Information and Software Technology 51 (2009) 749–768

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Introducing requirements traceability support in model-driven development of web applications

Pedro Valderas *, Vicente Pelechano

Department of Information Systems and Computation, Technical University of Valencia, 46022 Cami de Vera s/n, Valencia, Spain

ARTICLE INFO

Article history:
Received 26 March 2008
Received in revised form 29 July 2008
Accepted 7 September 2008
Available online 8 November 2008

Keywords:
Requirements traceability
Requirements engineering
Web engineering
Web applications
Model-to-model transformations
Model-driven development

ABSTRACT

In this work, we present an approach that introduces requirements traceability capabilities in the context of model-driven development of Web applications. This aspect allows us to define model-to-model transformations that not only provide a software artifact of lower abstraction (as model-to-model transformations usually do) but also to provide feedback about how they are applied. This feedback helps us to validate whether transformations are correctly applied. In particular, we present a model-to-model transformation that allows us to obtain navigational models of the Web engineering method OOWS from a requirements model. This transformation is defined as a set of mappings between these two models that have been implemented by means of graph transformations. The use of graph transformations allows us to develop a tool-supported strategy for applying mappings automatically. In addition, mechanisms for tracing requirements are also included in the definition of graph transformations. These mechanisms allow us to link each conceptual element to the requirements from which it is derived. In particular, we focus on tracing requirements throughout the navigational model, which describe the navigational structure of a Web application. To take advantage of these traceability mechanisms, we have developed a tool that obtains traceability reports after applying transformations. These reports help us to study aspects such as whether or not requirements are all supported, the impact of changing a requirement, or how requirements are modelled.

1. Introduction

Model-driven development (MDD) [1] promotes the use of models for developing software systems. MDD processes usually start with a requirements model that describes the user needs that the system must satisfy in a computation-independent way. Then, this model is refined into one or more conceptual models that describe the system without considering technological aspects. Finally, these models either (1) are refined into other models that describe the system by using concepts of a specific technology and are then translated into code or (2) are directly derived to code if they contain enough information.

This refinement of requirements into code is done by the application of several model-to-model and model-to-code transformations [2]. These transformations provide a model-based solution to a classical problem that the software engineering community has been trying to solve from its beginning: how to go from the problem space (user requirements) to the solution space (design and implementation) with a sound methodological guide. These model transformations clearly establish how requirements are transformed into conceptual models and how these conceptual models are transformed into code. However, they are usually defined as result-oriented transformations (their main goal is to provide us with a specific result: a software artefact of lower abstraction) and they pay little attention to providing us with feedback about how the transformation is applied to obtain this result. For instance, once a conceptual model is derived from a requirements model, usually no information is provided to indicate the requirements from which each conceptual element has been derived or the transformation mechanisms that have been applied to derive the conceptual elements. This aspect makes it difficult to validate whether transformations are properly applied or whether the different elements derived throughout the transformation process correctly satisfy the specified requirements.

We think that this aspect can be improved by introducing requirements traceability capabilities into the model transformations. Requirements traceability is the notion of establishing a relation between each unique requirement and the work products derived from it [18]. The problem that we face in this work is how to define model-to-model transformations that allow us to derive conceptual models from requirements models supporting requirements traceability. In particular, we face this problem in the context of Web application development where little support is provided for this aspect. Although there are several works such

* Corresponding author.
E-mail addresses: pvalderas@dsic.upv.es (P. Valderas), pele@dsic.upv.es (V. Pelechano).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.09.008

as [3–6] that provide traceability support in the transformation of requirements models into conceptual models, they are not enough from the perspective of Web application development where an aspect that has been poorly considered in traditional software has been extremely encouraged: Navigation.

Many MDD methods have been presented in the last decade for supporting the development of Web applications. They propose different navigational models to capture Navigation at the conceptual level [7–11]. However, few efforts have been made to properly specify Navigation at the requirements level (navigational requirements [32]) and to provide model-to-model transformations that transform these requirements into the proper navigational models. There are a few examples such as [12] and [13] where requirements models for Web applications are proposed together with mechanisms to translate navigational requirements into navigational models. However, there is a lack of tools that support analysts in the application of these mechanisms, and their application is dependent upon the experience and skills of analysts. This aspect makes the tracing of requirements a very complex activity. Our own work in [15] also faces the problems of specifying Web application requirements and deriving navigational models from requirements models. However, support for requirements traceability is not provided. We will give a more detailed overview of both our previous works and the related literature in Sections 2 and 6, respectively.

In this work, we introduce a model-to-model transformation that allows us to derive navigational models from Web application requirements models in a traceable way. This transformation is based on a set of mappings defined between the abstractions of our task-based requirements model presented in [15] and the abstractions of the navigational model proposed by the Web Engineering method OOWS [9]. We have improved our requirements model by introducing mechanisms that allow us to decouple requirements of different types (e.g. to decouple data requirements from navigational requirements), which facilitates the tracing of navigational requirements in later steps. In the definition of the mappings, we have analysed the impact that their application has on the navigational model according to the traceability classification proposed in [4], which was defined to consider traceability aspects in MDD environments. This classification and other definitions about requirements traceability will be explained in detail in Section 4.

Furthermore, the proposed mappings are implemented by a technique based on graph transformations [16]. Graph transformations can be automatically applied by using open-source tools such as AGG [17]. Thus, we provide a strategy that is based on this tool in order to automatically apply the model-to-model transformation. Additionally, we complement the graph transformations with traceability mechanisms that allow us to trace requirements through the automatically derived navigational model. To facilitate this, the tool *TaskTracer* has been developed. This tool analyses the derived navigational models and creates a traceability report that allows analysts to study how requirements are supported by conceptual elements. This information can be used to study aspects such as whether or not requirements are all supported in the navigational model, the impact of changing a requirement, or how requirements are modelled. This information can be used to validate the correctness and completeness of the model-to-model transformation.

The process of automatic derivation of navigational models from requirements models as well as the generation of traceability reports is illustrated in Fig. 1.

To sum up, the contributions of this work are the following:

- We present an improved version of the task-based requirements model for Web applications introduced in [15]. This model allows us to properly capture navigational requirements. Additionally, we have defined new mechanisms that facilitate the tracing of these requirements.

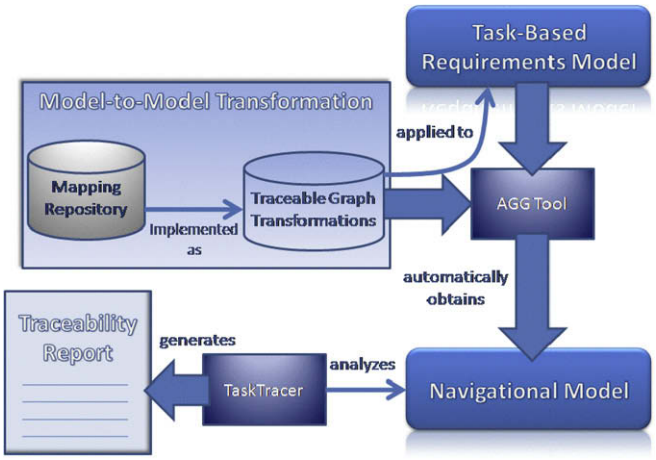


Fig. 1. Automatic derivation of Web navigational models with traceability support.

- We present a model-to-model transformation that allows us to automatically obtain OOWS navigational models from the task-based requirements model in a traceable way. This transformation is based on a set of mappings between two models that have been implemented by means of graph transformations.
- We present a tool that analyses the navigational models obtained after applying the model-to-model transformation and provides us with a traceability report. This report constitutes a valuable tool for taking pragmatic advantage of traceability information during the development of a Web application.

The rest of this paper is organized as follows: Section 2 gives a detailed account of the background required to properly understand this work. In particular, we present an overview of the OOWS navigational model and a study of the concept of traceability. Section 3 introduces our previous work about requirements engineering for Web applications. This section also includes the improved version of our task-based requirements model for Web applications. Section 4 introduces the set of mappings between task-based requirements models and OOWS navigational models. Section 5 introduces a technique based on graph transformations that allows us to automatically apply these mappings taking into account traceability aspects. Section 6 introduces both the TaskTracer tool and the traceability reports that this tool generates. We also study how these reports can help us in the development of Web applications. Section 7 analyses the related work. Finally, conclusions and further work are presented in Section 8.

2. Background

In this section, we introduce the background that is needed to better understand the new ideas introduced in this paper. In particular, we introduce:

1. An overview of the navigational model proposed by the Web engineering method OOWS.
2. The concept of traceability and the different types of traceability that have been presented along the published literature. We also determine the type of traceability that is supported by our work.

2.1. The OOWS navigational model: an overview

OOWS [9] is a Web engineering method that provides methodological support for Web application development. OOWS is the extension of an object-oriented software production method called

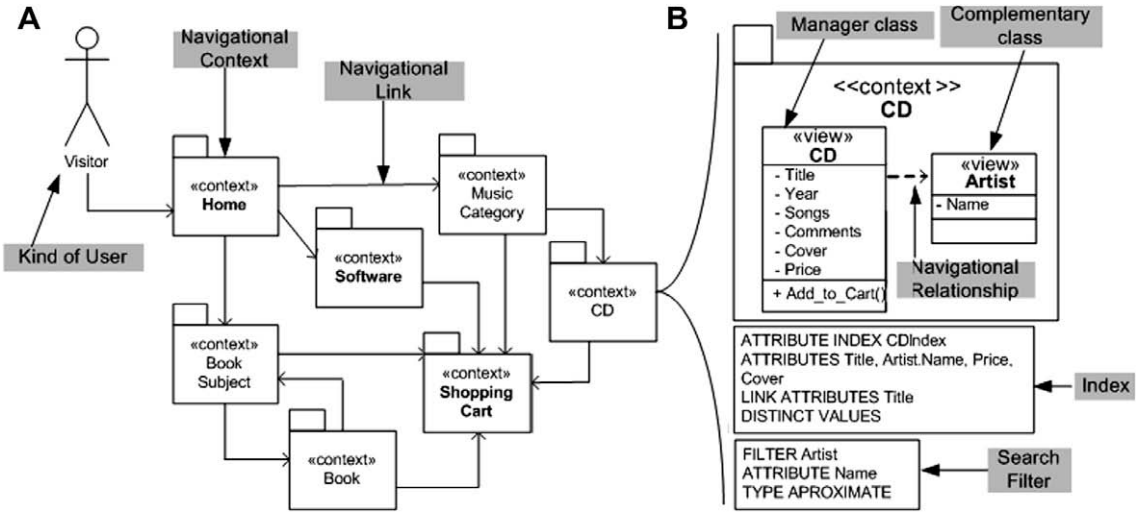


Fig. 2. The OOWS navigational model.

OO-Method [25]. OO-Method proposes several models to describe the different aspects of a system: the *class diagram* (to describe the static structure) and the *dynamic* and *functional* models (to describe the system behaviour). OOWS introduces the *navigational model* to describe the navigational aspect of a web application. Next, we briefly introduce this model.

The OOWS navigational model is made up of a set of navigational maps that describe the navigation that is allowed for each kind of user. A navigational map is represented by a directed graph (which defines the navigational structure) whose nodes are navigational contexts and whose arcs denote navigational links. Fig. 2A shows the visitor navigational map of the running example. A navigational context (represented by an icon[1] of a package stereotyped with the "context" keyword) defines a view over the class diagram that allows us to specify the information that is shown in the context (class attributes) and the operations that the user can activate (class operations). A navigational link represents navigational context reachability: the user can access a navigational context from a different one if a navigational link between the two contexts has been defined.

A navigational context is made up of a set of *navigational classes* that represent class views over the classes of the class diagram (including attributes and operations). These classes are stereotyped with the *view* keyword. Each navigational context has one mandatory navigational class, called *manager class*, and optional navigational classes to provide complementary information for the manager class, called *complementary classes*. All navigational classes must be related by unidirectional binary relationships, called navigational relationships, which are defined upon an existent relationship in the class diagram. Fig. 2B shows the navigational context CD that is made up of the manager navigational class (CD) and a complementary navigational class (Artist). These classes are related by means of a navigational relationship. This navigational context provides the user with the CD's title, the artist's name, the recording year, the songs, the front cover, the price, and some comments about the CD (navigational classes attributes). In addition, the user can invoke the Add_to_Cart functionality to add the CD to the shopping cart (manager class operation).

Furthermore, for each context, we can also define access mechanisms of two types: (1) Search filters, which allow us to filter the space of objects that retrieve the navigational context. For instance, the CD navigational context allows the user to find all the CDs of a specific artist (see search filter in Fig. 2B); and (2) Indexes, which are structures that provide an indexed access to the population of objects. Indexes create a list of summarized information allowing the user to choose one item (instance) from the list. This selection causes this instance to become active in the navigational context. For instance, the navigational context in Fig. 2B provides the user with a list of summarized information where the title, the artist's name, the price, and the cover are shown for each CD. If users select a CD from this list, they obtain the information defined in the context view (navigational classes) for the selected CD.

2.1.1. Implementation issues

In order to better understand the OOWS navigational model, we show an implementation of the context CD. This context is implemented by means of two Web pages which are shown in Fig. 3. The Page A provides the lists of CDs defined in the index (see Fig. 2B). For each CD, the title, the artist's name, the price and the front cover are shown. If users select a CD from this page, they access Page B. This page provides the user with the information defined in the navigational context view (see Fig. 2B). This information is the CD title, the year, the songs, some comments, the front cover, the price, and the artist's name. This page also allows users to add the CD to the shopping cart (according to the operation defined in the manager class, see Fig. 2B). Finally, notice both how a search engine is available in both pages (according to the defined search filter, see Fig. 2B) and how the menu at the left side allows users to access the shopping cart (according to the navigational link defined in the context CD, see Fig. 2A).

2.2. Requirements traceability

According to Gotel [18], *Requirements Traceability* refers to the ability to describe and follow the life of a requirement, in both a forward and backward direction. Forward traceability looks at both tracing the requirements source to the resulting requirements and tracing the resulting requirements to the work products that implement them. Backward traceability looks at both tracing each work product back to its associated requirements and tracing each requirement back to its source

[1] Note that the OOWS navigational model uses a concrete syntax that is similar to some UML diagrams [19]. However, it is not defined as a UML profile. The OOWS navigational model has been defined as a new modeling language by describing its meta-model using MOF. This means that, although its graphical notation is similar to UML diagrams, its semantics is different.

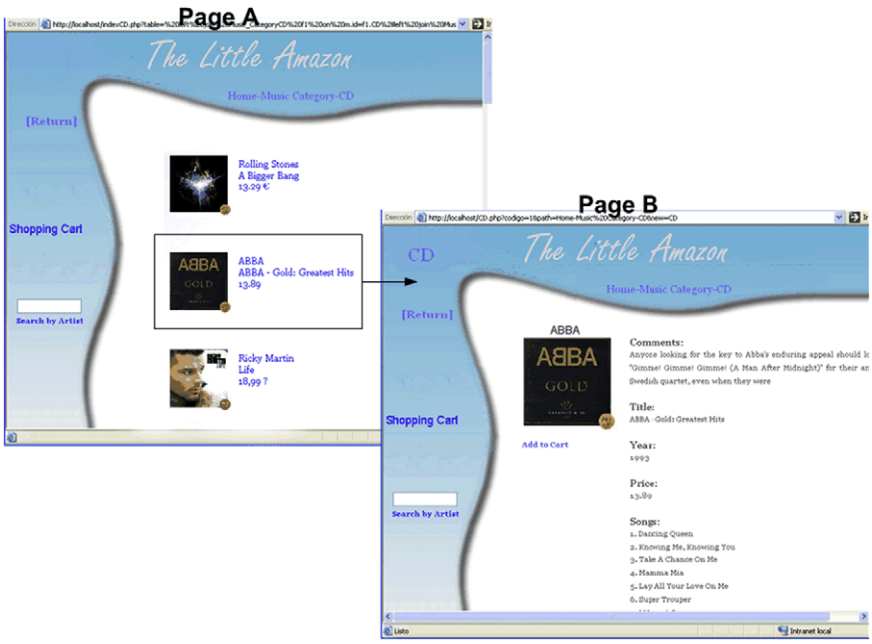


Fig. 3. Example of implementation of the context CD.

Requirements traceability allows us to:

- Validate whether or not requirements are all supported and whether the implementation is compliant with the requirements.
- Understand the user need that is addressed by each requirement.
- Verify the necessity of each requirement and how it is implemented.
- Check how each requirement has been interpreted by programmers.
- Establish the impact of changing a requirement on software artefacts.

Two different types of traceability are distinguished according to the classification proposed by Gotel:

- *Pre-requirements specification (pre-RS) traceability* is concerned with those aspects of a requirement's life prior to its inclusion in the RS (requirement production). This type of traceability is used to track the relationship between each requirement and its source. For example, a requirement might be traced from a business need, a user request, a business rule, an external interface specification, an industry standard or regulation, or to some other source.
- *Post-requirements specification (post-RS) traceability* is concerned with those aspects of a requirement's life that result from its inclusion in the RS (requirement deployment). This type of traceability is used to track the relationship between each requirement and the work products to which that requirement is allocated. For example, a requirement might trace to one or more architectural elements, detail design elements, object/classes, code units, tests, user documentation topics, and/or even to people or manual processes that implement that requirement.

In this work, we focus on *Post-RS traceability*. We introduce a technique that allows us to derive Web application navigational models from task-based requirements models in a traceable way. In this context, we know that two or more conceptual models can give support to the same requirements specification [50]. Thus, one requirement might be derived into more than one conceptual element, and all of them correctly support the requirement. However, there might be other requirements that can only be derived into a unique conceptual element in order to be correctly supported. In order to consider this type of relationship between requirements and conceptual elements, another traceability classification is proposed in [4]:

- *Weak traceability*, which is concerned with those aspects of the requirements model that will be translated to elements in the conceptual model that can be changed or even deleted on destination. This means that aspects with weak traceability will only be *proposed elements* in the conceptual model.
- *Strong traceability*, which is concerned with those aspects of the requirements model that will be translated to elements into the conceptual model that cannot be changed or deleted on destination. This means that aspects with strong traceability will be *mandatory elements* in the conceptual model.

3. Requirements engineering for web applications

Our previous work is centered on providing requirements engineering support in the context of model-driven development of Web applications. We initially faced the problem of Web application requirements specification in [15] where an initial version of a task-based requirements model is introduced. Next, the derivation of navigational models from this requirements model is handled in [33] where a derivation strategy based on MDA is proposed. In [34], we introduce a complete methodological view of our work and introduce a technique to generate Web application prototypes from the task-based requirements model. However, none of these works confront the problem of requirements traceability.

In this work, we extend our requirements engineering approach by introducing support for tracing Web application requirements throughout navigational models. We redefine the strategy used for deriving navigational models by including traceability mechanisms that allow us to know how navigational model elements

are derived from requirements. In addition, we present a new tool that analyses these new mechanisms and provides analysts with traceability reports. These extensions are presented in detail in Sections 3–5.

In order to define the proposed extensions, we must improve our task-based requirements model in order to properly support traceability aspects. In addition, it is crucial to know how requirements are captured in this model in order to properly understand the current work. Thus, we present a detailed description of the improved version of this model below.

3.1. A task-based requirements model for web applications

In this section, we present a requirements model for specifying Web Applications requirements. This model takes into account the navigational aspect of Web applications.

Navigation is encouraged in Web applications because the focus of Web applications is extensively based on communication [46]. Since the initial goal of the WWW is based on its role as an information medium, many Web applications are fully or partially developed as magazines or brochures, and their development mainly involves capturing and organizing a complex information domain and making that domain accessible to users. The information is organized in a structure made up of nodes, links, and anchors (according to the hypertext paradigm [47]), which allows users to navigate the information in a non-linear way.

To properly capture these navigation capabilities at the requirement level, we use the concept of task together with aspects related to the system-user interaction. We have chosen the concept of task because temporal relationships can be defined between tasks. This aspect provides us with valuable information that can be used to obtain a general overview of how users must navigate information. For instance, we can indicate that a task T2 must be performed after a task T1 finishes. Then, the final Web application must provide mechanisms that make users navigate first the information related to T1 and then the information related to T2.

In the specification of software requirements, the performance of tasks is traditionally described from the set of actions that both the system and the user perform during the task [48]. These descriptions are very suitable to capture functional requirements. However, when they are used to capture Web application requirements they fail in the specification of navigational requirements. This is why we complement tasks with interaction aspects. Due to the encouragement of Navigation, the way in which users interact with Web applications substantially changes from the way in which users interact with traditional software. In the case of traditional software, users interact with the system to activate some functionality in order to obtain specific results. In the case of Web applications, users can also interact with the system in order to just browse information. We think that by providing a mechanism to capture this type of interaction at the requirements level (how users browse information) we are providing a mechanism to properly capture the navigational requirements of Web applications.

In this context, the requirements model that we present in this work is defined from the following elements:

(1) A task taxonomy, which provides a general view of the different tasks that users need to perform by interacting with the Web Application. This taxonomy represents tasks in a hierarchical way and allows us to define temporal relationships.
(2) A set of task performance descriptions, which indicates how the different tasks must be performed by describing the interaction that users require from the system in order to perform them.
(3) Two types of templates that allow us to specify requirements related to the information that the system must handled. On the one hand, *information templates* allow us to specify the data that the system must store. On the other hand, *exchanged data templates* allow us to associate this data with the different system–user interactions defined in the task performance descriptions.

3.1.1. Defining the task taxonomy: a general view of the user's tasks

We propose to identify and represents the tasks that users must be able to perform by following a hierarchical task analysis process [20]. This task analysis is performed by taking general tasks and refining them into more specific ones. The refinements are presented in a hierarchical way, which creates a task taxonomy.

In order to perform the hierarchical task analysis, we take a Statement of Purpose (which describes the goal for which the application is being built) as the starting point. For instance, the Statement of Purpose for an E-commerce application[2] that allows users to purchase different types of products can be the following:

> The Web application under development is an E-commerce Application. The main goal of the system is to provide support for the on-line purchase of products. To achieve this, the user must be able to consult information about products, add them to a shopping cart and send purchase orders. Furthermore, tools for the information management must be provided.

Next, we consider the Statement of Purpose to be the most general task that users can perform by interacting with the Web application. Then, a progressive refinement is performed from this task to obtain more specific tasks. Tasks are decomposed into subtasks by following structural or temporal refinements. The *structural refinement* (represented by solid lines, see Fig. 2) decomposes complex tasks into simpler subtasks. The *temporal refinement* (represented by dashed lines, see Fig. 2) provides constraints for ordering tasks that are all children of a single task according to the task logic. To define these temporal constraints, we propose using the temporal relationships introduced by the CTT approach (ConcurTaskTree) [21]. For reasons of brevity, we only present the three relationships that are used in the case study of this paper:

- T1 []≫ T2, *Enabling with information exchange*: when T1 is terminated, T2 is activated. In addition, when T1 terminates, besides activating T2, it provides a value for T2.
- T1 |> T2, *Suspend-Resume*: T1 can be interrupted by T2. When T2 terminates, T1 can be resumed.
- T1*, *iteration*: the task can be completed several times.

Tasks inherit the temporal relationships of the ancestors. For instance, in Fig. 3, *Add CD* is a sub-task of *Add Products to Shopping Cart*. Since *Add Products to Shopping Cart* can be suspended by *Inspect Shopping Cart*, this relationship will also apply to *Add CD*.

The task taxonomy is finished when the elementary tasks are obtained. An *elementary task* is a task in which atomic actions are obtained when it is decomposed. We consider atomic actions to be either single actions of the user or single actions of the system. A single action of the user can be, for instance, the selection of information or the activation of an operation. A single action of the system can be, for instance, the inquiry of the state of the system or the modification of it.

Fig. 4 shows the task taxonomy that we obtain for the running example. It is described as follows:

[2] This E-commerce application is taken as running example in the rest of this paper.

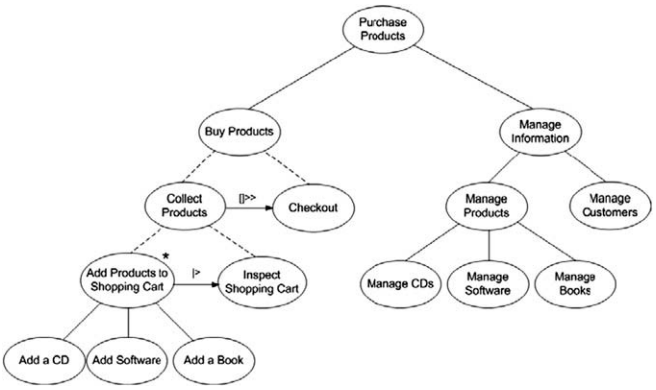


Fig. 4. The task taxonomy of the running example.

- The statement of purpose is decomposed by means of a structural refinement (solid line) into two tasks: (1) *Buy Products*, which involves the needs related to the process of buying products and (2) *Manage Information*, which involves the needs related to management activities.
- The task *Buy Products* is decomposed by means of a temporal refinement (dashed line) into *Collect Products* and *Checkout*. The relation between them is enabling with information exchange. Thus, the products must be collected into the shopping cart before checkout. The information that needs to be exchanged is the collected products.
- In order to collect products, users add them to the shopping cart. During this process, users can inspect the shopping cart when they consider it to be opportune. Thus, *Collect Products* is decomposed into *Add Product to Shopping Cart* and *Inspect Shopping Cart*. The relation between the two tasks is suspend-resume, which indicates that *Add Product to Shopping Cart* may be interrupted at any point by *Inspect Shopping Cart*. After each interruption, the task *Add Product to Shopping Cart* is resumed.
- *Add Product to Shopping Cart* is an iterative task. Thus, it can be performed as many times as the user needs. It is decomposed by means of a structural refinement into the tasks *Add CD*, *Add Software*, and *Add Book*. Therefore, when the user adds a product to the shopping cart, s/he is adding a CD, a software product or a book.
- The task *Manage Information* is decomposed by means of a structural refinement into Manage Products and Manage Customers. Users can manage information about either products or customers.
- Finally, the task *Manage Products* is decomposed by means of a structural refinement into *Manage CDs*, *Manage Software* and *Manage Books*. Users can manage information about CDs, software or books.

Once the task taxonomy is constructed, we describe how elementary tasks must be performed. We describe only elementary tasks because these tasks constitute the lowest level of the taxonomy. Then, in contrast to non-elementary tasks, which are described throughout their subtasks, there is no description in the taxonomy for elementary tasks. Next, we introduce a technique to perform these descriptions.

3.1.2. Task performance descriptions: describing elementary tasks in detail

In this section, we introduce a technique that allows us to describe elementary tasks from the interaction that users require from the Web application in order to perform each task. According to this technique, we describe elementary tasks as a set of user and system actions that is complemented with information about the system–user interaction by indicating explicitly when (at which exact moment) it is performed. To do this we introduce the concept of *interaction point* (IP).

Thus, the performance of an elementary task is considered to be a process where the system carries out several *system actions*, sometimes delaying them in order to interact with the *user* by means of *interaction points* (IP).

Two kinds of interactions can be performed in an IP:

1. *Output interaction*: the system provides the user with information and/or access to operations that are related to an entity.[3] The user can perform several actions with both the information and the operations: the user can select information (as a result, the system provides the user with new information), or the user can activate an operation (as a result the system carries out an action).
2. *Input interaction*: the system requests the user to introduce information of an entity. The system uses this information to correctly perform a specific action (for instance, the client information needed to carry out an on-line purchase). In this case, the only action that users can perform is the information input.

Two kinds of system actions are proposed:

1. *Functionality execution*, which are actions that change the system state.
2. *Information search*, which are actions that only query the system state.

In order to perform descriptions of this type, we propose the use of activity diagrams.[4] Each activity diagram has the following characteristics (see Fig. 3):

- Each node (activity) represents an IP (solid line) or a system action (dashed line). IPs are stereotyped with the *Output* or the *Input* keyword to indicate the interaction type. System actions are stereotyped with the *Function* or the *Search* keywords to indicate their types.
- For the Output IPs, the number of information instances[5] that the IP includes (cardinality) is depicted as a small circle in the top right side of the primitive.
- For the Input IPs, we consider that they exclusively depend on a system action (the user introduces information because the system needs it to correctly perform an action) and does not take part in the general process of the task. Then, nodes that represent both elements (input IP and system action) are encapsulated in dashed squares.
- Finally, each arc represents (1) a user action if the arc source is an IP or (2) a node sequence if the arc source is a system action. If an arc represents a user action (the arc source is an IP), it can be:
  - An activation of an operation: if the source is an Output IP and the arc target is a system action.
  - An information selection: if the source is an Output IP and the arc target is another Output IP. In this case, arcs can be doubly arrowed to indicate that after users finish studying

[3] Any object of the real world that belongs to the system domain (e.g. client, product, and invoice).

[4] We have used a concrete syntax that is similar to the UML activity diagram in order to facilitate its use to the software engineering community. However, our diagram is not defined as a UML profile but it is defined as a new modeling language by describing its meta-model using MOF. This means that, although its graphical notation is similar to the UML activity diagram, its semantics is different.

[5] Given a system entity (e.g. client), an information instance is considered to be the set of data related to each element of this entity (Name: Joseph; Surname: Elmer; Telephone Number: 9658789).

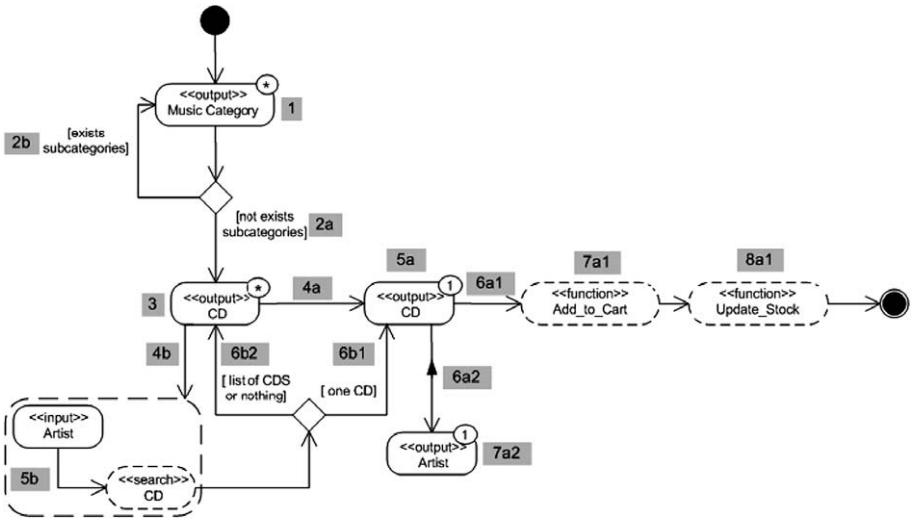


Fig. 5. Performance description of the elementary task *Add CD*.

the selected information (provided by the target IP), the user must return to the source IP in order to continue with the task performance. These situations represent those moments during the performance of a task in which users may access complementary information. It is not mandatory for users, however, to access this information in order to perform the task.

– An input of data: if the source is an Input IP and the arc target is system action.

Fig. 5 shows the task performance description associated to the elementary task Add CD (the shaded numbers are not part of the notation). According to this description:

- The elementary task Add CD starts with an Output IP where the system provides the user with a list (cardinality *) of music categories (1).
- From this list, the user can select a category (2a and 2b). If the category has subcategories, the system provides the user with a list of (sub) categories (2b).
- If the selected category does not have subcategories (2a), the system informs about the CDs of the selected category by means of an Output IP (3). The user can perform two actions from this IP:
  (A) Select a CD (4a), and then the system provides the user with a description of the selected CD (5a).
  (B) Activate a search action (4b), and then the system performs a system action that searches for the CDs of an artist (5b). To do this, the user must introduce the artist by means of an Input IP. If the search returns only one CD, the system provides the user with its detailed description (6b1). Otherwise, the system provides the user with a set of CDs (6b2).

- Finally, when the user has obtained a CD description (5a) s/he can:
  (A) Select the artist of the CD (6a2), and then the system provides the user with detailed information about the artist (7a2). In this case, it is mandatory for the user to return to the CD description in order to finish the task.
  (C) Activate the Add_to_Cart operation (6a1). When users activate this operation the system performs an action that adds the selected CD to the shopping cart (7a1). Next, the system updates the stock (8a1), and, finally, the task finishes.

IP-based descriptions allow us to indicate which type of information is exchanged between system and users (by means of the entity associated to each IP). However, details about this information are not described. These details are specified in later steps by means of an information template technique (which is introduced below). This allows us to provide a high level of independence among different kinds of requirements.

3.1.3. Information templates: describing the data requirements

Once user tasks have been identified and described, we propose to define the data that the system must store about each identified entity (e.g. CD, Artist, and Music Category). Furthermore, we also describe the data that is exchanged in each IP that is defined in the task performance descriptions. To do this, we propose a technique based on templates, which is inspired by techniques such as NDT [22], the CRC Card [23] or the approach presented in [24].

First, we propose to define an *information template* (see Fig. 6A) for each entity identified in the description of an elementary task performance. In each template, we indicate an identifier, the entity, and a specific data section. In this section, we describe the information in detail by means of a list of specific features associated to the entity. We provide a name, a description, and a nature for each feature. This nature expresses the specific type of the feature in an abstract way, without giving design details. It can be a simple nature, which is defined by predefined values such as string, number, text, etc., or it can be a complex, nature which is defined by another information template.

Identifier:	Id1		
Entity:	CD		
Specific Data:	*Name*	*Description*	*Nature*
	Title	Title of the CD	String
	Year	Recording year of the CD	Number
	Artist	Artist that has recorded the CD	Id02
	Songs	Songs list of the CD	List (String)
	Comments	A brief commentary of the CD	Text
	Front Cover	Front cover of the CD	Image
	Price	Price of the CD	Currency
	Purchase times	Times that the CD has been purchased	Number
	Client Profiles	Profiles of the clients that have purchased the CD	List (String)
	Stock	Quantity of units in stock	Number

Fig. 6. Information template associated to the entity CD.

Fig. 6A shows the information template associated to the entity CD (identified in the description of the elementary task Add CD, see Fig. 5). According to this template, the information that the system must store about a CD is (see the specific data section): the CD title, the recording year, the artist that has recorded the CD, the list of songs, some comments about the CD, the front cover, the price, the number of times that the CD has been bought, the profiles of the customers that usually purchase it, and the number of units in stock. Notice that all these features present a simple nature except for the artist of the CD. The nature of this feature is described by the information template whose identifier is Id02 (which is the identifier of the information template associated to the entity Artist).

Once the data that the system must store about each entity has been defined, we can use it to describe in detail the information that the system and the user exchange in each interaction point. We propose to associate the features defined for each entity (in the information templates introduced above) with the different IPs defined in the descriptions of elementary tasks. To do this, we use *exchanged data templates* (see Fig. 7).

Each exchanged data template is made up of the following fields:

- *Identifier*: the identifier of the template.
- *IP and elementary task*: These fields indicate the IP in which the data is exchanged. In order to textually identify an IP, we use the following notation: *Output* (*Entity*,*Cardinality*) for Output IPs, and *Input(Entity, System Action)* for Inputs IPs. We also indicate the elementary task in whose description the IP has been defined.
- *Retrieval condition*: This field can only be indicated for Output IPs. It allows us to define a condition that every entity instance provided in an Output IP must accomplish. This allows us to filter the instances that users access in an Output IP. This type of condition is defined by using the Object Constraint Language (OCL) [49]. Note that we only define the invariant of an OCL expression. The contextual element to which the OCL expression is defined is the entity associated to the Output IP for which the condition is applied. The declaration of this contextual element (the context of the OCL expression) is omitted to not overload the template definition. For instance, in Fig. 5, we are indicating that the IP Output(CD,*) only provides users with information about those CDs that are available in stock.
- *Exchanged data*: This field indicates the entity features that describe the exchanged data. In the case of Ouput IPs, these features constitute the data that is shown to users. In the case of Input IPs, these features constitute the data that is requested to users. For each feature, we indicate its name, the entity, and the identifier of the information template associated to the entity.

Fig. 7 shows the exchanged data template associated to the IP Output(CD,*), defined in the elementary task Add CD (see Fig. 3). According to this template, for each CD that is available in stock, this IP provides: the title of the CD, the price, the front cover, and the name of the artist that has recorded the CD.

Finally, it is worth noting that both the use of information templates for defining only data requirements and the use of exchanged data templates for associating data with user-system interactions are contributions of this work.

Identifier:	O2	
IP:	Output(CD,*)	
Elementary Task:	Add CD	
Retrieval Condition:	self.stock > 0	
Exchanged Data:	Entity and Template Id	Feature
	CD: id1	Title
	CD: id1	Price
	CD: id1	Front Cover
	Artist: id2	Name

Fig. 7. Exchanged data template associated to the IP Output(CD,*).

This contribution allows us to provide a technique in which mechanisms to describe the interaction between the user and the system (aspect which has been traditionally handled in the field of HCI [3] and which allow us to capture navigational requirements) are combined with mechanisms to capture traditional requirements such as functional requirements and data requirements (traditionally handled in the field of RE [57]). This combination however is performed with a high level of decoupling between requirements which allows us to describe requirements of different types separately. This aspect facilitates the definition of traceability links that relate navigational model elements with requirements that are exclusively related to Navigation.

4. From task descriptions to OOWS navigational models: a methodological guide

In this section, we introduce a methodological guide to support analysts in the construction of Web application navigational models from the requirements specified in a task-based requirements model. This guide proposes a set of mappings that associate each abstraction of the requirements model to the equivalent abstraction(s) of the OOWS navigational model. To do this, mappings are defined from the elements defined in the meta-model of each model. The meta-model of the task-based requirements model can be found in [53]. The meta-model of the OOWS navigational model can be found in [54].

These mappings are under continuous study and revision in accordance with the experience we acquire when applying them to different Web development projects. We present some representative examples of these mappings below. The whole set of mappings can be found in [26].

Each mapping is defined by means of the following structure:

- *RE Model*: element(s) defined in a task-based requirements model.
- *Maps to*: primitive(s) of the OOWS navigational model that correspond to the element(s) of the task-based requirements model identified above.
- *Except when*: situations (if there are any) in which this rule must not be applied.
- *Traceability type*: indicates if the traceability of the rule is strong or weak (see Section 4).

Mappings are grouped in three categories according to the conceptual elements that are derived. We propose mappings that derive:

- the navigational structure (contexts and links),
- the definition of each Navigational Context (navigational classes and navigational relationships),
- access mechanisms (indexes and search filters).

4.1. Mappings for deriving the navigational structure

The navigational structure of a Web application indicates how the information and functionality is organized in navigational contexts and links in order to be accessed by users. In this sense, the way in which users interact with the Web application to perform each task constitutes a valuable source of information for deriving the navigational structure.

For instance, we can derive the navigational contexts in which information must be organized by analyzing how users access

the information provided by the system during the performance of a task (Output IPs). An Output IP represents a step in the performance of an elementary task where the system provides the user with information. At the navigational model, we indicate the information that the Web application must provide to users by means of navigational contexts. Then we can derive navigational contexts from the Output IPs defined in the task performance descriptions. This derivation is supported by mappings such as the one in Fig. 8A.

The mapping in Fig. 8A has been defined with a strong traceability. This means that the set of identified navigational contexts is mandatory to correctly support the performance of each task. This mapping present an exception: it must not be applied if the Output IP provides information about multiple instances of an entity and also allows users to access another IP that provides information about only one instance of the same entity. These IPs represent a step in a task where the user compares elements of a list with each other (instances of the first IP) in order to select one and then obtain more detailed information about this instance (in the next Output IP). These situations are captured in the OOWS navigational model by means of the index primitive, which is handled by another mapping (further presented).

Fig. 8B shows the navigational contexts that are derived when the mapping is applied to the task Add CD (see Fig. 3). The navigational contexts are: Music Category, which must provide users with information about music categories; CD, which must provide users with information about CDs; and Artist, which must provide users with information about artists.

Navigational links can be derived by analyzing the sequence of Output IPs that users access when they are performing a task. Output IPs represent steps in a task where the system provides users with information. We know that these steps are supported at the conceptual level by means of navigational contexts (see previous mapping). Then, in order to preserve the sequence of Outputs IPs defined at the requirements level we must define the corresponding sequence of navigational contexts at the conceptual level. This sequence of navigational contexts is defined by connecting them throughout navigational links. Thus, we define a navigational link between two navigational contexts if the IPs from which the contexts have been derived are: (1) connected by means of an arc or (2) connected through an Output IP that has not derived into a context. This aspect is supported by mappings such as the ones in Fig. 9A. These mapping have a strong traceability. This means that users must be able to activate the derived navigational links in order to satisfy the task logic.

Fig. 9B shows the navigational links that are derived when the mappings are applied to the task Add CD (see Fig. 3). According to these links, users can access information about CDs from the context that provides information about music categories. Additionally, users can access information about artists from the context that provides information about CDs.

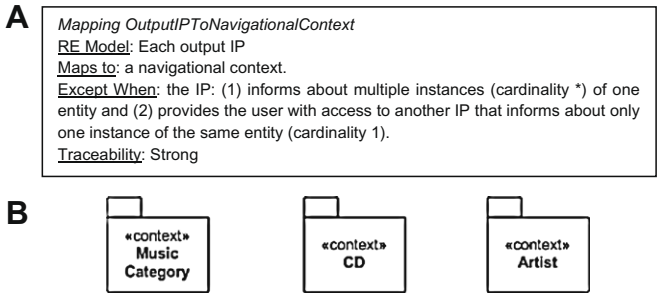


Fig. 8. Mapping between Output IPs and navigational contexts.

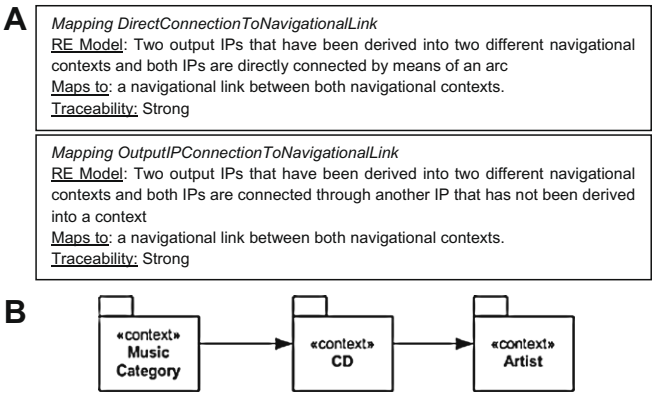


Fig. 9. Mapping between Output IP connections and navigational links.

Navigational links can also be derived from the different temporal relationships that are defined in the task taxonomy. For instance, if a suspend/resume relationship has been defined between two tasks, T1 and T2, it means that the user can interrupt T1 at any time to achieve T2. Then, in order to preserve this temporal constraint at the conceptual level, navigational contexts derived from T2 must be accessible from navigational contexts derived from T1. To obtain this, we must define navigational links among T1 navigational contexts and T2 navigational contexts. This aspect is supported by mappings such as the one in Fig. 10A. This mapping has a strong traceability. This means that the derived links are mandatory in order to allow users to perform tasks according to the temporal relationships defined in the task taxonomy.

Fig. 10B shows the navigational links that are derived from the suspend/resume relationship defined in the task taxonomy of the running example (see Fig. 2). Taking into account that tasks inherit the temporal relationships of their parent task, the Add CD elementary task is connected to the Inspect Shopping Cart elementary task by means of a Suspend/Resume relationship (inherited from the Add Products to Shopping Cart task). Thus, the navigational contexts derived from the task Add CD are linked to a navigational context derived from the task Inspect Shopping Cart.

4.2. Mappings for deriving the definition of each navigational context

Once navigational contexts have been derived we must derive the information that is included in each of them. This information is defined by the navigational views (made up of a manager

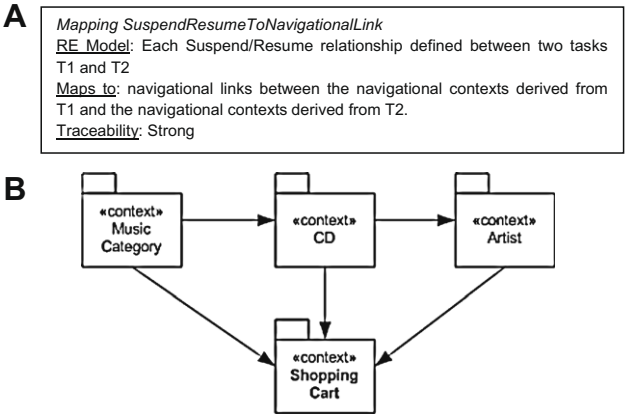


Fig. 10. Mapping between Suspend/Resume relationships and navigational links.

navigational class and a set of complementary navigational classes) that are attached to each navigational context. Taking into account that navigational contexts are derived from Output IPs, the manager class of each context can be directly derived from the entity associated to the IP (e.g. the manager class of the context CD, which provides information about CDs, is the class CD). However, we must also derive manager class attributes and operations as well as complementary classes.

On the one hand, the information that users and the system exchange in each Output IP (described in detail by information templates) can be used to derive manager class attributes and complementary classes. Exchanged data templates allow us to define the data that are provided to users in each Output IP. These data are defined by associating a set of entity features to the Output IPs. In the case that the Output has been derived into a navigational context, these features can be used to define manager class attributes and complementary classes.

On the other hand, operations are detected from the function system actions defined in the performance descriptions of elementary tasks. In particular, we analyse those function system actions that are connected to an Output IP. If an Output IP is connected to a function system action through an arc, the user can activate the system action after consulting the information provided by the Output IP. Then, if the Output IP has been derived to a navigational context, the navigational context must allow users to activate the system action. Then, each system action that can be activated from an Output IP that has been derived to a navigational context, defines an operation of the manager class of this navigational context.

All these derivations are supported by mappings such as the ones in Fig. 11A. The first mapping, which derives the manager class, has a strong traceability because it is mandatory for users to access information about this class. The second and third mappings, which derive manager class attributes and complementary classes, have a weak traceability because we consider that analysts could modify, delete or add either new attributes or complementary classes without critically altering the performance of a task. Forth mapping, which derives manager class operations, has a strong traceability because it is mandatory that navigational contexts provide users with access to the operations in order to properly perform the task.

Fig. 11B shows the navigational view of the context CD which is derived when the mappings are applied to the exchanged data template shown in Fig. 5 as well as to the performance description of the task Add CD (see Fig. 3). According to this view, when users access the context CD, they are provided with the title of the CD, the year, the list of songs, the front cover, some comments and the price; the name of the artist is also provided.

4.3. Mappings for deriving access mechanisms

Finally, information access mechanisms (indexes and search filers) can be derived from task performance descriptions. On the one hand, some Output IPs represent steps during the performance of a task where the user compares elements of a list with each other in order to select the desired one. These IPs are those that both inform about multiple instances of one entity (cardinality *) and provide the user with access to a second IP that informs about only one instance of the same entity (cardinality 1). In the OOWS navigational model, these lists of elements that are used to facilitate the access to a specific one are modelled by means of the index primitive. Thus, each Output IP that accomplishes the conditions mentioned above is derived to an index. On the other hand, Search system actions perform an inquiry over the system state in order to obtain specific information. Queries of this kind can be supported at the conceptual level by the OOWS search filter primitive. We can derive this primitive from the search system actions that are activated from an Output IP and they also query the system in order to obtain information about the same entity associated to the Output IP.

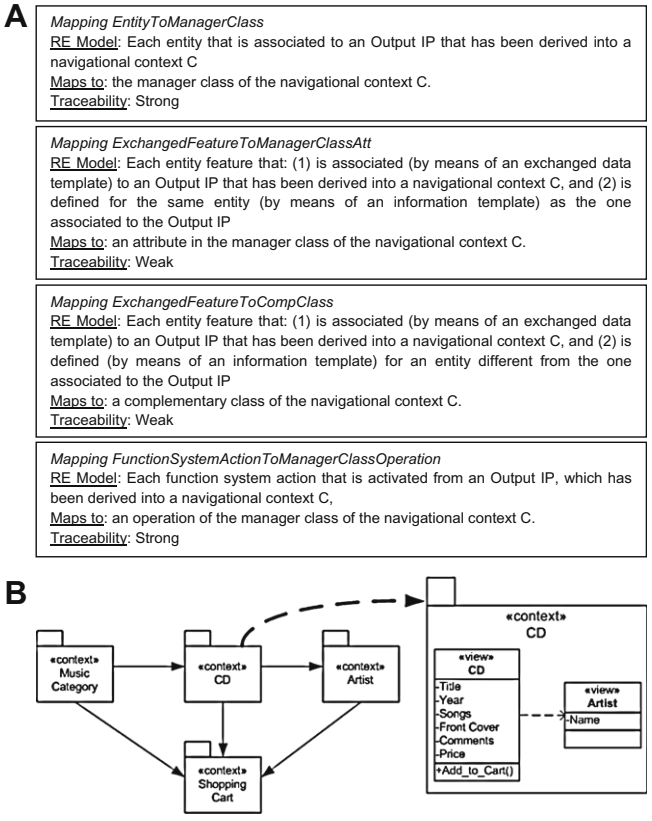


Fig. 11. Mappings for deriving context navigational views.

The two derivations introduced above are supported by mappings such as the ones in Fig. 12A. The first mapping, which derives indexes, has a weak traceability because software engineers may decide to support this list by means of other conceptual elements instead of an index. For instance, software engineers may consider a better solution to model this list of elements by means of another navigational context in order to obtain particular navigational characteristics. The second mapping, which supports search filters, has a strong traceability because users need to perform the searches in order to properly satisfy the task logic. In the OOWS navigational model, a search filter is the only primitive that allow us to represent these searches at the conceptual level. Therefore, the derived search filters are mandatory because no other conceptual elements can replace them.

Fig. 12B shows the access mechanisms that are derived when the mappings are applied to the task Add CD. These access mechanisms are: an index (derived from the Output IP that provides users with a list of CDs) and a search filter (derived from the search system action that allows users to search CDs of a specific artist). The attributes defined in each access mechanism are derived from the exchanged data templates associated to: (1) the Output IP from which the index is derived and (2) from the Input IP that is connected to the Search system action from which the filter is derived. These derivations are supported by mappings that are not presented in order to not overload this section.

5. Applying mappings in an automatic and traceable way

In this section, we present a strategy based on graph transformations that allows us to automatically apply the set of mappings

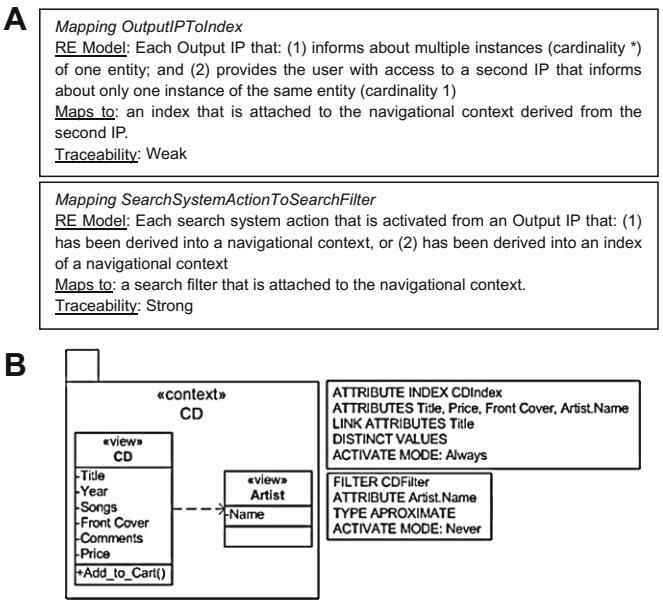


Fig. 12. Mappings for deriving access mechanisms.

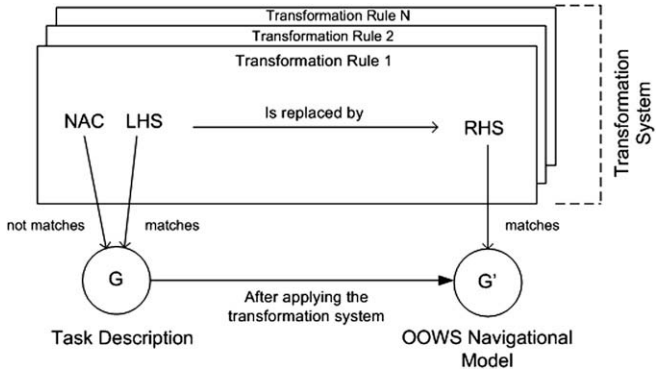


Fig. 13. A transformation system.

presented above in a traceable way. This strategy consists in defining each mapping as a graph transformation rule and then applying these rules by means of the AGG tool [17]. Additionally, each graph transformation rule has been defined in order to support traceability aspects that can be analysed by the TaskTracer tool (which is presented in Section 7).

Section 6.1 describes how graph transformations are defined and how traceability mechanisms are included in their definition. Section 6.2 explains how graph transformations are applied to task-based requirements models by means of AGG.

5.1. Defining graph transformations with traceability mechanisms

Graph transformations rely on the theory of graph grammars [16]. A graph grammar is defined from: (1) a transformation system, which is a set of transformation rules and (2) a graph to which the transformation rules are applied (called host graph).

The main reasons for using graph transformations are:

- *Formal*: Graph transformations are based on a sound mathematical formalism (graph grammars theory) and enables verifying formal properties on represented artifacts.
- *Visual*: Graph-transformation approaches are capable of expressing mappings in a declarative manner but using a graphical syntax that facilitates the comprehension of its internal logic and its application.
- *Correctness checkable*: Several mechanisms (such as conditional graph rewriting or typed graph rewriting) are provided in order to check the correctness of the artifact obtained after the application of each transformation rule.
- *Tool supported*: Several tools such as AGG [17], AToM3 [51], VIATRA [52] or VMTS [14] can be used for defining and applying graph transformations. These tools have been tested throughout multiple applications in real scenarios.

A graph transformation rule is a graph rewriting rule that is made up of a Left Hand Side (LHS), which represent a sub-graph of the host graph, and a Right Hand Side (RHS), which represent a sub-graph of the target graph. We can also define a Negative Application Conditions (NAC), which represents a sub-graph that must not be contained in the host graph. Fig. 13 illustrates how a rule of a transformation system is applied to a graph G: when the LHS matches G and the NAC does not match G, then the LHS is replaced by the RHS. G is transformed into G′. All elements of G that are not covered by the match are considered as unchanged. All elements that are contained in the LHS and are not contained in the RHS are considered as deleted.

To add more expressiveness to transformation rules, variables may be associated to attributes within a LHS. These variables are initialized in the LHS and their value can be used to assign an attribute to the expression of the RHS. An expression may also be defined to compare a variable declared in the LHS with a constant or with another variable. This mechanism is called 'attribute condition' [27].

Fig. 14 shows an example of a graph transformation rule. This rule implements the mapping *OutputIPToNavigationalContext* (see Fig. 7) together with the mapping *EntityToManagerClass* (see Fig. 10). This rule states that when an IP (LHS) is found, it must be transformed into a Navigational Context with its Manager Navigational Class (RHS). However, this rule is not applied if the IP provides access to another IP that provides the user with information about one instance of the same entity (NAC). In addition, a variable *x* has been defined in the LHS. This variable is initialized with the entity of the Output IP, and it is used to give name to the Navigational Context and the Navigational Class included in the RHS. This variable is also used in the NAC to check that both connected Output IPs have the same entity.

By applying rules such as the one presented above we are able to transform a graph that represents a task-based requirements model into a graph that represents an OOWS navigational model. However, they do not provide support for traceability purposes. Once a rule is applied, we obtain a conceptual element, but we lose the requirement element from which it is derived (since the LHS is replaced by the RHS). In order to solve this problem, we propose following the next two guidelines:

- The RHS part repeats the requirement elements defined in the LHS and add the derived conceptual elements.
- The RHS part includes traceability elements that have two functions:
  - To define a traceability link between the newly created conceptual elements and the requirement elements from which they are derived. To do this, each traceability element presents a source and a target arc. The source arc is connected to the elements included in the LHS. The target arc is connected to the elements that are created by the RHS.
  - To identify the mapping that has been applied in the derivation. To do this, we associate a unique identifier that represents each mapping to each traceability element.

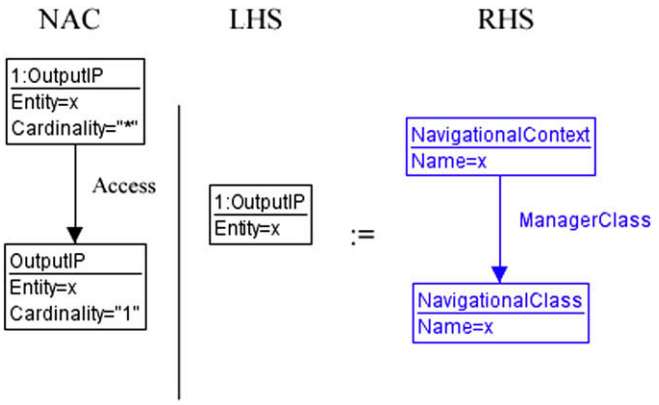


Fig. 14. Graph transformation rules.

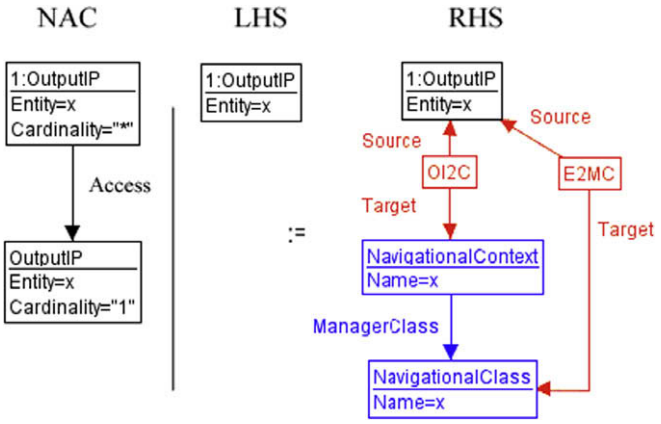


Fig. 15. Example of traceable graph transformation rules.

Fig. 15 shows the graph transformation rule present in Fig. 14 redefined according to the guidelines presented above. The RHS of this rule also derives a navigational context and a manager navigational class for each Output IP. However, this RHS maintains the Output IP defined in the LHS. Additionally, the RHS create two traceability elements:

- *OI2C* which links the newly created Navigational Context (*target*) to the Output IP from which it is derived (source). The name of this traceability element indicates that the navigational context has been derived by applying the mapping ***O**utput**I**PToNavigational**C**ontext* (see Fig. 8).
- *E2MC* which links the newly created Manager Navigational Class (*target*) to the OutputIP whose entity has been used to derive the manager class. The name of this traceability element indicates that the manager class has been derived by applying the mapping ***E**ntityTo**M**anager**C**lass* (see Fig. 11).

Fig. 16 presents another example of a traceable graph transformation rule. In this case, the rule implements the mapping *FunctionSystemActionToManagerClassOperation*. The LHS of this rule matches with an Output IP from which a function system action can be activated. Notice how this LHS uses the auxiliary element *OI2C* defined above in order to identify the navigational context (with its manager class) derived from the Output IP. The RHS maintains the structure defined in the LHS and adds a class operation that is connected to the manager class of the context. Additionally, the traceability element *F2O* is created. This element connects the newly created Operation (target) to the Function system action (source) from which it is derived. Additionally, it identifies the application of the corresponding mapping.

The set of traceable graph transformation rules that fully implement the derivation of OOWS navigational models from task-based requirements models can be found in [26]. Fig. 17 shows a representative example of the result that is obtained when this set of rules is applied. The upper side of this figure shows a host graph that represents a partial version of the task-based requirements model of the running example. In particular, it shows the representation of the performance description of the task Add CD. The lower side of Fig. 17 shows the graph that is obtained when the traceable graph transformation rules are applied. The middle of the figure shows the traceability elements that are created by the rules. Notice how the links created by these traceability elements allow us to easily identify the requirement elements from which conceptual elements are derived.

5.2. Applying traceable graph transformation rules

As introduced above, several tools provide support to the definition and application of graph transformations. In this work, we have used the AGG tool [17] to automatically apply the traceable graph transformation rules presented above.

AGG (Attributed Graph Grammars tool) can be considered to be a genuine programming environment based on graph transformations. It provides (1) a programming language that enables the specification of graph grammars and (2) a customizable interpreter that enables graph transformations. AGG was chosen because it allows the graphical expression of directed, typed, and attributed graphs (for expressing specifications and rules). It has a powerful library containing algorithms for graph transformation, critical pair

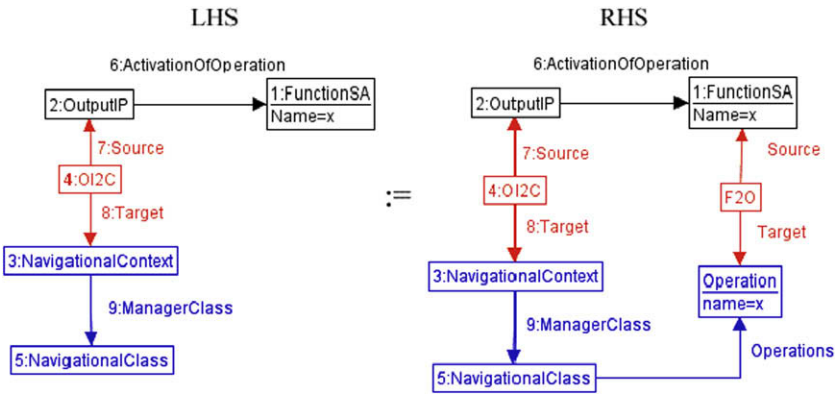


Fig. 16. Another Example of traceable graph transformation rules.

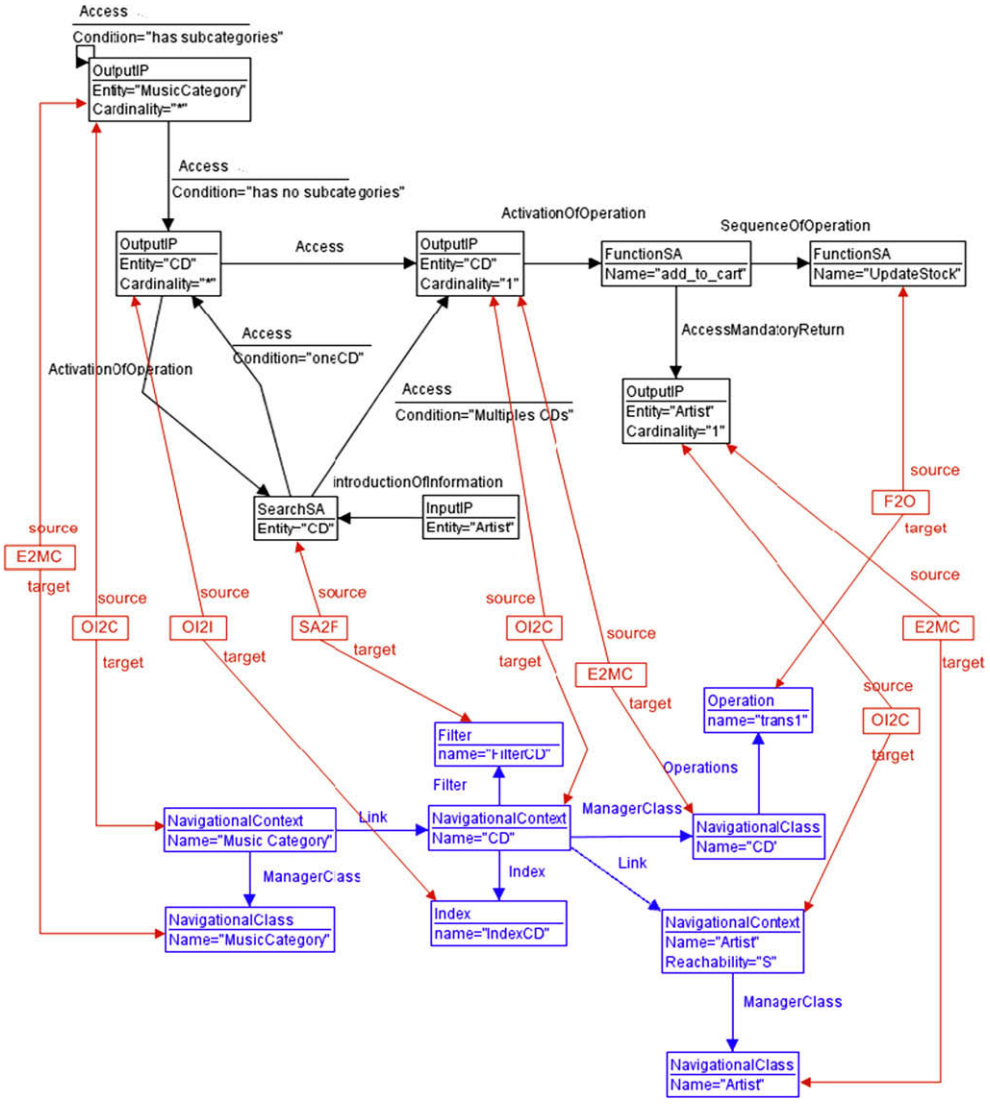


Fig. 17. Example of resultant graph.

analysis, consistency checking, and application of positive and negative conditions.

Fig. 18 shows a snapshot of the AGG user interface. Frame 1 is the grammar explorer. Frames 2, 3 and 4 enable the specification of sub-graphs that make up a production: a negative application (frame 2), a left-hand side (frame 3) and a right-hand side (frame 4). The host graph on which graph transformation rules are applied is represented in Frame 5.

The AGG tool allows us to automatically transform a source *graph* into a target *graph* by applying graph transformation rules. To do this, we just need to load the source graph into the AGG tool (in our case a graph that represents a task-based requirements model) in order to introduce the definition of the graph transformation rules, and then the system automatically applies the rules and obtains the target graph (in our case a graph that represents an OOWS navigational model).

However, our objective is to apply traceable graph transformation rules to task-based requirements models, which are not constructed as graphs. In order to solve this problem we take advantage of the way in which AGG store graphs. This tool stores graphs in an XML document by using simple XML elements such as *Node* and *Edge* (which define the graph), and *NodeType* and *NodeEdge* (which associate a type to the previous elements). Then, we specify a task-based requirements model in an XML document by means of elements defined directly from the concepts proposed in the task-based requirements model (*ElementaryTask*, *OutputIP*, *FunctionSA*, etc.). Next, a XSL Transformation is performed in order to translate this task-based XML specification into an XML AGG graph. This aspect is illustrated in Fig. 19.

In the same way that a task-based requirements model is handled in order to obtain an AGG graph, the graph obtained after applying rules (which represents an OOWS navigational model) must be handled in order to be transformed into the proper format. Next, we present a tool that handles this graph in order to provide analysts with a report for analyzing traceability aspects.

6. Reporting traceability links by means of tasktracer

In this section, we present a prototypical tool that analyses AGG graphs to generate a traceability report to study how requirements are supported by means of OOWS conceptual primitives. This tool is the *TaskTracer* (see Fig. 20). It navigates the AGG graph searching for traceability elements. The tool analyses this information and creates a repository of traceability links. Next, it generates a report in HTML format which accesses this repository. The user interface of this tool is very intuitive and is divided into two main frames:

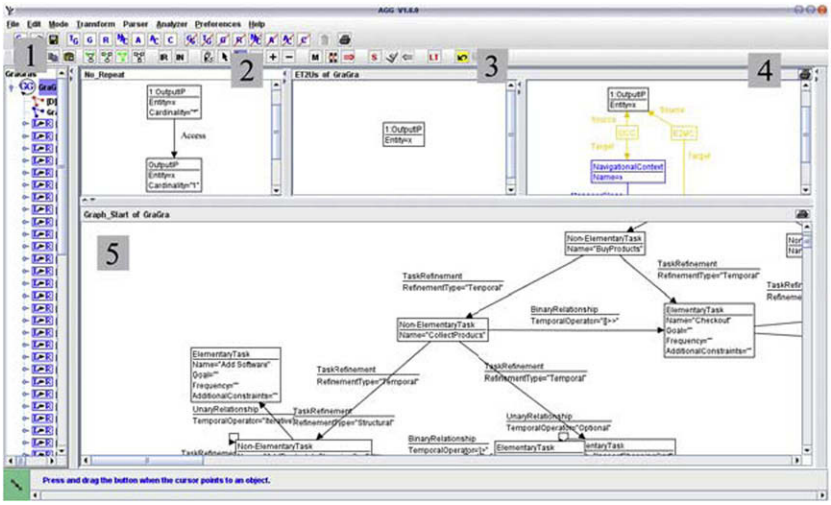


Fig. 18. AGG user interface.

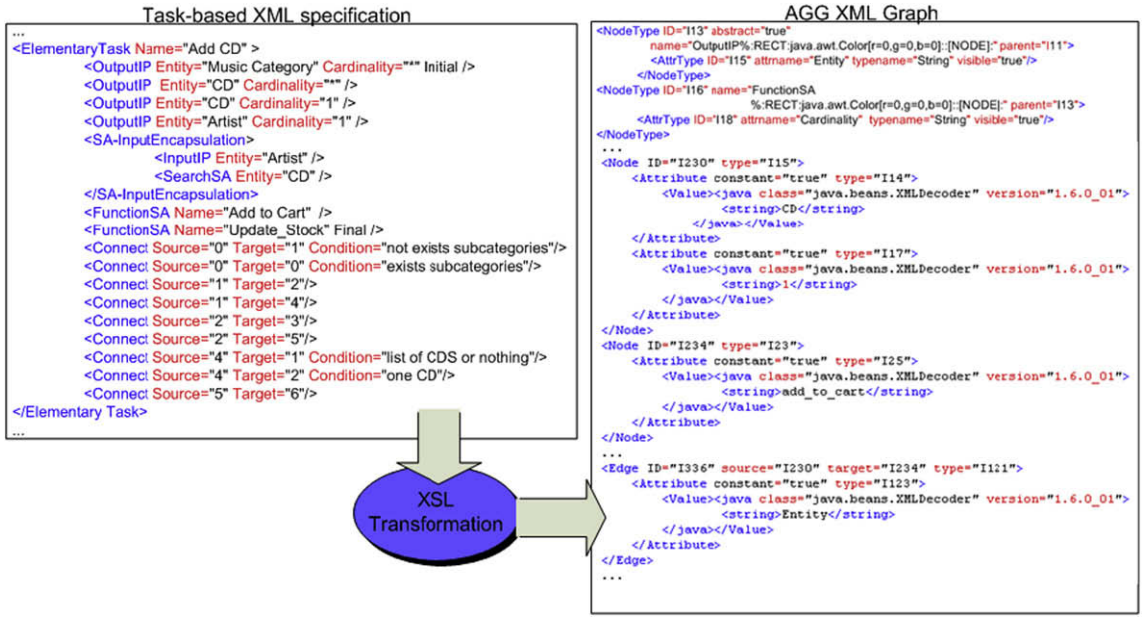


Fig. 19. Obtaining an AGG XML graph by means of an XSL Transformation.

(1) The Source File frame, which allows us to select the XML document in which the AGG graph is stored. (2) The Target Path frame, which allows us to indicate the path in which the traceability report must be created.

Next, we present both the traceability link repository and the HTML report.

6.1. The traceability link repository

As explained above, the TaskTracer tool takes as input an XML document that contains an AGG graph derived by applying the set of traceable graph transformation rules. Then, the tool searches for traceability elements. For each traceability element that is found, the tool stores a traceability link in a repository.

In order to define this repository, we have first defined the structure that must be used to store information about traceability links. To do this, we have defined a traceability link meta-model, which is shown in Fig. 21. This meta-model has been inspired by other works such as [28–30] where meta-models with traceability purposes are proposed.

According to the meta-model in Fig. 21, each traceability link presents an *Identifier*. Additionally, the links are defined by

- A *Requirement Element*, which references an element of the task-based requirements model.
- A *Conceptual Element*, which references an element of the OOWS navigational model.
- A *Mapping*, which references one of the mappings presented in Section 5.

The mappings presented in Section 5 have been assigned to a weak or a strong traceability. Mappings with a strong traceability are those whose derived conceptual elements are mandatory to properly support the specified requirements. Mappings with a weak traceability are those whose derived conceptual elements may be changed by analysts if they consider it opportune. In this

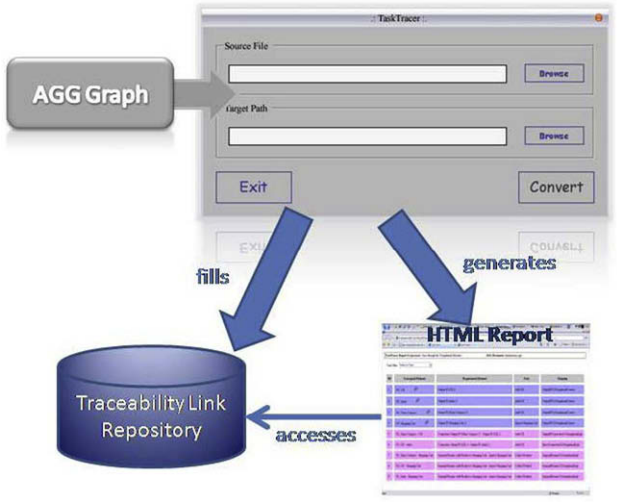


Fig. 20. TaskTracer tool.

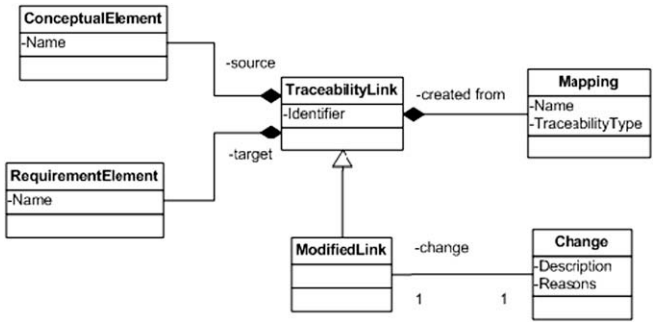


Fig. 21. Traceability link meta-model.

context, we also want to maintain a tracing of these changes. To do this, we have included *Modified Links* in the traceability meta-model. These links inherit all the information of the Traceability Links presented above. However, a *Change* is associated to them. This change indicates that the conceptual element derived by the mapping that is associated to the traceability link has been changed. Each Change is defined by a *Description* of the change performed and the *Reasons* for performing the change. In relation to Modified Links, a constraint must be satisfied: the mapping associated to every Modified Link must have a weak traceability assigned. This constraint is expressed in OCL as follows:

```
Context ModifiedLink
  Inv: self.allInstances -> forAll(l |
    l.Mapping.TraceabilityType=Weak)
```

Taking the meta-model introduced above as base, we have developed the traceability link repository as a relational database. The database server that has been used to facilitate the access and management of this database has been MySQL Server [31].

6.2. The traceability report

Once the TaskTracer tool has introduced the identified traceability links into the repository, an HTML report is generated. This report accesses the information stored in the repository and provides analysts with an intuitive description of how elements of the OOWS navigational model have been derived from the task-based requirements model. This aspect allows analysts to study both *how* the derived navigational model supports the specified requirements and *why* (according to which requirements) this navigational model has been derived.

On the one hand, the generated report provides analysts with a main table that explains which requirements have been considered in the derivation of the navigational structure. This table is shown in Fig. 22. The table associates each OOWS conceptual element that defines the Web application navigational structure (navigational contexts and navigational links) to the requirement element from which they are derived. The task in which the requirement element has been defined as well as the mapping used to perform the derivation is also indicated. Information about the type of traceability (weak/strong) is not included in this table because the mappings used to derive navigational contexts and navigational links all have a strong traceability.

In order to better analyse the mappings that have been applied, mappings that derive navigational contexts are highlighted in a different colour (blue) from mappings that derive navigational links (pink). Additionally, notice how the main table is complemented with a task filter (see left upper side of Fig. 22).[6] This filter allows us to select one of the tasks defined in the requirements model. When we do this, only the conceptual elements derived from this task are shown. This aspect allows us to study individually how each task is supported at the conceptual level. This filter also allows us to select one of the temporal relationships defined in the task taxonomy. In this case, the main table is filtered in order to show the navigational links derived from the temporal relationship as well as the navigational contexts that these navigational links connect. An example of how the task filter is used is shown in Section 6.3.

The traceability report also includes a table for each navigational context, which explains how the context has been derived. To access these tables, we just need to click over the magnifying glass icon that appears with the name of each context in the main table (see Fig. 22). Fig. 23 shows the traceability table associated to the navigational context CD. Each row of the table associates the conceptual elements that define the context with the requirements elements from which they have been derived. Additionally, we can also know the mapping that is applied in each derivation and the traceability type (strong or weak). In order to better analyse this last aspect, derivations with a strong traceability are highlighted with a different colour (dark blue) from the derivations with a weak traceability (light blue).

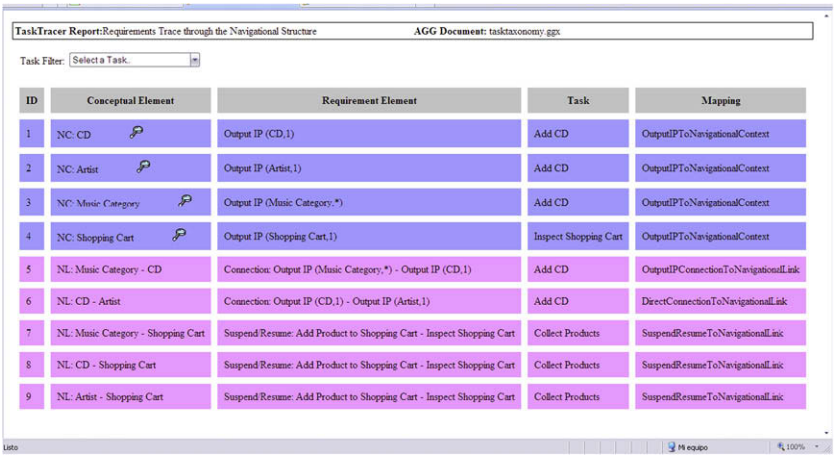
Conceptual elements derived by mappings with weak traceability can be modified. In order to trace these modifications, the traceability report includes a last column (Modified) that assigns a checkbox to each of these elements. If the checkbox is activated, the corresponding Traceability Link of the repository is transformed to a Modified Link. In addition, the *Changes* link appears at the left side of the checkbox (see the tenth Traceability Link in Fig. 23). By clicking over this link, the page in Fig. 24 appears. This page allows us to introduce (and further query) the changes that have been made and the reasons for them.

6.3. Benefits of using the traceability report

In this section, we introduce the main benefits that are obtained by using the traceability reports introduced above. These benefits are the following:

- To validate the completeness of the model-to-model transformation, i.e., validate whether or not all the navigational requirements are supported.

[6] For interpretation of the references in colour in Figs. 22 and 23, the reader is referred to the web version of this article.



TaskTracer Report: Requirements Trace through the Navigational Structure **AGG Document:** tasktaxonomy.ggx

Task Filter: Select a Task.

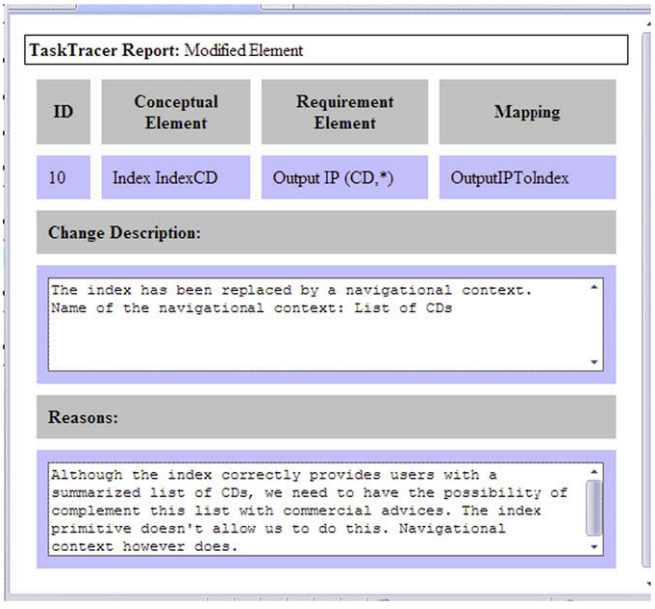
ID	Conceptual Element	Requirement Element	Task	Mapping
1	NC: CD	Output IP (CD,1)	Add CD	OutputIPToNavigationalContext
2	NC: Artist	Output IP (Artist,1)	Add CD	OutputIPToNavigationalContext
3	NC: Music Category	Output IP (Music Category.*)	Add CD	OutputIPToNavigationalContext
4	NC: Shopping Cart	Output IP (Shopping Cart,1)	Inspect Shopping Cart	OutputIPToNavigationalContext
5	NL: Music Category - CD	Connection: Output IP (Music Category,*) - Output IP (CD,1)	Add CD	OutputIPConnectionToNavigationalLink
6	NL: CD - Artist	Connection: Output IP (CD,1) - Output IP (Artist,1)	Add CD	DirectConnectionToNavigationalLink
7	NL: Music Category - Shopping Cart	Suspend/Resume: Add Product to Shopping Cart - Inspect Shopping Cart	Collect Products	SuspendResumeToNavigationalLink
8	NL: CD - Shopping Cart	Suspend/Resume: Add Product to Shopping Cart - Inspect Shopping Cart	Collect Products	SuspendResumeToNavigationalLink
9	NL: Artist - Shopping Cart	Suspend/Resume: Add Product to Shopping Cart - Inspect Shopping Cart	Collect Products	SuspendResumeToNavigationalLink

Fig. 22. Traceability report created by TaskTracer: main table.

TaskTracer Report: Requirements Trace through Navigational Context Definition **Navigational Context:** CD

ID	Conceptual Element	Requirement Element	Mapping	Traceability Type	Modified
1	Manager Class CD	Output IP (CD,1)	EntityToManagerClass	Strong	
2	MC Attribute: Title	Info.Temp: CD // Feature: Title	ExchangedFeatureToManagerClassAtt	Weak	☐
3	MC Attribute: Year	Info.Temp: CD // Feature: Year	ExchangedFeatureToManagerClassAtt	Weak	☐
4	MC Attribute: Songs	Info.Temp: CD // Feature: Songs	ExchangedFeatureToManagerClassAtt	Weak	☐
5	MC Attribute: Front Cover	Info.Temp: CD // Feature: Front Cover	ExchangedFeatureToManagerClassAtt	Weak	☐
6	MC Attribute: Comments	Info.Temp: CD // Feature: Comments	ExchangedFeatureToManagerClassAtt	Weak	☐
7	MC Attribute: Price	Info.Temp: CD // Feature: Price	ExchangedFeatureToManagerClassAtt	Weak	☐
8	Complementary Class Artist	Info.Temp: CD // Feature: Artist Name	ExchangedFeatureToCompClass	Weak	☐
9	MC Operation: Add_to_Cart	System Action: Add_to_Cart	FunctionSystemActionToManagerClassOperation	Strong	
10	Index IndexCD	Output IP (CD,*)	OutputIPToIndex	Weak	☑ Changes
11	Filter FilterCD	System Action: Search (CD)	SearchSystemActionToSearchFilter	Strong	

Fig. 23. Traceability report created by TaskTracer: context CD definition table.



TaskTracer Report: Modified Element

ID	Conceptual Element	Requirement Element	Mapping
10	Index IndexCD	Output IP (CD,*)	OutputIPToIndex

Change Description:

The index has been replaced by a navigational context.
Name of the navigational context: List of CDs

Reasons:

Although the index correctly provides users with a summarized list of CDs, we need to have the possibility of complement this list with commercial advices. The index primitive doesn't allow us to do this. Navigational context however does.

Fig. 24. Editor for introducing changes into weak traceability links.

We can perform this validation by means of the main table provided by the traceability report (see Fig. 25). First, by using the task filter, we can *validate whether or not every task has been derived into a set of navigational context and navigational links*. If the task appears in the drop-down list, the task is supported in the navigational model; otherwise, an error has occurred either during the derivation process or during the requirements specification. Second, once we have checked that every task appears in the drop-down list, we can validate if their descriptions are fully supported. To do this, we can use the task filter of the main table to show only the conceptual elements derived from a specific task. Next, by using this filtered main table and the context tables associated to each navigational context (see Fig. 22), we can *analyse whether or not a task is fully supported at the conceptual level*. In order to fully support a task at the conceptual level, each requirements element (e.g. Output IP, Search System, and Exchanged Feature) that has been defined in the task performance description must be associated to a conceptual element.

For instance, Fig. 25A shows how every task defined for the running example appears in the drop-down list associated to the task filter. Notice how temporal relationships also appear in this list. Fig. 25B shows the navigational contexts and the navigational links derived from the task Add CD. From this list, we can validate that the IPs Output IP(CD,1), Output IP(Artist,1), and Output IP(Music

Fig. 25. Analyzing if the task Add CD is fully supported at the conceptual level.

Category,*) have been derived into a navigational context. Furthermore, we can also validate that the connection between the Output IP(Music Category,*) and the Output IP (CD,1), and the connection between the Output IP (CD,1) and the Output IP(Artist,1), have been derived into a navigational link. Next, we can use the tables associated to the navigational contexts in order to validate whether or not the rest of elements defined in the task Add CD are derived into a conceptual element.

- To validate the correctness of the model-to-model transformation, i.e., validate whether or not the navigational model is compliant with the requirements.

We can use the traceability report to facilitate the validation of whether the derived navigational model is compliant with the specified requirements. To do this, we must validate that the conceptual elements fit the semantics of the requirements elements from which they are derived. For instance, in order to validate if navigational contexts are compliant with requirements, we must analyse (considering that a navigational context defines a chunk of information provided to users) if navigational contexts provide the information defined in the Output IPs from which they are derived. In the same way, we can analyse if the navigational links defined in the navigational model allow us to navigate the information in the same way as it is described at the requirements model. We can also study if searches, summarized lists of objects and operations are all provided in the way and time that is indicated in the task performance descriptions.

In this case, the traceability report helps us to know the requirement element from which a conceptual element is derived as well as to filter conceptual elements by tasks in order to study each task individually. However, to properly perform this validation, we need to precisely know the semantics of both requirement elements and conceptual elements, and manually analyse if they fit each other properly.

- To support requirements traceability in changes performed directly on the navigational model derived by the model-to-model transformation.

Some OOWS conceptual primitives derived by the model-to-model transformation are derived by mappings with a weak traceability. These elements are marked as proposed elements indicating that analysts can change them if they consider it opportune. We mark these elements as elements that can be modified because their modification has no critical consequences on the way in which users must perform the described tasks. Examples of these changes can be the replacement of a conceptual primitive by an equivalent one (such as in the example in Fig. 24), the change of the name of a class attribute in order to better adjust to its semantics, to hide class attributes in some contexts in order to not overload them, etc. These changes constitute modeling decisions that do not produce changes on the captured requirements and they are more intuitively done by directly modifying the navigational model. However, these changes must be properly linked with the corresponding requirements in order to maintain a correct trace of requirements.

The traceability report supports analysts in the performance of these modifications. On the one hand, weak conceptual elements are clearly marked in the traceability report (see Fig. 23) in order to help analysts to identify these elements and then study whether they need to be modified. On the other hand, the traceability report provides analysts with facilities to describe changes introduced manually in a traceable way. Fig. 24 shows the traceability report presents an editor for describing the changes introduced in weak conceptual elements. These descriptions allow us to link the new conceptual elements created in each modification with the requirements that they support.

- To establish the impact of changing a requirement on the navigational model.

It is well known that customers usually propose changing requirements throughout the development process. The traceability report allows analysts to study the impact of these changes on the navigational model before applying them.

We consider the impact of changing a requirement on a navigational model to be the number of conceptual primitives that must be modified due to the change. Then, we can initially guess this im-

pact by analyzing the conceptual elements that the traceability report associates to each requirement element. For instance, we can guess that the impact of changing a temporal relationship between two tasks will be defined by the modification of the navigational links that have been derived from this relationship. In the same way, if we want to change an Output IP, we can guess that the impact of this change will be defined by the modification of the derived index or navigational context.

Additionally, we must not forget that navigational models are derived from requirements models in an automatic way. In this context, we can check the impact of changing a requirement by directly performing this change and deriving the navigational model. If the change is not satisfactory, we just need to undo the change and derive the initial navigational model again.

7. Related work

This work is focused on providing support for tracing requirements in the context of Web application development. In particular, we have paid special attention on how navigational requirements can be traced. In this context, to trace navigational requirements through different software artifacts, we must first properly specify them. However, there is little support to properly perform this task.

Navigation is an aspect that has been encouraged in the development of Web applications. To properly handle this new aspect, several Web Engineering methods have appeared during the last decade. Some examples are OOHDM [7], UWE [8], WSDM [10], WEBML [11], W2000 [35], OOH [36], RMM [37] or NDT [38]. However, most of these approaches are only focused on providing new mechanisms to describe Navigation at the conceptual level; as far as the specification of navigational requirements, they prescribe the use of traditional techniques such as Use Case diagrams. In this context, we can find works such as [39–42] that state that new RE techniques are required for specifying the requirements of Web applications since RE techniques for traditional software are not appropriate for supporting distinctive characteristics of Web applications such as Navigation.

Only few approaches such as OOHDM, UWE and NDT take navigational requirements in consideration. These approaches propose the use of use cases in the requirements specification. However, they introduce additional techniques for complementing use cases in order to properly support navigational requirements. OOHDM introduces User Interaction Diagrams (UIDs) [12]. For each use case, a UID is defined. Each UID graphically describes the exchange of information between the users and the system without considering specific aspects of the user interface. UWE proposes complementing use cases with activity diagrams. Use case diagrams are used to represent an overview of the requirements while activity diagrams provide a more detailed view [13]. In this case, two types of use cases are proposed: Navigation use cases, which comprise a set of browse information activities, and WebProcess use cases, which include more complex activities that are expressed in terms of transactions that have been initiated by the user. Finally, NDT complements uses cases with formatted templates. NDT classifies requirements into the following types: storage information, actor, functional, interaction (which describe navigation) and non-functional requirements. For each type, NDT defines a special template, i.e., a table with specific textual fields that must be completed for specifying requirements.

As far as the derivation of navigational models from requirements specifications, different support is provided by the three approaches introduced above. On the one hand, OOHDM presents guidelines in order to help analysts to define the navigational model from the requirements specifications. These guidelines are defined from a set of rules presented in [12,43]. These rules indicate how the elements that define the navigational model can be systematically derived from UIDs. These rules are textually described by using natural language. No formalism is used in the rule definition. In this context, certain degree of ambiguity is introduced in the rule definition (derived from the use of natural language). This aspect becomes critical if we consider that rules must be manually interpreted and applied. No tool is provided to support the application of these rules.

On the other hand, UWE and NDT provide a common Web requirements meta-model (WebRe)[13] that fits the requirements specification of both approaches. Furthermore, a set of model-to-model transformations that take a description based on this meta-model as source allows us to systematically obtain navigational models. The derived navigational models are defined by using the UWE conceptual abstractions. The model-to-model transformations are defined in the QVT Relations standard [44]. However, authors do not provide a tool for automatically applying these transformations. They are currently looking forward to tools that will support the QVT transformation language. In the meantime, they are gathering experience with other transformation languages and techniques, such as ATL and graph transformations [45].

As far as the tracing of requirements through the navigational model, none of these approaches provide mechanisms that support analysts in this activity.

There are other approaches such as [3–6] that face the problem of requirements traceability in MDD. However, none of them offer traceability support for Navigation. In addition, we introduce new ideas that distinguish our work from these other ones:

In [3], a basic framework of requirements tracing is introduced to facilitate the correction of software in model-driven development. This framework establishes a set traceability patterns that relate UML models to each other according to the development process of the Business Object Analysis and Design methodology [55]. This is similar to the traceability rules that we have introduced in this work. However, the application of the patterns must be done by analysts after all the models are created. There is no mechanism that automates the application of these patterns. In addition, these patterns are not used to automate the creation of models like we have done in this work.

In [4], a set of traceability rules that helps analysts to create conceptual models from requirements models in introduced. These traceability rules take a use case based model as source and relate its elements to elements of the UML class diagrams. In addition, there is a tool that supports the application of these rules and therefore allows us to automatically derive partial class diagrams from use cases. However, this work is mainly focused on support forward traceability (that is, deriving class diagrams from use cases). Little work has been done to support backward traceability (that is, to identify use case elements from class diagram elements). We support this aspect by introducing the proper mechanisms into the definition of graph transformations that can be analysed by the TaskTracer tool.

In [5], an approach based on Behaviour Trees [56] is proposed to systematically derive a software design from functional requirements. This approach proposes representing behaviour in individual functional requirements by means of behaviour trees. Next, mechanisms are introduced to integrate these behaviour trees into a single one. Finally, the integrated behaviour tree is translated to a component-based software design. As happens in [4], this approach provides little support for backward traceability. In addition, there is no tool that supports the application of the proposed derivation and so they must be done manually.

In [6], a tool-supported approach is presented to define and analyse traceability relationships between models in a bidirec-

tional way (that is, forward and backward). This tool takes different models as source and automatically creates traceability relationships between models using a set of manually defined scenarios of use (which has a predefined structure). Thus, this tool can be used to improve the development of software in MDD methods that propose to construct models manually. In methods where models are automatically derived through transformations, relationships between models are already defined in model transformations; we just need mechanisms that allow these transformations to support traceability aspects. We have provided mechanisms of this type through the traceable graph transformations and the TaskTracer tool.

Our work differs from all the analysed works in the following aspects: (1) We provide a requirements model that properly considers the navigational aspect in the specification of Web applications requirements; (2) we provide a model-to-model transformation to automatically derive conceptual models from this requirements model in a traceable way; and (3) we provide a tool that analyses performed model-to-model transformations in order to provide us with traceability reports.

8. Conclusions and further work

In this paper, we have presented a strategy to improve a classical problem in software engineering: How to go from the problem space to the solution space with a sound methodological guidance. In particular, we have faced this problem from the context of Web application development by paying special attention to an aspect that is poorly considered in traditional software development: Navigation.

We have presented a methodological guide that allows us to derive OOWS navigational models from task-based requirements models. This guide has been defined as a collection of mappings. These mappings take the abstractions of a task-based requirements model as base and indicate how they can be supported at the conceptual level by means of primitives of the OOWS navigational model.

In addition, we have complemented these mappings with a strategy to apply them automatically and with traceability capabilities. To do this, we have implemented each mapping as a graph transformation rule. This aspect has allowed us to define a strategy based on the AGG tool for automatically applying graph transformation rules to task-based requirements models. Furthermore, in the definition of graph transformation rules, we have included traceability elements that have been used to generate a traceability report in a later step.

The tool TaskTracer has been implemented to obtain the traceability report. This tool takes an AGG graph obtained after applying traceable transformation rules as source. From this graph, the tool instantiates a repository of traceability links and generates a report based on HTML that allows analysts to easily study how requirements are supported by the navigational model.

As proof of concept, we have used our approach in the development of several small and medium-size Web applications. One of these applications is the E-commerce application used as running example in this paper. Other applications are the *ProS Research Center Website* (http://www.pros.upv.es) and the *DSIC Department Website* (http://www.dsic.upv.es). Through the development of these web applications we have checked that the navigational model is correctly derived. To do this, we have defined the navigational model of each application twice, first manually and next by using the model-to-model transformation. Then, the EMF Compare plugging of the Eclipse platform has helped us to compare both versions of the navigational model. This has allowed us to check that mappings are correctly applied to derive conceptual elements.

In addition, we have used the TaskTracer tool to validate requirements and the navigational model with stakeholders. This has allowed us to check that requirements were correctly linked to navigational model elements.

The TaskTracer tool has also arisen as a valuable mechanism to manage the changes in requirements that were introduced by stakeholders after creating an initial requirements specification. Some of these changes were considered by weak traceability mappings (those that define conceptual elements that can be changed by equivalent ones or even deleted without critically changing the specified requirements). They were easily considered and synchronized in both requirements and navigational model by using the editor for introducing changes of the TaskTracer tool. Other type of changes required a more complex modification of the navigational model. In these cases, the use of the TaskTracer tool helped us to study how the navigational model must be modified. When a requirement changed, we just needed to use the TaskTracer tool in order to identify which elements of the navigational model needed to be updated. This aspect allowed us to save time an effort to support the introduced changes in requirements.

As further work, we plan to develop a tool that allows us to perform changes in the web application prototype in a traceable way, i.e., changes that are performed in the implementation artefacts will automatically be reflected in the task-based requirements model.

References

[1] S.J. Mellor, A.N. Clark, T. Futagami, Model-driven development – Guest editor's introduction, IEEE Software 20 (5) (2003) 14–18.
[2] K. Czarnecki, S. Helsen, Classification of model transformation approaches, in: Proceedings of the 2nd Workshop on Generative Techniques in the Context of Model-Driven Architecture (OOPSLA'03), USA, 2003.
[3] T. Tsumaki, Y. Morisawa, A framework of requirements tracing using UML, in: Proceedings of 7th Asia Pacific Software Engineering Conference (APSEC 2000), pp. 206–213.
[4] E. Insfrán, A requirements engineering approach for object-oriented conceptual modeling, PhD Thesis, Department of Information Systems and Computation, Technical University of Valencia, Spain, October 2003.
[5] R.G. Dromey, From requirements to design: formalizing the key steps, in: Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03), 2003.
[6] A. Egyed, P. Grünbacher, Automating requirements traceability: beyond the record & replay paradigm, in: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02), 2002.
[7] D. Schwabe, G. Rossi, S.D.J. Barbosa, Systematic hypermedia application design with OOHDM, in: Proceedings of the Seventh ACM Conference on Hypertext, Bethesda, Maryland, United States, March 16–20, 1996, HYPERTEXT96, ACM, New York, NY, 1996, pp. 116–128.
[8] N. Koch, Software engineering for adaptive hypermedia applications, PhD Thesis, Ludwig-Maximilians-University, Munich, Germany, 2000.
[9] J. Fons, V. Pelechano, M. Albert, O. Pastor, Development of web applications from web enhanced conceptual schemas, in: Proceedings of the 22th International Conference on Conceptual Modeling (ER 2003), LNCS, Chicago, IL, USA, October 13–16, vol. 2813, Springer, Berlin, 2003.
[10] O. De Troyer, C. Leune, WSDM: a user-centered design method for web sites, in: Computer Networks and ISDN Systems, Proceedings of the 7th International World Wide Web Conference, Elsevier, Brisbane, Australia, 1998, pp. 85–94.
[11] S. Ceri, P. Fraternali, A. Bongio, Web modeling language (WebML): a modeling language for designing web sites, in: WWW9 Conference, Amsterdam, May 2000.
[12] P. Vilain, D. Schwabe, C. Sieckenius, A diagrammatic tool for representing user interaction in UML, in: Lecture Notes in Computer Science, UML'2000, York, England 2002.
[13] N. Koch, G. Zhang, M.J. Escalona, Model transformations from requirements to web system design, ICWE'06, July 11–14, 2006, Palo Alto, California, USA, 2006.
[14] Visual modeling and transformation system (vmts), <http://avalon.aut.bme.hu/~tihamer/research/vmts> (last time accessed 22-11-2007).
[15] P. Valderas, J. Fons, P. Pelechano, Developing E-Commerce applications from task-based descriptions, in: Proceedings of the 6th International Conference on E-Commerce and Web Technologies, EC-Web 2005.
[16] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, Singapore, 1997.
[17] Attributed Graph Grammar System v.1.6, <http://tfs.cs.tu-berlin.de/agg/>.

[18] O. Gotel, Contribution structures for requirements traceability, Ph.D. Thesis, Imperial College, Department of Computing, London, England, 1995.
[19] Unifed Modelling Language 2.0. Object Management Group, <Http://www.uml.org>.
[20] A. Shepherd, Hierarchical Task Analysis, Taylor & Francis, London, 2001.
[21] F. Paternò, C. Mancini, S. Meniconi, ConcurTaskTree: A Diagrammatic Notation for Specifying Task Models, in: INTERACT'97, Chapman & Hall, 1997, pp. 362–369.
[22] M.J. Escalona, Modelos y técnicas para la especificación y el análisis de la navegación en sistemas software, PhD. Thesis, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, 2004 (in Spanish).
[23] R. Wirfs-Brock, B. Wilkerson, L. Wiener, Designing Object-oriented Software, Prentice-Hall, 1990.
[24] A. Durán, B. Bernárdez, A. Ruiz, M. Toro, A requirements elicitation approach based on templates and patterns. International Workshop on Requirements Engineering (WER'99), Buenos Aires, Argentina, 1999, pp. 17–29.
[25] O. Pastor, J. Gómez, E. Insfrán, V. Pelechano, The OO-method approach for information systems modelling: from object-oriented conceptual modeling to automated programming, Information Systems 26 (7) (2001) 507–534.
[26] P. Valderas, A set of graph transformation rules for transforming requirements specifications into OOWS conceptual models, Technical Report, Department of Information Systems and Computation, Technical University of Valencia, Spain, <https://oomethod.dsic.upv.es/files/TechnicalReport/GraphTransformationRuleCatalogue.pdf>, 2007.
[27] H. Partsch, R. Steinbruggen, Program transformation systems, ACM Computing Surveys 15 (3) (1983) 199–236.
[28] D.S. Kolovos, R.F. Paige, F.A.C. Polack, On-demand merging of traceability links with models, in: Proceedings of the 2nd EC-MDA Workshop on Traceability, Bilbao, Spain, 2006.
[29] M. Didonet Del Fabro, J. Bézivin, P. Valduriez, Weaving models with the eclipse AMW plugin, in: Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany, 2006.
[30] F. Jouault, Loosely coupled traceability for ATL, in: Proceedings of the ECMDA-Traceability Workshop (ECMDA-TW), 2005.
[31] MySQL 5.0 Community Server 5.0, <http://www.mysql.org/>.
[32] M.J. Escalona, N. Koch, Requirements engineering for web applications: a comparative study, Journal on Web Engineering 2 (3) (2004) 193–212.
[33] P. Valderas, J. Fons, V. Pelechano: transforming web requirements into navigational models: an MDA based approach, ER 2005, pp. 320–336.
[34] P. Valderas, V. Pelechano, O. Pastor: a transformational approach to produce web application prototypes from a web requirements model, International Journal on Web Engineering and Technology 3 (1) (2007) 4–42.
[35] L. Baresi, F. Garzotto, P. Paolini, Extending UML for modeling web applications, in; Proceedings of the 34th Hawaii International Conference on System Sciences, 2001.
[36] J. Gomez, C. Cachero, O. Pastor, Extending a conceptual modelling approach to web application design, in: Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAISE), LNCS, Stockholm, Sweden, June 5–9, vol. 1789, Springer, Berlin, 2000, pp. 79–93.
[37] T. Isakowitz, E. Stohr, P. Balasubramanian, A methodology for the design of structured hypermedia applications, Communications of the ACM 8 (38) (1995) 34–44.
[38] M.J. Escalona, Modelos y técnicas para la especificación y el análisis de la navegación en sistemas software, PhD. Thesis, Departamento de Lenguajes y Sistemas Informáticos. Universidad de Sevilla, 2004 (in Spanish).
[39] E. England, A. Finney, Managing Multimedia: Project Management for Interactive Media, second ed., Addison-Wesley, Reading, MA, 1999.
[40] J. Burdman, Collaborative Web Development, Addison-Wesley, Reading, MA, 1999.
[41] S.P. Overmyer, What's Different about Requirements Engineering for Web Sites?, in: Requirements Engineering, Springer-Verlag London Limited, 2000, pp 62–65.
[42] D. Lowe, Web System Requirements: An Overview, Requirements Engineering, vol. 8, Springer-Verlag London Limited, 2003.
[43] P. Vilain, D. Schwabe, Improving the web application design process with UIDs, in: Proceedings of the 2nd International Workshop on Web Oriented Software Technology (IWWOST'2002) Málaga, Spain, June 10, 2002.
[44] Meta Object Facilities (MOF) Query/ Views/ Transformations (OCL) 1.0. Object Management Group. <Http://www.omg.org> (accessed 22.11.07).
[45] S. Segura, D. Benavides, A. Ruiz-Cortés, M.J. Escalona, From requirements to web system design. an automated approach using graph transformations, IV Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM'07), In conjunction to the XII Jornadas de Ingeniería del Software y Bases de Datos, Zaragoza, Spain, 2007.
[46] P. Greenspun, Philip and Alex guide to Web publishing, <http://photo.net/wtr/thebook>, 1999 (last time accessed 22-11-2007).
[47] J. Conklin, Hypertext: an introduction and survey, IEEE Computer 20 (9) (1987) 17–41.
[48] S. Lauesen, Task description as functional requirements, IEEE Software March–April 2003 20 (2) (2003) 58–65.
[49] Object Constraint Language (OCL) 2.0. Object Management Group. <Http://www.omg.org> (accessed 22.11.07).
[50] M. Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.
[51] J. De Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, Gabriele Taentzer3Attributed graph transformation with node type inheritance, Theoretical Computer Science 376 (3) (2007) 139–163 (Special Section on FASE'04 and FASE'05).
[52] D. Varró, A. Pataricza, Generic and meta-transformations for model transformations engineering, in: 7th International Conference on the Unified Modeling Language (UML 2004).
[53] P. Valderas, A meta-model for a task-based requirements model, Technical Report, Department of Information Systems and Computation, Technical University of Valencia, Spain, <http://oomethod.dsic.upv.es/oomethod/>, July 2007.
[54] P. Valderas, The OOWS approach in a nutshell, Technical Report, Department of Information Systems and Computation, Technical University of Valencia, Spain, <http://oomethod.dsic.upv.es/oomethod/>, July 2007.
[55] BON: The analysis and design method for reliability, reusabilility and reversibility, <http://archive.eiffel.com/products/bon.html>.
[56] K. Winter, Formalising behaviour trees with CSP, in: Proceedings of the International Conference on Integrated Formal Methods (IFM'04), 2004.
[57] R. Young, The Requirements Engineering Handbook, Artech House Publishers, Norwood, MA, USA, 2003.