6 MULTIPATH TRANSFORMATIONAL DEVELOPMENT OF USER INTERFACES WITH GRAPH TRANSFORMATIONS

Quentin Limbourg* and Jean Vanderdonckt**

*SmalS-MvM
Av. Prince Royal, 102 – B-1050 Brussels, Belgium.
**Belgian Laboratory of Computer-Human Interaction (BCHI),
Louvain School of Management (LSM),
Université catholique de Louvain
Place des Doyens, 1 – B-1348 Louvain-la-Neuve, Belgium.

Abstract. In software engineering, transformational development is aimed at developing computer systems by transforming a coarse-grained specification of a system to its final code through a series of transformation steps. Transformational development is known to bring benefits such as: correctness by construction, explicit mappings between development steps, and reversibility of transformations. No comparable piece exists in the literature that provides a formal system applying transformational development in the area of user interface engineering. This chapter defines such a system. For this purpose, a mathematical system for expressing specifications and transformation rules is introduced. This system is based on graph transformations. The problem of managing the transformation rules is detailed, e.g., how to enable a developer to access, define, extend, restrict or relax, test, verify, and apply appropriate transformations. A tool supporting this development paradigm is also described and exemplified. Transformational development, applied to the development of user interfaces of inter-

active systems, allows reusability of design knowledge used to develop user interfaces and fosters incremental development of user interfaces by applying alternative transformations.

6.1 INTRODUCTION

A recent survey of the area of Human-Computer Interaction (HCI) compared to the area of Software Engineering (SE) would find the former to be mainly empirical, experience-based, and relying on implicit knowledge and the latter to be notoriously and deliberately structured, principle-based, and relying on explicit knowledge. The development lifecycle of highly-interactive systems in general and of their User Interface (UI) in particular form the cornerstones of HCI, which has been observed to suffer from several shortcomings that are intrinsic either to the type of interactive systems being developed or to the existing practices used. Among these shortcomings are the following observations:

- *Lack of rigorousness*: on the one hand, the development lifecycle of interactive systems cannot be based on the same rigorous models that are typically used in SE (Brown 1997). On the other hand, HCI lifecycle is submitted to a high order of complexity that is neither reflected nor well supported in existing models and methods (Wegner 1997).

- *Lack of systematicity*: as SE aimed for a well-structured method for developing highly complex systems, so did HCI for developing interactive systems. However, the systematicity and the reproducibility found in SE methods cannot be transferred straightforwardly to HCI: the development lifecycle remains intrinsically open, ill-defined, and highly iterative (Sumner et al. 1997) as opposed to the domain of SE where it is structured, well-defined, and progressive (D'Souza and Wills, 1999).

- *Lack of a principle-based approach*: where SE proceeds in the development from one step to another according to well-established principles, in contrast HCI usually advances in a more opportunistic way when the current result is usable enough to proceed to the next step (Bodart et al., 1995; Puerta, 1997).

- *Lack of explicitness*: not only the knowledge required to properly conduct the development lifecycle of interactive systems is not as principled as in SE, but also it is implicitly maintained in the mind of experienced designers. This knowledge is therefore harder to communicate from one person to another, although initiatives exist that make this knowledge more explicit through design patterns, usability guidelines. Even more, when this knowledge is made more explicit, nothing can guarantee that it is applied uniformly and consistently within a same development project or across various development projects.

The aforementioned comparison holds as long as significant efforts toward structured, principle-based, and explicitly based process devoted in SE remain unparalleled with the area of HCI. This chapter seeks to contribute to reestablish a balance between HCI

and SE regarding this aspect by providing an effort in the direction of a true development process for UI engineering with the same quality factors that are usually found in SE. For this purpose, it is expected that a model-driven approach of UI development could represent an engineering effort attempting to systematize UI development. It does so by constructing high-level requirements and, progressively, transforms them to obtain specifications that are detailed and precise enough to be rendered or transformed into code. This type of approach is referred to in the SE literature as the *transformational approach*. More recently, along with the Model Driven Architecture (OMG, 2006) proposal (Miller and Mukerij, 2003), model processing and transformation has gained a particular importance in the software engineering literature (Rensik, 2003; Kuske et al., 2002; Gerber et al., 2002).

Several ingredients are lacking in existing HCI methods to fully achieve a transformational approach in the development of UI. Conceptually, there is no systematic understanding of the relationships among all development artifacts (i.e., models) needed to build a UI. Furthermore, the design knowledge required to feed these models and to make them smoothly evolve over time from one development step to another is often implicitly maintained in the minds of developers and designers and/or hard-coded in supporting software. When such design knowledge exists, it is not always systematically, consistently, and correctly applied throughout the project or across projects.

Sumner et al. (1997) explain that the development process, as usually conducted in HCI, is a process that is eminently open (several development steps can be conducted or considered simultaneously), ill-defined (the initial requirements are usually largely incomplete, if not inconsistent), and mostly iterative (it seems impossible to conduct a development step in such a way that its outputs are definitive).

Nanard and Nanard (1995) report that the development lifecycle of an interactive application consists of a sophisticated Echternach process that does not always proceed linearly in a predefined way. It is rather an interwoven set of development steps, which alternate bottom-up and top-down paths, with selecting, backtracking, and switching among several actions. Thus any method and development tool is expected to effectively and efficiently support a flexible development lifecycle, which does not stiffen the mental process of expert designers in a fixed procedural schema. On the other end, when we consider the needs of moderately experienced designers, the method and its supporting tool should enforce a minimum number of priority constraints. These constraints should define which development artifacts must be specified before others, suggesting for example how and when to proceed from one development step to another.

The variety of the approaches adopted in organizations and the rigidity of existing solutions provide ample motivations for a UI development paradigm that is flexible enough to accommodate multiple development paths and design situations while staying precise enough to manipulate information required for UI development. To alleviate these problems, a development paradigm of **multipath UI development** is introduced that is characterized by the following principles:

- *Expressiveness of UI*: any UI is expressed depending on the context of use via a suite of models that are analyzable, editable, and manipulable by software (Puerta, 1997).

- *Central storage of models*: each model is stored in a model repository where all UI models are expressed according to the same UI Description Language (UIDL).

- *Transformational approach*: each model stored in the model repository may be subject to one or many transformations supporting various development steps (Eisenstein *et al*. 2001).

- *Multiple development paths*: development steps can be combined together to form development paths that are compatible with the organization's constraints, conventions, and context of use. For example, a series of transformations may derive a presentation from a task model.

- *Flexible development approaches*: development approaches are supported by following alternate development paths (Nanard and Nanard, 1995) and enable designers to freely shift between these paths depending on the changes imposed by the context of use (Calvary et al., 2003).

To address the above requirements, this chapter presents a method for expressing models that are relevant to HCI, but expressed in an SE way so that HCI development paths can be supported with a level of flexibility that is desired in HCI, while keeping the rigorousness brought by SE. For this purpose, the present chapter is structured as follows: Section 6.2 presents existing work that is related to the issue of structuring the HCI development process via the model-based approach similarly to what MDA is doing in SE. Section 6.3 introduces and motivates the choice of graph grammars and graph transformations to ensure a transformational approach guaranteeing expressiveness and flexibility. The methodology introduced in this chapter supports model transformation based on these concepts. Section 6.4 analyzes how traditional development approaches found in SE can be addressed in a parallel way in HCI by identifying a series of levels of abstractions between which transformations can be applied. Throughout this section, ample examples of design knowledge manipulated at each level are provided. Section 6.5 summarizes the main benefits brought by our methodology and perceived shortcomings.

6.2 RELATED WORK

Model-Based Approach of User Interface (MBAUI) has been around for many years, basing its power on models in order to develop interactive systems. MBAUI can be assimilated to a larger trend in software engineering called the transformational development paradigm. Its *modus operandi* resides in the performance of model-to-model transformations to support model engineering activities of UI development. To provide relevant concepts and a stepwise development cycle is essential in the definition of a development lifecycle. Support for a developer in accomplishing development steps is also highly desirable. In the context of MBAUI, the nature of the support provided to a developer can consist for multiple elements (Puerta, 1997): a simple syntax editor, a graphical model editor, a well-furbished and exemplified documentation system, a structured knowledge base, a model derivation module, a model analyzer, and

a code generator. Such a methodology combining all these advantages does not exist today.

Historically, MBAUI has exploited models of various types and for various uses. MECANO (Puerta, 1996) automatically generates presentation and dialog models as intermediary steps toward a NeXT GUI from a domain model expressed in an object-oriented language. JANUS (Balzert et al. 1996) exploits relationships such as inheritance, aggregation and generalization of a domain model to deduce a UI structure. GENIUS (Janssen et al. 1993) derives UI code from an extended entity-relationship model and, so called, dialog nets based on Petri nets. MOBI-D (Puerta, 1997) uses as input scenarios, a task model and a domain model to automatically generate a GUI. MOBI-D is equipped with a module called TIMM learning from a designer's choices to sharpen a widget selection process.

TEALLACH (Griffiths et al., 2001) allows designing database UIs while allowing co-evolutionary design of a user's task model design is the first tool to integrate explicitly in the design process the concept of model mapping. More recently tools like ARTSTUDIO (Thevenin, 2001) or TERESA (Mori et al. , 2004) exploited the information contained in a user's task model, to derive context-specific presentation of a UI.

All of the above-cited tools and methods perform some model mapping and transformation, somehow. None of them provides an explicit mechanism to represent and manipulate heuristics (or patterns) governing the model transformation process. Some tools do involve some transformational mechanism, but it is built-in so that their modifiability is impossible.

MBAUI is suffering from a bottleneck in the consolidation and dissemination of the knowledge used to realize model transformation. From this statement we may define two requirements to fill our research agenda:

- *Core requirement 1*: an easy-to-understand and uniform description of models subject to transformation. This description would cover various viewpoints on a UI system.
- *Core requirement 2*: an explicit formalism to specify and perform UI model transformations.

6.3 EXPRESSING THE UI DEVELOPMENT CYCLE WITH GRAPH TRANSFORMATIONS

Developing a UI according to an MBAUI philosophy can be seen as an activity of transforming a high-level specification into a more concrete specification (or code). Unfortunately, no generic solution has been proposed to address the problem of defining a computational framework for expressing, manipulating, and executing model transformation involved in engineering approaches to UI construction. To achieve this goal, several requirements have been identified (Limbourg and Vanderdonckt, 2004a, 2004b):

- A definition of each manipulated artifact capturing different *viewpoints* necessary to UI development.

- A definition of relationships between different *viewpoints*. These relationships are essential in order to obtain an integrated view of a specification.
- A representation of the knowledge needed to perform model-to-model transformations.
- A mechanism to manipulate the knowledge to perform a derivation. UI model derivation is heuristic by nature. A satisfactory solution implies at least a possibility, for a developer, to choose between different derivation heuristics. Ideally, a developer should be able to alter or redefine these heuristics.
- A mechanism to check desirable properties on derived models. These properties might be consistency, correctness, or usability.

6.3.1 Approaches for Model Transformation

In the next paragraphs we survey a number of existing techniques and evaluate their relevance to our goal. Imperative languages provide a means to perform model transformation:

- Text-processing languages like Perl or Awk are popular for performing small text transformation. These tools cannot be considered to specify complex transformation systems as they force the programmer to focus on very low-level syntactic details.
- Several environments provide APIs to manipulate and transform models and, often, their corresponding to specific metamodels: Jamda (Boocock, 2003), UMLAUT (Ho *et al*., 1999), dMof (Queensland University, 2002).

Relational approaches (Akehurst et al., 2003; Gerber et al., 2002) rely on the specification of mappings between source and target element types along with the conditions in which a mapping must be instantiated. Mapping rules can be purely declarative, and non executable, or executable thanks to a definition of an execution semantic. Relational approaches are generally implemented using a logic-based programming language and require a clear separation of the source and target model.

XSLT transformations are a good candidate as models have, generally, a syntactical representation in an XML-compliant format. The way XSLT proceeds is very appealing as it (1) searches for matches in a source XML document (2) executes a set of procedural instructions, when a match is found, to progressively construct a target XML file. Unfortunately, some experiences (Gerber *et al*., 2002) showed that XSLT transformations are not convenient to compute model transformation for two main reasons (1) their verbosity has been identified as a major problem to manage complex sets of transformation rules (2) their lack of abstraction: progressively constructing a target XML file entails an inclusion, in transformation rules, of syntactic details relative to the target file.

Common Warehouse Metamodel (CWM) Specification (Object Management Group, 2003) provides a set of concepts to describe model transformation. Transformations can be specified using a black box or a white box metaphor.

Transformations are grouped in transformation tasks (some meaningful set of transformations), which are in turn themselves grouped in transformation activities. A control flow of transformation can be defined between transformation tasks at this level (with the concept of transformation step). Even if transformations allow a fine-grained mapping between source and target element, CWM does not provide a predefined language to specify the way these elements are transformed one to another.

Graph grammars and graph transformations have been used for many years to represent complex transformation systems. It has been used notably in the software engineering field for representing, for instance: software refactoring (Mens et al., 2001), software evolution (Heckel et al., 2002), multiagent system modeling (Depke et al., 2002), modeling language formalization (Varro *et al.*, 2002). Graph grammars have been proved an 'efficient in time' formalism for specifying and computing any model-to-model transformation (Agrawal et al., 2003). As main advantages to our approach, graph transformation specification: (1) are rather declarative (they are based on graph patterns expression) (2) provide an appealing graphical syntax which does not exclude the use of a textual one (3) are executable thanks to an grounded execution semantic based on push-out theory (4) offer modularity by allowing the fragmentation of complex transformation heuristics into small, independent chunks. In the context of UI development with graph transformations, two pioneering work can be mentioned (Freund et al., 1992; Sucrow, 1998). Both approaches make an interesting use of graph transformations but have a too narrow conceptual coverage to address a fully defined UI development cycle.

6.3.2 Our Methodology

Our methodology proposes a framework (Figure 6.1) coping with the development of UIs for single and multiple contexts of use. To achieve this goal, this methodology relies on a set of models structured in four levels of abstraction: (1) an implementation level contains UI code. The UI code is generated from models contained at the model level (2) a model level contains models developed for an actual system. A model at model level is an instance of a meta-model at meta-model level (3) a meta-model level contains a definition of all concepts needed to build UI models (4) a meta-meta model level contains the basic structure definition used to define the meta-model (and transitively, the model level), i.e., a directed, attributed, labeled graph structure.

In a model-based approach of UI development, a designer's task consists mainly in defining models and producing UI code according to these previously defined models. At each phase of the development cycle, specific artifacts are defined; these artifacts correspond to, so called, *viewpoints* on the system. We propose four viewpoints on UI systems:

1. *Computation-independent viewpoint* contains elements enabling the description of a UI system independently of any computer-related considerations. This viewpoint is composed of a *task model and domain model*. A task model is a hierarchical decomposition of the tasks to be carried out by a user to achieve her goal. After a comparison of a dozen task modeling techniques (Limbourg and Vanderdonckt, 2003), an altered version of ConcurTaskTree (CTT) (Mori

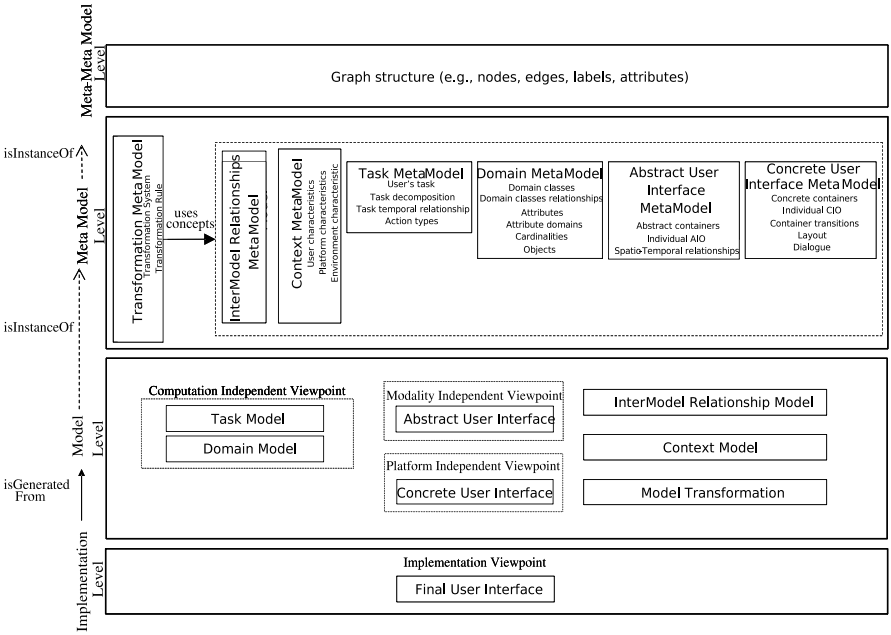

Figure 6.1 Overall framework of our methodology

et al., 2004) has been chosen to represent user's tasks and their logical and temporal ordering. CTT has been altered in the sense that a task taxonomy has been introduced to better describe the nature of a basic task, leaf of a task decomposition. This taxonomy facilitates a mapping between tasks and interaction objects supposed to support this task. A *domain model* contains domain-oriented concepts as they are required by the tasks described in a task model. A domain model describes the real-world concepts and their interactions as understood by users (D'Souza and Wills, 1999). Our domain model is a UML class diagram populated with classes, attributes, methods, objects (Larman, 2001). Concepts contained in a domain model are at a certain point manipulated by systems users. By manipulated, it is meant that domain concepts are at a certain point subject of an exchange (an input or/and an output) between the user and the system. Consequently, domain concepts can be mapped onto elements describing a UI structure and behavior (e.g., abstract UI, concrete UI).

2. *Modality-independent viewpoint* contains elements that are independent of the modality (e.g., graphical interaction, vocal interaction, speech synthesis and recognition, video-based interaction, virtual, augmented, or mixed reality) in which the UI they describe will be rendered. This viewpoint contains an *Abstract UI* (AUI) specification. An AUI defines abstract containers by grouping subtasks according to various criteria (e.g., task model structural patterns, cognitive load analysis, and semantic relationships identification), a navigation

scheme between the Abstract Containers (AC) and selects one or several *Abstract Individual Component* (AIC) for each basic user's task. Each "abstract individual component" is attached to one or several facets describing its function in a UI. We identify four types of facets: input, output, navigation, and control. Abstract interaction objects can be mapped onto: (i) a task or a set of tasks they support, (ii) a domain class or a set of attributes they represent, (iii) a concrete interaction object.

3. *Platform-independent viewpoint* contains a viewpoint that is (1) independent of the computing platform for which the system will be implemented and (2) dependent of a particular modality. This viewpoint contains a *Concrete UI* specification. A CUI concretizes an abstract UI for a given context of use. A CUI is populated with *Concrete Interaction Objects* (CIOs) (Vanderdonckt and Bodart, 1993).

4. *An implementation viewpoint* is a viewpoint containing a coded UI i.e., any UI running on a particular platform either by interpretation (e.g., through a browser) or by execution (e.g., after compilation of code).

Three other models are defined in our framework: (1) an inter-model relationship model, (2) a context model, and (3) a transformation model. These models do not define any particular viewpoint but rather are needed in a UI development process at every phase: (1) contains a set of mapping declarations linking elements belonging to different viewpoints, (2) contains a description of all the context considered during the development process, and (3) contains a set of rules enabling the transformation of one viewpoint into another or to adapt a viewpoint for a new context of use.

Our viewpoint structuring can be compared (Figure 6.2) with respect to the Model-Driven Architecture (MDA) proposal provided by the Object Management Group (Miller and Mukerij, 2003). MDA proposes a set of concepts and methodological recommendations to address the development of systems in a context characterized by a diversity of evolving computing platforms. MDA *viewpoints* are: (1) a Computation-Independent Model (CIM), sometimes called business model, shows a system in a way that is totally independent of technology (typically a business class diagram in OO methods). (2) A Platform-Independent Model (PIM) provides a view of the system independently of any details of the possible platform for which a system is supposed to be built. (3) A Platform-Specific Model (PSM) provides a view of a system that is dependent on a specific platform type for which a system is supposed to be built. (4) An implementation is a specification providing all details necessary to put a system into operation.

6.3.3 Transformation Is the Name of the Game

Our methodology enables expressing and executing model transformation based on UI viewpoints. Figure 6.3 illustrates the different kinds of transformation steps in our framework:

- *Reification* is a transformation of a high-level requirement into a form that is appropriate for low-level analysis or design.

Table 6.1 A comparison of terms used in MDA and our methodology

Model-Driven Architecture	**Our methodology**
Computing-Independent Model	Computation-independent viewpoint: task and domain models
Platform-Independent Model	(1) Modality-Independent viewpoint: Abstract UI model; (2) platform independent viewpoint: Concrete UI
Platform-Specific Model	Implementation viewpoint: UI Code
Platform Model	Context Model

- *Abstraction* is an extraction of a high-level requirement from a set of low-level requirement artifacts or from code.
- *Translation* is a transformation a UI in consequence of a context of use change. The context of use is, here, defined as a triple of the form *(E, P, U)* where E is an possible or actual environments considered for a software system, P is a target platform, U is a user category.
- *Reflection* is a transformation of the artifacts of any level onto artifacts of the same level of abstraction, but different constructs or various contents (Calvary et al., 2003).
- *Code generation* is a process of transforming a concrete UI model into a compilable or interpretable code.
- *Code reverse engineering* is the inverse process of code generation.

The different transformation types are instantiated by *development steps* (each occurrence of a numbered arrow in Figure 6.3). These development steps may be combined to form *development paths*. Development paths are detailed in Section 6.4. The content of Section 6.4 is detailed right of Figure 6.3).

While code generation and code reverse engineering are supported by specific techniques (not covered in this chapter), we use graph transformations to perform model-to-model transformations i.e., reifications, abstractions and translations.

The models have been designed with an underlying graph structure. Consequently any graph transformation rule can be applied to any UI specification. Graph transformations have been shown convenient formalism (Limbourg and Vanderdonckt, 2004a, 2004b). The main reasons are (1) an attractive graphical syntax, (2) a clear execution semantic, and (3) an inherent declarativeness of this formalism. Development steps

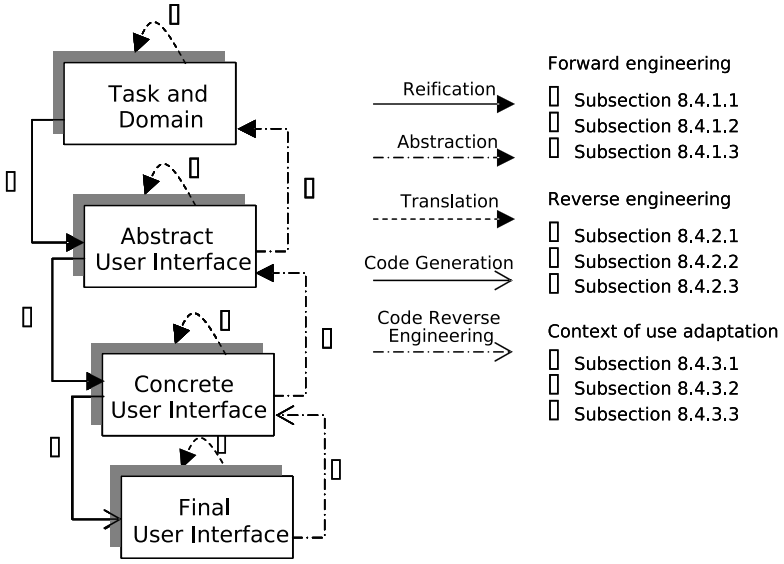

Figure 6.2 Transformations between viewpoints

are realized with transformation systems. A transformation system is a set of (individual) transformation rules. A transformation rule is a graph rewriting rule equipped with negative application conditions and attribute conditions (Rozenberg, 1997).

Figure 6.3 illustrates how a transformation system applies to a specification: let G be a specification, when (1) a Left-Hand Side (LHS) matches into G and (2) a Negative Application Condition (NAC) does not matche into G (note that several NAC may be associated with a rule), and (3) the LHS is replaced by a Right-Hand Side (RHS). G is resultantly transformed into G', a resultant specification. All elements of G not covered by the match are considered as unchanged. All elements contained in the LHS and not contained in the RHS are considered as deleted (i.e., rules have destructive power). To add to the expressive power of transformation rules, variables may be associated to attributes within an LHS. Theses variables are initialized in the LHS; their value can be used to assign an attribute in the expression of the RHS (e.g., LHS : button.name:=x, RHS : task.name:=x). 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*.

As shown in Figure 6.4, transformation rules have a common meta-model with our models. Furthermore, to preserve the consistency of transformed artifact, resultant UI models are checked upon their meta-model. Transformation rules resulting in a non-consistent resulting graph are just not applied.

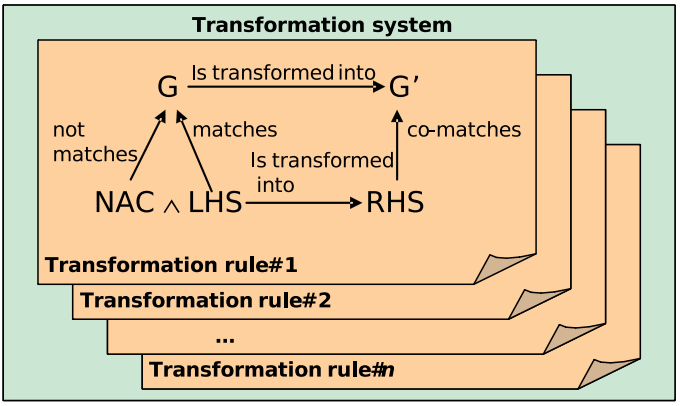

Figure 6.3 A transformation system in our methodology

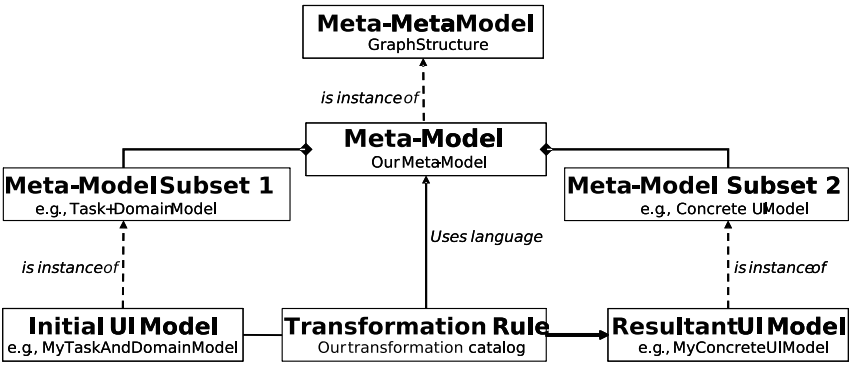

Figure 6.4 Framework for model transformations

6.4 DEVELOPMENT PATHS

Transformation types have been introduced in Section 6.3.3. These transformation types are instantiated into *development steps*. These development steps may be composed to form *development paths*. Several types of development paths are identified:

- **Forward engineering** (or **requirement derivation**) is a composition of *reifications and code generation* enabling a transformation of a high-level viewpoint into a lower-level viewpoint.
- **Reverse engineering** is a composition of *abstractions* and *code reverse engineering* enabling a transformation of a low-level viewpoint into a higher level viewpoint.
- **Context of use adaptation** is a composition of a *translation* with another type of transformation enabling a viewpoint to be adapted in order to reflect a change in the context of use of a UI.

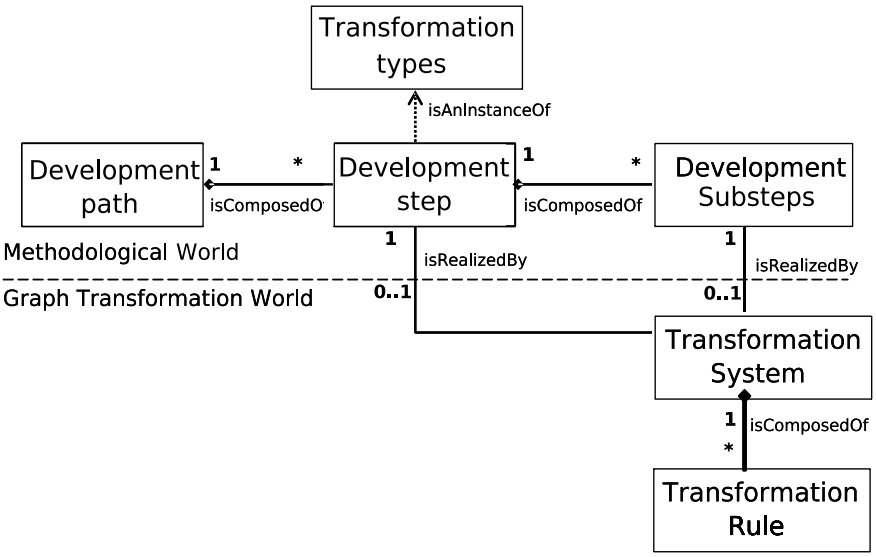

Figure 6.5 Transformation paths, step and substep

As show in Figure 6.5, development paths are composed of development steps. Development steps are instances of transformation types described in Section 6.3.3. Development steps are decomposed into development substeps. A development substep realizes a basic goal assumed by the developer while constructing a system. Some basic goals have been identified by Luo (1995). It may consist, for instance, of selecting concrete interaction objects, defining navigation, etc. Development steps and development substeps may be realized by transform system. In the remainder of this section, subsections 6.4.1, 6.4.2, and 6.4.3 respectively illustrate main development paths (forward, reverse engineering, and context of use adaptation). An example for each development step and substep is provided. All examples use the graphical formalism of the tool AGG (Ehrig et al., 1999).

6.4.1 Forward Engineering

As shown in Figure 6.6, the starting point of UI forward engineering is the construction of a task specification and a domain model. This initial representation is then transformed into an abstract UI which is then transformed into a concrete UI model. The concrete UI model is then used to generate UI code. A forward engineering process is fully illustrated hereafter.

From Task & Domain to Abstract User Interface. Step T1 (Figure 6.6) concerns the derivation of an AUI from models at the computation-independent viewpoint (e.g., a task, a domain, or task and domain model). This development step may involve the following development substeps:

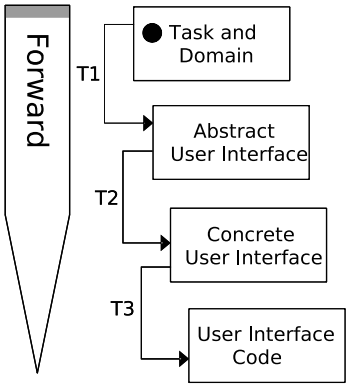

Figure 6.6 Forward transformational development of UIs

- **Identification of Abstract UI structure** consists of the definition of groups of abstract interaction. Each group corresponds to a group of tasks tightly coupled together. The meaning of "task coupling" may vary from one method to another. It goes from very simple heuristics like "for each group of task child of a same task generate an interaction space" to sophisticated heuristics exploiting temporal ordering and decomposition structure between tasks (e.g., enable task sets method followed by Mori et al. ,2004) or information flow between tasks e.g., TRIDENT method proposed by Vanderdonckt and Bodart (1993).

Example 1 is a transformation system composed of two rules enabling the creation of a simple hierarchical structure containing abstract individual components and abstract containers.

- Rule 1 (Figure 6.7): For each leaf task of a task tree, create an Abstract Individual Element. For each task, parent of a leaf task, create an Abstract. Link the abstract container and the Abstract Individual Element by a containment relationship.
  - Rule 2 (Figure 6.8): create an Abstract Container structure similar to the task decomposition structure.

- **Selection of abstract individual component** consists of finding the best abstract individual component type to support one or several user's tasks. Task type, attribute types and domain of value of domain concepts, structure of the domain model are notably important information to perform an adequate AIC selection.

Example 2 is composed of rule 3. It exploits information on task action types to attach appropriate facets to corresponding abstract individual components.

- Rule 3 (Figure 6.9): for each abstract individual element mapped onto a task the nature of which consists in the activation of an operation and this task is

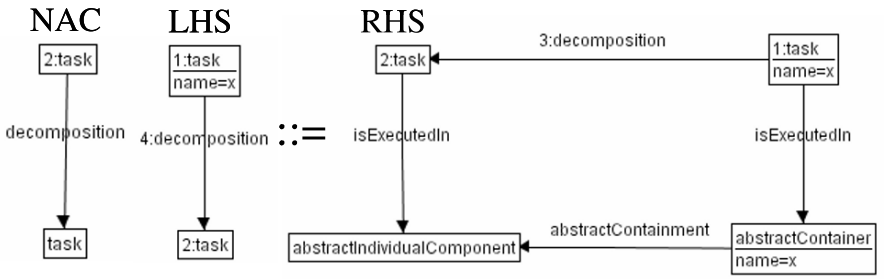

Figure 6.7 Creation of abstract individual components derived from task model leaves

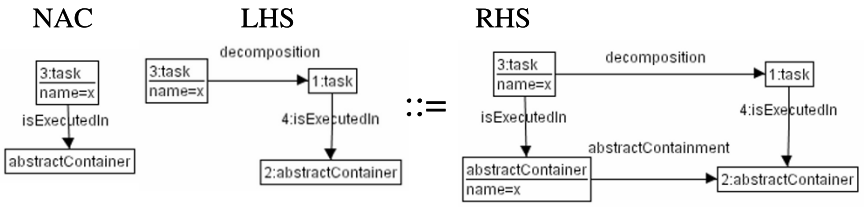

Figure 6.8 Creation of abstract containers derived from task model structure

mapped onto a class, assign to the abstract individual component an action facet that activates the mapped method.

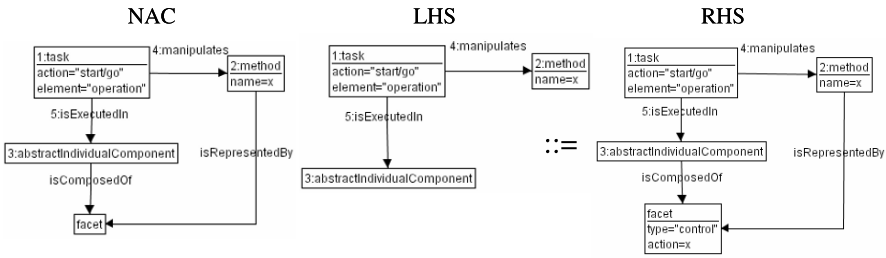

Figure 6.9 Creation of a facet for an abstract individual component derived from task action type

- **Identification of spatiotemporal arrangement of abstract individual components and abstract containers**. The structure of a task model is exploited to derive spatiotemporal arrangement of elements contained in an AUI. This, temporal relationships defined between tasks can be respected in the abstract specification. This is an essential guarantee of usability of the UI to be created. Spatiotemporal relationships between abstract elements are done using Allen

temporal relationships generalized for 2D and specialized for describing any arrangement of a pair of widgets (Trevisan et al. 2002). Limbourg et al. (2005) detail this mechanism more thoroughly. Two levels of arrangement are identified: (1) intra-container level (example 3) concerns the arrangement of abstract individual components within the same abstract container (2) inter-container level (example 4) concerns the definition of a navigational structure among abstract containers.

Example 3 is composed of rule 4. It places abstract individual components in precedence relationship ("isBefore") based on the fact that the tasks they represent are sequential ("≫"). To perform a complete arrangement every type of task temporal relationship should be covered by a rule.

- Rule 4 (Figure 6.10): for every couple of AIC belonging to a same abstract container and these AIC are mapped onto sister tasks that are sequential "≫", create a relationship of type "isBefore" between these AIOs.

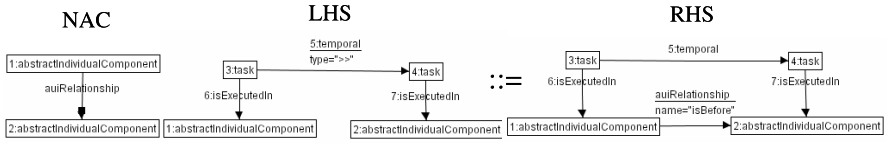

Figure 6.10 A sequentialization of abstract individual component derived from task temporal relationships

Example 4 is composed of rule 5. It defines spatiotemporal arrangement between abstract containers. It uses the same principle as example 3

- Rule 5 (Figure 6.11): For an abstract container (ac1) mapped onto a task (taskX). TaskX is related to a task (taskY) that is mapped onto an AIO (aio2) belonging to an abstract container (ac2) different than ac1, then create an "is simultaneous" spatiotemporal relationship between them.

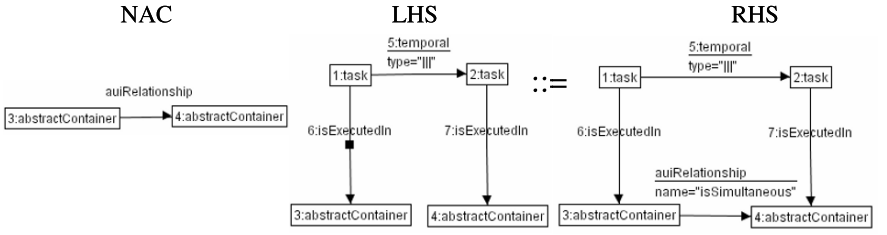

Figure 6.11 A placement of abstract container derived from task temporal relationships

From Abstract User Interface to Concrete User Interface . Step T2 consists of generating a concrete UI from an abstract UI. This development step may involve the following development substeps:

- **Reification of abstract containers into concrete containers.** An abstract container may be reified in different types of concrete containers. Factors influencing this transformation are: modality, context of use, interaction style, designer's preference. A major difficulty of this step resides in the problem of choosing an appropriate level to group abstract containers into a concrete container (typically a window for a graphical modality). A minimal choice would be to create a concrete container (e.g., a window) for each group of sibling leaf tasks. A maximal solution would be to group all abstract individual components and all abstract containers into a single concrete container (e.g., one window).

Example 5 is a transformation system composed of rules 6 and 7. This system transforms into window, abstract containers at a certain depth in the abstract container hierarchy. All abstract containers content is reified and embedded into the newly created window.

- Rule 6 (Figure 6.12): Each abstract container at level "leaf-l" is transformed into a window. Note that an abstract container is always reified into a, so called, box at the concrete level. This box is then embedded into a window.

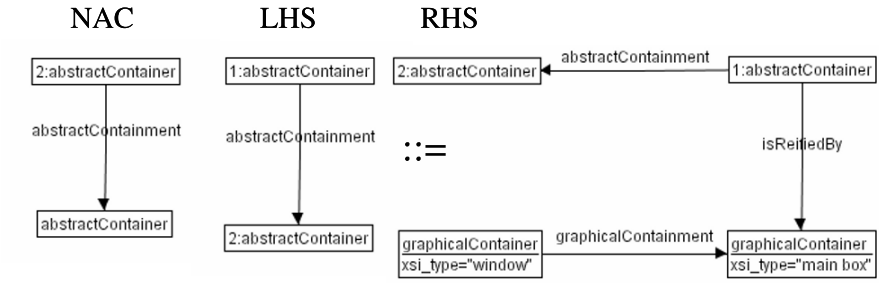

Figure 6.12 A creation of windows derived from containment relationships at the abstract level

- Rule 7 (Figure 6.13): Each abstract container contained into an abstract container that was reified into a window is transformed into a horizontal box and embedded into the window.

- **Selection of concrete individual components**. Functionalities of abstract individual component are identified with their facet. Selection of concrete individual components consists of choosing the appropriate concrete element that will support whole or a part of the facets associated with an abstract individual component.

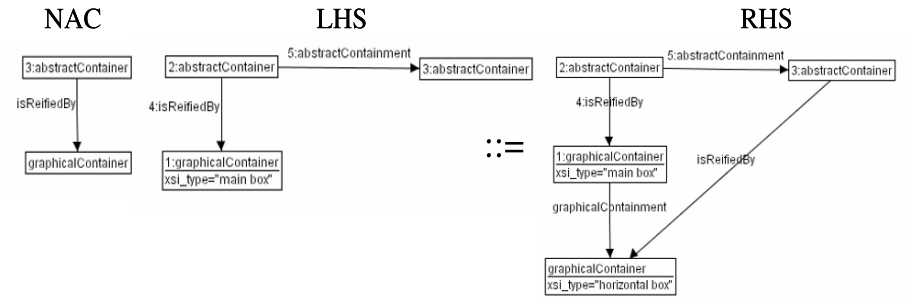

Figure 6.13 A generation of window structure derived from containment relationship at the abstract level

Example 6 is composed of rule 8. It creates an editable text component (i.e., a text box) to reify an AIO with an input facet.

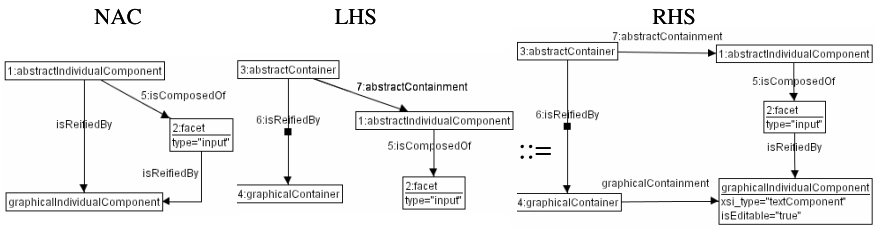

Figure 6.14 Creation of an editable text component (i.e., an input field) derived from facets type of abstract components

- Rule 8 (Figure 6.14): Each input facet of an abstract individual component is reified by a graphical individual component (a type of concrete individual component) of type "editable text component" (i.e., a text box).
  - **Arrangement of concrete individual component.** Allen relationships used to specify spatiotemporal relationships among abstract interaction objects are interpreted in order to provide information on the relative placement of a concrete individual component with respect to other elements of this type.

Example 7 is composed of rule 9. This example transforms an AUI into a concrete model for the graphical modality. It chains concrete individual components according to abstract individual component ordering.

- Rule 9 (Figure 6.15): For each couple of abstract individual components related by a "isBefore" relationship and reified into concrete individual components, generate a "isAdjacent" relationship between the concrete individual components.

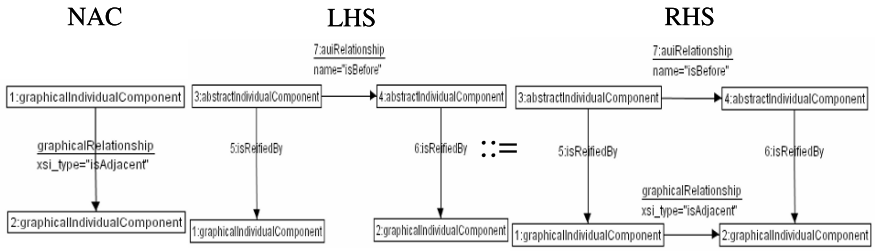

Figure 6.15 A placement of graphical individual components derived from spatiotemporal relationships at the abstract level

– **Definition of navigation**. Container transition relationships are transformed into navigation relationships. Ad hoc navigation objects may be created for this purpose (e.g., a menu, a tabbed dialog box bar may be created).

Example 8 is composed of rule 10. It generates a button to enable navigating between two windows.

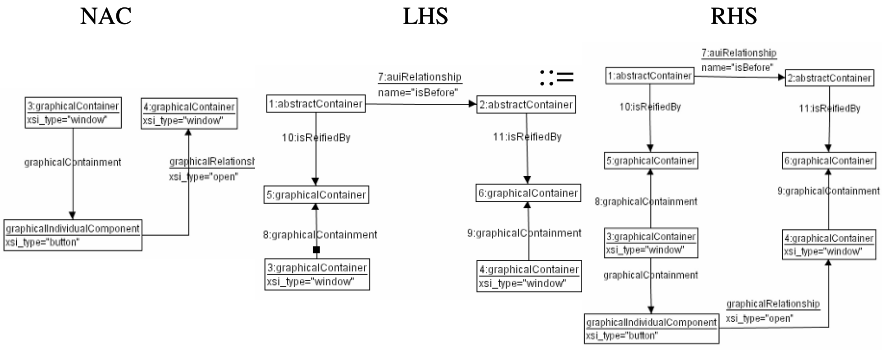

Figure 6.16 A window navigation definition derived from spatiotemporal relationships at the abstract level

- Rule 10 (Figure 6.16): For each container related to another container belonging to different windows and their respective abstract container related by a "is before relationship", generate a navigation button in source container pointing to the window of target container.

From Concrete User Interface to Code. Step T3 consists of code generation from a CUI. Code generation techniques for UI have been surveyed in various domains such as generative programming and model to code approach in Visitor-based approach and template based approach (Czarnecki and Eisenecker, 2000).

6.4.2 Reverse Engineering

As shown in Figure 6.17, the starting point of UI reverse engineering is the UI code. This code is analyzed and transformed into a higher level representation i.e., a concrete UI. From this CUI model, an AUI and, finally, a task and domain model are retrieved.

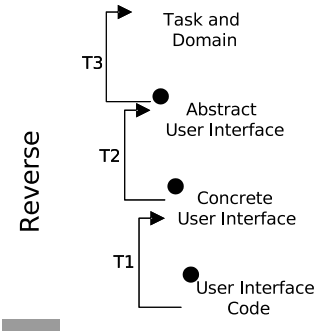

Figure 6.17 Reverse transformational development of UIs

From Code to Concrete User Interface . A state of the art in reverse engineering of UIs can be found in Bouillon et al., (2004) expressed according to the IEEE Terminology (Chikofsky and Cross, 1990). Transition T1 is notably supported by ReversiXML (Bouillon, Vanderdonckt, and Chieu, 2004), an on-line tool functioning as a module of an Apache server which performs reverse engineering into UsiXML. It takes as input a static HTML page, a configuration file containing a set of user-defined options, and produces a concrete and/or abstract UI.

From Concrete User Interface to Abstract User Interface. Transition T2 consists of deriving a more abstract UI specification from a concrete one. This derivation is trivial because the source model holds more information than the target model. Nevertheless, several development substeps may be identified: abstraction of CIO into AIO, abstraction of arrangement relationships, abstraction of navigation, etc. *Example 9 is composed of rule 11. It consists of obtaining an abstract individual component equipped with an input facet.*

- Rule 11 (Figure 6.18): For each editable graphical individual component create an abstract individual component equipped with an input facet.

From Abstract User Interface To Task & Domain. Transition T3 is the derivation of a task and concept specification. This transition has been considered very extensively in the area of reverse engineering where several techniques exist that contribute to recover a design model from existing information such as code. Indeed, the conceptual gap between AUI level and task and domain level is so important that little information may be extracted from an AUI model to retrieve a task or domain specification. Static analysis of Web pages examines the code of a Web page without

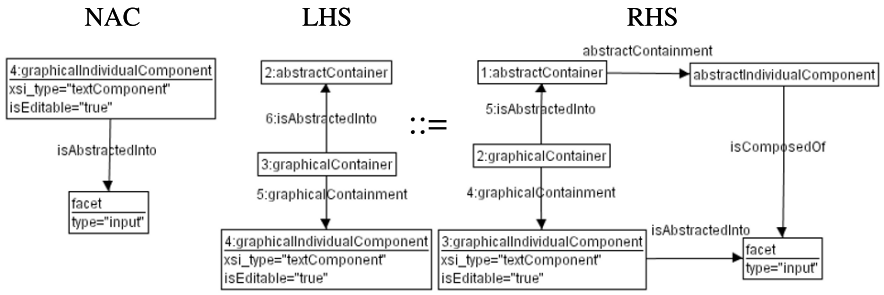

Figure 6.18 Creation of a facet at the abstract level derived from a type analysis of graphical individual components

interpreting or executing it in order to understand aspects of the website. Since static analysis has been successfully used in software testing and compiler optimization, it has been extensively used for analyzing the HTML code of Web pages. However, this technique leaves untreated all non-HTML parts of the Web page. Therefore other techniques need to be investigated such as the following methods. *Pattern matching* parses the code of a Web page to build a manipulable representation of it. Then slicing techniques are used to extract interface fragments from this representation, and a pattern matcher identifies syntactic patterns in the fragments. Using the code fragments as a basis, details about modes of interaction and conditions of activation are identified with control flow analysis. *Syntactic Analysis and Grouping* relies on a recognition algorithm that identifies input/output statements and attempts to incorporate them into groups. The grouping information is then used to define screens from the original user interface. This is particularly appropriate for scripting languages. *Cliché and Plan recognition* automatically identify occurrences of clichés, stereotyped code fragments for algorithms and data structures. The cliché recognition system translates the original code into a plan calculus, which is then encoded into a flow graph, producing a language-independent representation of the interpretation's flow that neutralizes syntactic variations in the code.
Example 10 is composed of rule 12. This example derives information on task action type from the abstract UI level.

- Rule 12 (Figure 6.19): For each abstract individual component equipped with a navigation facet create a task of action type "start/go" on an item of type "element".

6.4.3 Context of Use Adaptation

Context adaptation (illustrated in Figure 6.20) covers model transformations adapting a viewpoint to another context of use. This adaptation may be done at different levels.

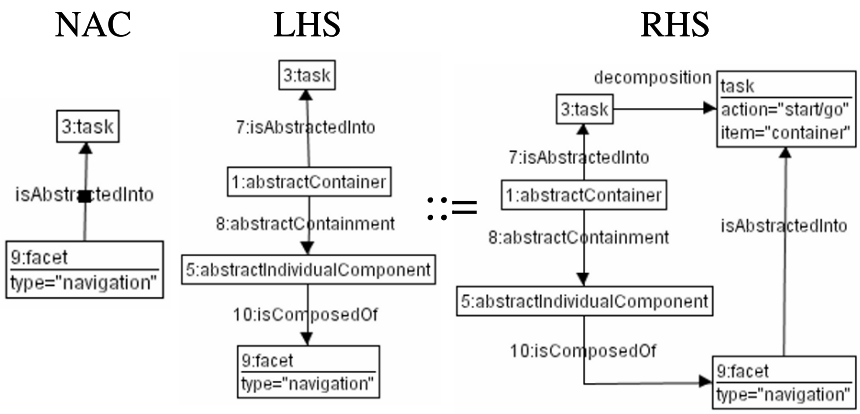

Figure 6.19 Definition of task action types derived from an analysis of facets at the abstract level

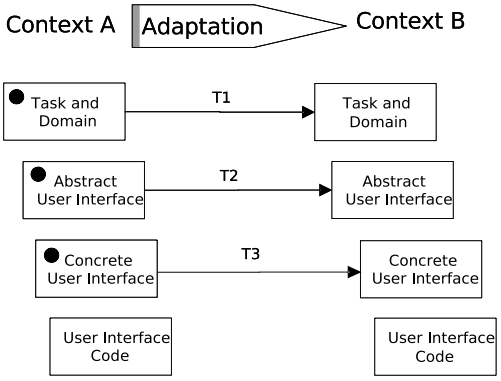

Figure 6.20 Context adaptation at different levels of our framework

From Task & Domain to Task & Domain. We propose one development substep type to exemplify adaptation at T1 level (Figure 6.20): Transformation of a task model.

- **Transformation of a task model:** Transformation of a task model may be useful to adapt a task specification to different categories of users, to different environments. For instance, an expert user needs less structuring in the accomplishment of a task than a novice user. This has an influence on the relationships between tasks. Another example is the management of user's permissions. Some users may not be allowed to perform certain tasks (e.g.,., editing a document), transformation rules may be defined to adapt a task specification to these constraints.

Example 11 is a transformation system composed of rule 13 and rule 14. A task hierarchy is "flattened" to allow an (expert) user to perform all tasks at the same time (i.e., concurrently).

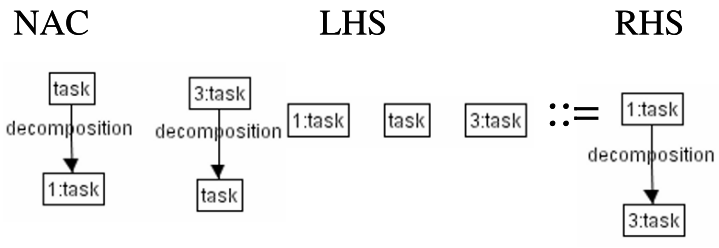

Figure 6.21 Flattening of a task tree structure

- Rule 13 (Figure 6.21): This rule (1) erases each intermediary task (i.e., non-leaf and non-root tasks) and (2) attaches every leaf task to the root.
- Rule 14 (Figure 6.22): For each sister tasks change their temporal relationship into concurrent.

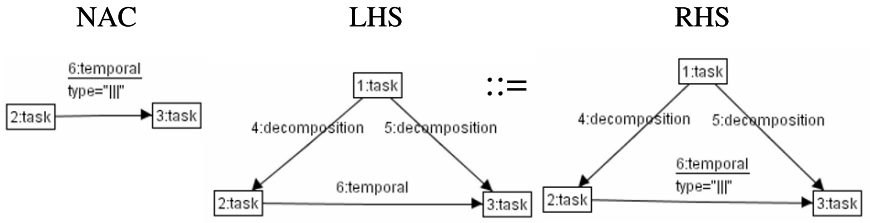

Figure 6.22 Transforming all temporal relationship to concurrent

From Abstract User Interface to Abstract User Interface Adaptation at this level. Adaptation at the abstract level concerns abstract container reshuffling and abstract individual component modification (e.g., facet modification, facet splitting, facet merging). We propose an example of abstract individual component modification.

- **Abstract individual component facet modification:** A modification of an abstract individual component affects its facets in their specification (e.g., an input facet is mapped onto a different domain concept) or their structuring (e.g., a facet is transferred onto another abstract component, a facet is erased).

Example 12 is a transformation system containing rules 15 and 16. It merges the facets of two abstract individual components mapped onto concurrent tasks. This example is

based on the assumption that the tasks of a system must be concentrated into a lesser number of abstract components. This means that concrete components resulting from the abstract specification will have to assume more 'functionalities' than in the source version of the specification.

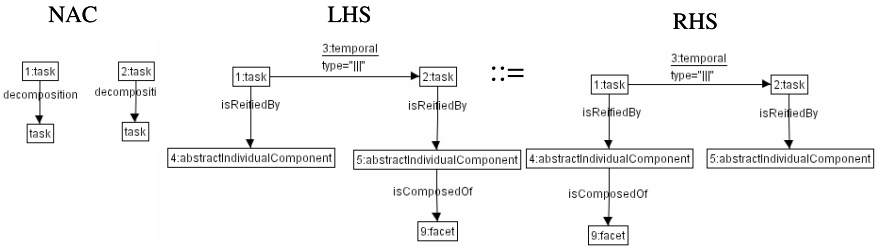

Figure 6.23 A merging of facets of abstract individual components

- Rule 15 (Figure 6.23): For each pair of abstract individual components mapped onto concurrent tasks. Transfer all facets of the abstract individual component that is mapped onto the task that is target of the concurrency relationship, to the other abstract individual component.
- Rule 16 (Figure 6.24): Erase all abstract individual components that have no facets left.

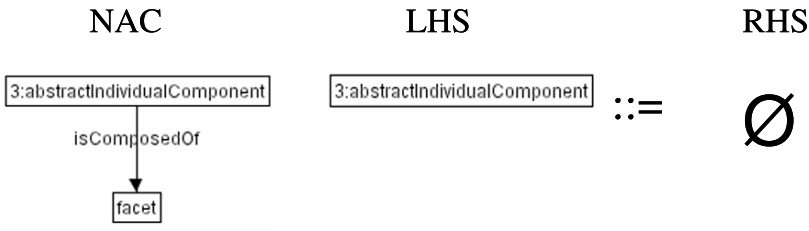

Figure 6.24 Erasing abstract individual components with no facets left

From Concrete User Interface to Concrete User Interface. Adaptation at the concrete level consist of several development substeps like container type modification (called concrete container reformation), modification of the types of concrete individual components (called concrete individual components reselection), layout modification (layout reshuffling), or navigation redefinition. We provide hereafter examples for these first three adaptation types.

- **Concrete container reformation:** Concrete container reformation may cover situations like container type transformation (e.g., a window is transformed into

a tabbed dialog box) or container system modification (e.g., a system of windows are merged into a single window).

Example 13 is a transformation system composed of rules 17, 18 and 19. This transformation adapts a window into a tabbed dialog box and transfers window content into several "tabbed items".

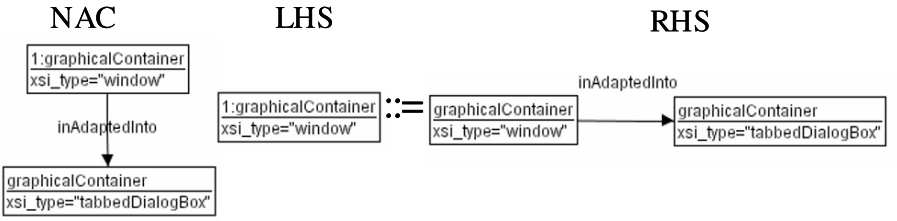

Figure 6.25 Initializing of the adaptation process by creating graphical component to adapt into

- Rule 17 (Figure 6.25): Each window is selected and mapped onto a newly created tabbed dialog box.
- Rule 18 (Figure 6.26): Transfers every first level box of the window to adapt into tabbed item composing a tabbed dialog box.
- Rule 19 (Figure 6.27): Cleans up the specification of remaining empty main boxes.

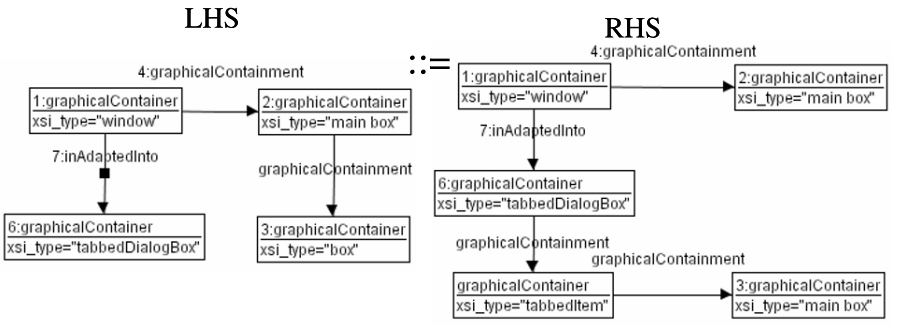

Figure 6.26 Creation of tabbed item and transfer of the content of the adapted window

- **Concrete individual component reselection:** Reselection transformations adapt individual component into other individual components. This covers individual component merging or slitting, or replacement.

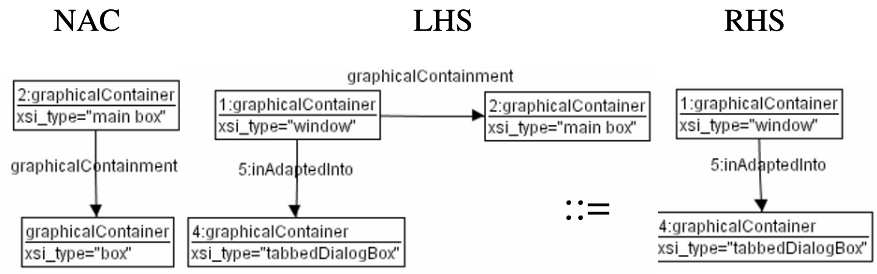

Figure 6.27 Deletion of unnecessary containers

Example 14 is composed of rule 20. It merges a non-editable text component (i.e., a label) and its adjacent editable text component into one editable text component. The content of the non-editable text component is transferred into the editable text component

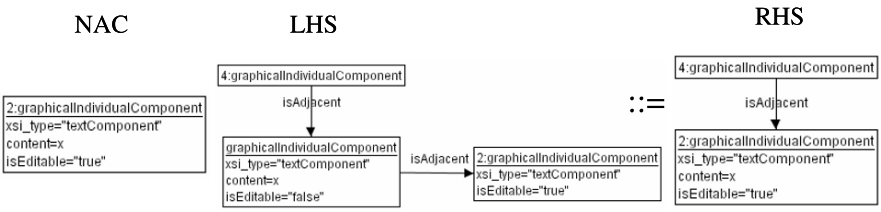

Figure 6.28 Merging of a non-editable text component (e.g., a label) and an editable text component (e.g., an input field) into one single editable text component

- Rule 20 (Figure 6.28): For each couple of adjacent editable text component and non-editable text component. Erase the editable text component and transfer its content into the non-editable text component (unless some contents have already been transferred).
- Rule 21(Figure 6.29): Each box is transformed into a vertical box and every individual component is glued to left.
  - **Layout reshuffling:** A layout at the concrete level is specified with horizontal and vertical boxes. An element contained in a box may be glued to an edge of this box. Any transformation modifying this structuring is categorized as layout reshuffling transformation.

Example 15 is composed of rule 21. It squeezes all boxes of a UI.

1. **Alternate and Composed development paths**

Other development paths could be equally expressed depending on their entry and exit points. Some of them are partially supported by various tools based on the UsiXML language (Vanderdonckt, 2005).

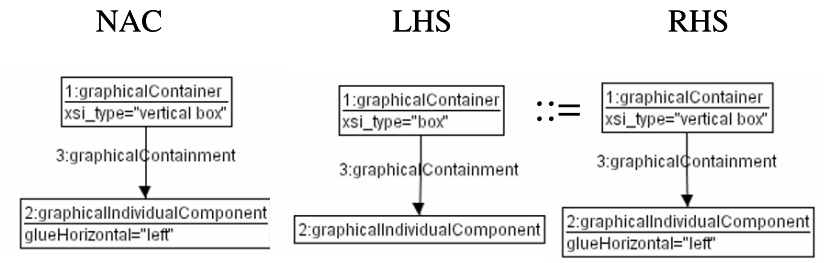

Figure 6.29 Squeezing of a layout structure to display vertically

- **Retargeting**: This transition is useful in processes where an existing system should be retargeted, that is, migrated from one source computing platform to another target computing platform that poses different constraints. Retargeting is a composition of reverse engineering, context adaptation, and forward engineering. In other words a UI code is abstracted away into a CUI (or an AUI). This CUI (or AUI) is reshuffled according to specific adaptation heuristics. From this reshuffled CUI (or AUI) a new interface code is created along a forward engineering process.

- **Middle-out development**: This term coined by Luo (1995) refers to a situation where a developer starts a development by a specification of the UI (no task or concept specification is priorly built). Several contributions have shown that, in reality, a development cycle is rarely sequential and even rarely begins by a task and domain specification. Literature in rapid prototyping converges with similar observations. Middle-out development shows a development path starting in the middle of the development cycle e.g., by the creation of a CUI or AUI model. After several iterations at this level (more likely until customer's satisfaction is reached) a specification is reverse engineered. From this specification the forward engineering path is followed.

- **Widespread development** (Hartson and Hix, 1989): In this development path, the designer may start wherever she wants (e.g., at any level of the development process), perform the rules that are relevant at this level, evaluate the results provided by these rules and proceed to other development steps as appropriate. This is a somewhat extreme position where everything is open and flexible, perhaps somewhat too much.

- **Round-trip engineering** (Demeyer et al. 1999): This development path is unique in the sense that it is a genuine path, but not directly for development. It results from applying manual modifications to code which has been generated automatically, after a model-to-code transformation. If a manual change has been operated on some piece of code generated automatically, then this change will be lost the next time a model-to-code transformation is applied. In order not to lose this effort, it is desirable to propagate the manual change into an abstraction which is relevant to the CUI.

6.5 CONCLUSION

In this chapter, a method has been introduced and defined that supports multiple paths in the domain of development of UIs. These paths are basically expressed on three types of transformation (i.e., abstraction, reification, and translation) so that any development path, consisting of development steps, can be supported by a transformational approach by combining transformations of the three types. To uniformly and consistently apply the transformations in a rigorous framework, graph grammars and graph transformations have been exploited. Correctness of transformations is an issue that may emerge when talking about model transformation. Two types of correctness may be considered.

Syntactic (structural) correctness stipulates that for any well-formed source model, and transformation rule enabled to provide a well-formed target model. While semantic correctness is hard to prove, syntactic correctness is easily guaranteed within our framework by two essential elements: Model type checking and consistency checks mechanism. Graph type checking ensures that a given transformation will not be applied if the resulting model it produces violates the meta-model it is supposed to conform to. Deriving a model to another may endanger consistency between different representations. For this purpose some basic consistency rules can be expressed with the technique of graph consistence rules. A graph of types may also be accompanied with the expression of specific consistency constraints inexpressible within the graph of types. OCL is used for this purpose in Agrawal et al. (2003), pre and post-condition with graph patterns (Akehurst et al., 2003).

Semantic correctness stipulates a semantic adequacy between a source and a target model. In our context, semantic correctness proving is hard to consider as by definition the domain of discourse of source model and target model are different.

Other important properties of interest that denote the powerfulness and the limitations of our method can be discussed equally.

Incompleteness of the method. There are few criteria to judge the quality of the method. It is also impossible to prove that a general solution is optimal. We can only prove sometimes formally, sometimes informally that a solution meets several quality criterias.

Seamlessness. This is a quality attribute attached to certain methodologies in the field of software engineering. It qualifies a small gap between concepts used at the analysis level and concepts relevant to implementation. Graph grammars, as used in this work, contribute to reach seamlessness of our method as manipulated structures from the requirements analysis to the design are graphs. Furthermore, the knowledge used to perform development steps are graphs.

Traceability. The identification and documentation of derivation paths (upward) and allocation or flow down paths (downward) of work products in the work product hierarchy. Important kinds of traceability include: To or from external sources to or from system requirements; to or from system requirements to or from lowest level requirements; to or from requirements to or from design; to or from design to or from implementation; to or from implementation to test; and to or from requirements to test (IEEE, 1998).

Consistency The degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component (IEEE, 1998).
Iterative development cycle. Iteration is well supported as graph productions supporting transitions in the development cycle may be undone to retrieve the source artifact as it was before transformation. This artifact may be modified by a developer and reused as source of a new derivation.

Last, and although empirical studies have already proven the advantages of using an MDA-driven approach for the development of software applications (Bettin, 2002), specific metrics should be precisely defined and applied to determine the effort and quality of the models and code obtained by using on the one hand any UI methodology on its own and on the other hand such methodologies like the one introduced in this chapter. A comparative analysis of several projects conducted through a traditional development method and through an MDA-driven method like the one presented here represents a huge amount of work, but would certainly be very interesting to establish.

Acknowledgments

The authors acknowledge the support of the CAMELEON European project and the SIMILAR network of excellence (`www.similar.cc`) on multimodal interfaces funded by European Commission.

References

Agrawal, A., Karsai, G., and Lédeczi, A. (2003). An end-to-end domain-driven software development framework. In Crocker, R. and Steele, G. L. Jr., editors, *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 8–15. ACM.

Akehurst, D. H., Kent, S., and Patrascoiu, O. (2003). A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239. On-line: http://www.cs.kent.ac.uk/pubs/2003/1764.

Bettin, J. (2002). Measuring the potential of domain-specific modeling techniques. In *Proceedings of the 2nd Domain-Specific Modeling Languages Workshop, Working Papers W-334, Helsinki School of Economics*, pages 39–44.

Boocock, P. (2003). The Jamda project. http://jamda.sourceforge.net/.

Bodart, F., Hennebert, A. M., Leheureux, J. M. and Vanderdonckt, J. (1995). A model-based approach to presentation: A continuum from task analysis to prototype. In F. Bodart, *Focus on Computer Graphics Series*, p. 77–94. New York: Springer-Verlag.

Bouillon, L., Vanderdonckt, J., and Chieu, K. (2004). Flexible reengineering of websites. In *Proceedings of the* 8^{th} *International Conference on Intelligent User Interfaces*, Multiplatform interfaces, pages 132–139.

Brown, J. (1997). Exploring human-computer interaction and software engineering methodologies for the creation of interactive software. *ACM SIGCHI Bulletin*, 29(1):32–35.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. (2003). A unifying reference framework for multitarget user interfaces. *Interacting with Computers*, 15(3):289–308.

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, Readign: MA.

Demeyer, S., Ducasse, S., and Tichelaar, S. (1999), Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering, *In Proceedings UML '99 (The Second International Conference on The Unified Modeling Language)*, Kaiserslautern, Germany, pages 630–644.

Depke, R., Heckel, R., and Küster, J. M. (2002). Formal agent-oriented modeling with UML and graph transformation. *Science of Computer Programming*, 44(2):229–252.

D'Souza, D. F. and Wills, A. C. (1999). *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Readign: MA.

Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (1999). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. Singapore: World Scientific.

Eisenstein, J., Vanderdonckt, J., and Puerta, A. (2001). Applying model-based techniques to the development of UIs for mobile computers. In *Proceedings of the 2001 International Conference on Intelligent User Interfaces*, pages 69–76, New York. ACM Press.

Freund, R., Haberstroh, B., and Stary, C. (1992). Applying graph grammars for task-oriented user interface development. In Koczkodaj, W. W., Lauer, P. E., and Toptsis, A. A., editors, *Computing and Information - ICCI'92, Fourth International Conference on Computing and Information, Toronto, Ontario, Canada, May 28-30, 1992, Proceedings*, pages 389–392, IEEE Computer Society.

Gerber, A., Lawley, M., Raymond, K., Steel, J., and Wood, A. (2002). Transformation: The missing link of MDA. In Corradini, A., Ehrig, H., Kreowski, H.-J., and Rozenberg, G., editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer.

Griffiths, T., Barclay, P. J., Paton, N. W., McKirdy, J., Kennedy, J. B., Gray, P. D., Cooper, R., Goble, C. A., and Silva, P. P. (2001). Teallach: a model-based user interface development environment for object databases. *Interacting with Computers*, 14(1):31–68.

Hartson, H. R. and Hix, D. (1989). Toward empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies*, 31(4):477–494.

Heckel, R., Mens, T., and Wermelinger, M. (2002). Workshop on software evolution through transformations: Towards uniform support throughout the software lifecycle. In Corradini, A., Ehrig, H., Kreowski, H., and Rozenberg, G., editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, Oc-*

tober 7-12, 2002, Proceedings, volume 2505 of *Lecture Notes in Computer Science*, pages 450–454. Springer.

Ho, W. M., Jézéquel, J. M., Le Guennec, A., and Pennaneac'h, F. (1999). UMLAUT: An extendible UML transformation framework. In *14th IEEE International Conference on Automated Software Engineering*, pages 275–278. IEEE Computer Society Press.

IEEE, *IEEE 830: Recommended Practice for Software Requirements Specifications*. IEEE Computer Society Press.

Kuske, S., Gogolla, M., Kollmann, R., and Kreowski, H.-J. (2002). An integrated semantics for UML class, object, and state diagrams based on graph transformation. In Butler, M. and Sere, K., editors, *3rd Int. Conf. Integrated Formal Methods (IFM'02)*, pages 11–28, Springer-Verlag.

Larman, C. (2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Englewood Cliffs: Prentice Hall.

Limbourg, Q. and Vanderdonckt, J. (2003). Comparing task models for user interface design. In Diaper, D. and Stanton, N., editors, *The Handbook of Task Analysis for Human-Computer Interaction*, pages 135–154. Lawrence Erlbaum Associates.

Limbourg, Q. and Vanderdonckt, J. (2004a). Transformational development of user interfaces with graph transformations. In Jacob, R. J. K., Limbourg, Q., and Vanderdonckt, J., editors, *Proceedings of the* 5^{th} *International Conference on Computer-Aided Design of User Interfaces CADUI*, pages 105–118. Kluwer.

Limbourg, Q. and Vanderdonckt, J. (2004b). UsiXML: A user interface description language supporting mul-tiple levels of independence. In Matera, M. and Comai, S., editors, *Engineering Advanced Web Applications*, pages 325–338. Rinton Press.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and Lopez, V. (2005). UsiXML: a language sup-porting multipath development of user interfaces. In *Proc. of* 9^{th} *IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS'2004*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220, Springer-Verlag.

Luo, P. (1995). A human-computer collaboration paradigm for bridging design conceptualization and implementation. In Paternó, F., editor, *Design, Specification and Verification of Interactive Systems '94*, Focus on Computer Graphics, pages 129–147, Springer-Verlag. Proceedings of the Eurographics Workshop in Bocca di Magra, Italy, June 8 – 10, 1994.

Mens, T., Van Eetvelde, N., Janssens, D., and Demeyer, S. (2001). Formalising refactoring with graph transformations. *Fundamenta Informaticae*, 21:1001–1022.

Miller, J. and Mukerij, J. (2003). MDA guide version 1.0.1. On-line: www.omg.org.

Mori, G., Paternò, F., and Santoro, C. (2004). Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Software Eng*, 30(8):507–520.

Nanard, J. and Nanard, M. (1995). Hypertext design environments and the hypertext design process. *Communications of the ACM*, 38(8):49–56.

Object Management Group (2003). Common warehouse specification version 1.1, vol. 1. http://www.omg.org/docs/formal/03-03-02.pdf.

Puerta, A. R. (1996). The MECANO project: Comprehensive and integrated support for model-based interface development. In Vanderdonckt, J., editor, *Computer-Aided Design of User Interfaces I, Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces CADUI'96, June 5-7, 1996, Namur, Belgium*, pages 19–36. Presses Universitaires de Namur.

Puerta, A. R. (1997). A model-based interface development environment. *IEEE Software*, 14(4):40–47.

Queensland University (2002). An OMG Meta Object Facility Implementation. The Corba Service Product Manager, University of Queensland, 2002, On-line: `http://www.dstc.edu.au/Products/CORBA/MOF/`.

Rensik, A. (2003). Proceedings of the 1st Workshop on Model-Driven Architecture: Foundation and Application MDAFA'03. CTIT Technical Report TR-CTIT-03-27, University of Twente, Twente. On-line: http://trese.cs.utwente.nl/mdafa2003/.

Rozenberg, G. (1997). *Handbook on Graph Grammars and Computing by Graph Transformation 1 (Foundations)*. World Scientific, Singapore.

Sucrow, B. (1998). On integrating software-ergonomic aspects in the specification process of graphical user interfaces. *Transactions of the SDPS Journal of Integrated Design & Process Science*, 2(2):32–42.

Sumner, T., Bonnardel, B., and Harstad, B. (1997). The cognitive ergonomics of knowledge-based design support systems. In *presented at Proceedings of the Conference on Human Factors in Computing Systems (CHI'97)*, Atlanta, GA.

Thevenin, D. (2001). *Adaptation en interaction homme-machine : le cas de la plasticité*. Ph.D. thesis, Universite Joseph-Fourier - Grenoble I, France.

Trevisan, D. G., Vanderdonckt, J., and Macq, M. (2002). Analyzing interaction in augmented reality systems. In Pingali, G. and Jain, R., editors, *roceedings of ACM Multimedia 2002 International Workshop on Immersive Telepresence ITP'2002*, pages 56–59, ACM Press.

Vanderdonckt, J. (2005). A MDA-compliant environment for developing user interfaces of information systems. In Pastor, O. and Cunha, J. F., editors, *Proceedings of the 17th International Conference on Advanced Information Systems Engineering CAiSE 2005, Porto, Portugal, June 13-17*, volume 3520 of *Lecture Notes in Computer Science*, pages 16–31. Springer-Verlag.

Vanderdonckt, J. and Bodart, F. (1993). Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems, Amsterdam*, pages 424–429. ACM Press.

Varró, D., Varró, G., and Pataricza, A. (2002). Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91.