i$, $label(a_{x-k})=label(a'_{y-k})$ (that is when the last part of the path coincides in n_1 and n_2). After applying the refactoring, n_2 becomes $n_2=\{<v'_1,a'_1,v'_2>, <v'_2,a'_2,v'_3>,\ldots, <v'_{y-i},a_{y-i},v_{x-i+1}>,\ldots,<v_{x-1},a_{x-1},v_x>\}$

The application of the second refactoring over the graph of Fig. 8 results in the new graph shown in Fig. 9.

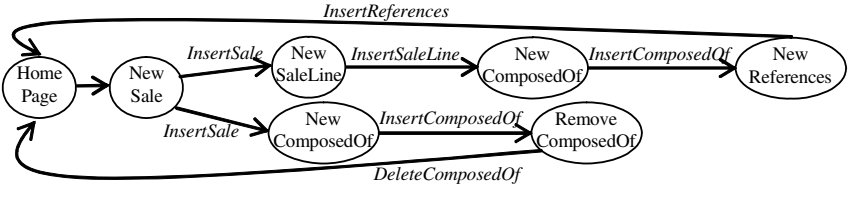

Fig. 9. Graph after applying the second refactoring

Apart from these automatic refactorings we could also provide several manual refactorings (i.e. refactorings that the designer must decide where and when to apply them). For instance, we could merge the nodes *NewSaleLine*, *NewComposedOf* and *NewReferences* in a single page with an outgoing link including the (ordered) sequence of operations included in the outgoing links for the three pages.

5 Related Work

Two kinds of related research are relevant to this paper: methods proposing properties to determine the quality of navigation models and methods devoted to the automatic generation of navigation models from data models.

Regarding quality aspects of navigation models, current proposals are rather limited. CASE tools provide limited verification facilities, mainly purely syntactic analysis of the correctness of the models regarding the syntax and semantics of the modelling languages. Other common supported properties include basic verifications of the navigation structure, as the reachability of all pages from the home page [4], [5]. [11] accepts the definition of a *route* for each conceptual user service. A route represents the sequence of steps that the user must follow to complete the service. Then, it checks that the structure of the navigation model is consistent with the defined routes. A few powerful formal verifiers also exist (see [6] as an example) though they are not yet fully integrated with current web development methods, which hamper their practical usability. Moreover, none of these proposals consider the specificities of navigation models with content-modification operations or the relationship between navigation models and their related data models as the quality properties we have defined in this paper.

With respect to the automatic generation of navigation models, existing approaches (as [1], [7] and [2]) derive the structure of the navigation model based on the relationships among the entity types in the data model (outside the web community, we find similar proposals, devoted to the automatic generation of the application graphical-user interface, see [14] as an example). Nevertheless, the generated models are just read-only navigation models for browsing the data; they do not include data modification functionalities.

Other important kinds of quality properties, as usability and accessibility of web applications [16] are outside the scope of this paper.

6 Conclusions and Further Work

We have presented two new quality properties (completeness and correctness of navigation models) that focus on the relationship between navigation and data models defined for the same web application. With these properties we can check whether a navigation model conforms to its data model so that inconsistencies between them can be detected in the early stages of the web application development process.

Our properties complement current quality checks for navigation models, which do not consider the data modification operations that may appear in those models. We believe that our properties are relevant to current web development methods addressing the definition and generation of fully-fledged web applications.

Also, we have shown how these properties can be used to automatically generate a preliminary version of a navigation model once the data model has been specified. This initial navigation model can then be refined by the designer in order to obtain the final navigation model for the web application.

Regarding further work, we would like to extend our quality assessment by detecting not only errors in the navigation model but also other kinds of problematic situations (“warnings”) that should be revised and by exploring the applicability of our graph-based representation to check additional properties (some properties may be reduced to path problems over our graph [15]) Besides, we also plan to improve our

current generation method for navigation models to include additional patterns and refactorings that make the model closer to the final one expected by the designer. Finally, we plan to validate our approach using an industrial case study.

Acknowledgments

We would like to thank the people of the GMC group and the anonymous reviewers for their many useful comments in the preparation of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnología under the project TIN2005-06053 and the integrated action HI2006-0208.

References

1. Albert, M., Pelechano, V., Fons, J., Rojas, G., Pastor, O.: Extracting Knowledge from Association Relationships to Build Navigational Models. LA-WEB'03, pp. 2–10 (2003)
2. Assossou, D., Wack, M.: Transformation Rules from Conceptual Model to Navigational Model in Hypermedia Applications. WEBIST'06 (1) pp. 428-434 (2006)
3. Bollobás, B.: Modern Graph Theory, p. 394. Springer-Verlag, Heidelberg (1998)
4. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33(1-6), 137–157 (2000)
5. Comai, S., Matera, M., Maurino, A.: A Model and an XSL Framework for Analyzing the Quality of WebML Conceptual Schemas. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 339–350. Springer, Heidelberg (2002)
6. Deutsch, A., Marcus, M., Sui, L., Vianu, V., Zhou, D.: A Verifier for Interactive, Data-driven Web Applications, SIGMOD'05, pp. 539–550 (2005)
7. Falquet, G., Guyot, J., Nerima, L., Park, S.: Design and analysis of active hypertext views on databases, Information Modeling for Internet Applications, pp. 40–58. Idea Group Publishing (2003)
8. Fowler, M.: Refactoring: Improving the design of existing code, p. 464. Addison-Wesley, London, UK (1998)
9. Hall, A., Chapman, R.: Correctness by construction. IEEE Software 19(1), 18–25 (2002)
10. Jakob, M., Schwarz, H., Kaiser, F., Mitschang, B.: Modeling and Generating Application Logic for Data-Intensive Web Applications, ICWE'06, pp. 77–84 (2006)
11. Lucas, F.J., Molina, F., Toval, A., de Castro, M.V., Cáceres, P., Marcos, E.: Precise WIS Development. ICWE'06, pp. 71–76 (2006)
12. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Trans. Software Eng. 30(2), 126–139 (2004)
13. Pastor, O., Fons, J., Pelechano, V., Abrahao, S.: Conceptual Modelling of Web Applications: The OOWS approach. In: Web Engineering, pp. 277–302. Springer-Verlag, Heidelberg (2006)
14. Pizano, A., Shirota, Y., Iizawa, A.: Automatic Generation of Graphical User Interfaces for Interactive Database Applications. CIKM'93, pp. 344–355 (1993)
15. Tarjan, R.E.: Fast algorithms for solving path problems. Journal of the ACM 28(3), 594–614 (1981)
16. Vanderdonckt, J., Beirekdar, A.: Automated Web Evaluation by Guideline Review, Journal of Web Engineering 4(2), 102–117 (2005)