"Multi-Path Development of User Interfaces"
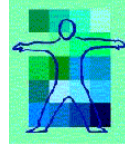
Limbourg, Quentin

Abstract

In software engineering transformational development is a paradigm consisting in the progressive refinement of abstract models into concrete models, until program code. This thesis applies transformational development concepts to User Interfaces (UIs). It enlarges the paradigm of transformational development by defining a methodology allowing the realization of various types of development paths (e.g., forward engineering, reverse engineering, context of use adaptation) in order to realize multi-path development, we propose an ontology of concepts defining various viewpoints that can be maintained on a UI system. Viewpoints are hierarchically structured depending on their level of abstraction. They describe user tasks, classes of objects, presentational and behavioral aspects of UIs, context of use, and a set of mappings between these representations. The underlying mathematical formalism of our ontology being a graph structure (directed, identified, labeled, constrained, and typed g...

Document type : *Thèse (Dissertation)*

Référence bibliographique

Limbourg, Quentin. *Multi-Path Development of User Interfaces.* Prom. : Vanderdonckt, Jean

Multi-Path Development of User Interfaces

By Quentin Limbourg

A dissertation submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy in
Management Sciences

of the Université catholique de Louvain

Committee in charge:

Prof. Jean Vanderdonckt, Advisor
Prof. Manuel Kolp, Examiner
Prof. Thierry Van den Berghe, ICHEC, Examiner
Prof. Joëlle Coutaz, Université J. Fourier, Reader
Prof. Oscar Pastor, Universidad Politécnica de Valencia, Reader

Autumn 2004

Table of Contents

To Nancy, Eloïse, those who left and those who come.

Acknowledgement

I would like to express my thanks to:

- My advisor, Professor Jean Vanderdonckt, for his constant support and enthusiasm regarding my work.

- Professors Joëlle Coutaz, Oscar Pastor, Manuel Kolp, and Thierry Van den Berghe for accepting to participate to the jury of this dissertation.

- My colleagues from IAG school of management at Université catholique de Louvain.

- My family and friends.

Abstract

In software engineering transformational development is a paradigm consisting in the progressive refinement of abstract models into concrete models, until program code. This thesis applies transformational development concepts to User Interfaces (UIs). It enlarges the paradigm of transformational development by defining a methodology allowing the realization of various types of development paths (e.g., forward engineering, reverse engineering, context of use adaptation) in a unique framework. Such methodology is referred to as *multi-path development*. In order to realize multi-path development, we propose an ontology of concepts defining various viewpoints that can be maintained on a UI system. Viewpoints are hierarchically structured depending on their level of abstraction. They describe user tasks, classes of objects, presentational and behavioral aspects of UIs, context of use, and a set of mappings between these representations. The underlying mathematical formalism of our ontology being a graph structure (directed, identified, labeled, constrained, and typed graphs), we transform one viewpoint into another by the application of conditional graph rewriting rules gathered in graph grammars. These enable us expressing a wide variety of transformational heuristics so as to be able to express multiple development paths. Ontology and transformations may be stored in an XML format called UsiXML (User interface eXtensible Markup Language) allowing the dissemination, the capitalization, and the consolidation of UI specifications and transformation catalogs.

Chapter 1   Introduction

This dissertation is located in the discipline of *Engineering for Human-Computer Interaction* (EHCI). This discipline is at the crossroad of two disciplines: *Software Engineering* (SE) and *Human-Computer Interaction* (HCI). SE can be defined as "the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software; that is, the application of engineering to software" [IEEE90]. HCI can be defined as "a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them" [Hewe96].

Interactive computing systems are computer systems allowing a certain level of control by a human agent. This control is operated through a *User Interface* (UI). A UI can be defined as any software and/or hardware piece allowing a user to communicate with a computer system. In other words, a UI is a software component, a hardware component, or a series of such components enabling a user to interact with an application so as to reach her task's goals. Typical user's goals are information retrieval, browsing, visualization, resource management in the large, process or automation control, etc.

1.1   User Interfaces in the Scope of the Software Crisis

The software crisis is a concept that emerged in the early seventies [Dijk72]. The development of hardware computing power and storage capacity has progressively pushed software systems to an unprecedented level of complexity. Software development became a slow, tedious, expensive and error prone activity. As failure stories accumulated in the history of software development [Meye97, Stan95], it was acknowledged that software had to be taken out of the hands of crafted developers and brought to the ones of engineers. Software systems because of their growing complexity could not be handled neither by a single

person anymore nor by a group of *cognoscenti*. All speaking the same language, the one of the machines! Software development had to become a team activity where communication and coordination takes, at least, as much time as coding.

The software crisis is still going on. Software development today is still closer to craft than to science. The quality of software artifacts produced intrinsically depends on the skills of developers, as the technique they use is rarely measurable or reproducible.

Many people dreamed about a computer discipline with a level of rigour that is usually found in "hard" sciences like mathematics, physics or chemistry. Since the early days of software crisis diagnosis, a lot of efforts have been devoted to make software projects more structured, predictable, and controllable. This is the role of the software engineering community.

Numerous research works are conducted in fields such as project management, requirements analysis, specification languages, software architectures, programming languages, verification methods and quality testing. These researches fostered results such as, but not limited to, management practices, methodological recommendations (e.g., software project management), formal methods (e.g., B specifications [Abri96]), Computer-Aided Software Engineering (CASE) tools (e.g., DB-MAIN [Engl99]), programming practices (e.g., use of patterns [Fowl96]) and concepts (e.g., abstract data types [Lizk74], design by contract [Meye97]).

Model engineering (i.e., a discipline that is concerned with the development of models) is part of the numerous solutions proposed to overcome the software crisis.

A *model* can be defined as an intentional and simplified representation of a real-world thing. Model primitives (i.e., model building blocks) are gathered in *meta-models* i.e., models describing other models' concepts and relationships. "Intentional" stresses that there is always some intent or goal behind the identification of abstractions populating a model. A model can, for instance, facilitate understanding, simulation, or testing. A model is said to be a simplification as it abstracts away details that do not seem relevant to the goal of the modeling activity. The same real-world artifact could consequently be modeled with different abstractions.

1. Introduction

The *viewpoint* concept characterizes the different perspectives that can be maintained about some real-world thing. Each viewpoint permits the construction of a peculiar view on the same real-world thing. The concept of *real-world thing* should not abuse the reader. A real-world thing is not necessarily something that has a material existence. A real-world thing refers to a thing, i.e., a separate and distinct individual quality, fact, idea or usually entity [Merr04]. In other words, a real-world thing is something that can be distinguished from other things.

Model-driven development is a development paradigm that relies on model engineering i.e., in the power of models to build computer systems. It advocates that software development should be guided as much as possible by the construction, and refinement of software models at various levels of abstraction. Most of current development methodologies have been influenced by, can be affiliated to, or are totally in debt with, this paradigm, for instance: object-oriented methodologies, database engineering, or agent-oriented methodologies.

Transformational development can be considered as a sub-paradigm of model-driven development. It presents the development of software as a progressive refinement of abstract models into concrete models, until program code [Somm99]. In order to do this, transformational development, it relies on catalogs of transformations able to (semi-)automatically perform model-to-model and model-to-code transformations. Transformational development has attracted the attention of the SE from the beginning (for instance: [Balze72, Chea81, Bend83]).

More recently, along with the Model Driven Architecture (MDA) proposal [Mill03], model processing and transformation have gained particular importance in the software engineering literature [Rens03, Kusk02, Gerb02, Mell03]. The main motivation of these works is to tackle the problem of computing platform heterogeneousness. For this purpose MDA defines a set of abstraction layers able to factor out specificities of implementation platforms. In this context, explicit model-to-model transformations enable the realization of the development process.

1.2   On Fragile Bridges between HCI and SE

During the last years, the area of HCI has been compared to the area of SE, the former being described as mainly empirical, experience-based, and relying on implicit knowledge as opposed to the latter being notoriously and deliberately structured, principle-based, and relying on explicit knowledge. The development life cycle of highly-interactive systems in general and of their UI in particular forms the cornerstone of HCI. It has been observed for suffering from several shortcomings that are intrinsic either to the type of interactive systems developed or to the adopted existing practices. For many years, there has been a dream of developing any UI in such a way to preserve the quality properties of the SE area. Among the criticisms addressed to HCI are the following observations:

- *Lack of rigour*: The development life cycle of interactive systems shared by HCI and SE does not involve the same level of rigour that is typically used in SE [Brow97]. In addition, HCI development life cycle is estimated to involve an order of complexity higher than those found in SE [Wegn97].

- *Lack of systematization*: as SE dreamed of a well-structured method for developing highly complex systems, so did HCI for developing interactive systems. However, the systematization, and the reproducibility found in SE methods cannot be transposed equally in HCI: the development life cycle remains inherently open, ill-defined, and highly iterative [Sumn97] as opposed to the domain of SE where it is structured, well-defined, and progressive [DSou99].

- *Lack of a principle-based approach*: where SE methodologies define system development as a succession of one stage after 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 stage [Puer97].

- *Lack of explicitness*: not only is the knowledge required to properly conduct the development life cycle of interactive systems is not as principled as in SE, but also is it 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, etc. [Szek96, Pate00]. Even more, when this knowledge is made more explicit, nothing can guarantee that it is applied uniformly and consistently within the same development project or across various development projects.

It can be observed that the above comparison holds as long as significant efforts towards structured, principle-based and explicitly-based process realized in SE remain unparalleled within the area of HCI.

On the other hand, it can be argued that software engineering has paid very little attention to the problem raised by UIs development. A user interface is an essential component of software systems. It determines how easily a user may control underlying functions of a computer program. A program equipped with powerful functionalities but with a poor UI has little value if its user audience is supposed to be large and varied. Numerous studies show the importance of a well designed UI system [Niel94, Sinh04]. But what is a well designed UI? How to build it? Unfortunately, SE methodologies rarely propose concepts and practices to achieve the development of UIs. UIs are treated as any pieces of software. And, from our opinion, they are not!

Although efforts have been undertaken since more than 10 years to bridge the gap between SE methods and HCI methods [Tarb93, Lim94, Balz95, Bart95, Bod95a, Boda95b, Robe98, Nune00, Sanc01], it can be said that SE methods rarely adopt a particular insight on an essential aspect to interactive systems development: the user.

User Centered Design [Cons99] was proposed as a development paradigm that focuses on the quality of interaction in the first place (referred as usability).

UCD represents both a collection of user-centric methods and more generally a philosophy for approaching technology design. UCD methods engage the user, its activities and its environment in all stages of an interactive application's analysis and design. UCD typically uses an extensive initial research phase, coupled with methods for conceptualizing users and its activities. UCD methods are particularly suited for constraining contexts of use, complex tasks or critical situations that require user characteristics, goals and environment to be well understood.

A community of research tries to combine the objectives of UCD with the ones of software engineering. This community has given birth to numerous research

1. Introduction

providing results, notably, in formal methods [Thimb90, Harr90, Dix91, Unge96, Pala97, Chat99, Pate00], quality inspection (i.e. usability engineering and evaluation methods), requirement analysis (i.e., user engineering), methodological recommendations (see methods cited above), software architecture, CASE tools, etc.

Model-Driven Development of User Interface (*MDDUI*) has emerged from this research stream as a field relying on models to guide the development of user interfaces. The use of models in HCI to reason on abstract properties of user interfaces has been a long tradition since [Parn67]. A lot of efforts have been made in applying formal methods to HCI (see references above), and identifying the abstractions that were appropriate to HCI with respect to software engineering. Similarly to what happened in SE, a portion of the scientific community tries to identify how user interface models can be automatically, or semi-automatically, refined to come closer to the implementation of the UI system itself.

Transformational development of user-interfaces (TDUI) is a sub-paradigm of model-driven development of UIs. By analogy with transformational development in SE, it defines the development of user interface systems as a successive application of transformations to an initial representation. This generally implies a progressive refinement of abstract models into concrete models, until program (here UI) code.

Since the mid-nineties, numerous engineering methods have been proposed to support transformational development of user interfaces. Most of them are concentrated on deriving UI code from abstract models, others are focused on recovering a model from a UI implementation. A more recent trend gave birth to methods specifically devoted to the adaptation of a UI system to multiple contexts of use.

Like for the MDA mentioned above, transformational development of UI finds its root motivations in the concept of *heterogeneousness*. In this case the heterogeneousness concerns the variety of contexts of use (referred as a triple <user, computing platform, physical environment> [Thev01]) for which a UI is designed. This heterogeneousness stresses the need for abstractions able to factor out details relevant to specific contexts. From these abstractions, it is possible to obtain context specific representations by progressive refinements. The advantage of accessing to such representations is to be able to reason on one single model and obtain many different UIs.

1. Introduction

Another strong motivation for transformational development of UI is to resist a constant pressure imposed on UIs: *change*. UIs is a component that has to rapidly evolve, this may be due, but not limited to, a change in the:

- *Organizational structure* [Brow97]: it may be a task redefinition, task reallocation among workers, redefinition of the organization structure, adaptation to a dynamic business environment, transfer of task from one user to another one.

- *Organizational process:* as a matter of fact, organizations react to changes in very different ways in their UI development processes. For instance, one organization starts by recovering existing input/output screens, by redrawing them and by completing the functional core when the new UI is validated by the customer (*bottom-up approach*). Another prefers to modify the requirement of the system and remaps it to screen design (*top-down approach*). A third one tends to apply in parallel all the required adaptations where they occur (*wide spreading approach*). A fourth one relies on an intermediate model and proceeds simultaneously to requirement models, and the screen design (*middle-out approach*) [Luo94].

- *Hardware platforms:* support of new computing platforms [Gaer03], migration from stationary platforms to mobile computing [Mori03], adaptation to dynamic environments [Luyt03].

- *Software platform:* change of the computing language, redesign due to obsolescence [Boui04].

- *User's requirements:* evolution of users with more demands, increasing need for more usable UIs, evolution of the domain of application [Agra03].

To address the challenges posed by the pressure of change, the existing development processes are not always considered as appropriate, as they do not reflect the implication of change throughout the complete application life cycle.

[Sumn97] emphasize the fact 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

1. Introduction

usually broadly incomplete, if not inconsistent), and mostly iterative (it seems impossible to conduct a development step in a way that its output is definitive). Nanard and Nanard [Nana95] report that the development life cycle 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 life cycle, which doesn't 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 stage to another.

Relying on model transformations to build a software product allows to better face change as these changes do not have to be understood in terms of implementation but in terms of abstract concepts.

A state of the art in transformational development of user interface reveals two families of shortcomings: ontological and methodological. We address these in the following sub-sections.

Ontological shortcomings

Ontological shortcomings concern the conceptual frameworks of the approaches defined so far.

A certain coarse-grain convergence in the concepts that are used to model UI may be observed. In most of the methods a domain and a task model are used as an expression of the requirements of the UI system being built. A domain model describes the objects of an application domain, a task model describes a logical and temporal ordering of tasks as performed by users in interaction with a system. Most of the surveyed methods are also equipped with a set of abstractions enabling a description of the UI itself. These abstractions enable a description of a UI appearance and behavior in a way that is independent of implementation details.

1. Introduction

Unfortunately, these similarities hide an important heterogeneousness in the way each of these models are defined and communicated to the method's user.
Four main shortcomings may be pointed out from these observations:

- *Lack of ontological explicitness* - A few methods define in an explicit manner their underlying concepts. Concepts are generally bounded to tools or methodological recommendations, thus preventing a designer to grasp the conceptual foundations of the methodology.

- *Lack of ontological rigour* - When a method explicitly defines its ontology, the preciseness of concepts definitions highly varies from one method to another. In addition, concepts are seldom formally expressed, especially the relationships between the ontological concepts.

- *Lack of ontological commitment* - The ontological commitment refers to a shared understanding of concepts among a scientific community. The fact that a few ontologies have been defined so far prevents convergence around a set of concepts.

- *Lack of communication of concepts* - Research teams tend to conduct their research and development on their own models. Conceptual consolidation across methods is difficult as cross-method understanding is a tedious and time-consuming activity, requiring the full understanding of each method and establishing correspondence between them. As a consequence, communication among researchers is made complex.

- *Lack of extensibility of concepts* – When available, the concepts manipulated by methods are hardly extensible. This prevents the adaptation of methodologies to cover new model concepts, notably, the ones related to new interaction modalities.

Methodological shortcomings

Methodological shortcomings concern the way existing approaches concretize transformational development with the definition of methodological stages, steps (i.e., transitions between stages), and transformation catalogs to perform these steps.

1. Introduction

- *Lack of methodological explicitness* - Existing approaches are seriously lacking of explicitness in the way they propose their catalog of transformations both to the designer and to researchers. The transformation catalogs are often implicitly maintained in the head of developers and designers and/or hard-coded in supporting software. Consequently, the transformational process proposed in the literature consists essentially in a black box.

- *Lack of methodological rigour* - When development steps and transformation catalogs are made explicit the preciseness of their expression is limited. We are not aware of any formally defined transformation catalog in the domain of HCI.

- *Lack of consistency in applying methodology* - When such design knowledge exists, it is generally not systematically, consistently and correctly applied throughout the project or across projects. Methodological steps remain open to interpretation while lack of methodological explicitness hampers any structured reasoning on the application of transformations.

- *Lack of communication of transformation catalogs* - Consequently to the lack of explicitness, the exchange of knowledge regarding transformation catalogs can be hardly achieved. Even when transformation catalogs are made explicit in tools, their heterogeneous formats prevent the reuse of transformations outside the context for which they were designed.

- *Lack of predictability of transformation* - The implicitness of transformations decreases the predictability of the transformation results. This causes a frequent reproach made to transformational development [Myers95,00].

- *Lack of modifiability of transformation catalogs* - Developing UIs is about making heuristic decisions in a vast design space. Transformations have consequently an inherent heuristic nature as they try to translate into algorithms part of these design decisions. Proposed methods offer very little possibilities to the designer to modify built-in heuristics: adding, deleting, modifying, reusing transformations is almost impossible.

- *Lack of flexibility in methodological steps.* Methods generally come with their models, their development steps. Due to the implicitness of their transformation formalism it is almost impossible to adapt the proposed

methodological steps to the designers' needs and the project context. Flexibility is a notorious requirement for user interface development methods [Brow97].

These shortcomings lead us to conclude that transformational development of user interfaces can be improved along several dimensions.

1.3   Thesis

Thesis statement

This dissertation addresses the shortcomings previously outlined for achieving transformation-driven development of user interface. This dissertation provides an:

(1) ontological framework based on an explicit and rigorous representation of concepts relevant to UI development.

(2) methodological framework based on the ontological framework previously introduced. This methodological framework introduces a new paradigm for UI development called **multi-path development of UIs** that is characterized by the following principles:

- *Transformation driven*: a development method is composed of development stages. A development step is a transition from one stage to another one. Development steps rely on explicit and rigorous transformation catalogs.

- *Multiple-path:* The context of development projects may involve variable arrangements of development steps. A development path refers to a particular arrangement of steps. Multi-path development refers to the capacity of a method to accommodate to various development paths.

Validation

Two kinds of validation are provided to assess the validity of this thesis. A theoretical validation confronts the methodological framework introduced by this thesis to the requirements identified after a state of the art of existing

transformation-driven development methods. A practical validation is provided by illustrating how the methodological framework can be instantiated on two case studies.

Scope

The scope of this thesis is delineated by the following statements:

- We focus on a specific kind of software i.e., information systems. Information systems is "a means of recording and communicating information to satisfy the requirements of all users, the business activities they are engaged in and the objectives established for them"[Olle88].

- We consider that the typical user interfaces of information systems are either:

   - Graphical User Interfaces (GUI), which are 2-dimensional and based on widgets that belong to standard toolkits and window managers.
   - Vocal User Interfaces (VUI), which are UIs exploiting the auditory channel by standard speech synthesizers and voice recognizers.

- We target this dissertation primarily to the research community and those persons who define development methodologies in organizations.

1.4 Reading Map

In addition to the introduction and the conclusion, this dissertation is organized in four chapters.

Chapter 2 reports on some significant pieces of work related to the paradigm of transformational development of user interfaces. We survey in this chapter more than 20 different approaches and try to identify and compare their conceptual content along with their transformational development process. A set of observations and shortcomings is raised in conclusion of a comparative analysis. From these observations, we establish a list of requirements for addressing the observed shortcomings. This list of requirements will help us to assess the appropriateness of our solution.

1. Introduction

Chapter 3 introduces the reference representations that are used throughout this work. This addresses the principles of expressiveness and rigour of models identified above. In this chapter, we rely on conceptual schemas to provide a view of each important abstraction populating our framework. We present a structuring of concepts in viewpoints, capturing various levels of abstraction that can be maintained on a UI. After that, we present the abstract syntax that has been used to represent our concepts, namely: directed, identified, labeled, and typed graphs. Finally, we present two concrete syntaxes (i.e., graphical and textual) used to represent our concepts.

Chapter 4 shows how transformations are represented and executed thanks to conditional graph rewriting and graph grammars. In a first part, a theoretical explanation on the formalism and the way we exploit it is provided. An illustration of the graphical syntax used to represent transformations allows the reader to quickly understand the examples provided in the rest of the chapter. An application of this formalism is then presented by proposing several types of development paths with graph transformations i.e., forward engineering, reverse engineering, context of use adaptation. Finally, we expose the type of tool support that has been realized to achieve multi-path transformational development of user interfaces.

Chapter 5 illustrates the principles of multi-paths transformational development for two case studies. The first one concerns the development of an on-line polling system. The second one concerns the development of a virtual travel agent. We conclude this chapter by an evaluation of the two case studies.

Chapter 6 concludes by discussing the appropriateness of the solution proposed in this dissertation. Our contributions are summarized and by future works are proposed.

Chapter 2   State of the Art

2.1 Current Approaches in User Interface Development

Two elements define the smallest common denominator of what a UI is:

1. A *presentation* - concerns the physical description of the UI. It consists of a static representation using available interactors as building blocks. Without any presentation, a UI has no appearance (or "look").

2. A *dialog* - concerns the dynamic behavior of the presentation elements. It describes the input/output flows between a user and an interactive application (mediated by the UI). Without any dialog specification, a UI has no behavior (or "feel").

Various approaches to build these two UI elements have been reported in the literature as well as experienced by practitioners. To characterize these approaches, we rely on three starting points for initiating UI construction, as defined in the Diane methodology [Bart88] (Fig. 2-1):

1. The *internal view* – relates to the UI implementation and its description as it is relevant for the UI developer.

2. The *external view* – relates to the interface appearance and its behavior, as perceived by the end user

3. The *conceptual view* – provides an insight on the logical structure underlying a UI in designer's terms. A conceptual view provides the designer with a set of abstract concepts facilitating reasoning on the artifact that is being

built (e.g., a finite state machine, a class diagram).

These three views define three possible points where the process of UI construction can be initiated. All possible transitions between these representations enable a definition of nine theoretical approaches for constructing the UI can be identified (Fig. 2-1).
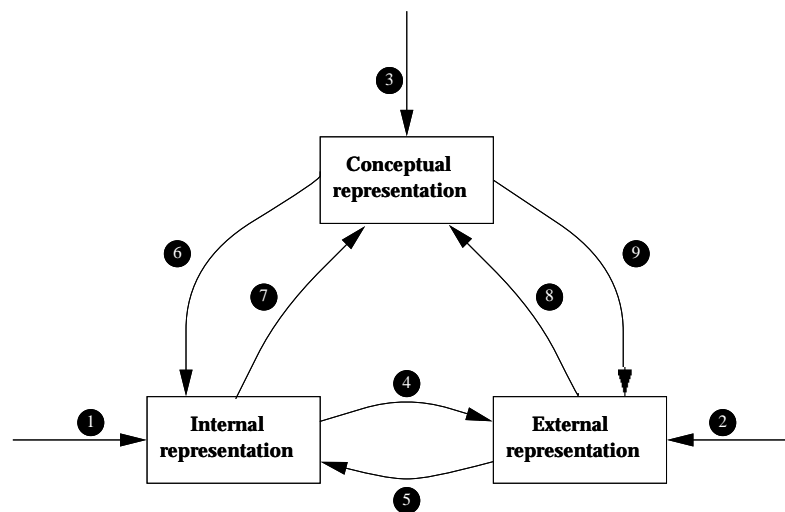
Figure 2-1 A framework for classifying approaches of UI development practices (adapted from [Bart88])

- A *programmatic approach* (**transition 1**). In this approach, the internal representation is obtained by directly coding the UI in its target computer language, e.g., HTML for a markup language or Basic, Pascal for imperative languages and Java as object-oriented language. Theoretically, a UI can be coded with any of these languages. Practically, some languages provide designers with a better support by, offering sets of pre-defined components especially tailored for UI construction. Several development transitions can be defined when starting from an internal representation:

  - An *internal-external generation* approach (**transition 4**) - derives an external representation from an internal representation (e.g., interpretation of HTML code and its rendering within a computing platform).

  - An *internal-conceptual derivation* approach (**transition 7**) - derives from the internal representation a conceptual representation (e.g., reverse

engineering HTML code to obtain an abstract view of its presentation, reverse engineering of a domain model from a form displayed. A state of UI reverse engineering can be found in [Boui04]).

- An *exploratory approach* (**transition 2**). In this approach, a developer firstly provides an external representation of the UI (e.g., with a graphical editor like those found in Integrated Development Environments like Visual Basic, or Visual C++, or a mock-up produced by a drawing tool such as Microsoft Visio).

  – An *external-internal representation* approach (**transition 5**) – derives an internal representation from an external representation (e.g., code generation from forms built in Visual Basic editor).

  – An *external conceptual* (**transition 8**) derivation - derives a conceptual representation from an external representation (e.g., Cellest tool [Elra01] reverse engineers an abstract specification of the presentation from screen dumps of a UI).

- A *specification-based approach* (**transition 3**). This development approach starts with an abstract representation of a UI (i.e., any UI model).

  – A *conceptual-external generation* approach (**transition 9**) – derives an external representation from the conceptual representation. For instance, Genova [Geno04] produces a UI preview (an external representation) before generation based on a selection of relevant information from a class diagram (a conceptual representation)

  – A *conceptual-internal generation* approach (**transition 6**) - derives an internal representation from the conceptual representation. In this case, the tool directly produces the code from the conceptual representation without any intermediate representation. For instance, MacIDA directly generates MacApp code from its Activity Chaining Graph that serves as a conceptual representation [Boda95]

As all these approaches are relevant to our domain of research, these development approaches and the concepts they manipulate are detailed in the next subsections.

2.2 Exploratory Approach

The exploratory approach consists of initiating a UI development by a graphical representation. Two major trends are clearly identified in the exploratory approach: a mockup approach and a visual programming approach.

2.2.1 Mock-up Approach

Exploratory approach is often used in consulting companies to quickly produce a working draft to convince a potential client at low cost and to validate the mockup with the customer so as to begin coding as soon as possible. This approach finds its extreme in paper prototyping where the UI is drawn on pieces of paper and post-it that are manipulated by a person to animate the behavior of the UI depending on the customer's requests. An exploratory approach consists of either a hand-drawing or a mock-up constructing. For the first category, general purpose drawing software (e.g., Corel Draw, Microsoft Powerpoint, Aldus Persuasion) can be used. Software dedicated to industrial drawing (e.g., Microsoft Visio, ABC Flowchart) can be preferred as they already encompass the vectorial drawing of the widgets composing a UI. Even more, DEMAIS [Bail03] already supports main graphical mechanisms fro expressing presentation constructs and dialog transitions for a simple multimedia application. For the second category, sketching tools like Silk [Land96], Denim [Hong01] or JavaSketchit [Caet02] can be used. A state of of the art on sketching tools can be found in [Coye04].

The Denim tool [Hong01, Newm03] is such a tool that allow designers to draw a sketch of the presentation of any web page and then to link these sketches together with arrows (Fig. 2-2). The transition between presentation and navigation is smoothly ensured thanks to a zoom visualization. The tool never recognizes however the layout or the widgets as Silk or JavaSketchIt do. This intuitive and simplistic tool allows the developer sketching a future UI without being interfered by low level details nor be distracted by physical attributes. Similarly in FreeForm [Plim04], the developer can then mock-up screen by screen the future application. Part of the dynamic behavior (e.g., window transitions) of the UI can be added to simulate UI state transitions.
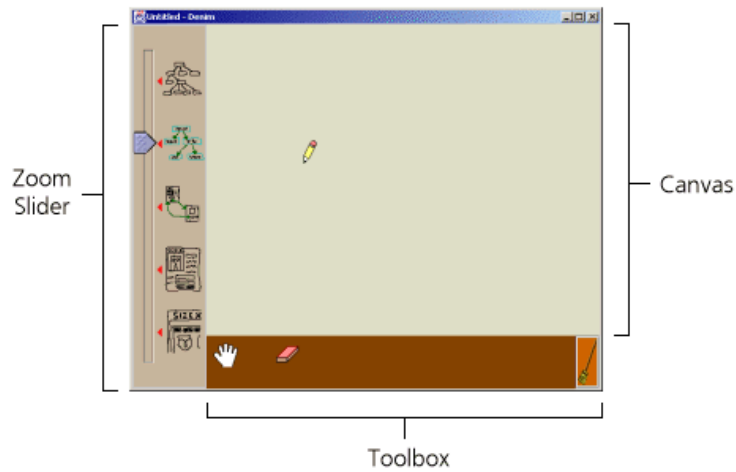
Figure 2-2 The Denim tool

The mock-up approach is adequate for preliminary studies. It is very suggestive, especially for the users. But when it comes to turn a mock-up into a shippable package, part of the effort might be lost when there no recognition of the mock-up. Indeed, if no computer support is provided to get an internal representation, a developer has to completely re-implement the previously done job in a genuine programming language with no guarantee of consistency with what was sketched before.

2.2.2 Visual Programming

Visual programming is the most popular way to construct a UI. Most of the major programming languages or toolkits possess their proprietary visual environment. Visual specification systems allow the developer to build a UI by combining two types of aspects:

1. A visual aspect by direct manipulation of widgets: each widget is dragged from a palette of widgets and dropped onto a working area. As such, visual programming may be used for constructing mock-ups.

2. A programming aspect of the application underlying a UI. Callbacks procedures are programmed in a high-level programming language or a scripting language.
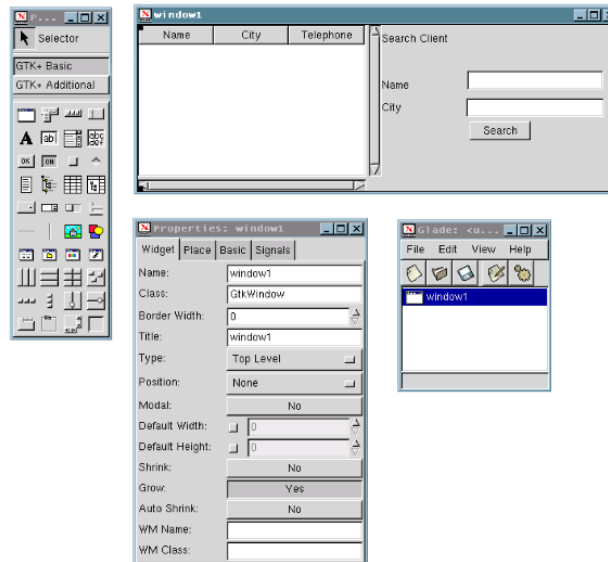
Figure 2-3 Visual programming with Glade environment

Fig. 2-3 shows the Glade environment [Glad04], the visual environment for GTK+ [GTK04]. This environment is rather typical of all UI builders. In the top left corner, a toolbox permits to select different sorts of widgets that the developer will place by direct manipulation on a working area (in the top right corner) depicting the UI. Properties such as graphical appearance or event they respond to can be configured in a property sheet (bottom left corner). The multiplicity of working spaces (windows) is managed by a project manager (bottom right corner).

Visual programming is typically based on UI toolkits. Visual programming a UI is extremely easy and natural as GUIs are visual by nature. It is very efficient for building simple interfaces. Visual programming increases developers' productivity but suffers from the same shortcomings as for programming approaches i.e., mainly, a risk of unstructured programming and a difficulty of reasoning on the properties of the artifact that is being built.

Programmatic Approach

Interface development practices have significantly evolved with programming languages development. Roughly, major steps of UI programmatic approach are: low level programming (i.e., assembly code), high level programming (e.g., C++), toolkit programming (e.g., Java Foundation Classes [SUN04,Flan99]), scripting, and use of mark-up language. Note that this classification is not a total partition. It is based on development practices that evolved with contingent factors such as technology constraints and application domain requirements.

2.2.3 Low Level Programming

Low level programming consists in providing instructions in machine or assembly language. Low-level programming for UI development is no longer a common practice today. It is still used where UI response time is critical (e.g., computer games programming, real-time applications). Low level programming dramatically reduces portability and reusability. It requires a high computing knowledge, advanced programming skills and is time consuming.

2.2.4 High Level Programming

High-level programming develops a UI faster than low level programming thanks to a set of human-understandable symbols replacing cryptic assembly or machine instructions. Interface programming using a high level language solely is a line-consuming activity. High-level programming tries to rely on reusable libraries containing drawing routines and/or graphical components. These libraries are often called *toolkits*.

UI portability is increased with high level languages. One could think that the interface could be executed on any platform. Unfortunately, this is not the case: interface systems strongly rely on peculiar OS services for rendering the interface. By themselves, high-level languages do not intrinsically support UI portability. Furthermore, they remain cryptic for the non-expert (and sometimes for the expert too!). Consequently, a developer spends a lot of time to solve implementation and his prevented to develop higher level reasoning on the

artifact that is being built.

2.2.5 Toolkit Programming

Toolkits are UI program libraries. They contain common widgets used to build the interface like input fields, buttons, menus, pre-defined dialog boxes, etc. They also provide support functions for manipulating widget like events and I/O handling. Thanks to toolkits, low level issues related to the widget manipulation can be disregarded by developers. Popular toolkits are Microsoft Foundation Classes [Feur97](MFC) for Windows operating system, Abstract Windowing Toolkit (AWT) [SUN04, Flan01] or Swing components [Ecks98] for Java Virtual Machine, GTK for Linux, Tk [Oust94] or Motif [Foun00] for Unix operating systems. Fig. 2-4 shows an 'HelloWorld' program using AWT toolkit frame object.

```
import java.awt.*;

public class Hello extends Frame {
  public static void main(String argv[])
  {
   new Hello();
  }

 Hello() {
 Label hello = new Label("Hello World");
 add("Center", hello);
 resize(200, 200);
 show();
 }
}
```

Figure 2-4 Toolkit programming provides high level constructs
(a Java/AWT excerpt for drawing a window)

The main advantage of toolkits is that they provide a great flexibility and an improved control over the UI elements while maintaining a relative ease of use. A problem with toolkits appears in tradeoffs between the number of features, the ease of use and the portability. For instance, it turns out that AWT is highly portable across various platforms but very poor in terms of number or variety of widgets and customization as it only supports the smallest common denominator of platforms. OSF Motif, on the other side, is extremely rich and customizable, yet poorly portable as it only works on top on Unix BSD4.2.

Toolkits require a high learning curve. Several months are generally needed to

master a specific toolkit. UI quality largely depends on the programmer's experience. As toolkits allow developing complex UIs without any constraint, the problem of "spaghetti callbacks" becomes predominant. Callbacks are calls to procedures notifying the application that a user action has been achieved. When the amount of callbacks between the application and the UI is increasing, their management becomes intractable if no structured approach is adopted.

2.2.6 Mark-up Languages

Mark-up languages [Luyt04] are at the fringe of programming approach and specification-based approach. Mark-up languages are declarative languages. They describe what a UI is rather than what to do to produce it. Mark-up languages are especially good at describing presentation elements of the interface and static properties such as widget layout, style characteristics,... The considerable success of mark-up languages for UI development is due to its ease of use. Initially designed for data, mark-up languages provide a raw description of the UI elements that can be interpreted by browsers compliant with the mark-up language. A survey of UI languages has been provided in [Cout02]. Fig. 2-5 gives an example of dialog described with the User Interface Mark-up Language (UIML) [Abra99,UIML04].

```
<UIML>
<HEAD>
  <AUTHOR>Hubert Lingot</AUTHOR>
  <DATE>July 16, 2001</DATE>
</HEAD>
<APP CLASS="App" NAME="DialogApp">
  <GROUP CLASS="Dialog"
  NAME="PrintFinishedDialog">
    <ELEM CLASS="DialogMessage"
    NAME="PrintFinishedMessage"/>
    <ELEM CLASS="Dialog Button"
    NAME="OKButton"/>
  </GROUP>
</APP>
<DEFINE NAME="OkButton"W
  <PROPERTIES>
  <ACTION
    VALUE="DialogApp.EXIST=false"
    TRIGGER="Selected"
  />
  </PROPERTIES>
</DEFINE>
</UIML>
```

Figure 2-5 A UIML Dialog Description Example

Mark-up languages are purely descriptive. A purely descriptive language is

generally insufficient for describing dynamic aspects (i.e., the behavior) of the interface. Mark-up languages are generally complemented with scripting languages. Mark-up languages are easy to understand, even for novice developers. They allow the developer to concentrate on the content of the UI rather than on presentation aspects. They are very resistant to bit-errors. These reasons explain the success of XML family languages. From a portability point of view, mark-up languages rely on platform specific implementation of programs called "renderers" (or sometimes "browser").

2.3 Specification-Based Approach

In software engineering, specification-based (or model-driven) approach relies in the power of models to construct and reason about software systems.

A model is a simplified and intentional view of real-world things. A model is a simplification as it withdraws details of real-world objects and tries to identify properties of interest of real-world objects. Identifying these properties requires some kind of judgment. That is why modeling is said to be an intentional activity. One never models for the sake of modeling. Real world concepts can be abstracted away in different ways. In other words, modeling is not a deterministic process resulting from observation of the real world.

The goal of specification-based, or *model-based approach*, for user interface development is to propose a set of abstractions, development processes and tools enabling a engineering approach of user interface development. The characteristics of an engineering approach are its systematic (development based of rational principles), its reproducibility, its orientation towards quality criteria.

Compared to programming, specifying a UI means to describe it at a higher level of abstraction which is independent of the implementation. As argued by [Schn98], the default form for specification in any field is the natural language. It holds inconvenience of being ambiguous, lengthy and vague. Furthermore natural language specifications are difficult to prove consistent, correct or complete. The specification approach uses some form of formal or semi-formal notation to describe a UI. It has the advantage of being very specific to the interface part to be described. It presents a disadvantage of being longer to learn but ensures abstraction.

We present specification-based approaches in three steps. Throughout Sec. 2.4.1 to Sec. 2.4.5, we provide an overview of different abstractions and models defined in the literature to achieve a specification-based development of UIs. In Sec 2.4.6, an overview of development methodologies and tools that are considered significant with respect to specification-based approaches is delivered. This overview is based on the abstractions and models introduced in Sec. 2.4.1 to 2.4.5. Sec. 2.4.7 concludes this discussion by providing a systematic comparison of these methodologies and their associated tools.

2.3.1 Abstractions

By examining the existing literature, abstractions related to the UI development can be categorized in three families:

- *Computing-independent abstractions* are abstractions enabling a UI description of the system to be built without any reference to the computing resources with which a UI will be implemented. Computing independent abstractions encompass task models and domain models.
- *UI focused abstractions* are abstractions enabling a UI description to be built while taking into account details of its design. For instance, a given modality, a particular computing-platform or a widget set. UI focused abstractions are gathered in two models: a presentation model and a dialog model.
- *Context of use abstractions* are abstractions concerning contextual information describing "situations" for which a system is designed. A context model or a user model contains such abstractions.

2.3.2 Task Model

User-Centered Design (UCD) has yielded many forms of design practices in which various characteristics of the context of use are considered. Among these, task analysis is widely recognized as one fundamental way not only to ensure some user-centered design [Hack98] but also to improve the understanding of how a user can interact with a user interface to accomplish a given interactive task.

A task model is often defined as a description of an interactive task to be performed by the user of an application through the application's user interface. Individual elements in a task model represent specific actions that the user may undertake. Information on subtask ordering as well as conditions on task execution is also included in this model.

Task analysis methods have been introduced from disciplines with different backgrounds, different concerns, and different focuses on task. The disciplines include:

- *Cognitive psychology* or *ergonomics* [Stan98]. Task models are used to ensure the understanding of how users can interact with a given user interface for carrying out a particular interactive task. Task analysis is useful for identifying the cognitive processes (e.g., data manipulation, thinking,

problem solving) and structures (e.g., the intellectual skills and knowledge of a task) exploited by a user when carrying out a task and for showing how a user can dynamically change them as the task proceeds [John84]. It can also be used to predict cognitive load and rectify usability flaws.

- *Task planning and allocation.* Task models are used to assess task workload, to plan and allocate tasks to users in a particular organization, and to provide indicators to redesign work allocation to fit time, space, and other available resources [Kirw92].
- *Software engineering.* Task models can capture relevant task information in an operational form that is machine understandable. This is especially useful where a system needs to maintain an internal task representation for dynamic purposes, such as to enable a control on the system state, or an adaptation to variations in the context of use [Lewi94, Smit96].
- *Ethnography.* Task models can focus on how humans interact with a particular user interface in a given context of use, possibly interacting with other users at the same time.

Existing task models show a great diversity in terms of formalism and depth of analysis. They are also used to achieve a range of objectives [Boms98, Boms99]:

- To inform designers about potential usability problems, as in HTA [Ann67].
- To evaluate human performance, as in GOMS [Card83].
- To support design by providing a detailed task model describing task hierarchy, objects used, and knowledge structures exploited while interacting, as in TKS [John92] or CTT [Pate00].
- To generate a UI prototype, as in TERESA tool [Pate04].

In general, any task analysis method may involve three related poles:

- *Models* – capture some facets of the problem and translate them into systems specifications.
- A *stepwise approach* – in which a sequence of steps is used to work on models.
- *Software tools* – support the approach by manipulating the appropriate models.

We focus on the first pole, that is, on models. It is assumed that the structuring of a method's steps for modeling tasks should remain independent of the task

model's contents. Therefore, the methodological part of each task model was taken to fall outside the scope of our analysis. A tool clearly facilitates the task modeling activity in hiding the model notation from the analyst and helping her to capture it, edit it for any modification, and exploit it for future use (e.g., task simulation, user interface derivation). Most models presented below are software supported.

Lots of task models have been proposed in the literature. Some representative task models are CTT [Pate99], Diane+ [Bart88,Tarb93], GOMS [John96], GTA [Weli98], HTA [Shep95], MAD* [Scap89, Gamb97], MUSE [Lim94], TAG [Payn86], TAKD [Diap89], TKS [John92]. In [Limb03], we proposed a meta-model expressing in a common way the concepts manipulated by these task models. From this survey, core concepts for engineering an interactive system were identified:

- A set of task attributes enabling a description of the nature of the task independently of its concrete realization on a particular computer system.
- A hierarchical decomposition of tasks allows a structuring of tasks starting from high level tasks onto leaf tasks representing user's actions.
- A task temporal ordering. Sister tasks may be temporally arranged with a set of temporal operators (e.g., sequencing, parallelism, choice).
- A set of relationship to domain concepts enable to express the "things" on which a task is operated on.

2.3.3 Domain Model

A domain model captures concepts from the semantics of the application domain. Without domain concepts a UI description would be an empty shell.

Domain modeling comes from software engineering [Dsou99]. It represents an essential ingredient to UI engineering methods as it describes its informational content. The domain model is usually developed by software engineers and provide "as is" to the UI designers e.g., under the form of an Application Programming Interface (API). UI designer's job consists afterwards in connecting a UI to the provided API.

Historically, the role and content of domain models used for UI development has evolved from hard-coded data models [Balz95], to entity-relationship-attributes schemas [Boda94b, Boda95], and to conceptual class diagrams and class diagrams with methods [Grif02].

Entity-Relationship-Attribute model (ERA) finds its roots in philosophy (theory of ontology) and database engineering. ERA models seek to represent real-world objects as *entities* equipped with *attributes. Relationships* can be defined among these entities to express the possible interaction within entities. ERA is complemented with a constraint mechanism allowing a limitation on relationships instances defined among entities e.g., a limitation on the number of instances of entities participating into a same relationship type (i.e. cardinality constraints). Trident [Boda95a,b] uses an entity-relationship to describe concepts manipulated by users while interacting with the system.

A *class diagram* is an extension of ERA model in the context of Object-Oriented (OO). It is defined [Breu97] as the description of the static structure of a system consisting of a number of c*lasse*s and their *relationships*. A class describes the properties of a set of objects and contains *attributes* and, potentially, *methods* which are process manipulating classes' instances. Structural relationships between classes of objects can be defined, theses relationships being called *associations*. Certain types of associations are so common across different systems that a precise has been defined for them, they are called *generic associations*. Two generic relationships are popular among analysts: *Generalization* is an association between a more general class (called superclass) and a more specific class (called subclass). A subclass holds all features of its superclass and adds some; *Aggregation* represents a whole part relationship. Class diagrams have been used notably in [Grif02].

In some cases, a domain model is simply hard coded in the system. This technique is still considered as part of modeling practices since, in systems using this technique, abstract characteristics of the code are extracted to enable an application of model derivation heuristics. For instance, Janus system [Balz95] uses C++ class structure to derive a presentation model.

Some domain models are expressed in ad hoc formalism. By ad hoc, it is meant a formalism that is not commonly found within usual CASE tools. The Mimic language in Mecano [Puer96] expresses the domain model with a structured declarative language.

2.3.4   User Interface Model

User interface models propose abstractions to improve comprehension, reasoning and manipulation of what a UI is. The real-world objects abstracted away in this

case concern all manifestations of a UI in the real world i.e., UI appearance (i.e., *presentation model*) and behavior (i.e., *dialog model*). Methodologies described in the literature vary according various dimensions:

▪ *Coverage.* Some methods concentrate on behavioral specification only (e.g., for property checking) and leave aside the problems related to UI appearance e.g., Petri Nets [Pala97] or Process Algebra. The integration of these methods with presentational aspects is still a hot research topic.

▪ *Separation of concerns.* Some methods do not make an explicit distinction between dialog and presentation. For instance, ADEPT [John92] relies on the task model as only description of the dynamics of the system.

▪ *Level of abstraction.* UI models proposed in the literature show a great diversity in terms of levels of abstraction of their concept. Three levels of abstraction, and corresponding model, are recurrently mentioned in the literature: abstract UI model, concrete UI model and final UI (also called implementation or code level). Abstract and Concrete UI raise many interpretation issues: What is abstract? What is concrete? With respect to what? In Fig. 2-6, we identified several levels of abstract and their correspondence.
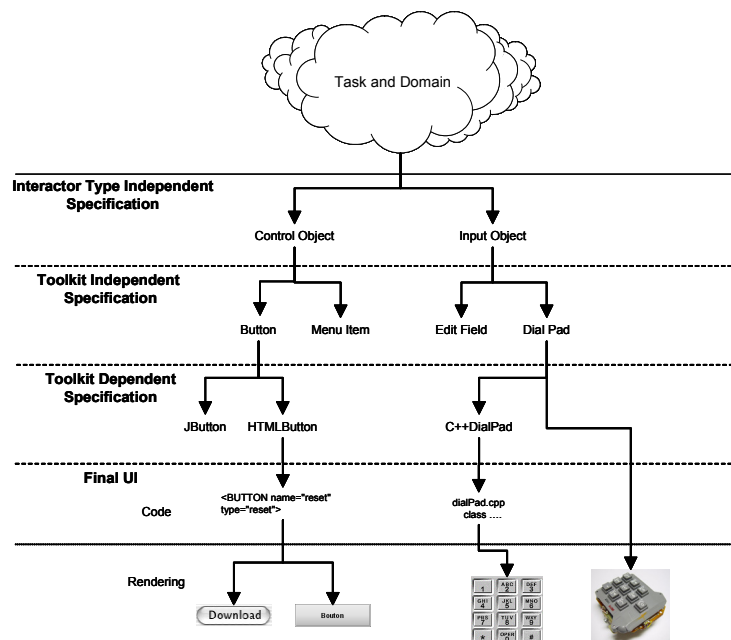
Figure 2-6 Various abstraction levels for UI models

A *final UI*, is composed of two sub-levels. The *rendering level* concerns the way a piece of UI related code is rendered on the screen (or other

interactive space) and made perceivable by a user. Note that this level may also cover physical devices enabling the interaction with the system. The *code level* is the implementation of the user interface. This implementation is realized using a programming language.

A *toolkit dependent specification* is a representation of a UI that makes explicit reference to elements of a specific programming language (or toolkit) while abstracting away syntactic details of this language. Fig. 2-7 provides such an example for a UIML? specification making reference to Java toolkit

```
<structure>
  <part    id="TopLevel" class="JFrame">
    <part id="L"          class="JLabel"/>
    <part id="Button"    class="JButton"/>
  </part>
</structure>
```

Figure 2-7 UIML specification atToolkit Dependent Level

A *Toolkit Independent level* manipulates a set of concepts that do not make any reference to specific toolkits. At this level interactor (i.e., widgets) are defined e.g., a button, a menu, a window. Generally, this level realizes an abstraction of several toolkits at the same time. This level allows a description of a UI that can be refined later on for different target languages (and environments).

A *Interactor-Type Independent level* provides us with a description of the UI in terms that are independent of interactor types. The concepts proposed at this level make a reference to a function endorsed by an interactor (e.g., selection, input of a value, output of a value). Assuming that several interactor types may endorse a same function, a description at this level allows a refining towards a wide variety of UIs.

Task and domain however not part as is of UI models may be presented at the top of this framework as they represent the most abstract viewpoint that can be defined on a UI system.

2.3.4.a   Presentation model

A presentation model is a description of the appearance of a user interface. Most presentation models found in the literature concern graphical, 2-D, widget-based UIs that is to say WIMP interfaces (Windows, Icons, Menus and Pointing device). Some presentation models allow a representation of voice interfaces [Klem00].

The content of presentation models varies across different methodologies. Two dimensions may help to categorize them:

(1) *Type of elements* in the scope of the model. The type of elements may be differentiated across the various levels of abstractions presented in Fig. 2-6.

(2) *Layout mechanism* exploited. Nearly all models are based on a hierarchical organization of elements populating an interface. Additional information may be used to refine a layout description: spatial constraints (e.g., alignment, adjacency) are used in [Thev02], a mechanism of box embedding is used in Latex typesetting system [Mitt99] or XUL [Gind01], a specification of absolute coordinates (in most of programming toolkits).

Constraint language allows specifying constraints on interface elements [Huds96]. Constraints languages provide a natural mechanism for expressing relationships within the interface structure. They are widely used for layout managers. Some guidelines are particularly well expressed with constraint languages. Alignment or groupings of widgets are elements that can be expressed in constraints. Constraints are generally small chunks of knowledge that are verified at run-time.

X.height = X.parent.height
X.width = [X.parent.width] - 10

Figure 2-8 Definition of widget constraints

Fig. 2-8 shows constraints expressing that a graphical object x should have the same height as its parent, but a width of 10 pixel less.

A problem with constraint languages is the ambiguity they leave when only a few constraints are specified (e.g., the usual case). Consistency of constraints is also something that is to check before rendering. On the other hand, constraints systems allow a designer to only specify constraints that are relevant in her model

and leave aside all layout details that do not matter for her at rendering.

Box embedding systems rely on a structuring of elementary widgets within abstract boxes determining the layout of its content. A box can impose a vertical or horizontal alignment of its content, an arrangement in grid structures, etc. Box embedding are used in most of the UI mark up languages [Cout02]. They bring the advantage of a precise interpretation as they are totally unambiguous while omitting any reference to absolute coordinates i.e., the layout declaration stays logical. Furthermore, because of their unambiguousness they are very cheap (in terms of computational power) to render.

A layout based on absolute coordinates is done through the definition of a set of coordinates on a display surface where pixels are used as unit references. Absolute coordinates are absolutely unambiguous, cheap to render. Absolute coordinate layout presents two major disadvantages: (1) it requires from the designer an enormous amount of time for specifying each element's coordinates (2) because of its lack of abstraction it obfuscates the logical structure of the UI.

2.3.4.b Dialog model

Dialog models enable to reason about the behavior of a UI system. For many, dialog models are a continuation of task model concepts [Gram96]. We hereafter give a brief survey of dialog modeling methods that percolated into the field of HCI methods.

- Backus-Naur Form (BNF) grammars

BNF grammars are typically used to specify command languages. Command languages express commands that modify the state of the UI on the users initiative. Fig. 2-9 exemplifies a grammar describing a file manipulation in UNIX.

```
file-op[Op] := command[Op] + filename + filename |
command[Op] + filelename + directory
command[Op=copy] := 'cp'
command[Op=move] := 'mv'
command[Op=link] := 'ln'
```

Figure 2-9 A BNF grammar for fille manipulation with UNIX

Grammars are particularly good in detecting inconsistencies within command sets. An inconsistent UI may contain unordered or unpredictable interaction. Inconsistency renders the UI error prone and hard to learn. [Reis81] proposed an action grammar to describe GUIs. [Payn86], with their Task-Action Grammar (TAG) extended this grammar by, namely, addressing three levels of inconsistency: lexical, syntactic, and semantic. These established TAGs accuracy in predicting. Grammars are effective for expressing sequential commands or users actions but when it comes to multimodal or direct manipulation they tend to be heavy to manipulate.

- State Transition Diagrams

A state transition diagram is a finite state machine representation that consists in a graph of nodes linked by edges. Each node represents a particular state of the system. Each edge species the input (i.e., event) required to go from one state to another. State transition diagrams like statecharts presented bellow provide a mean for specifying the dynamic behavior of the interface. Fig. 2-10 is such a representation applied to an event/action specification, very frequent in UI field.
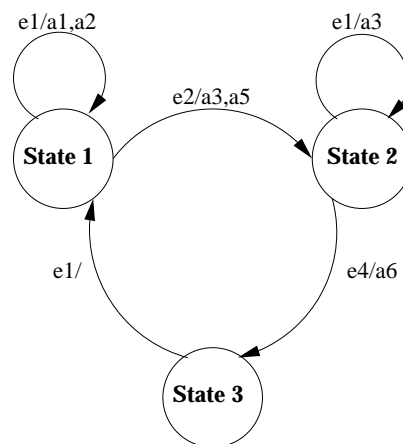
Figure 2-10 State-transition diagram specifying UI events and actions.

Each edge of the type $E_x = a_x; a_y; a_{...}$ where E_x is an event associated to a set of actions $a_x; a_y; a_{...}$ which ensure the transition to a target state.

State transition diagrams present several drawbacks for modeling the UI. Indeed,

today's UI tend to be modeless where one state can lead to many states. Furthermore this can be done using many different widgets of the UI. Theses two requirements match the quality criteria of reachability and device multiplicity. In consequence, state transition diagrams are prone to a combinatorial explosion and tend to replace nodes by screen prints. [Schn98] reduces the transition space to window managers in graphical state transition diagrams.

- ▪ Statecharts

Similarly to state transition diagrams, statecharts support a graphical representation of dynamic aspects of systems. Some work specially address the modeling of UI behavior with statecharts [Horr98]. Statecharts represent state variables with rounded rectangles called states. State changing mechanisms are represented with edges between states. State changing is triggered by events and can be further conditioned (Fig. 2-11). Statecharts facilitate the representation of state nesting, state history, concurrency and external interrupts.
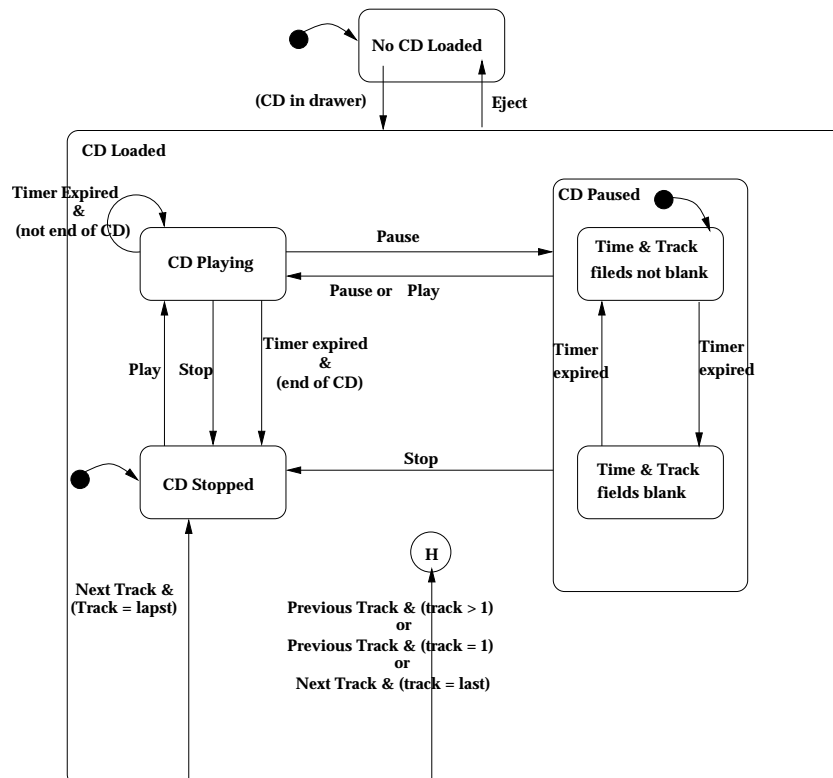
Figure 2-11 A CD player behavior with statecharts

Statecharts [Hare87] propose solutions to the shortcomings of state transition diagrams: statecharts have representational capacities for modularity and abstraction. The number of states with respect to the complexity of the modeled system increases slower with statecharts than with state transition diagrams. Statecharts avoid the problem of duplicating states and transitions. States in statecharts are hierarchical and capable of representing different levels of abstraction. Statechart are more convenient for multimodal interfaces as they provide nesting facilities, external interrupt specification and con-currency representation.

- ▪ Petri Nets

Petri Nets is a graphical formalism associated with a formal notation. Petri nets are best suited to represent concurrency aspects in software systems. Fig. 2-12 represents a cash withdrawal operation with an automatic telling machine. Petri nets represent systems with state variables called places (depicted as ellipses), and state-changing operators called *transitions* (depicted as rectangles).
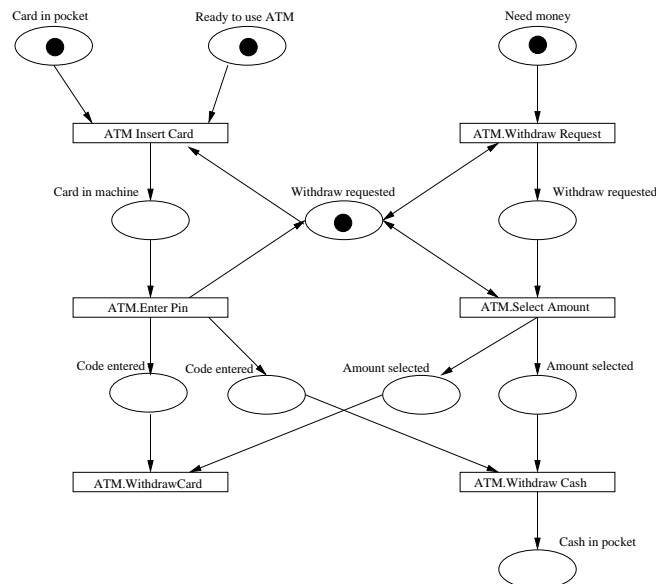
Figure 2-12 Petri net representing a cash withdrawal operation (from [?])

Connections between places and transitions are called arcs (represented by edges). State marking mechanism called tokens (represented by black solid circles

distributed around places). State change is the consequence of a mechanism called firing. A transition is red when all of its input places contain tokens. Firing involves the redistribution of tokens in the net i.e., input tokens are withdrawn from input places and output tokens are added in output places. Like State Charts, Petri nets hold mechanisms to represent additional conditions and nested states. Petri nets have the advantage of being entirely formalized (State Chart concurrency mechanism, for instance, has no formal background). Petri nets allow a checking of several properties of the represented information (e.g., a dialog model expressed with a Petri net can be checked for completeness or coherence). In the context of UI they have been used notably by [Pala94,97].

- ▪ Event-Response Languages

Event languages treat input stream as a set of events. Events are addressed to event handlers. Each handler responds to a specific type of event when activated. This type is specified in a condition clause. The body of the event generates another event, changes internal state of the system or calls an application procedure.

Several formalisms are suited for event-response specification. They can be distinguished following their capacity in managing dialog state variables and concurrency control. Production rules are often used to describe event-response specifications.

```
Sel-line              -> start-line <highlight 'line'>
C-point start-line -> rest-line <rubber band on>
C-point rest-line  -> rest-line <draw line>
D-Point rest-line  -> <draw line> <rubber band off>
```

Figure 2-13 Event-Response specification using production rules (from [Dix98])

The example given in Fig. 2-13 shows a set of four production rules representing a poly-line drawing dialog sequence. User events begin with upper case. Sel line event represents user's selection of line option (e.g., in a menu bar), C-point and D-point represent user's clicks on the drawing space. System events begin with lower case, rest-line event means the recording on coordinates of users first point selection. System events keep trace of the dialog state. System responses are enclosed within '<>'. They are perceivable events.

2.3.5 MBIDE Methods and Tools

A wide panel of UI development methodologies exploiting have been proposed in the literature. These methodologies have been called *Model-Based (user) Interface Development Environments* (MBIDEs). A selection of MBIDEs was made on the basis of the following criteria: originality of the concepts, originality of the development cycle, definition of an explicit methodology, minimal tool support.

For each MBIDE presented hereafter a summary of the underlying concepts and development process is proposed. A unified iconographic representation is used to present development processes (Fig. 2-14). Each symbol represents a type of models. This constitutes a first attempt to harmonize conceptual frameworks of these different approaches. Note that the difference between "interactor-type independent" and "toolkit independent" representation (see Fig. 2-6) is stressed by two different icons associated with abstract UI. A dialog model is also represented as a separated entity when the dialog model is substantially important in the development process. Solid arrows represent the derivation of one model from one or several other ones. Dashed arrows represent a significant knowledge adjunction in the design process (e.g., a designer manually determines a layout, a template is chosen to drive the derivation).
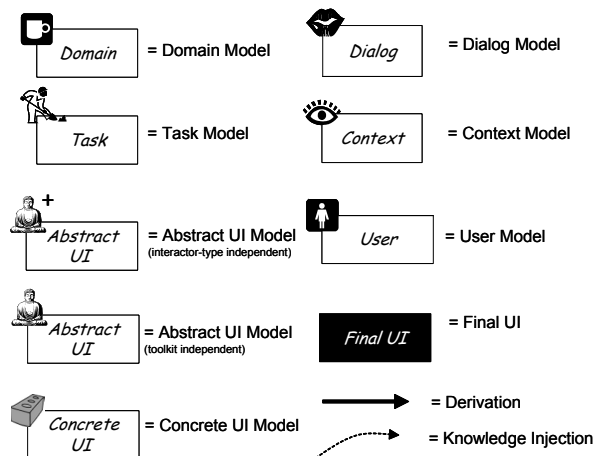
Figure 2-14 Symbols used for the state of the art on CADUI tools

2.3.5.a Adept

Adept adopts a user centered design approach [Wils96]. It is based on a sophisticated task model called Task Knowledge Structure (TKS) [John89,

John92]. Adept holds a users model [Kel92]. Both task and users models are elicited during the requirement stage of the application development (Fig. 2-15).
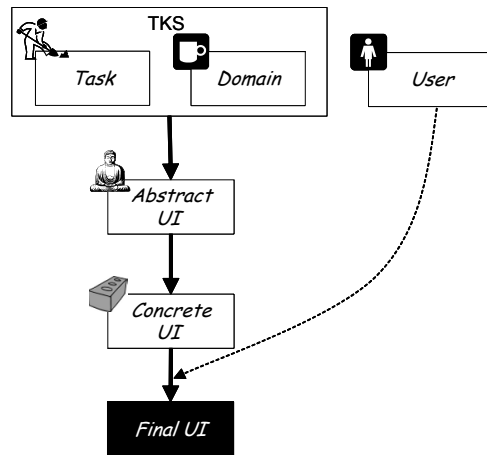
Figure 2-15 Adept development steps

A task model is, here, defined thanks to a Task Knowledge Structure (TKS). A TKS is a conceptual representation a person has stored in her memory about a particular task. Each task in TKS has an associated goal, procedure, actions and domain objects. Note the inclusion of the domain model into the task model.

A user model in Adept consists of a rule-based system associating design rules with user stereotypes. It is constructed thanks to a questionnaire that segregates users characteristics from design options. Retained characteristics are for instance knowledge of the domain, computer experience, motivation and attitude towards the system. Production rules are used to derive a UI model. For instance, a rule "IF Experience =high THEN textual commands" allows a selection of an interaction style depending on some assessment of a user experience.

After defining a task and user model, an Abstract Interface Model (AIM) is derived from the task model. The AIM defines abstract interaction objects to be manipulated by the leaf tasks of an Adept task specification. As it contains information about the commands to execute to accomplish task, it can be partly considered as a dialog model. Subsequently, a concrete interface model (CIM) is built. This model is derived from the AIM and the users' model, it instantiates Abstract Interaction Objects (AIOs) into concrete interaction objects. The CIM is then translated into Smalltalk code to produce an executable interface.

Design knowledge in Adept is expressed under the form of production rules. A tool (Fig. 2-16) allows a designer to choose among possible conflicting rules, to modify rules, to add new rules. The right part of the Fig. 2-6 represents production rules available in the system. For instance, the highlighted rule states that if an abstract object associated with a task of type "selection" widget and the manipulated object is of the type "range" then derived concrete object has to be of a type "slide bar". The left window of the figure shows a trace of the dialog between the system and the designer to resolve conflicts between rules.
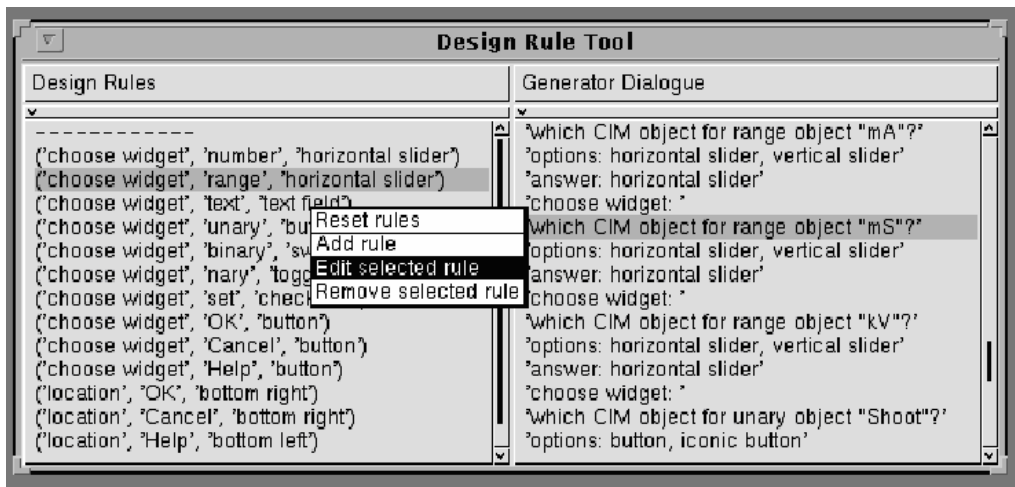
Figure 2-16 Selecting transformation heuristics with Adept

Adept offers as tool support: a model editor, a user interface design assistant, an implementation tool.

2.3.5.b   Art Studio

ArtStudio [Thev02] consists of a development environment for producing multi-target UIs embedded with support of plasticity at design-time. For this purpose, ArtStudio starts from a task model and its relationship to a domain model (referred to as a *concept model* in ArtStudio). The task model is decorated with various mechanisms to indicate variations of the task model depending on variation of the context of use, primarily the type of platform. This way, the context model consists of a combination of a platform, a user, and its surrounding environment.

The decorated task model initiates then a process where an abstract UI is derived from the task model so as to identify abstract workspaces (or Presentation Units). Abstract workspaces, in turn, give rise to a definition of the UI in concrete terms. A final UI in Java is produced from the concrete specification (Fig. 2-17). ArtStudio is original in that the UI that is generated supports to some degree the property of plasticity, that is, the capability of the UI to adapt itself to the current context of use while maintaining predefined usability properties [Calv01].
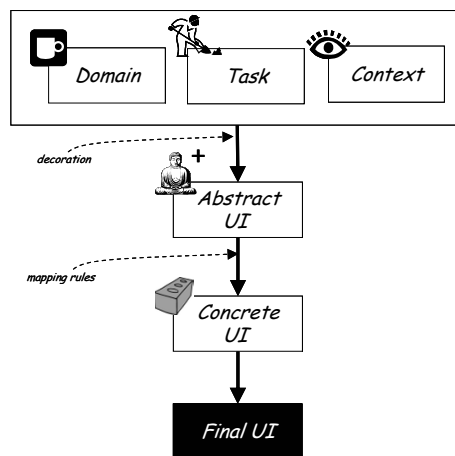
Figure 2-17 The approach followed in Art Studion.

2.3.5.c Trident

Trident [Boda94,Boda95b] uses a task model, a domain model (including a flow of control specification), and a presentation model. Trident development process (Fig. 2-18) starts with a *contextual analysis*. Context in the sense of Trident means tasks, users and organizational contexts identification. Trident task model is a Task Knowledge Structure (TKS) (see above). This task specification embeds part of a domain description and user mental model.
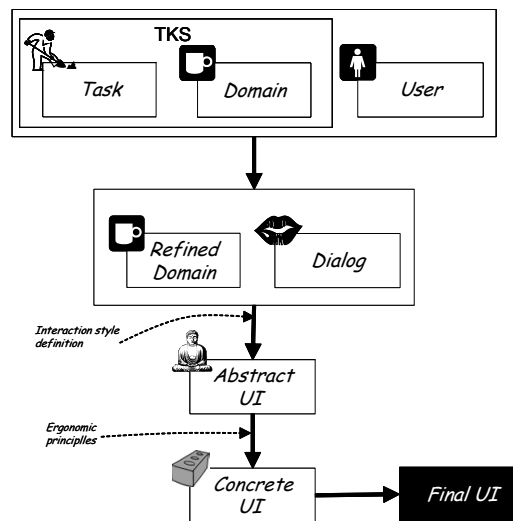
Figure 2-18 Trident development process

From this, so called, contextual analysis several elements can be derived. A domain model is derived under the form of an entity-relationship-attribute model; an dialog model skeleton is defined from the specification of actions contained in a task analysis. Dialog model is specified using an activity chaining graph formally showing sequences of functions and their respective input/output flow. A designer selects between several interaction styles. Interaction styles help to determine dialog mode, dialog control, function triggering mode, and metaphor style.

Presentation model elements are derived from the activity chaining graphs. Subgraphs are mapped onto presentation units that are decomposed into logical windows. Those logical windows are then populated with abstract interaction objects (i.e., independent of a target platform). This whole process is guided by the application of ergonomic rules exposed [Vand97]. AIOs are then mapped onto concrete interaction objects (i.e., platform-dependent objects). A knowledge of the target environment is necessary to achieve this step. CIOs are finally arranged thanks to a sophisticated placement algorithm. Within the environment, the UI generated can be interpreted for testing purposes and edited. Once the beautification process is finished, the FUI can be generated for Microsoft Visual Basic 3.0.

Although Trident contains some extensive design knowledge in a knowledge base (hundreds of production rules are provided for supporting the development

process), these rules are hard-coded and totally embedded (i.e., totally unavailable to a designer) in the system. It is possible to view them, to control their application (in a mixed-initiative way), but any modification of such a rule would require a direct intervention of a highly skilled developer who is familiar with, for instance, the underlying expert system (i.e. AION/DS) and the mechanism of selection trees.

2.3.5.d   FUSE (Formal User Interface Specification Environment)

FUSE [Lonc96] generates a UI code from the task, domain and user model. An FUSE is also capable of generating help and guidance files, which is original with respect to other environments. A task model in FUSE consists of a hierarchy of tasks successively decomposed into sub-tasks until leaf tasks. Leaf tasks are associated with a function implemented in C++. Other models are also exploited in Fuse as depicted on Fig. 2-19.

A domain model is a set of algebraic specifications of the functions and data structures. Algebraic functions are under the form $Spec_{AI} =< sum_{AI}, Ax_{AI} >$ where sum_{AI} is a declaration of data types and function signatures and Ax_{AI} is a set of pre- and post-conditions associated with functions.

A user model describes user groups and individual users. Users are stereotyped along three dimensions: motivation, knowledge on the computer system, task knowledge (each rated low, medium or high). FUSE also holds a dynamic system to record user's properties at run-time.
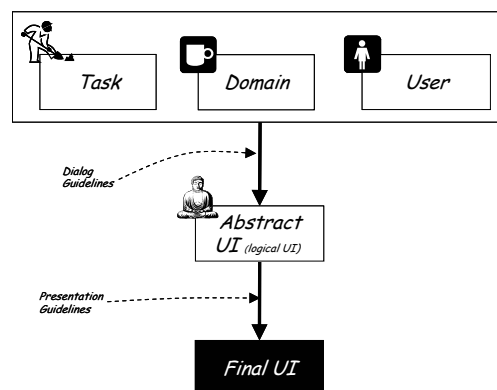
Figure 2-19 Fuse development process

Once created, FUSE input models are transformed into a *logical UI* model.

"Dialog guidelines" are provided to achieve this transformation operation. A logical UI is a sort of abstract UI model, it consists in a set of views containing a description of user actions, system actions, and objects associated with user tasks. A logical UI is represented with a formalism called Hierarchic Interaction graph Template (HIT). This formalism is based on attributed grammars and on data flow diagrams. From a logical UI and a set of "layout guideline" specification, a C++ user interface (along with help and guidance files) is generated at run-time.

2.3.5.e   Genova

Genova is a commercial tool that generates a UI code for several languages (Java, C++, Visual Basic) starting with an enriched class diagram (Fig. 2-20).
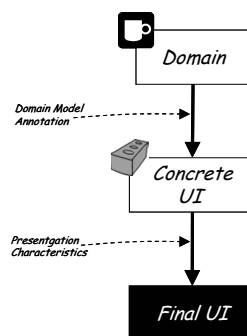
Figure 2-20 Genova Development Process

As a start, a designer defines an application class diagram. Classes that have to be represented on the user interface are annotated. From this information, a draft presentation model is generated. Roughly, this draft presentation model consists of a filtering of the input class diagram according to designer's annotations.

From this draft presentation model, a designer is able to choose between presentation templates called style guides. She uses built-in transformation rules for this purpose. For instance, the developer can map data types to abstract interaction objects (i.e., types of objects independent of any target platform) or choose among a dozen of heuristics to determine the choice of dialog units.

Genova consists of a plug-in on top of the Rational Rose CASE tool that is used to edit the class diagram. Its UI design assistant proposes dialog windows to customize pre-existing transformation rules (Fig. 2-21).
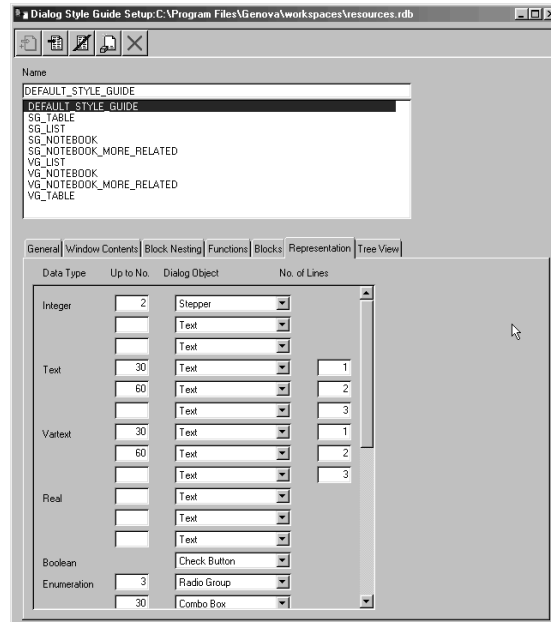
Figure 2-21 Defining presentation characteristics using Genova

Although Genova is distinguishable as it enables designers to modify dialog templates, it does not make explicit neither the internal format of models nor the transformation rules between them. In other words, the customization process is limited to the tool options.

2.3.5.f Janus

Janus [Balz93,95,96] allows its user to perform a derivation of a presentation model on the basis of an object model. Janus application model is an object oriented model issued from an object oriented analysis (Fig. 2-22).
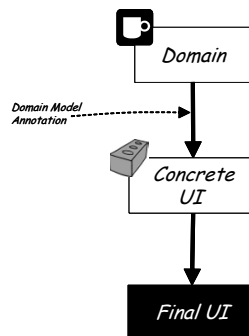
Figure 2-22 Janus Development Process

Janus transformations rules consist in window definition and widget selection rules. For instance a rule asserts that every domain class (that is not abstract) must give rise to a window. A knowledge base associates method names with concrete interaction objects. Janus uses generic relationships like aggregation or generalization in order to derive, notably, windows transitions.
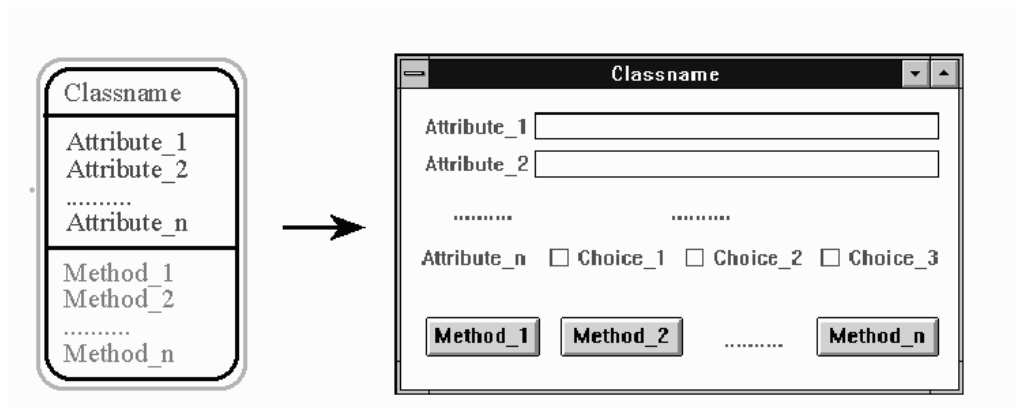
Figure 2-23 Illustration of a transformation heuristic in Janus

Janus development cycle is presented in Fig. 2-22. After being transformed into an internal format (JSD), a domain model is transformed into a form-based presentation that is linked to methods implementations in C++. Fig. 2-23 graphically represents how an object class of the domain model (left part) is mapped onto a window in the presentation model (right part). Although this graphical representation facilitates the understanding of the aims and scope of the transformation, little or no information is provided on their format. Being hard coded in the sofware, nobody can access them, unless perhaps the developers of Janus. Janus has been recently packaged in a commercial tool, called Otris

(www.otris.de), that suffers from the same shortcomings.

2.3.5.g JustUI

JustUI consists of a UI development method starting from a CTT task model and a domain model expressed as a UML class diagram specified in Oliva Nova Model Editor (ON ME) [Moli02]. From the information contained in both models, a pattern-matching approach is adopted that identifies stereotypical sub-tasks and map them onto typical patterns for information systems. These patterns are structured into three levels in a Hierarchical Action Tree (HAT) (Fig. 2-24):

- Level 1: The first level contains the HAT pattern, providing the access to the application.
- Level 2: The second level contains the Interaction Units. The UI is decomposed in several scenarios to support user tasks.
- Level 3: The third level is composed of patterns that add semantics for interaction units.

Once these patterns are applied, an abstract UI is generated that consists of a hierarchical decomposition of Interaction Units, a generalization of Presentation Units [Boda95c] to both presentation and dialog. This decomposition of Interaction Units is exploited to give rise to the final code, which can be generated for multiple computing platforms, including JavaBeans, ColdFusion, HTML, C++ and Visual Basic.
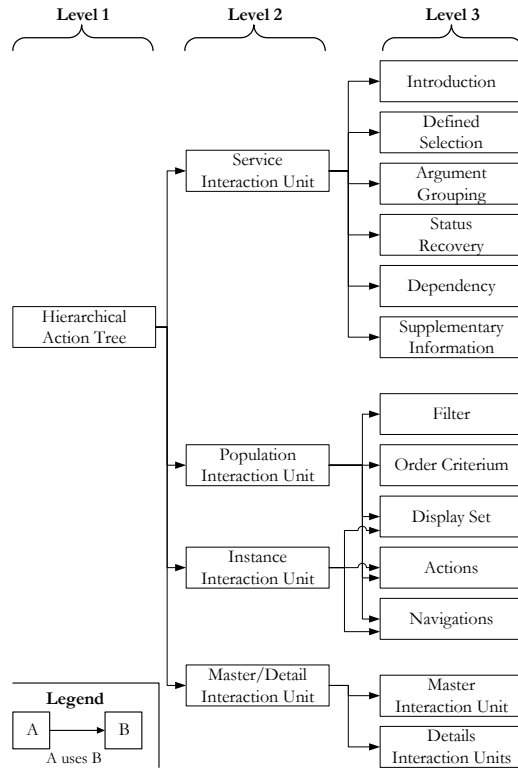
Figure 2-24 The Hierarchical Action Tree of JustUI.
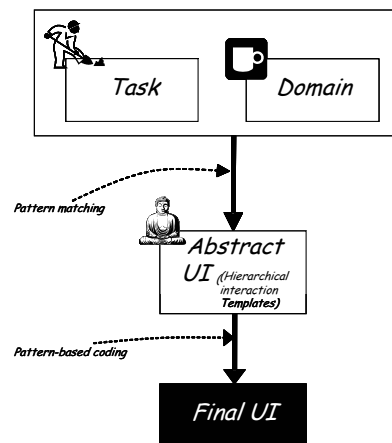
Figure 2-25 Illustration of the pattern-matching approach followed by JustUI

2.3.5.h Mastermind (Models Allowing Shared Tools and Explicit Representations Making Interfaces Natural to Develop)

Mastermind is the continuation of Humanoid [Luo93,Szek90,Szek92] and UIDE [Fole91,Fole95].
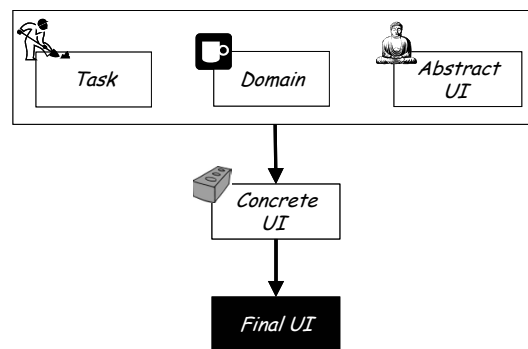
Figure 2-27 Mastermind's development process

Mastermind has three component models: task, presentation and application. These models are implemented with CORBA IDL. The use of a unique language facilitates the establishment of mappings between models (Fig. 2-27).

A task model consists of a hierarchy of tasks decomposed in necessary sub-tasks. A task is described by several attributes such as: goal and effect (i.e., task post-condition), task type, associated methods signature, and associated presentation elements (i.e., widgets). A task model in Mastermind covers aspects of dialog specification as it also specifies user's interaction with the system. The task model plays a central role in Mastermind architecture as it expresses constraints on the sequencing of behavior in presentation and application model.

A domain model (application model is Masterminds terminology) is based on OO design techniques. It contains application class structures and methods signatures. Objects are extended with a mechanism of pre-condition and notification i.e., a publish-subscribe event language. This event language allows task or presentation model elements to declare interest in any modification of the domain model.

An abstract UI model specifies both static (in terms of toolkit independent widgets) and dynamic aspects (a combination of statecharts and Petri nets). An event response language (called interaction diagram) operates as a binding between presentation and dialog. Mastermind takes into account direct

manipulation interfaces, by using a mechanism for expressing functional constraints between the presentation elements.

Mastermind starts with the definition of task, domain and presentation models. No tool is provided to support this task. Models can be defined in parallel but some co-ordination is needed. Models are linked by hand. An implementation tool generates C++ code from the declarative specifications in Corba IDL.

Humanoid is complemented by Kurt Sirewalt [Stir9,Stir98, Stir99,Stir00] works on Mastermind Dialog Language (MDL). MDL formalizes models as concurrent agents that synchronize on common events. It seeks efficiency by implementing a dialog component (i.e., implementation of the task model) that synchronizes presentation and application components. Humanoid is more interested in resolving the problem of composing models to form a run-time system than to define a transformational mechanism allowing a derivation of a component from another.

2.3.5.i   MOBI-D

Mobi-D [Puer97] is the continuation of Mecano environment [Puer96]. Mobi-D involves five basic models in the development Process (Fig. 2-28): task, domain, user, presentation, and dialog model. Mobi-D distinguishes between two levels of abstraction among models. The abstract level concerns task, domain, and user model. A concrete level concerns presentation, and dialog model.
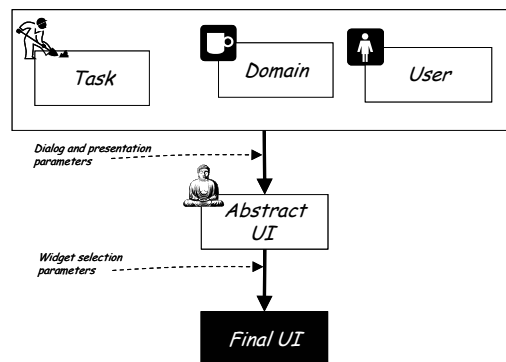
Figure 2-28 Mobi-D development process

Mobi-D development cycle is the following. A task, domain, and user model are

obtained from a requirement analysis. A tool (U-tel) is provided to extract elements of these three models from scenarios. Mobi-D offers two software tools to improve UI derivation. The first tool is a model editor that allows users to manually link elements of models by dragging and dropping elements onto each other. This editor does not provide any guidance in how to set the mappings and is intended to be used for mapping models of the same level of abstraction (i.e., abstract level). Inter-level mapping is supported by a tool called TIMM. TIMM's role is to explore all concrete elements that could potentially be linked with the current abstract element and to select a subset of these interactors using the rules of its knowledge base. Developers can freely choose to use one of the proposed interactors or to set their own mappings. They can access to the knowledge base and modify existing rules. TIMM also allows the establishment global orientations for the task and domain to abstract UI model: navigational preferences, style, number and size of windows, etc. Fig. 2-29 exemplifies a dialog window allowing a developer to configure transformation rules enabling a derivation of CIOs from AIOs.
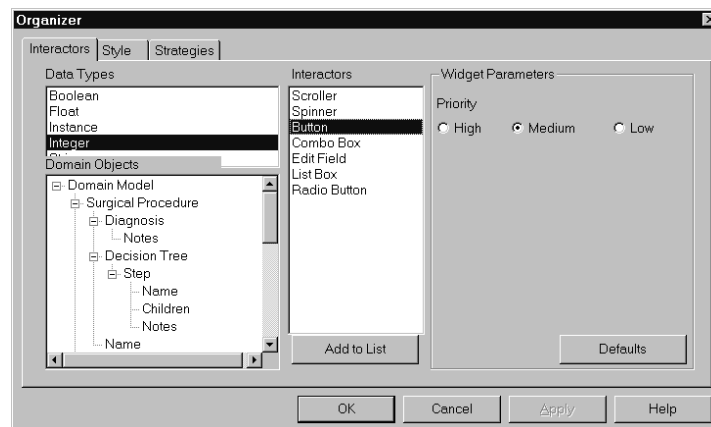
Figure 2-29 Abstract Interaction Object Selection in Mobi-D

At the end of the design phase, the Mobi-D interface-builder tool exploits all parameters provided by a designer to generate a C++ UI. Models components are declaratively described using a language called Mimic. The entire mapping between elements of the interface is declared using MIMIC in a sixth model component called the design model. Following Mimics's grammar, a design model is an unordered collection of design mappings. Each design consists of a mapping between model elements that can be conditioned.

2.3.5.j   TADEUS

TADEUS [Elwe95] uses four models: task, domain, user and dialog to generate an executable UI specification (Fig. 2-30).
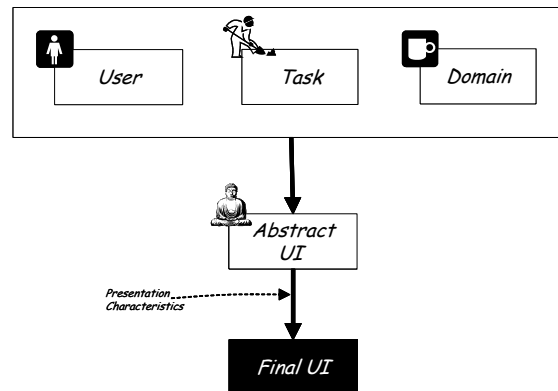
Figure 2-30 Tadeus development process

Instead of generating executable code, Tadeus generates specification file executable by an external UIMS (User Interface Management System), namely the ISA system. A task model in Tadeus is similar to the one used in Adept (see above). It consists in a hierarchy of task/goals, sub-tasks/sub-goals ordered by temporal operators. Tasks are linked to users interested in the task performance and onto domain objects manipulated during the task execution. Task and domain models are part of a single graphical representation used to construct a dialog model.

A domain model is the result of an OO-like analysis. It consists in a class diagram schema with attributes and methods for each object.

A user model, in Tadeus, is also inspired from TKSs. It categorizes users of the system and maps them onto roles. The relationship between roles and task has several attributes such as task execution frequency, preferred input device, etc. These attributes govern the selection of interaction techniques, including widgets.

An abstract user interface consists of a so-called dialog model that intertwines structural and behavioral aspects. This dialog model is built in two steps. Dialog modeling assumes a prior construction of task, domain and user model. The first step of dialog modeling is to define views on task/domain model. The definition

of views consists of designating tasks and domain objects that should be represented on a same window. The developer performs this step by annotating a task/domain tree. This annotation results of a static state specification of the dialog model. A second step consists in building a dynamic specification with *dialog graphs* [Schl96]. Dialog graphs are a notation for specifying multiple instances of windows, hierarchical dialog structuring and modal dialogue boxes. Dialogue graphs theory makes a distinction between intra-view dialog called *processing dialog* and inter-views dialog called *navigation dialog*.

Intra-view dialog is defined through interaction tables. It consists mainly in grouping concrete objects together and determining a position for groups. Intra-view dialog is specified after the abstract to concrete mapping. Inter-views dialog is specified via dialog graph in itself.

Dialog graphs distinguish different dialog view types (single, multimodal, complex,...) and transition types (concurrent or sequential). A graph manipulation algorithm reduces the graph complexity (a major shortcoming of state transition diagram).

At the end of the development process, a designer has to define several layout attributes such as background color, default interactors to display on windows. Additionally, for each view determined in the dialog modeling, a designer defines which abstract object should represent the view. Abstract objects are then mapped onto one or several concrete object. Rules are provided for the selection of concrete objects. These rules essentially exploit data types.

Design knowledge is mentioned to be used in Tadeus. Due to the lack of information provided it is unsure to determine its nature and use. No external mechanism is provided to manipulate rules. We suppose they are hard-coded in the system.

2.3.5.k   Teallach

Teallach [Brif98,Barc99,Grif01] is an MBIDE specifically designed for OO databases access applications. It exploits model mapping in a very explicit manner. It also holds a flexible development cycle allowing a construction and cross-consolidation of different models simultaneously. Teallach contains a domain, task, and presentation models. Dialog elements are distributed across task and

presentation models.

A domain model is expressed with an object-oriented data along with a specification of operations manipulating these data. Data and operations are defined in the context of an object-oriented database on top of which an application is built. Teallach domain model is automatically generated from an object database schema under the ODMG format. Transient and persistent objects are represented the same way. Originally, Teallach allows definitions of object states.

A task model holds, here, information about the dynamic aspects of the interaction of the user with the system and data processing requirements. Leaf tasks (called *primitive tasks*) are either action tasks or interaction tasks. Action tasks are activities that can be carried out by the application or by the user. They can be mapped to a domain object operation. Interaction tasks are interactive behavior carried out by the user. An interaction task is mapped onto a presentation object. To specify the information flows within the task model elements, Teallach uses a mapping between tasks and domain object states or presentation elements states. A domain object state or presentation element state is associated with any composite task (e.g., intermediary level tasks).

A presentation model is a specification of the interface appearance. It is hierarchical. The presentation uses external resources i.e., existing widget sets or any developer defined widget. Two levels of abstraction are explicitly defined within Teallach:

The final user interface generated by Teallach relies on Java Swing's widget set. Designer is allowed to define custom widgets. An abstract presentation model defines high-level categories of widget. Categories are established on the base of roles a widget can play within an interface. Categories are: free container (i.e., a top level container), container (i.e., a non-top level container), inputter (e.g., an input box), displayer (e.g., a label), editor (e.g., an updatable table), chooser (e.g., a radio button), action item (e.g., a button).
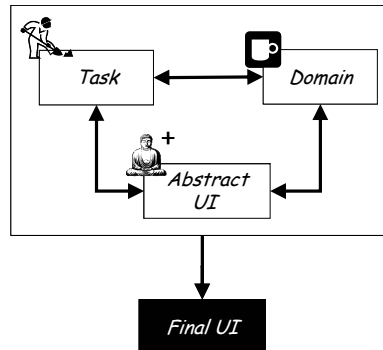
Figure 2-31 Teallach Development Process

Teallach development process (Fig. 2-31) consists of two activities: component models construction and component model mapping. Mappings have been divided into two categories: linking and deriving. *Linking* consists of simply associating two component model elements. *Deriving* consists of constructing one component model element from another one. Both activities are further explained later. Teallach development cycle has the originality of being lowly constraining in the sequence of development steps. Indeed, Teallach allows the developer to start with any of the three component models construction. In the same way, mappings can be done in any order. Teallach generates a running interface in Java.

Design knowledge in Teallach surely exists at least in the head of their authors but is not documented anywhere. Therefore, we were not able to specify these design knowledge properties.

Tool support consists of a model editor, a design assistant, and an implementation module. Fig. 2-32 represents the model editor. The left window is the domain model editor. The top right window is the task model editor. The bottom right window is the presentation model editor. Models can be mapped graphically by drawing lines between model elements. Dialog boxes appear for specifying further information on a newly created mapping.
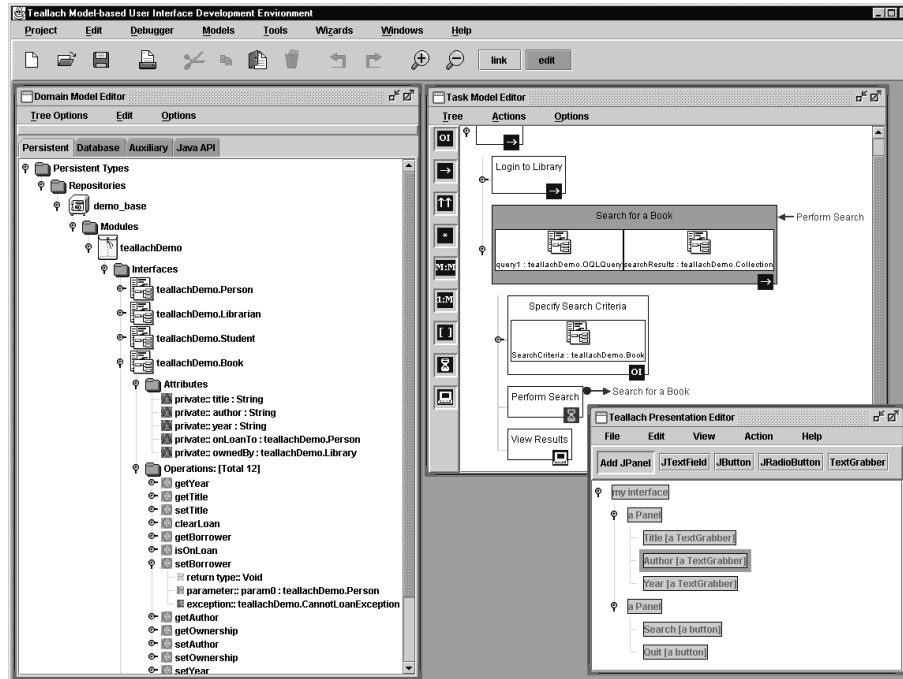
Figure 2-32 Teallach Model Editor

In addition, Teallach holds a mechanism for maintaining links integrity through an alert system that warns the developer for any link breaking.

2.3.5.1 Teresa

[Pate03] introduced a method for producing multiple FUIs for multiple computing platforms at design time. They suggest starting with the task model of the system, then identifying the AUI specifications in terms of its static structure (the presentation model) and dynamic behavior (the dialog model): such abstract specifications are exploited to drive the implementation. This time, the translation from one context of use to another is operated at the highest level: task and concepts. This allows maximal flexibility, to later support multiple variations of the task depending on constraints imposed by the context of use. Here again, the context of use is limited to computing platforms only. The whole process is defined for design time and not for run-time. For instance, there is no embarked model that will be used during the execution of the interactive system, contrarily to the Seescoa approach analyzed below. Fig. 2-33 graphically depicts how the TERESA tool supports this approach. At the AUI level, the tool provides designers with some assistance in refining the specifications for the different computing

platforms considered. The AUI is described in terms of Abstract Interaction Objects (AIOs) [Vand93] that are in turn transformed into Concrete Interaction Objects (CIOs) [Vand93] once a specific target has been selected.
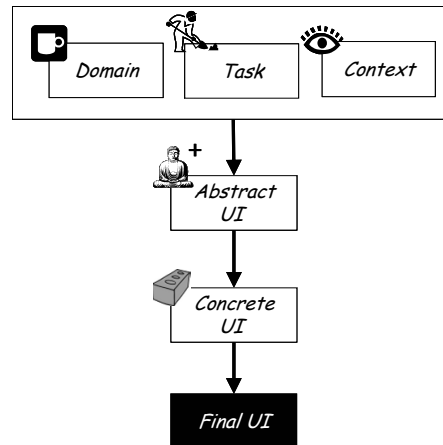
Figure 2-33 The TERESA development approach

2.3.5.m Seescoa

Seescoa [Luyt03] consists of a suite of models and a mechanism to automatically produce different FUIs at runtime for different computing platforms, possibly equipped with different input/output devices offering various modalities (e.g. a joystick). This system is context-sensitive as it is expressed first in a modality-independent way, and then connected to a specialization for each specific platform. The context-sensitivity of the UI is here focusing on computing platforms variations. An AUI is maintained that contains specifications for the different rendering mechanisms (presentation aspects) and their related behavior (dialog aspects). These specifications are written in a XML-compliant User Interface Description Language (UIDL) that is then transformed into platform-specific specifications using XSLT transformations. These specifications are then connected to a high-level description of input/output devices. A case study is presented that automatically produces three Final UIs at run-time: for HTML in a Web browser, for Java with Abstract Window Toolkit (AWT) on a Java-enabled terminal, and for Java with Swing library. Although the process is straightforward, generated UIs appear to have the same layout of final objects, but coming from the same CIOs.

Fig. 2-34(a) graphically depicts the process followed in this work to produce context-sensitive UIs. A translation is performed at the abstract level before going down in the framework for each specific configuration (here restricted to a

platform). No concepts or task models are explicitly used in this version. The entry point of this forward engineering approach is therefore located at the level of Abstract UIs. Dygimes [Luyt04] is an extended version of Seescoa that adopts the same approach as in Seescoa, except that the AUI is obtained from a CTT task model [Pate00] that is progressively transformed into a priority tree as a starting point for obtaining the AUI. These differences are highlighted in Fig. 2-34(b).
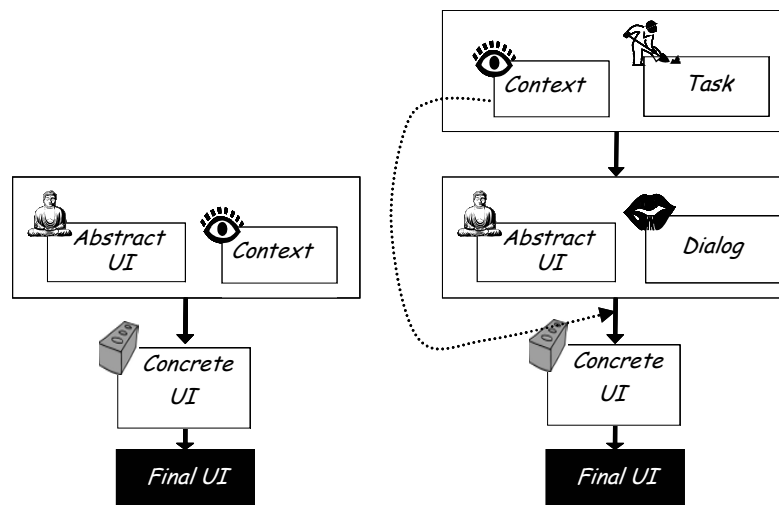
Figure 2-34 The Seescoa development approach (a) and its Dygimes extension (b)

2.3.5.n   Vista

Vista is a system supporting a so called *co-evolutionary design* of interactive systems. Co-evolutionary design is defined as the co-evolution (i.e., concurrent) of as set of design artifacts maintained by different stakeholders in a development process (i.e., software engineers and HCI specialists). The following *views* are considered in Vista:

- A task model codifies the human activities that the computer system is meant to support. A dialog model (termed task-oriented specification) is based on the task hierarchy. The task-oriented specification is formalized in User-Action Notation (UAN) [Hart90]. UAN specifies the user actions, interface feedbacks, and interface state resulting from the action.

- A domain model is an architectural model called *clock architecture*. Clock architecture is a layered Model-View-Controller (MVC) cluster [Grah96]. Components in this model are responsible for the processing of user inputs and maintenance of display devices.

▪ A final UI, at code level, is also presented to designers. This enables a visualisation at the code level of any change done to other models.

Links between these representations are established in a mixed-initiative, that is partly by hand and partly through automated analysis conducted by the system. This automation is based on a set of syntactic linking rules maintained in a dictionary. Application code is represented by the FUI rectangle.

Fig. 2-36 shows possible design representations. Window (a) is the task model. A link goes from leave tasks to action specification in window (b). The UAN specification is linked onto architectural components on window (c). Architectural components are in their turn linked to the application code.
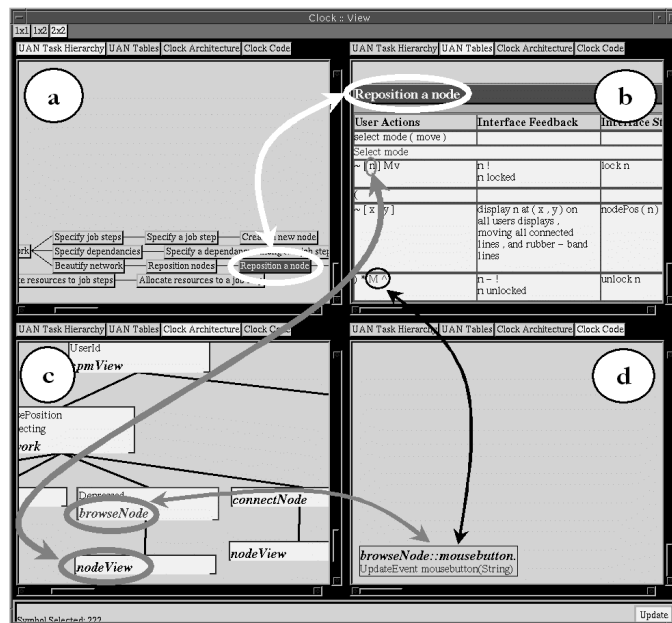
Figure 2-36 Model editor of Vista Tool (model linking is represented by arrows)

2.3.5.o Morph

MORPH [Moor96, Moor97] identifies basic user interaction tasks (i.e., interactor-type independent level of Fig. 2-6) in legacy code by applying static program analysis techniques, including: control flow analysis, data flow analysis, and pattern matching (Fig. 2-37). The resulting model is then used to transform the detected abstractions in a graphical environment from a specific widget toolkit.
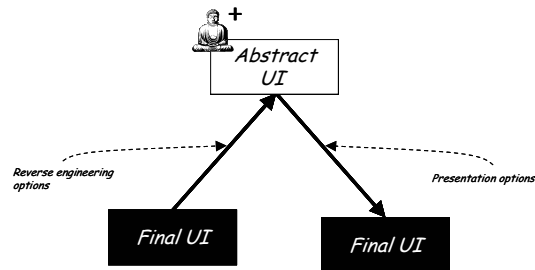
Figure 2-37 Development process in Morph

The original code is then modified to take into account the new dialogue structure of GUIs. MORPH is part of a larger environment called MORALE supporting complete reengineering process. Morph actually supports separation of concern with only one level of abstraction, the engine being driven by some parameters that give already some degree of flexibility. Operationalization is partially achieved: introducing a new heuristic or rule requires some intervention of MORPH's developer.
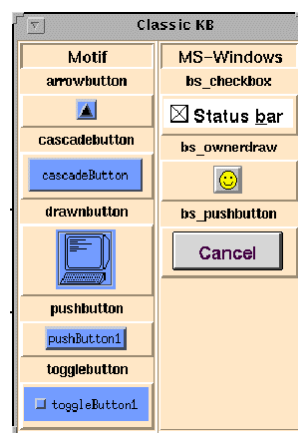
Figure 2-38 Selection rules in Morph

2.3.5.p   More

MORE [Berg04] produces applications that are device independent. A Platform Independent Application (PIA) can be created either by a design tool or by abstracting a FUI thanks to a generalization process (Fig. 2-39).

Generalization is done by reverse engineering of HTML code. This process starts with the detection of interaction elements. Secondly, the properties and semantic

information of these elements can be inferred. A specialized engine with a device profile then creates another application specialized for this particular device. Similarly to MORPH, operationalization and designers's control remain partial.
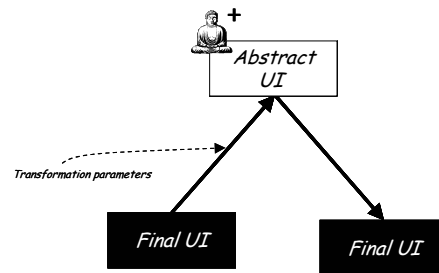
Figure 2-39 More Development Process

2.3.5.q   Tamex

The reengineering process in TAMEX [Elra01] allows one to produce HTML UIs composed of data contained in several other Web pages. The approach (Fig. 2-40) followed by Tamex is based on the concept of task-specific mediation: information sources within an application domain are encapsulated in wrapper agents (data extraction) interacting with an intelligent intermediary agent, the mediator (aggregate data).
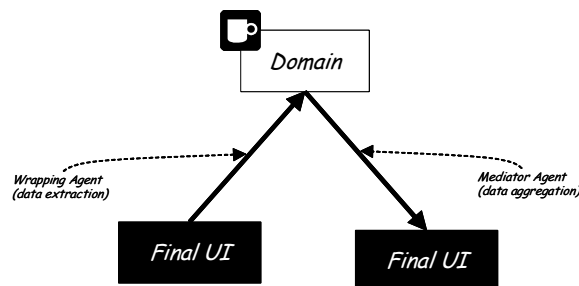
Figure 2-40 Tamex Development Process

XML is used as an intermediate data structure for information exchange and as a modeling language for the mediator's domain ontology and task structure. The information extraction is done with an XPath-based algorithm for generating extraction rules from HTML.

2.3.5.r   WebRevenge

WebRevenge [Paga03] analyzes Web site HTML code to automatically detect an underlying logical interaction design (Fig. 2-41). Such a design is represented through task models that describe how activities should be performed to reach users' goals. WebRevenge is capable of deducing a task model from web pages, which is different from ReversiXML [Boui04], which only recovers the presentation of the concrete UI.
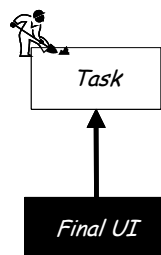
Figure 2-41 Web Revenge Design Process

2.3.6   Comparison on MBIDEs

The present section gives a synthetic overview of MBIDEs surveyed in this chapter. For this purpose, two families of properties are segregated: properties regarding conceptual content of methodologies, and properties regarding the model transformations underlying these methodologies. Table 2-1 and Table 2-2 sum up this comparative analysis.

2.3.6.a   Ontological properties

Mod. (Models): describes the type of models manipulated by the methodology. T = task, Do = Domain, Di = dialog, AUI =abstract presentation, AUI = abstract presentation in terms of user actions and/or interaction space definition, CUI = concrete user interface,   U = user, C = context.

Lnk (Linking): indicates the type of manual linking between different models involved in the methodology. For instance, T ↔ Do indicates that task model concepts could be manually linked to domain model concepts.

Sep. (Separation of concerns): indicates the extent to which concepts relevant to each category are separated within the methodology.

Trg (Target Language): designates the languages of the UI to produce.

Uni. (Uniformity of formalism): refers to the representation formats of different models in the methodology, whether they are represented using a same formalism. Value: √ for yes and ✘ for no.

Avl. Mod. (Availability of models): refers to the possibility for an external tool to process the manipulated models. Possible values:

- ✘: models are stored in an internal format not made explicit e.g., models are tightly coupled with the tools.
- √: means that an external format for models exists e.g. models are available under a machine understandable format. A typical form is an XML language.

Ext. Mod. (Extensibility of models definition): refers to the possibility of extending definitions of models with new elements.

- Orig.: means that models were intended to be extensible, but only by the originator of the methodology. This guarantees some interoperability of tools around a language.
- Design: means that models are extensible and the designer (e.g., the tool user) is responsible for this extension.
- ✘ : means that no mechanism supports model extension e.g., the system is bundled with a particular set of model definitions.

2.3.6.b Methodological properties

Dvt (Development path): indicates the type of development path that is covered by the methodology. Possible values:

- Fo: means forward engineering
- Re: means reverse engineering
- Ad: means adaptation to context of use.

Tra. (Transformation types): expresses the type of transformations that are supported automatically or semi-automatically by the tool. Ex "(T, Do, C) ↦ AUI illustrates the transformation process in Teresa e.g., an AUI is transformed from the combination of a task model, a domain model, and a context model.

Tra. For. (Transformation formalism): refers to the formalism exploited to represent

transformation rules. Possible values:

- OA: stands for opportunistic algorithm. It means that the transformations are hard coded and varying according to the type of manipulation at hand. In other words there is no homogeneity in the way models are processed.
- PR : stands for production rules e.g., rules under the format "If Condition Then assertion".
- GR: stands for graph rewriting. The technique exposed in this dissertation.

(Avl Tra.) Availability of transformation rules: refers to the availability of the transformation rules for an external tool or method. Possible values: √ for yes and ✖ for no.

Extensibility of transformation rules: refers to the possibility of extending definitions of transformation rules with new rules.

- Orig.: means that an extension mechanism exists for rules but solely by the originator of the methodology.
- Design: means that rules are extensible by the designer.
- ✖: means that no mechanism supports transformation rules extension e.g., the rules are bundled with the system.

Traceability of transformations: indicates whether the application of transformation may be observed or not. Value: √ for yes and ✖ for no.

Pattern support: refers to the possibility of using patterns either as building blocks for model construction or as support for model transformation. Value: √ for yes and ✖ for no.

2. State of the Art

	Mod.	Lnk	Trg	Sep.	Avl Mod.	Ext. Mod
Adept	T, Do, AUI, CUI	T↔Do, T↔U	SmallTalk	Med.	✗	✗
Art Studio	T, C, Do, AUI, CUI	T↔C, T↔Do	Java	High	✗	✗
Trident	T, U, Do, Do+, Di, AUI, CUI	T, U ↔Do		Med.	✗	✗
FUSE	T, U, Do, AUI	T↔U, T↔Do	C++	Med.	✗	✗
Genova	Do, CUI	—	C++, Java,VB	High	✗	✗
Janus	Do, CUI	—	C++	High	✗	✗
JustUI	T, Do, AUI	(T↔Do) ↔AUI	Java, HTML, VB, Cold Fusion, C++	High	✗	✗
Mastermind	T, Do, AUI, CUI	T↔Do, T↔AUI, AUI ↔Do	C++	High	✗	✗
Mobi-D	T, Do, U, AUI	T↔Do, T↔U, Do↔U	C++	High	√	✗
Tadeus	T, Do, U, AUI	T↔U, T↔Do, (T,D,U) ↔AUI	UIMS file	High	✗	✗
Teallach	Do, T, AUI+	T↔Do, Do ↔ AUI+, T ↔ AUI	Java	High	✗	✗
Teresa	T, AUI, CUI, (Do), C	T↔Do, T↔C	HTML, VXML	Med	√	✗
Seescoa	T, Do, AUI, CUI, Di, C	T↔Do+, T↔C	Java	High	√/✗	✗
Vista	T, Di, Do	T↔Di; T↔Do, Di↔Do, Di↔FUI	C++	High	✗	✗
Morph	AUI+	—	Motif, MFC	?	✗	✗
More	AUI+	—	HTML	Low	✗	✗
Tamex	Do, CUI	—	HTML	Low	✗	✗
Web Revenge	T	—	cttXML	Low	√	✗
Our methodology	T, Do, C, AUI, CUI	T↔Do, T↔AUI, AUI ↔ CUI, CUI ↔AUI, T↔CUI, (T, Do,AUI,CUI) ↔C	Java, XHTML, Flash	High	√	Orig.

Table 2-1 Comparison of ontological aspects

2. State of the Art

	Dvt.		Tra For.	Avl. Tra.	Ext. Tra.	Tra. Tra.	Pat.
Adept	FO	(T, Do) ↦ AUI ↦ CUI (+U) ↦ FUI	PR	✗	✗	✗	✗
Art Studio	FO + AD	(T, Do, C) ↦AUI+↦CUI ↦ FUI	OA	✗	✗	✗	✗
Trident	FO	(T, Do, U) ↦ Do+,Di↦AUI ↦ CUI ↦ FUI	PR	✗	✗	✗	✗
FUSE	FO +AD	(T,Do,U) ↦ AUI ↦ FUI	OA	✗	✗	✗	√
Genova	FO	Do ↦ CUI ↦ FUI	OA	✗	✗	✗	✗
Janus	FO	Do ↦ CUI ↦ FUI	OA	✗	✗	✗	✗
JustUI	FO	(T, Do) ↦ AUI ↦ FUI	OA	✗	✗	✗	√
Mastermind	FO	(Do, T, AUI) ↦ CUI ↦FUI	OA	✗	✗	✗	✗
Mobi-D	FO + AD	(T,Do,U) ↦ AUI ↦ FUI	OA	✗	✗	✗	✗
Tadeus	FO	(T,Do, U) ↦ AUI ↦ FUI	OA	✗	✗	✗	✗
Teallach	FO	Do ↦ T, AUI ↦ T, T ↦ AUI, Do ↦ P, (Do,AUI,T) ↦ FUI	OA	✗	✗	✗	✗
Teresa	FO + AD	(T,Do,C) ↦AUI ↦ CUI ↦FUI	OA	✗	✗	✗	✗
Seescoa	FO + AD	T ↦ (AUI, Di) ↦ CUI ↦FUI	OA	✗	✗	✗	✗
Vista	FO	—	OA	✗	✗	✗	✗
Morph	RE	FUI↦ AUI+ ↦FUI	OA	✗	✗		
More	RE + AD	FUI↦ AUI+ ↦FUI	OA	✗	✗		
Tamex	RE	FUI↦ Do ↦CUI ↦FUI	OA	✗	✗		
WebRevenge	RE	FUI ↦ T	OA	✗	✗		
Our Methodology	FO + AD + RE	FO : (T ,Do) ↦ AUI ↦ CUI ↦ FUI RE : FUI ↦ CUI ↦ AUI ↦ (T, Do) AD: (T, Do,C) ↦ (T',Do',C'), (AUI, C) ↦ (AUI', C'), (CUI, C) ↦ (CUI',C') Or any other combination of transformation	GR	√	Design.	√	√

Table 2-2 Comparison of methodological aspects

76

2.4 Conclusion

Three UI development approaches were considered in this state of the art:

An *exploratory approach* provides us with a means of quickly collecting user's feedback. In this sense, it allows early evaluation and contributes to interface quality. The problem is that when it comes to produce a running UI, the developer has to completely re-implement the previously done job in a genuine programming language. This offers no guarantee of consistency with what was done before and, consequently, endangers the benefits obtained from an early user feedback. In consequence, exploratory approach should be taken as it is and stays confined in the limits of early requirement development steps.

A *programming approach* allows a straightforward implementation of a final interface. In terms of quality criteria, these approaches vary depending on the degree of portability, the resource consumption (expressed in time units, monetary units, lines of code, etc), and the ease of use (which depends on provided tool support, intuitiveness of the concepts, legibility of the code, etc). The programming approach provides no guarantee of quality *per se*. Programming (and maintaining) a UI without any method can be a haphazard activity. It gives no guarantee for regularity. "Rushing to code" without any structure favors a "trial and error" method. The result of such a work will highly depend on contingency factors such as the developer's experience or the development context. Furthermore, communicability between stakeholders is hindered. Programming languages are poor communication mechanisms. Stakeholders will hardly reason about UI properties in the programming approach. Programming approach should be taken as it is. Programming an interface is not engineering it !

A *specification-based* approach provides us with means to specify relevant properties of a UI at various levels of abstraction. This approach has many benefits notably of being reproducible and allowing high level reasoning.

To conclude on these approaches we operate a three step analysis:

First, a set of selected *observations* is provided. An observation is a synthetic and *descriptive* assessment (as opposed to a *normative* assessment) that is made regarding properties of surveyed transformational methodologies.

Second, *shortcomings* are outlined from observations. A shortcoming is a *normative* assessment that is made regarding a property of surveyed transformational methodologies. A shortcoming is normative in the sense that it positions the state of the art with respect to ideal properties identified in the software engineering literature.

Third, a set of *requirements* for a solution to overcome the above mentioned shortcomings is identified. The internal validity of the solution proposed in this dissertation will be assessed with respect to this set of requirements.

2.4.1   Observations

Observation 1: Methodological diversity. Surveyed tools can be categorized into different categories depending on their main goal. Older tools like Janus, Trident, Teallach focus on forward engineering user interface starting from models that abstract away definitions of the interface itself. Mostly coming from the reverse engineering community, tools like Tamex, More, Morph, and WebRevenge shed a new light on the need for rich abstract models able to describe UI aspects in a way that is independent of implementation details. More recent contributions like Teresa, Seescoa, and ArtStudio introduce new abstractions to tackle the, so called, problem of multi-context user interfaces. Again these methods raise the need for high level abstraction allowing a description of a UI independently of the context in which it is supposed to be implemented. This question is still a hot research topic. Two methods adopt slightly different goals than the one described above. Mastermind's originality lies in its proposal to compose models at run-time. For this purpose it proposes a set of inter-model relationships that becomes first class citizens when it comes to generate a final UI. The problem of model-to-model transformation is out of the scope of Mastermind. Vista has a different focus as its main goal is to correlate artifacts being used during a software development process. Vista also proposes a set of inter-model relationships. Interestingly, part of the discovery of these links is done automatically.

Observation 2: Inter-method conceptual similarities. A striking convergence in using certain concepts is to be noticed among all the tools that have been presented. A domain model is defined in all the presented methods, a task model is used in a great share of them, a use of UI models (at various levels of abstraction) is also present in all methods. Regarding dialog models, things seem a

little bit more heterogeneous. Dialog graphs are presented in Tadeus, Activity-Chaining Graphs in Trident, Statecharts and Petri Nets in Mastermind, State Transition Networks in Seescoa, Hierarchical Interaction Templates in Fuse. In Teresa, the user task model itself constitutes the pivot element to organize the dialog of the interface elements. A presentation model seems in all methods a hierarchic decomposition of elements realizing an abstraction from toolkit specificities. A major common aspect of all the methods is that they all propose one or several levels of abstractions for describing the presentation of the UI.

Observation 3: Inter-method conceptual dissimilarities. If a coarse-grain similarity can be assessed (observation 2), a low level comparison of concepts reveals important dissimilarities between methods. Comparing these model variations is a tedious activity. As a consequence, cross-method understanding is hard to gain.

Observation 4: Intra-method heterogeneity. Inside methods, specification models are numerous and do not easily relate to each other. Each model gives an insight on some particular aspect of the UI. One model is good for expressing constraints, another for expressing the behavior and the other the presentation. While these models may be necessary, they remain hard to integrate because their modeling primitives are heterogeneous, because they may appear inconsistent with each other, because their relationships are not explicitly defined, because their heterogeneous syntaxes raise barriers for their integration.

Observation 5: Conceptual closeness. No method seems to have a particular concern for extension possibilities of their underlying ontology.

Observation 6: A focus on graphical modality. All (except one) environments deal at the first place with graphical modality. Only Teresa is concerned with auditory interfaces.

Observation 7: Transformations are not first class citizens. Transformations are in most methods hidden to the designer (i.e., built-in), untraceable and, not modifiable. In some environments, though, rules can be parameterized by dialog wizards (Mobi-D, Genova), by adjunction of templates (FUSE, Trident), or by model annotation (Janus, Artstudio). In no method a designer is provided with a stand-alone language allowing her to define custom transformation rules.

Observation 8: Multiplicity of transformation formats. For the approaches where transformations can be identified, a multiplicity of formats, and underlying paradigms, can be observed. Adept and ArtStudio use production rules, Seescoa use XSLT transformations and afterwards Java code, Trident uses a prolog-like expert system and selection trees, Mobi-D uses C++ algorithms, Genius is based on declarative correspondence tables.

Observation 9: Methodological conglutination of concern. transformation rules proposed in methods (when they are visible) are not dissociated according the type of operations they realize on models. A rule partitioning is nowhere made explicit to the designer.

Observation 10: Single entry point, single exit point. Methods define their development process with one single entry point (i.e., the development process starts from an imposed artifact) and one single exit point (i.e., the artifact resulting from the development cycle is fixed by the method).

Observation 11: Methodological closeness. Proposed methodologies commit to the definition of development steps. The sequence in which development steps may be arranged is in no case modifiable.

2.4.2   Shortcomings

From these observations, we can conclude by presenting several shortcomings:

Methodological shortcomings

Shortcoming 1: Lack of ontological explicitness – A few methods define in an explicit manner their underlying concepts. Concepts are generally bounded to tools or methodological recommendations, thus preventing a designer to grasp the conceptual foundations of a methodology (Obs. 1, 3, 4).

Shortcoming 2: Lack of ontological rigour – When a method explicitly defines its ontology, the preciseness of concepts definitions largely varies from one method to another. In addition, concepts are seldom formally expressed., especially the relationships between the ontological concepts (Obs. 1, 2, 3, 4).

Shortcoming 3: Lack of ontological commitment – The ontological commitment refers to a shared understanding of concepts among a scientific community. The fact that a few ontologies have been defined so far prevents convergence around a set of concepts (Obs. 1, 2, 3, 4).

Shortcoming 4: Lack of communication of concepts – Research teams tend to conduct their researches and developments on their own models. Conceptual consolidation across methods is difficult. Cross-method understanding is a tedious and time-consuming activity because it necessitates understanding each peculiarity of each method and establishing correspondence between them. As a consequence, communication among researchers is made complex (Obs. 1, 3, 4).

Shortcoming 5: Lack of extendibility of concepts – When available, the concepts manipulated by methods are hardly extendible. This prevents the adaptation of methodologies to cover new model concepts, notably, the ones related to new interaction modalities (Obs. 5, 6).

Methodological shortcomings

Methodological shortcomings concern the way existing approaches concretize transformational development with the definition of methodological stages, steps (i.e., transitions between stages), and transformation catalogs to perform these steps.

Shortcoming 6: Lack of methodological explicitness – Existing approaches seriously lack of explicitness in the way they propose their catalog of transformations both to the designer and to researchers. The transformation catalogs are often implicitly maintained in the head of developers and designers and/or hard-coded in supporting software. Consequently, the transformational processes proposed in the literature consist essentially in black boxes. This lack of explicitness dramatically hampers methodological guidance (Obs. 1, 7, 8).

Shortcoming 7: Lack of methodological rigour – When development steps and transformation catalogs are made explicit the preciseness of their expression is limited. We are not aware of any formally defined transformation catalog in the domain of HCI (Obs. 8, 9).

Shortcoming 8: Lack of consistency in applying methodology – When such design knowledge exists, it is not always systematically, consistently and correctly

applied throughout the project or across projects. Methodological steps remain open to interpretation while lack of methodological explicitness hampers any structured reasoning on the application of transformations (Obs. 4, 9).

Shortcoming 9: Lack of communication of transformation catalogs – Consequently to the lack of explicitness, the exchange of knowledge regarding transformation catalogs can be hardly achieved. Even when transformation catalogs are made explicit in tools, their heterogeneous formats prevents the reuse of transformations outside the context for which they were designed (Obs. 7, 8).

Shortcoming 10: Lack of predictability of transformation – The implicitness of transformations decreases the predictability of the transformation results. This causes a frequent reproach made to transformational development of user interfaces [Myers95] (Obs. 4, 7, 8, 9).

Shortcoming 11: Lack of modifiability of transformation catalogs – Developing UIs is about making heuristic decisions in a vast design space. Transformations have consequently an inherent heuristic nature as they try to translate into algorithms part of these design decisions. Proposed methods offer very little possibilities to the designer to modify built-in heuristics: adding, deleting, modifying, reusing transformations is almost impossible (Obs. 7, 10, 11).

Shortcoming 12: Lack of flexibility in methodological steps – Methods come usually with their models, their development steps. Due to the implicitness of their transformation formalism it is almost impossible to tailor the proposed methodological steps to the designers' needs and the project context. Flexibility is a notorious requirement for user interface development methods [Brow97] (Obs. 7, 8, 10, 11).

These shortcomings lead us to conclude that transformational development of user interfaces can be improved along several dimensions. We provide hereafter a list of requirement we seek to address with this dissertation. Some of these requirements are motivated by the above observations and shortcomings, some are desirable properties found in the literature that apply on any methodology.

2.4.3   Ontological Requirements

Requirement 1: Ontological explicitness – states that our ontology should be defined externally to any methodology manipulating it and in an explicit way that facilitates its dissemination and manipulation among stakeholders (Motivation: Short. 1).

Requirement 2: Expressivity – means that a conceptual framework should provide enough details to address problems that motivated the elicitation of its constituent concepts. In our context models should, at least, provide enough details to allow an implementation of the system it describes. This essential requirement is not fulfilled by many formal methods, for instance those focusing on verifying state properties of the system that is being built (Motivation: general principle in software engineering, Obs. 6).

Requirement 3: Human readable – means that the provided ontology should be proposed in a format that enables its legibility by a human agent (Motivation: Short. 1, 3, 4).

Requirement 4: Formality – states that models are expressed in such a level of accuracy that it enables automatic reasoning on their properties. (Motivation: Short. 2, 4).

Requirement 5: Machine readable – states that the proposed ontology should be legible by a machine. (Motivation: Short. 1, 2, 3, 4).

Requirement 6: Ontological separation of concern – states that models should differentiate aspects of the problem at hand [Parna72, Dijk76]. Models defined in our methodology should capture and, segregate, different levels of abstractions (Motivation: general principle of software engineering).

Requirement 7: Verifiability of specification – is defined as: "the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases" [Meye97]. Applied to specification, verifiability refers to the possibility of checking easily desirable properties (e.g., consistency, usability criteria). This requirement is facilitated by formality and explicitness. (Motivation: general principle of software engineering, Short. 2).

Requirement 8: Ontological homogeneity – refers to the property of a set of concepts of being defined using a common syntax. All models concepts should be described in a single formalism that facilitates their integration and processing (Motivation: Short. 4).

Requirement 9: Reuse of specifications – refers to the possibility of reusing whole or a part of a specification for another system. The proposed framework should facilitate reusing specifications (Motivation: Short. 3, 4, general principle in software engineering [Meye97]).

Requirement 10: Ontological extendibility – refers to the ease of adapting a conceptual structure to the occurrence of newly elicited concepts. HCI is a vast area covering the definition of multiple types of interfaces, interaction techniques, and interaction contexts. A specification language should be equipped with extension mechanisms to allow its evolution in parallel with the artifact it seeks to model. (Motivation: Short. 5, general principle of software engineering).

Requirement 11: Standards – states that the expression means used to represent our ontology should rely on well accepted standards in the software engineering community. (Motivation: Short. 3, 4).

2.4.4 Methodological Requirements

Requirement 12: Methodological explicitness – states that the constituent steps of our methodology should be defined in a way that facilitates the comprehension of its internal logic and its application. (Motivation: Short. 6).

Requirement 13: Methodological flexibility – refers to the ability to initiate the development from any development stage (i.e., multiple entry points) and to terminate it at any development stage (i.e., multiple exit points). (Motivation: Short. 12).

Requirement 14: Methodological formality – states that development steps should be expressed in such a level of accuracy that it enables an unambiguous interpretation of the process they describe. (Motivation: Short. 7, 9).

Requirement 15: Executability – states that development steps should be expressed in such a level of accuracy that it is possible to execute them by an automaton. (Motivation: general principle with transformational development [Send03]).

Requirement 16: Methodological separation of concern. – refers to a partitioning of methodological steps according to the process types they realize (Motivation: Obs. 9, Short. 8, general principle in software engineering).

Requirement 17: Methodological extendibility – refers to the ability left to the designer to extend the development steps proposed in a methodology. (Motivation: Short. 11, 12).

Requirement 18: Methodological Homogeneity – refers to the property of methodological steps of being defined using a common syntax. All transformation steps should be described in a single formalism that facilitates their understanding and processing (Motivation: Short. 8, Obs. 8, 4).

Requirement 19: Predictability – refers to the possibility provided by a methodology to foretell the result of the application of development steps. (Motivation: Short. 10).

Requirement 20: Traceability – is defined [IEEE90] as the "degree to which a relationship can be established between two or more products (i.e., here models) of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another" (Motivation: general principle in software engineering, Short. 6).

Requirement 21: Correctness – may be defined as the ability of a software to perform their exact tasks [Meye97]. In the context transformational development, correctness can be defined as the adequacy of an artifact A with respect to the other artifact(s) B such that B is the source artifact that was used to derive A (Motivation: Short. 7)

Requirement 22: Support for tool interoperability – Tool interoperability refers to the possibility of reusing the output provided by a tool into another tool. Our method should foster interoperability of tools working on specification models e.g., editors, critiquing tools, code generators, interpreter (Motivation: Short. 9).

Requirement 23: Methodological reuse – refers to the possibility in a methodology to capitalize on the knowledge defined by designers to perform development steps and re-using this knowledge for other developments. (Motivation: Short. 9, general principle in software engineering).

Chapter 3 An Ontology for User Interface Specification

3.1 Introduction

Chapter 2 concluded with a list of observations on the state of the art in transformational development of UIs. From these observations a set of shortcomings outlined some deficiencies in the approaches described so far in the literature. A set of requirements was finally elicited to address these shortcomings.

This third chapter addresses the ontological shortcomings and requirements of Chapter 2 by defining an original ontology aimed at describing various concepts relevant to UI development.

The word "ontology" initially comes from the area of philosophy where it is a branch of metaphysics concerned with the nature and relations of being.
It is also defined as a particular theory about the nature of being or the kinds of existents [Merr04]. In the context of information sciences, an ontology is a formal specification of a conceptualization [Grub93].

A conceptualization is a simplified representation of the world produced for some purpose. An ontology is, thus, a set of descriptions of the concepts and relationships within a field of knowledge used among a community of agents (humans or computers). Ontologies constrain the interpretation of concepts within a domain.

3. An Ontology for User Interface Specification

The main purposes of an ontology is to enable communication between computer systems in a way that is independent of the individual system technologies, information architectures and application domain.

A key ingredient of an ontology is a vocabulary of basic terms and a precise definition of what those terms mean. The terms in an ontology are selected with great care, ensuring that the most basic (abstract) foundational concepts and distinctions are defined and specified. The terms chosen form a complete set, whose relationship one to another is defined using formal or semi-formal techniques.

Defining an ontology is not only about enumerating its constituent concepts. Formal foundations used to build an ontology have to be identified and defined precisely. One should also, and finally, define the appearance of the ontology considering that an ontology containing the same concepts may be materialized in the real world through different communication channels. The concepts of "apple" may be for instance simply orally pronounced or written, but can also be drawn on a piece of paper or mimed.

The framework presented in Fig. 3-1 (inspired from [Bare02]) distinguishes three essential components to introduce any ontology: a conceptual content (i.e., abstract concepts), the formal foundations used to represent the ontology (i.e., abstract syntax), the definition of the appearance of the ontology (i.e., concrete syntax). The structure of this chapter reflects these three aspects.
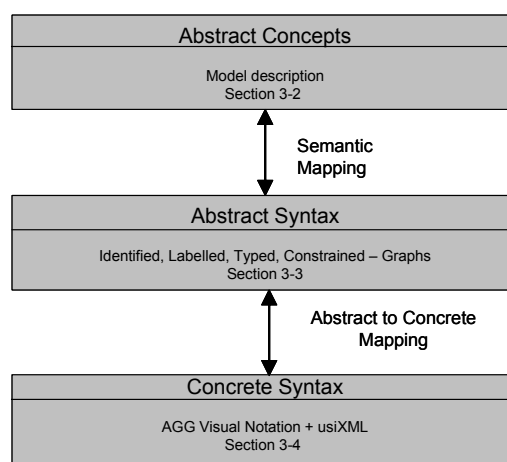
Figure 3-1 Our language structure

3. An Ontology for User Interface Specification

Section 3-2 presents the conceptual content of our language. After introducing the concept of viewpoint, each model type will be described along with its constituent concepts. UML class diagrams are used for this purpose along with definitions in natural language.

Section 3-3, presents the mathematical structures underlying our ontology, i.e., its abstract syntax. The notion of "directed, identified, labeled typed graph" is introduced, motivated and exposed.

Section 3-4 presents the concrete syntax of our language. Two syntaxes are going to be presented: a visual (i.e., graphical) and a textual one (i.e., an XML language called UsiXML).

3.2 Conceptual Content of the Language

Separation of concerns has been defined after Dikstra [Dijk76]. This principle states that the different aspects of a problem should be isolated from one to each other. Separation of concerns allows studying fractions of a matter in an independent manner while modularizing this matter. A concern gathers properties relevant to one perspective that can be maintained on an artifact.

A *viewpoint* is the materialization of a concern. A viewpoint is associated with a perspective of the thing it models. For instance, in the field of architecture, it is possible to maintain several viewpoints on a building with respect to a specific property of this building. Preliminary sketches enable a global vision of the building to build. A "general prescription document" describes the main functionalities according to what was agreed with the customer. A "detailed prescription document" provides a detailed vision of the technical solutions that must be implemented to meet the general prescription document. The detailed prescription document may segregate aspects such as electrical equipment, or sanitary equipment.

In an analogous manner, a structuring in viewpoints, in the context of software development processes, allows segregating different aspects of the application being built. Viewpoints may be organized in hierarchy according to the level of abstraction they provide.

Several viewpoints are identified and motivated in [Calv03]. Our conceptual framework relies on this work. These viewpoints are hierarchically organized around a property of independence it holds with respect to the context in which the system is being built. Four viewpoints are defined:

1. *Final UI* (FUI): is the operational UI i.e. any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after compilation of code in an interactive development environment). The final UI has two possible representations, the code and the rendering. The code concerns the UI representation either as a set of instructions (in a procedural language) or as a set of assertions (in a declarative language), or a mix of both. The rendering of the system is a user perceivable representation of the UI.

2. *Concrete UI* (CUI): provides a specification of the user interface in terms of Concrete Interaction Objects and concrete relationships. Concrete objects and relationships provide a vocabulary that is as independent as possible of any programming language or toolkit used to implement the UI. A CUI is an abstraction of the FUI. A CUI defines widgets, layout and interface navigation and detailed behavior. Although a CUI makes explicit to the final Look & Feel of a FUI, it is still a mock-up that runs only within a particular environment. A CUI can also be considered as a reification (i.e., a concretization) of an AUI at the upper level.

3. *Abstract UI* (AUI): provides a specification of the user interface in terms of Abstract Interaction Objects (AIO) and abstract relationships. Abstract objects and relationships provide us with a vocabulary that is as independent as possible of any modality (e.g., graphical interaction, vocal interaction, speech synthesis and recognition, video-based interaction, virtual, augmented or mixed reality). An AUI can also be defined as a canonical expression of the rendering of the domain concepts and tasks in a way that is independent from any modality of interaction. For example, in ARTStudio [Thev01], an AUI is a collection of related workspaces. The relations between the workspaces are inferred from the task relationships expressed at the upper level (task and concepts). AIOs are said to be widget-type independent (see Fig 2-6). An AUI defines interaction spaces by grouping AUIs (and implicitly tasks of the task model) according to various criteria (e.g., task model structural patterns, cognitive load analysis, semantic relationships identification). A set of abstract relationships is provided to organize AIOs in such a way that a derivation of navigation and layout is possible at the concrete level. An AUI is considered as an abstraction of a CUI with respect to modality.

4. *Task & Domain* (T&D): describe the various tasks to be carried out by the user in interaction with the system along with the domain-oriented concepts as they are required by these tasks to be performed. Domain objects are considered as instances of classes representing the concepts manipulated.

In addition to these viewpoints a context model is introduced to enable to associate any model element to the context(s) for which it is valid.

From a software engineering background, it is interesting to note that our viewpoint structuring can be compared (Table 3-1) to the Model-Driven

3. An Ontology for User Interface Specification

Architecture proposal provided by the Object Management Group [OMG01,Mill03]. Model-Driven Architecture 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 (note that the concept of platform remains very fuzzy in MDA proposal)..

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.

Model Driven Architecture	Our method
Computing Independent Model	Task and Domain
Platform Independent Model	(1) Abstract UI
	(2) Concrete UI
Platform Specific Model	—
Implementation	Final UI
Platform Model	Context Model

Table 3-1 A comparison of MDA models and our method

The current chapter describes the concepts needed to realize multi-path development of user interfaces. These concepts were elicited after a state of the art, partially presented in Chapter 2.

Several ontological formats have been proposed in the literature to formalize ontologies. OWL Web Ontology Language [W3C04b] is mostly referenced in the literature. To describe the concepts of our ontology, UML class diagrams [OMG03a] are used along with natural language explanation and graphical illustrations when relevant. The use of class diagrams is motivated by the following arguments. First, our ontology was developed in the context of the "Cameleon european project". Communicating easily the content of our ontology was a major requirement. UML appeared to us an ideal vector as this notation has become a *de facto lingua franca* in the software engineering community. Second,

UML is an appropriate notation for describing conceptual schemas [Mart98]. Third, UML is supported by a wide variety of tools (e.g., graphical editors, documentation, generation) which was not the case of most existing ontology languages (including OWL) when our ontology was initiated. Fourth, UML class diagrams proved very useful as a documentation for our XML-based textual syntax as one class is associated with one XML element, one class attribute is associated with one XML attribute, composition relationships are associated with embeddings of XML elements and finally generalizations are translated by XML schemas [W3C01] generalizations.

3.2.1 Task Model

> *A task model* describes the various tasks to be carried out by a user in interaction with an interactive system.

After a comparison of a dozen of task modeling techniques [Limb03], an extended version of ConcurTaskTree (CTT) [Pate97] has been selected to represent user's tasks along with their logical and temporal ordering. This choice has been done for the following reasons:

- *Software engineering orientation*. CTT, proposes a set of task attributes and task relationships, that is more oriented to software engineering than psycho-cognitive analysis (e.g., TKS [John92]).
- *Formalism*. CTT combines hierarchical structuring of tasks to temporal ordering of elements with a subset of LOTOS operators. LOTOS is a grounded formal notation in software engineering for specifying the ordering of processes in time [Pate97].
- *Communication*. CTT is supported by a usable tool (i.e., CTTE) and a graphical notation that facilitates its dissemination and communication among practitioners.
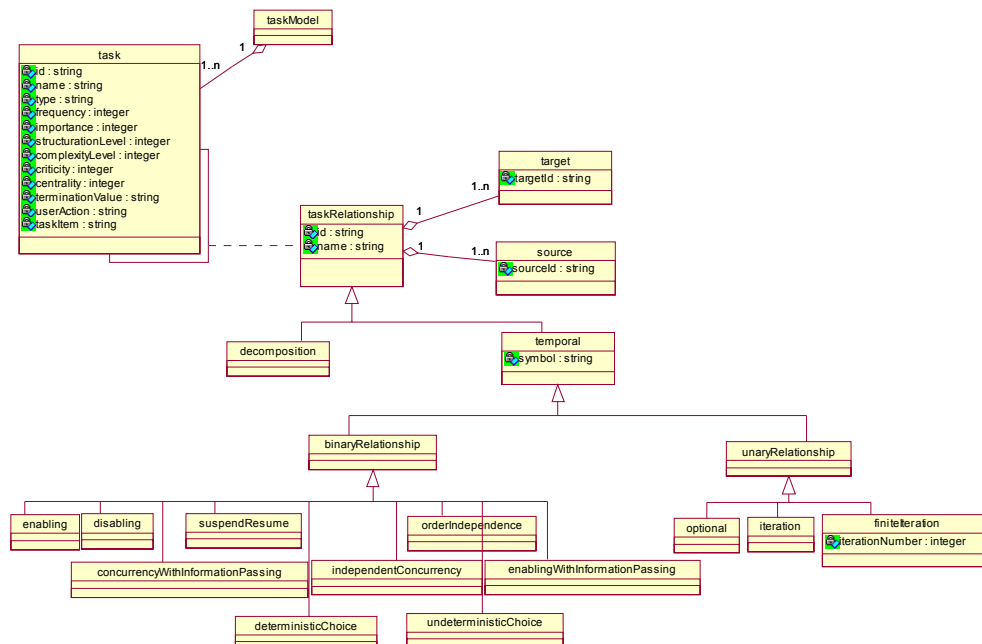

Figure 3-1 Conceptual view on the task model

A task model is therefore composed of *tasks* and *task relationships* (Fig. 3-1).

Tasks are, notably, described with a name, and a type. Task *type* may be: user's, interactive, system or abstract. A user task refers to a cognitive action like taking a decision, or acquiring information. User tasks are notably useful to predict a task execution time. An interactive task involves an active interaction of the user with the system (e.g., selecting a value, browsing a collection of items). A system task is an action that is performed by the system (e.g., check a credit card number, display a banner). An abstract task is an intermediary construct allowing a grouping of tasks of different types. Tasks can also have attributes. A task *frequency* attribute is an assessment of the relative frequency of execution of a task. Task frequency is evaluated on a scale from 1 to 5. A task importance attribute assesses the relative importance of a task with respect to main user's goals. Task *importance* is evaluated on a scale from 1 to 5. A value of 1 means that a task has a low importance, 5 means that a task is very important. Frequency and importance are interesting attributes when it comes to adapt a UI to a constraining context imposing a UI to be pruned of some of its elements (e.g., as display space decreases it may be interesting to filter out widgets that allow the execution of unimportant tasks).

Action type and *action item* enable a refined expression of the nature of leaf tasks (sometimes called action tasks or leaf tasks). This expression is based on a taxonomy introduced by [Cons03] to qualify a UI in terms of abstract actions it supports (Fig. 3-2). The taxonomy is twofold: a verb describes the type of activity at hand; an expression designates the type of object on which the action is operated. By combining these two dimensions a derivation of interaction objects supposed to support a task becomes possible.

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
✔	action/operation*	Print symbol table, Color selected shape
✔	start/go/to	Begin consistency check, Confirm purchase
✔	stop/end/complete	Finish inspection session, Interrupt test
✔	select	Group member picker, Object selector
✔	create	New customer, Blank slide
✔	delete, erase	Break connection line, Clear form
✔	modify	Change shipping address, Edit client details
✔	move	Put into address list, Move up/down
✔	duplicate	Copy address, Duplicate slide
✔	perform (& return)	Object formatting, Set print layout
✔	toggle	Bold on/off, Encrypted mode
✔	view	Show file details, Switch to summary
SYMBOL	**INTERACTIVE FUNCTION**	**EXAMPLES**
☐	container*	Configuration holder, Employee history
⊟	element	Customer ID, Product thumbnail image
☰	collection	Personal addresses, Electrical Components
⚠	notification	Email delivery failure, Controller status

Figure 3-2 Constantine's Canonical Abstractions

Task relationships are of two main types: decomposition and temporal.

- *Decomposition* enables representing the hierarchical structure of a task tree.
- *Temporal* allows specifying a temporal relationship between sibling tasks of a task tree. LOTOS [Pate97] operators are used here.

To illustrate task temporal relationships we propose an interpretation of each operator based on the "design by contract" paradigm introduced by [Meye97].

This paradigm promotes a use of a contract mechanism established with pre- and post conditions between different methods populating a (object-oriented) system. In this perspective, a pre-condition is an assertion on the system's state that a method *requires* to guarantee to *ensure* a post-condition after its execution.

To apply such a mechanism to task modeling concepts and their temporal relationships, two important types of assertions on state variables of the task model have to be defined: task *termination* and task *initiation*.

Task termination represents a possible conjunction of events allowing asserting that a task has been performed. For instance, if a task consists in "inputting" a value, it will terminate when a system event confirms the proper input of this value. A task consisting, for a train driver, to monitor a value (e.g., the train speed), each 15

minutes, will be considered has terminated if the driver explicitly pushes on a physical button.

Task initiation represents a possible conjunction of events allowing asserting that a task has been initiated. For instance, a user has initiated the input of a value but it was not sent to the database yet.

In this perspective, the execution of an interactive task consists in "accumulating" events required to its termination. In consequence, it is, by principle, impossible to guarantee that any interactive task will terminate somewhere in the future as its termination conditions depend on a user task. With respect to the discussion in [Dix90], we could ironically argue that the user's behavior is the first cause of non-determinism of interactive systems. For system tasks, a value may be transmitted from the system to assess the termination of an operation.

We propose hereafter a set of tables providing an interpretation for each LOTOS operator. For each operator we define what a task requires to be initiated and what it ensures. A termination condition is also provided for each operator. This condition tells when two temporally related tasks can be considered as terminated.

- ▪ Enabling (T1 has to be finished in order to initiate T2)

T1 >> T2	T1	Requires: Ø Ensures: ?
	T2	Requires: T1.Termination Ensures : ?
	Termination	T1.Termination AND T2.Termination

- ▪ Non-deterministic choice (Once one task is finished the other cannot be accomplished anymore)

T1 π T2	T1	Requires: NOT (T2.Termination) Ensures: ?
	T2	Requires: NOT (T1.Termination) Ensures : ?
	Termination	T1.Termination XOR T2.Termination

- Deterministic Choice (Once one task is initiated, the other cannot be accomplished anymore)

T1 [ ] T2	T1	Requires: NOT (T2.Initiation) Ensures: ?
	T2	Requires: NOT (T1.Initiation) Ensures: ?
	Termination	T1.Termination XOR T2.Termination

- Parallelism (T1 is interleaved with T2)

T1 ||| T2	T1	Requires: Ø Ensures: ?
	T2	Requires: Ø Ensures : ?
	Termination	T1.Termination AND T2.Termination

- Sequential independence (Is equivalent to (T1>>T2) OR (T2 >>T1))

T1 |=| T2	T1	Requires: NOT(T2.Initiation) XOR T2.Termination Ensures: ?
	T2	Requires: NOT(T1.Initiation) XOR T1.Termination Ensures : ?
	Termination	T1.Termination AND T2.Termination

- Deactivation (T2 may interrupt T1 before the termination of T1;. T1 cannot be resumed after T2 has terminated.)
-

T1 |> T2	T1	Requires: Ø Ensures: Ø
	T2	Requires: T1.Initiation AND NOT(T1.Termination) Ensures: ?
	Termination	Termination XOR T2.Termination

- Suspend/Resume (T2 may interrupt T1 before the termination of T1. Once T2 is finished, T1 may be resumed.)

T1 |> T2	T1	Requires: Ø Ensures: ?
	T2	Requires: T1.initiation Ensures: ?
	Termination	T1.termination OR T2.Termination

- About Information Passing

Several temporal operators may be decorated with an information passing symbol: Enabling with information passing (symbol: "[]>"); parallelism with information passing (symbol: "|[]|"). Information passing means that two tasks synchronize on a same piece of data.

For these operators we have to introduce a new assertion on the task model state regarding the passing of data from one task to another, let's call it *data synchronized*.

- Enabling with information passing (T1 has to be finished in order to initiate T2 and T2 is synchronized with T1 on some piece of data)

T1 >> T2	T1	Requires: Ø Ensures: ?
	T2	Requires: T1.termination and dataSynchronized Ensures : ?
	Termination	T1.termination AND T2.Termination

- Parallelism with information passing (T1 is interleaved with T2 while they synchronize on some data)

T1 |[]| T2	T1	Requires: NOT(T2.initiated) OR dataSynchronized Ensures: ?
	T2	Requires: NOT(T1.initiated) OR dataSynchronized Ensures : ?
	Termination	T1.termination AND T2.Termination

Additional operators may affect a task in relation with itself, these operators are called unary temporal relationships (see Table 3-2).

T* (Iteration)	T can be iterated an infinite number of times
T(n) (Finite Iteration)	T can be iterated n times
[T] (optional)	T is optional

Table 3-2 Unary task relationships

Several additional constraints may be formulated on the consistency of a task model:

- There exists a maximum of one binary (i.e., temporal or decomposition) relationship between two tasks.
- If a task is decomposed into another task then this last task must have a brother task.

- There is only one root task. This means that there is only one element with no decomposition relationship pointing to it.

3.2.2   Domain Model

A *domain model* describes the real-world concepts, and their interactions as understood by users and the operations that are possible on these concepts [DSou99].

The domain model is generally developed by software engineers and given "as is" (often under the form of an Application Programming Interface (API)) to UI designers. The rest of the job consists of connecting the UI to the functional core API while respecting some architectural principles (e.g., Pac [Cout87], MVC [Reen79,Kras88]).

As shown in Chapter 2, many formal notations have been introduced to represent systems of concepts: frames, semantic networks, entity relationship schemas, structured data models. We selected UML class diagrams as the basis of expression for our domain model. We considered UML class diagrams as Extended Entity Relationship model (EER) [Teor86]. The main reason for this choice is that UML has become a lingua franca in the domain of software engineering (Req. 11: Standards) and is widely used in industrial practice.

Our meta-model of an UML class diagram is presented in Fig. 3-3. Several features have been added to the initial UML standard in order to better tackle the problem of transformational development of UIs. For instance, the domain n of values attached to attributes is described with a richer precision in order to allow widget selection (e.g., enumerated domains can be described extensively).
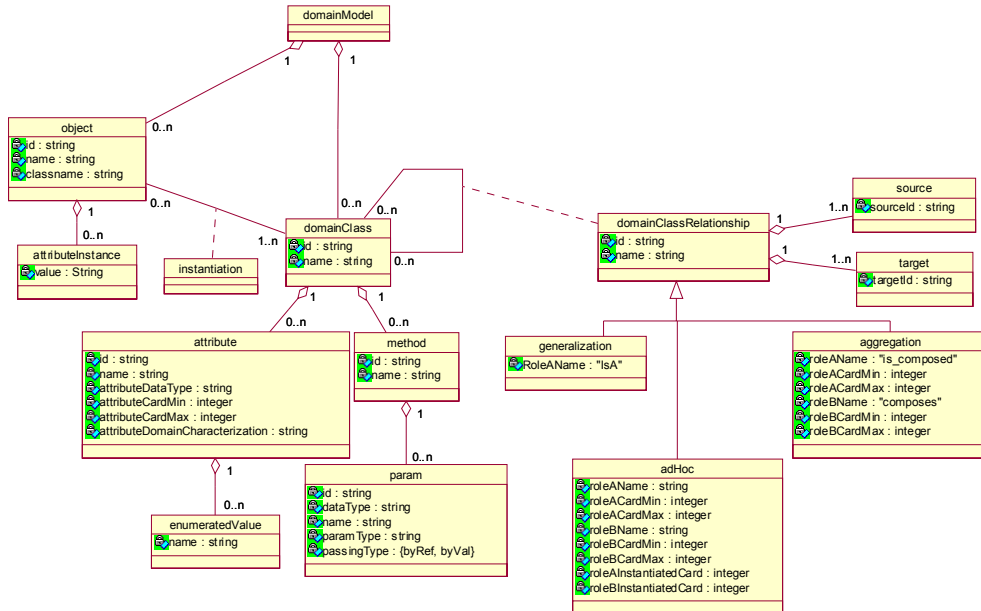
Figure 3-3 Conceptual model for a domain model

Domain model concepts are classes, attributes, methods, objects and domain relationships.

A *class* describes the characteristics of a set of objects sharing a set of common properties. A class is described with a name. A class may be composed of two types of features: attributes and methods.

An *attribute* enables a description of a particular feature of a class. The type of an attribute refers to common data types found in most programming language i.e., Boolean, char, string, integer, float. The type attribute may also make reference to an object type. The cardinality of an attribute indicates the number of values an attribute may be associated with. A cardinality can be specified by providing two integers: a minimal cardinality and a maximal cardinality. An original typology allows characterizing a type of domain for an attribute. Indeed, attributeDomainCharacterization takes the value of: interval, continuous interval, discrete interval, linear interval, circular interval, set[n] (where n is the number of possible values in an attribute domain). When used in combination with a task model, this typology helps to map domain attributes to a type of interaction object by which it will be rendered. For instance, a "choose element" task on an attribute with a circular interval enables the derivation of a (multi-state) toggle button.

Methods (in this context) are presences which are called either by objects of the domain or by user interface components. Methods manipulate object's attributes. Methods are, here, described with their signature i.e., with their name, type, and parameters. A set of predefined method names inspired from object oriented patterns are proposed to facilitate the definition of generic design heuristics. For instance, the CRUD pattern is used to refer to any method realizing a Create, Read, Update or Delete operation [Larm01].

Objects are instances of a class. An object is composed of attribute instances which may have values and define the state of an object.

Finally, *domain class relationships* describe various types of relationships between classes. They can be classified in three types: *generalization*, *aggregation*, and *ad hoc*. Class relationships are described with several attributes enabling the specification of role names and cardinalities.

3.2.3   Abstract User Interface Model

An *Abstract User Interface (AUI) model* is a user interface model that represents a canonical expression of the renderings and manipulation of the domain concepts and functions in a way that is as independent as possible from modalities and computing platform specificities.

An AUI (Fig. 3-4) is populated by *Abstract Interaction Objects (*AIO*)* and *abstract user interface relationships.* These concepts constitute a vocabulary that is independent of the modality and the computing resources for which a system is targeted at.

A *modality* (also called interaction technique) can be defined more precisely, after [Niga95], as the coupling of a physical device d with an interaction language *L: <d, L>*. Our language supports, at the concrete level, two modalities: speech (i.e. *auditory*) input and output and graphic (i.e., *graphical*) input and output.

Abstract Interaction Object (AIO) may be of two types *Abstract Individual Components (*AIC*)* and *Abstract Containers (*AC*).*
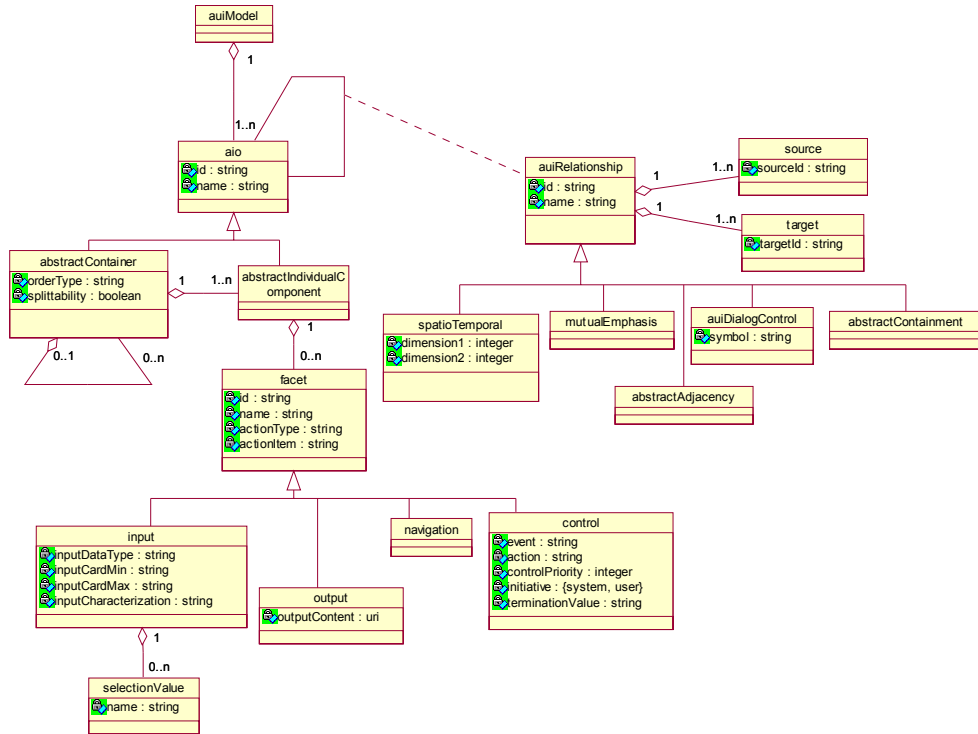
Figure 3-4 Concept model for abstract user interfaces

An *Abstract Individual Component* (*AIC*) is an abstraction that allows the description of interaction objects in a way that is independent of the modality in which it will be rendered in the physical world. Fig. 3-5 presents three possible reification of a set of abstract interaction objects for, respectively from left to right, tactile modality, auditory modality and 2D graphical modality. An AIC may be composed of multiple facets. Each facet describes a particular function an AIC may endorse in the physical world. Four main facets are identified:

- An *input* facet describes the input action supported by an AIC.
- An *output* facet describes what data may be presented to the user by an AIC.
- A *navigation* facet describes the possible container transition a particular AIC may enable.
- A *control* facet describes the links between an AIC and system functions i.e., methods from the domain model when existing.

A single AIC may assume several facets at the same time. The AIO that reifies this multi-facetted AIO will assume all those 'functionalities'. For instance, a CIO

may display an output while accepting an input from a user, ensure a transition between windows and trigger a method defined in the domain model.

Figure 3-5 Different reifications in different modalities for a set of control AIC (courtesy of C. Stephanidis)

An *Abstract Container* (AC) is an entity allowing a logical grouping of other abstract containers or abstract individual components. AC are said to support the execution of a set of logically/semantically connected tasks. They are called presentation units in [Boda95c] and work spaces in [Thev01]. An AC may be reified, at the concrete level, into one or more graphical containers like windows, dialog boxes, layout boxes or time slots in the case of auditory user interfaces.

Abstract User Interface Relationships (AUI relationship) are relationships that can be drawn between abstract interaction objects of all kinds.

Five types of abstract relationships may be defined at this level:

- *Decomposition* relationship allows specifying a hierarchical structure of abstract containers and abstract individual components.
- *AbstractAdjacency* relationship indicates that two AIO are logically adjacent.
- *Spatio-temporal* relationship allows a specification of a very precise layout in time or space in a way that is independent of any modality. For this purpose, the thirteen possible temporal relationships from Allen [Alle83] are considered. Basically, there are two types of temporal relationships (Table 3-3): before (sequential relationship) and simultaneous (that can be equal, meets, overlaps, during, starts, or finishes relationships). Each basic relationship has an inverse relationship, except the equal relationship which is symmetric. Although Allen relationships have been introduced to characterize temporal intervals, they are suitable for expressing constraints for space and time thanks to a space-time value. For example, in an "x before y" relationship, there is a space-time value greater than zero between x and y while in the "x meets y"

relationship the space-time value is equal zero between x and y. As relationships are abstract at the AUI level, the space-time value is left unspecified until needed at the CUI level.
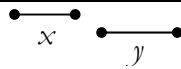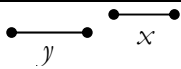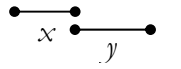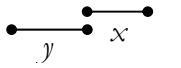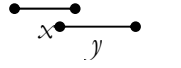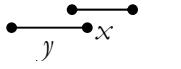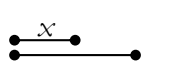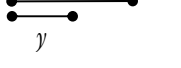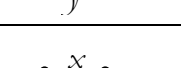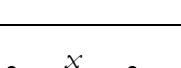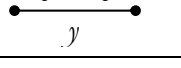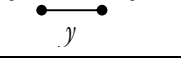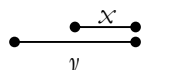
Relationship	Presentation	Relationship	Presentation
x before y (1)		x after y (8)	
x meets y (2)		y meets x (9)	
x overlays y (3)		y overlays x (10)	
x starts y (4)		x starts y (11)	
x during y (5)		y during x (12)	
x finishes y (6)		y finishes x (13)	
x equal y (7)			

Table 3-3 Allen relationships and their inverse.

Example 1: a *before* relationship can be used to specify that an AIO InputFirstname will be presented before another AIO InputLastname. At the CUI level, this will be turned into for instance: (1) In a graphical desktop: an edit box for entering the first name, followed by a second edit box for the last name in the same dialog box. In this case, the relationship represents in a 2-dimensional space along with a particular space interval. (2) In a mobile phone interface: a field for entering the first name is first displayed and when the user has completed the entry, a second field for entering the last name is displayed. In this case, the relationship represents navigation between two screens of a UI system.

So far, Allen relationships allow expressing physical constraints according to one dimension (1D) only: time or space. Allen relationships can be generalized to *n* dimensions for expressing similar constraints in a *n*D space, and consequently gain in precision. Here, the 2D generalization is kept to

express space relationships more precisely in any type of UI involving spatial expressions [Sung02, Lee03]. To exemplify this, let us assume two AIOs *A, B*. The spatial relationship between *A* and *B* is defined as follows: *Spatial_Composition* (*A,B*) = (*Ri* , *Rj*), where i, j $\in$ {1,…,13}, *Ri* is the identifier of the spatial relationship between *A* and *B* according to the *X* axis and *Rj* is the identifier of the spatial relationship between *A* and *B* according to the *Y* axis in the matrix reproduced in Fig. 3-7. When a spatial arrangement is expressed only according to one dimension, *Ri* = $\varnothing$ or *Rj* = $\varnothing$.
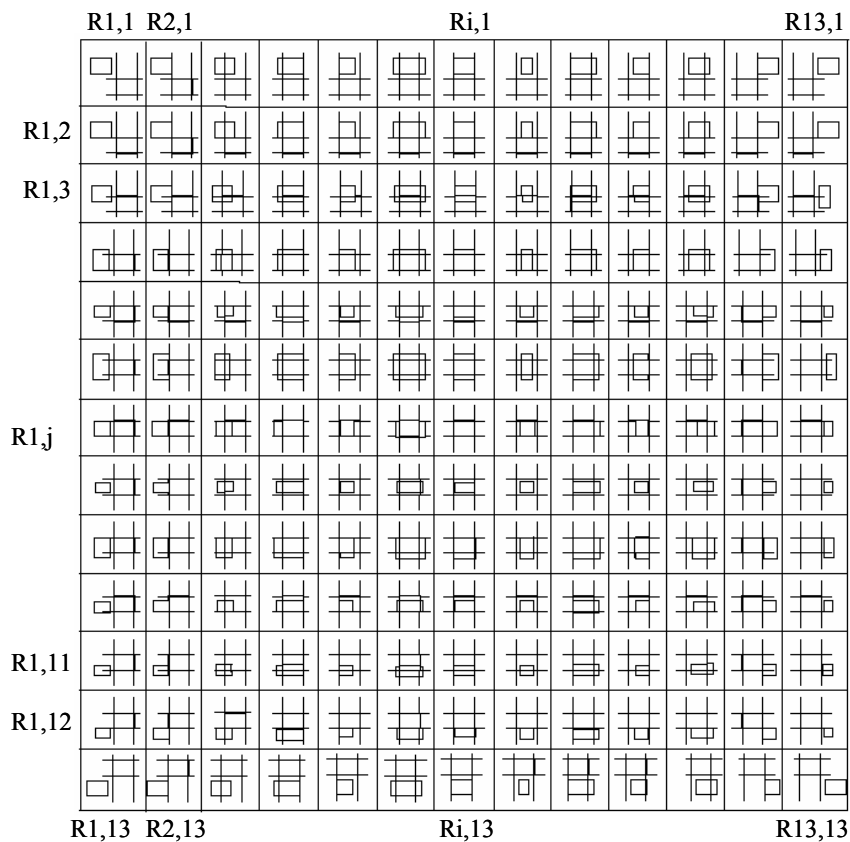
Figure 3-6 Matrix of 2D Allen Relationship

- *Dialog control* relationship allows a specification of a flow of control between the abstract interaction objects. Like for task models, LOTOS (see Sec. 3.2.1) operators are used for this purpose. For instance a relationship AIC1.EnterCountry []> AIC2.EnterProvince, indicates that AIC2 cannot be initiated while AIC1 is not achieved and that AIC1 has provided a value for the data on which the two components synchronize with. Like for tasks, an interpretation for each type of LOTOS operator may be provided in terms of

         pre/post-conditions, termination and initiation states. Dialog control relationships are defined in a same manner at the concrete UI level.

- *Mutual emphasis* relationship allows specifying that two components should be somehow differentiated at the concrete level. This relationship may be useful in a user interface where the probability of confusing two UI elements is high (e.g., in an airplane cockpit, a field displaying the angular speed and the absolute speed).

An additional constraint may be formulated about abstract models:

- There is only one root in the decomposition tree of abstract containers.

3.2.4   Concrete User Interface Model

> A *Concrete User Interface* (*CUI*) model is a UI model allowing a specification of an appearance and behavior of a UI with elements that can be perceived by users

By definition, a CUI is modality dependent as any CUI instance refers to the interaction modalities that have been selected for this UI. This reference can be unique in case of a "mono-modal" CUI or multiple in case of a multimodal CUI. Our language supports two modalities for the moment, speech and graphic. In reference to the definition of the modality found in [Niga95] and provided in Sec. 3.2.3. Speech modality is a combination of speech input and speech output. Speech Input is described as the couple <microphone, pseudo-natural language NL>, where NL is defined by a specific grammar. Speech output is similarly described as the couple <speech synthesizer, pseudo natural language NL>. Graphic modality is composed of a combination of graphic input and graphic output. Graphic input is described in terms of <pointing device PD, direct manipulation> where PD is generally a mouse. Graphic output corresponds to the couple <screen, drawing language> where a drawing language can be, for instance, procedural or declarative, pixel based or vector based.

In contrast to its modality dependence, a CUI remains toolkit independent as no CUI instance does refer to any physical element (i.e., toolkit elements or widget) of the computing platform. Nonetheless, a CUI description can be detailed enough to allow a complete rendering of a user interface.

A CUI model (Fig. 3-7) is composed of *Concrete Interaction Objects* (CIO) and *concrete relationships*. Concrete interaction objects and relationships are further refined into graphical objects and relationships and auditory objects and relationships. Other

types might complement these two categories as more modalities could be taken into account.
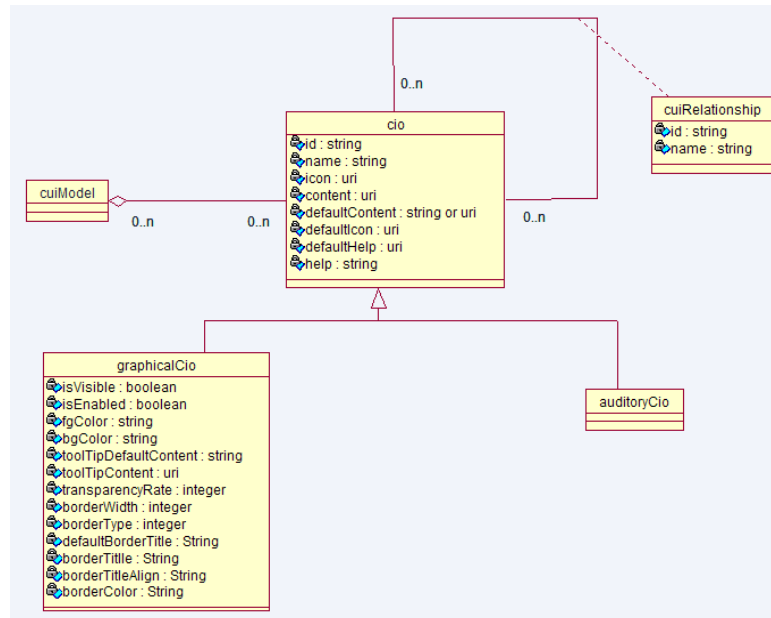
Figure 3-7 Root elements of the concrete user interface model

A Concrete Interaction Object (CIO) is defined as an entity that users can perceive and/or manipulate (e.g., a push button, a list box, a check box, a sound). A CIO realizes an abstraction of widget sets found in popular toolkits: graphical (Java Swing [Ecks98], HTML 4.01 [W3C99], Flash DRK6 [Macr04]) and auditory (earcons [Crea99] and VoiceXML 2.0 [W3C04]). In other words, CIOs allows an expression of UI elements that is independent of their actual rendering. Fig. 3-8 shows an example of different renderings for a menu element on three different platforms.
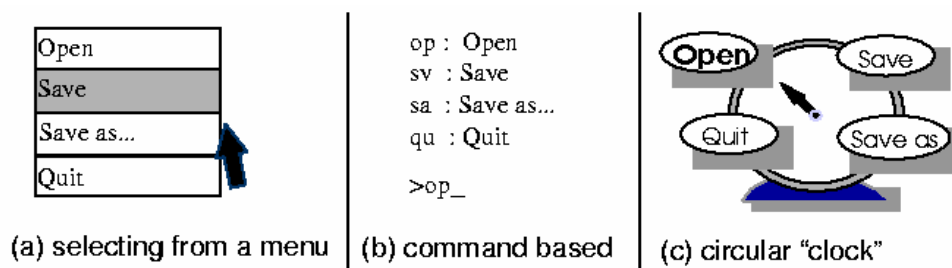
Figure 3-8 Examples of different graphical CIOs corresponding to a selection AIO [Courtesy of C. Stephanidis, ICS Forth]

CIOs are further classified depending on the modality they allow to support: graphical and auditory (Fig. 3-7). Graphical and auditory CIOs are further decomposed into containers and individual components. We have, thus, elements called Graphical Containers (GC), Graphical Individual Components (GIC), Auditory Containers (AC), Auditory Individual Components (AudIC).

The graphical part is the most detailed part of the language as a consequence of the complexity of this type of UI compared to pure auditory ones.
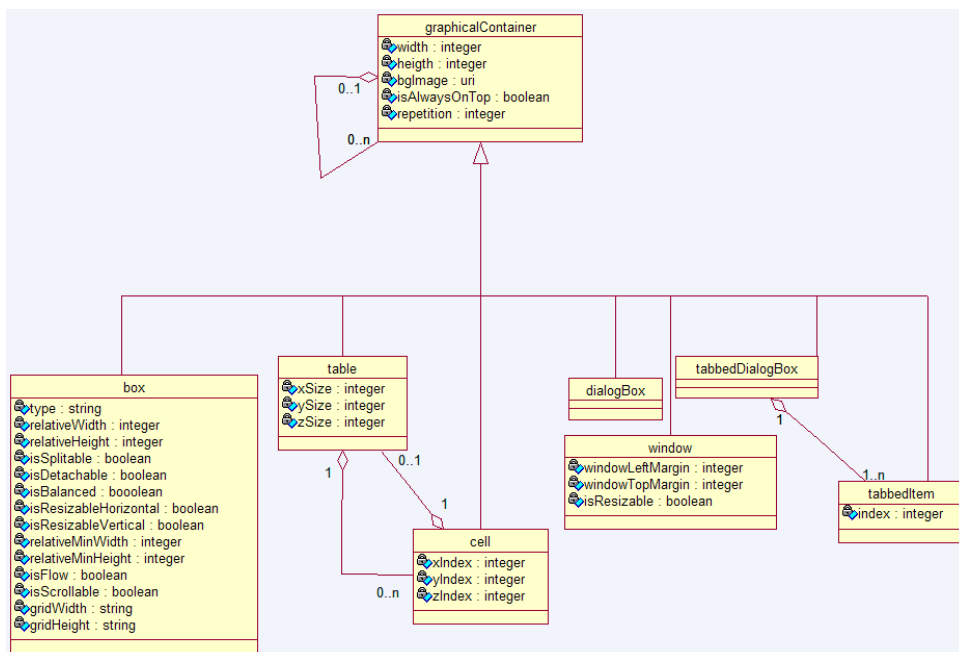
Figure 3-9 Graphical containers

Graphical containers are detailed in Fig. 3-9. They are classified in the following types:

- A *window* is a container that is found in nearly all 2D graphical toolkits. A window is equipped with native behavior such as close, tile, restore, minimize, maximize. A window may contain other graphical containers.

- A *tabbed* dialog box can be viewed as a set of windows stacked onto each other whose access in enabled by a set of tabs. Tabbed dialog boxes are composed of tabbed items.

110

- A *table* is composed of cells, a table may contain any other graphical container, including another table. A table is not considered, as in some languages (e.g., HTML 4.0), a layout mechanism.

- A *dialog box* is an independent box equipped by default of a confirmation or cancellation control.

- A *menu bar* is a container type that hosts menu and menu items.

- A *box* is the basic layout mechanism in our language. A box can only contain another box of a graphical individual container. We use a Tex-like [Mitt04] boxing system, allowing a very precise description of the layout while omitting any reference to an absolute coordinates positioning of elements. Boxes may be of several types (differentiated by their attribute *Type*): horizontal boxes, vertical boxes, horizontal grid and vertical grid. Fig. 3-10 shows an example of an embedding of boxes within a window. Window1 contains a vertical box (VBox1). This vertical box is further decomposed into a vertical box (VBox2) and a horizontal box (HBox1). The individual components are displayed accordingly to their mother box type.
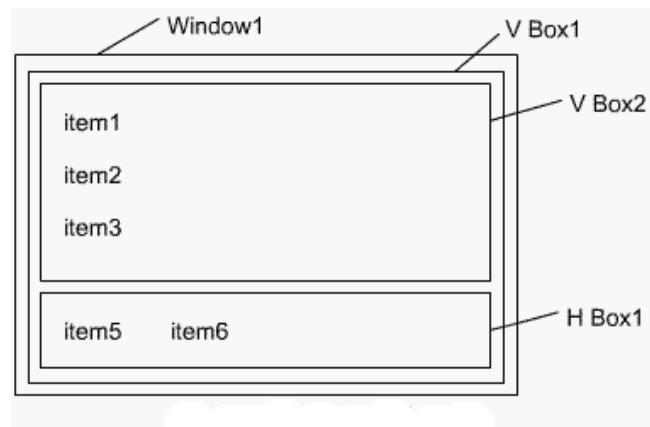
Figure 3-10 Illustration of the embedded box system to specify a 2D layout

Graphical Individual Components (GIC) are detailed in Fig. 3-11. *Text components* represent text-based components like a label, an input field, a password field, a multi-line input field, a complex textual output as a rtf file. A property (*isEditable*) allows differentiating text components subject to input or not. *Image components*

represent all images and can be divided into sub-zones. *Video components* enable specifying the insertion of a streaming of images into a UI. A whole range of CUI enables control, or choice: *button*, *toggle button*, c*heck boxes, radio buttons. Spin, combo box* and *tree* can be populated with an *item* CIO. A *menu* is populated with *menu items.* A *slider* may be associated with one or two *cursors.* Common composed components are also proposed: *drawing canvas*, *color picker, file picker, date picker, hour picker,* and *progression bar.*
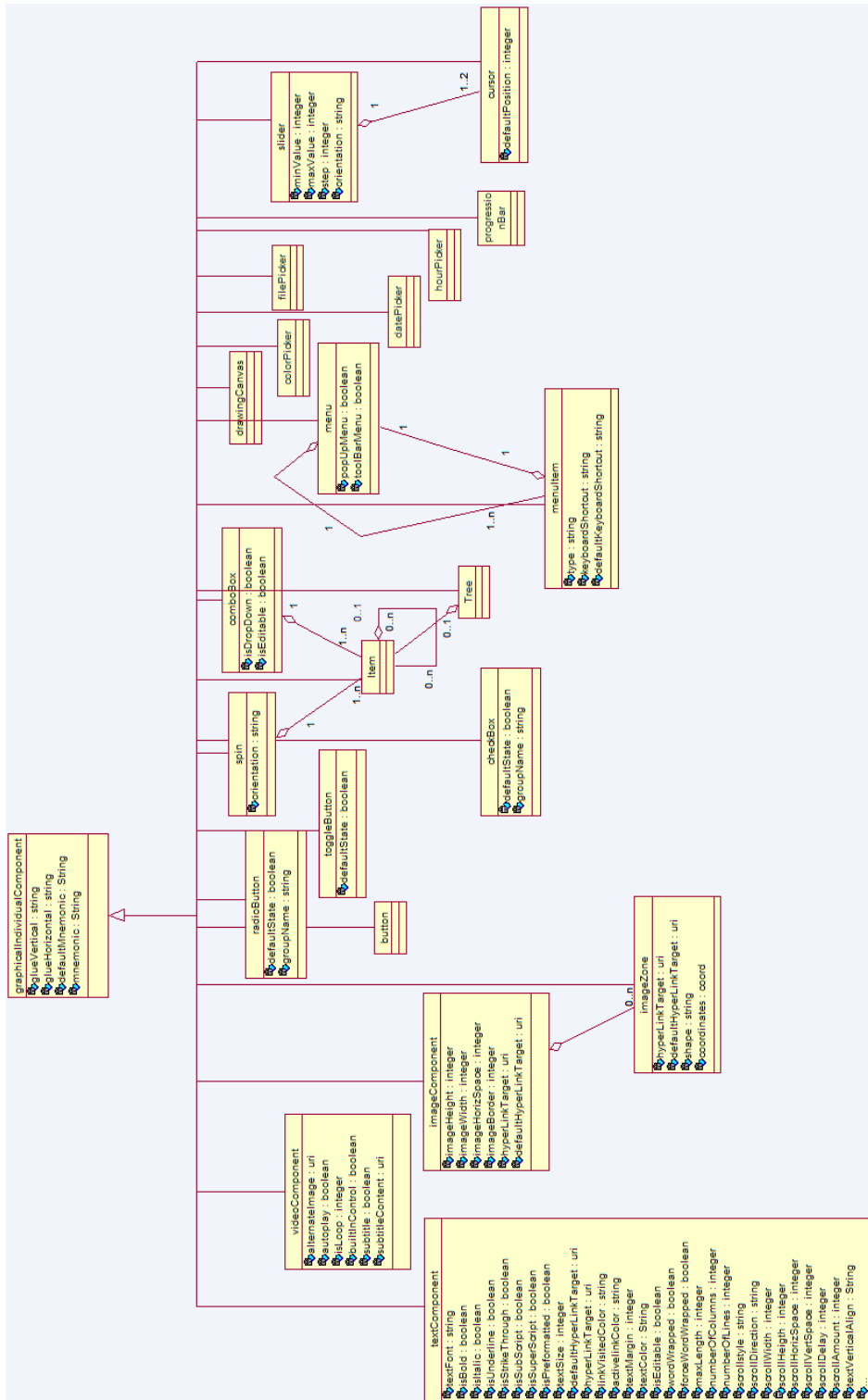
Figure 3-11 Graphical Individual Components Types

Auditory interaction objects are represented in Fig. 3-12. *Auditory Containers* represent a logical grouping of other auditory containers or auditory individual components. *Auditory individual components* are of two types: auditory output which may consist in music, voice or a simple "earcon" (i.e., an auditory icon) or auditory input which is a mere time slot allowing the user to provide an auditory input using her voice, or any other physical device able to produce sound.
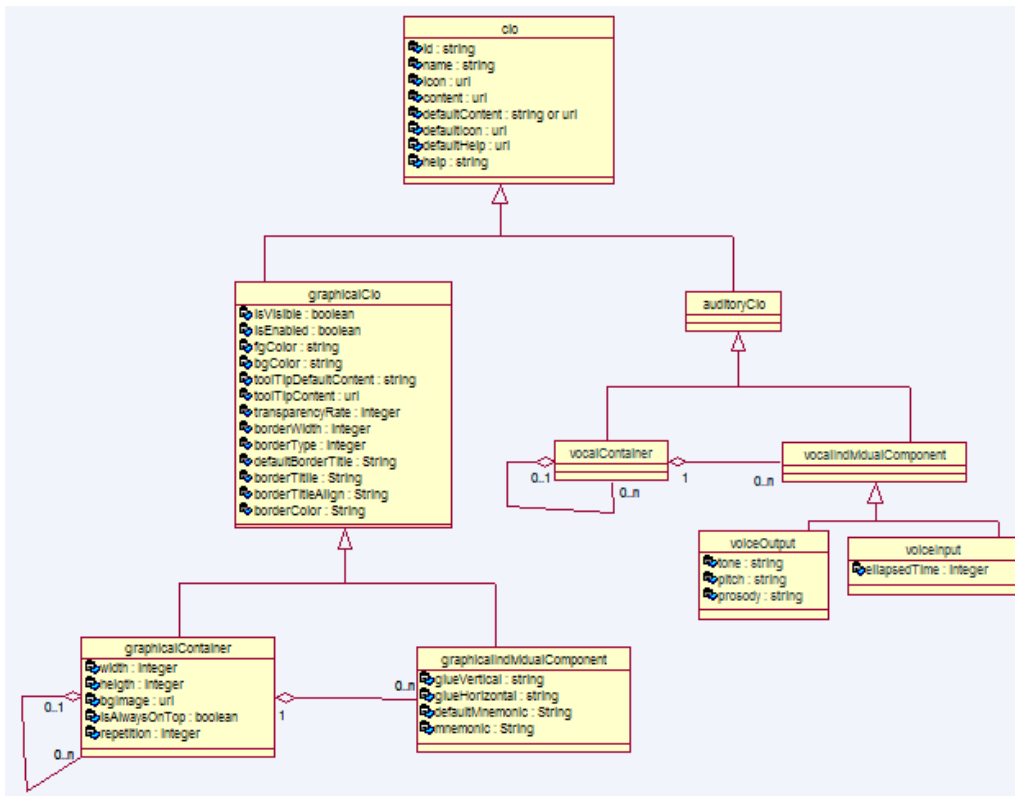
Figure 3-12 Graphical interaction objects and auditory interaction objects (detail from CUI)

CUI relationships are exposed in Fig. 3-13. Similarly to Concrete Interaction Objects they are divided into *auditory relationships* and *graphical relationships*. *Dialog control relationship* can be defined between both types of interaction objects.

Auditory relationships are of two types: *auditory transition* enables to specify a transition between two auditory containers. An attribute transition type determines the type of transition e.g., open, mute, reduce volume, or restore volume. A transition effect allows to specify an auditory effect to the transition

e.g., fade-out, fade-in. A relationship *auditoryAdjacency* indicates a time adjacency between two auditory components. A delay expressed in seconds indicates a time space between these components.

Graphical relationships are of four types. *Graphical transition* specifies navigation links between the different containers populating the UI. Transition types are : open, close, minimize, maximize, tile and restore. Some transition effects may be specified like "box-out", "box-in", "fade-out", "fade-in". The source of a graphical transition is generally a graphical individual component. An *alignment* may also be specified among any individual component belonging to the same window. *Adjacency* indicates that two components are topologically adjacent. A graphical emphasis indicates that two interaction objects must be differentiated using rendering artifacts (e.g., using two different colors).

Dialog control allows a specification of a flow of control between the concrete interaction objects. As so a dialog control may be specified independently of a task model. LOTOS (see Sec. 3.2.1) operators are used for this purpose. For instance a relationship CIC1.EnterCountry []> CIC2.EnterProvince, indicates that CIC2 cannot be initiated while CIC1 is not terminated and that CIC1 has provided a value for the data on which the two component synchronize with. An interpretation for each type of LOTOS operator may be provided with pre/post conditions, termination and initiation states. A Dialog control at the concrete level is differentiated from dialog control at the abstract level. While initiation and termination of objects cannot be fully specified at the abstract level (indeed, abstract objects cannot be mapped onto events), they may be at the concrete level. For instance, an event may be associated with the termination of a CIO e.g., a container terminates if such button is pressed.

Any CIO may be associated with any number of *behaviors* (see Fig. 3-14).
A *behavior* is the description of an event-response mechanism that results in a system state change. The specification of a behavior may be decomposed into three types of elements: an *event,* a *condition,* and an *action.*
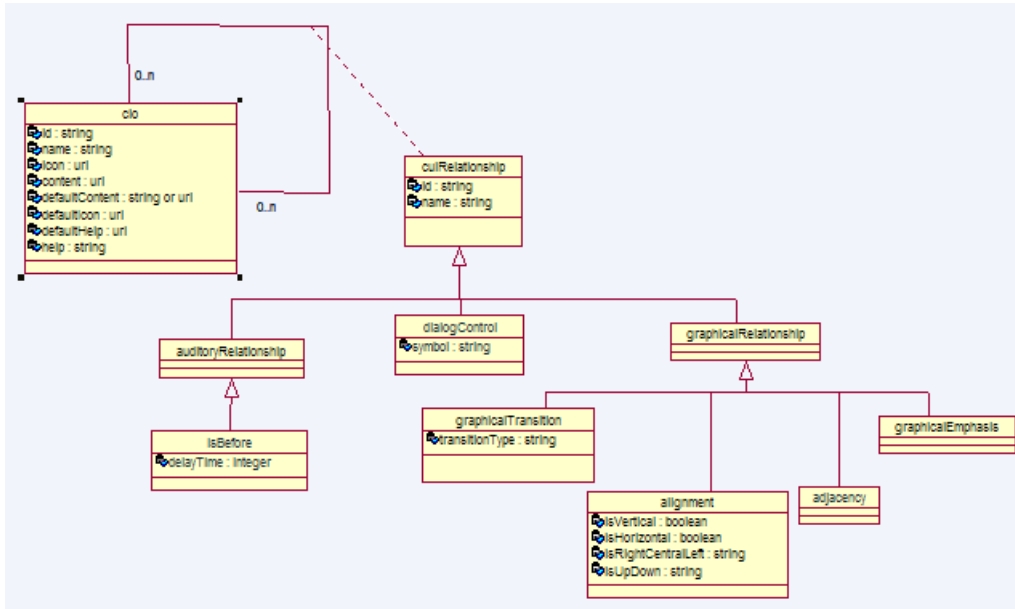
Figure 3-13 Relationships ate the concrete user interface level

An *event* is a description of a run-time occurrence that triggers an action. Some typical events are described in table 3-4. They consist of any system event (i.e., issued from a process belonging to the domain), user interface event (i.e., issued in the context of the user interface). For instance movePointer([X], [device]) refers to an event that consists in moving a pointer in the context of a CIO [X]. Events cannot make any reference to coordinates, as CUI does not. Like User Action Notation [Hart90], the concept of context of an object (identified by its id) is used to reference a display area where a particular object is rendered. Note that, the negative expression of an object context is also allowed. For instance, depress(NOT[X]), [device]), refers to a depress event (e.g., a mouse down) outside the context of [X]..[X] can also be unimportant in the realization of an event in such a case a value null is referenced. The [device] parameter makes reference to the device from which the event is generated. Each device or device part, is referenced in a device model (not in the scope of this dissertation) with a unique identifier.
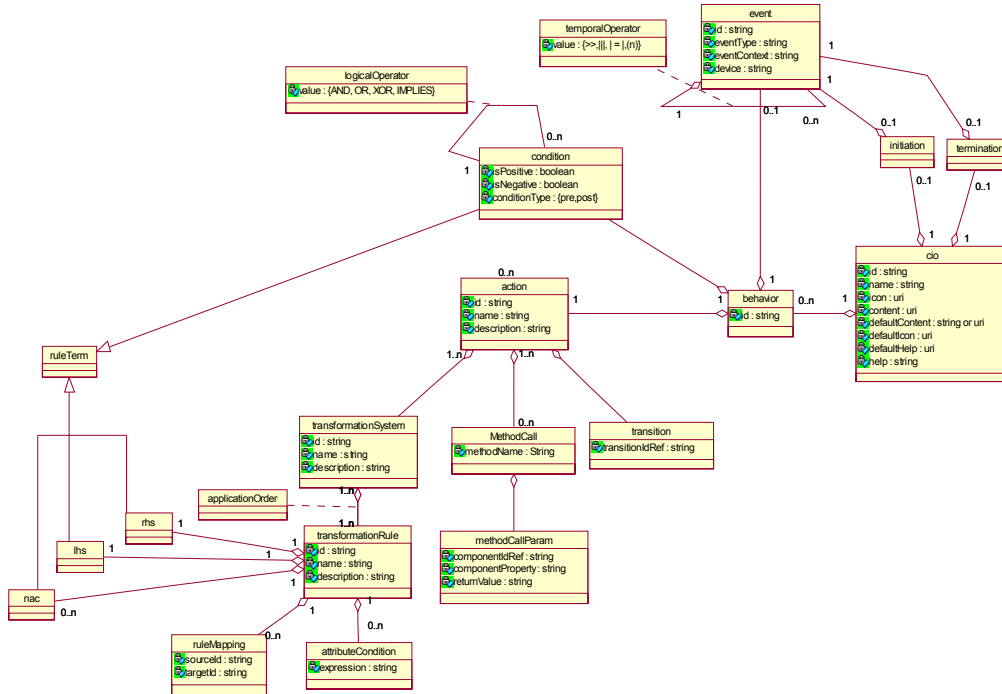
Figure 3-14 Behavioral specification at the concrete level

CIO	Events
System event	ellapsedTime(n), systemEvent(eventName)
Graphical User Interface Events	
All graphical CIOs	movePointer(X,device), pointerOver(X,device), moveOutPointer(X,device), click(X,device), doubleClick(X,device), depress(X,device), release (X,device), dragOver(X,Y,device), dragDrop(X, Y,device), hasFocus(X), lostFocus(X).
graphicalContainer	resize(xFactor,yFactor)
textComponent	Change
Slider	move(cursor,x)
Spin	spinUp, spinDown
Auditory User Interface	depress(tone)

Table 3-4 List of typical events

Events can be composed into more complex event expressions using a subset of the LOTOS operators introduced earlier (Sec. 3.2.1). "|||" indicates a concurrence of events (to be interpreted as a disjunction). ">>" indicates a strict sequence of events. "|=|" indicates an order independent sequence of events. "(n)" indicates a finite iteration of events where *n* is an integer indicating the

117

iteration factor. For instance, click(Button1, Mouse1LeftBut) |=| depress(null, KeyBrd_Z) is an event that is an order independent composition of a mouse click on a button and a keyboard depress.

A *condition* is the expression of a state that has to hold true before (pre-condition) or after (post-condition) an action is performed. A condition may be positive or negative. We express condition as patterns (i.e., a partial description of a state) on the user interface specification itself. Conditions may be composed using traditional logical operator. "AND" indicates a conjunction of conditions. "OR" indicates a disjunction of conditions. "XOR" indicates an exclusive disjunction of conditions. "IMPLIES" indicates an implication between two conditions.

An *action* is a process that results in a state change in the system. An action can be of three types: a *method call*, a *transformation system*, or a *transition*.

A *method call* is a call to a method that is external to the UI. If a domain model exists, all method calls must reference a method belonging to this model. A method call is normally specified with the name of the method (under the form Class.methodName), but other referencing techniques are not forbidden. The method call parameters can be specified by making a reference to the value of a property of an object belonging to the CUI.

A *transformation system* is the expression of any property change at the UI level. We use a mechanism to specify property changes on the UI. This mechanism is similar to the one that will be introduced in Chapter 4. To avoid too much forward reference, it can be said that a transformation system can be explained as follows: when a pattern is found in CUI specification, changes should occur on the elements matching the pattern. A transformation system might be, for instance, "when a green button is found in the specification, change the color property of this button to red" or "For all text components belonging to the main window, increase their font by a factor of 2".

A *transition*, also called *navigation*, is a description of a change in the container's visibility property of a user interface system. A transition has a source (a navigation individual component) and a target (generally a container). Depending on the type of modality, transitions may be of different types (see above in this Section).

3.2.5 Context Model

A context model (Fig. 3.15) is a model describing the three aspects of a context of use in which an end user is carrying out an interactive task with a specific computing platform in a given surrounding environment [Thev01]. Consequently, a context is hereby defined as a triple of the form $<e, p, u>$ where e is an element of the environments set considered for the interactive system, p is an element of the platforms set considered for the interactive system and u is an element of the users set for the interactive system.

A User model consists of a user stereotypes. A user stereotype is any set of users sharing similar characteristics. Stereotypes can be arranged in hierarchy. As so, a stereotype can be decomposed into sub-stereotypes.

A Platform model captures relevant attributes for each couple software-hardware platform and attached devices that may significantly influence the context of use in which the user is carrying the interactive task. Our context model has been developed in [Flor04]. A platform specification can consist of a series of physical hardware devices (hardware platform components), a series of software components (software platform), the characteristics of the network to which the platform is connected, the capability to support wireless (WapCharacteristics), and the capability of browsing web pages (BrowserUA).

An Environment model describes any property of interest of the physical environment where the user is using the UI on the computing platform to accomplish her interactive tasks. Such attributes may be physical (e.g., lighting conditions), psychological (e.g., level of stress), and organizational (e.g., location and role definition in the organization chart).

The context model will not be used in the transformational process described in this dissertation. Although any UI specification model may be attached to any number of context specification thanks to the "hasContext" relationship described in next section.
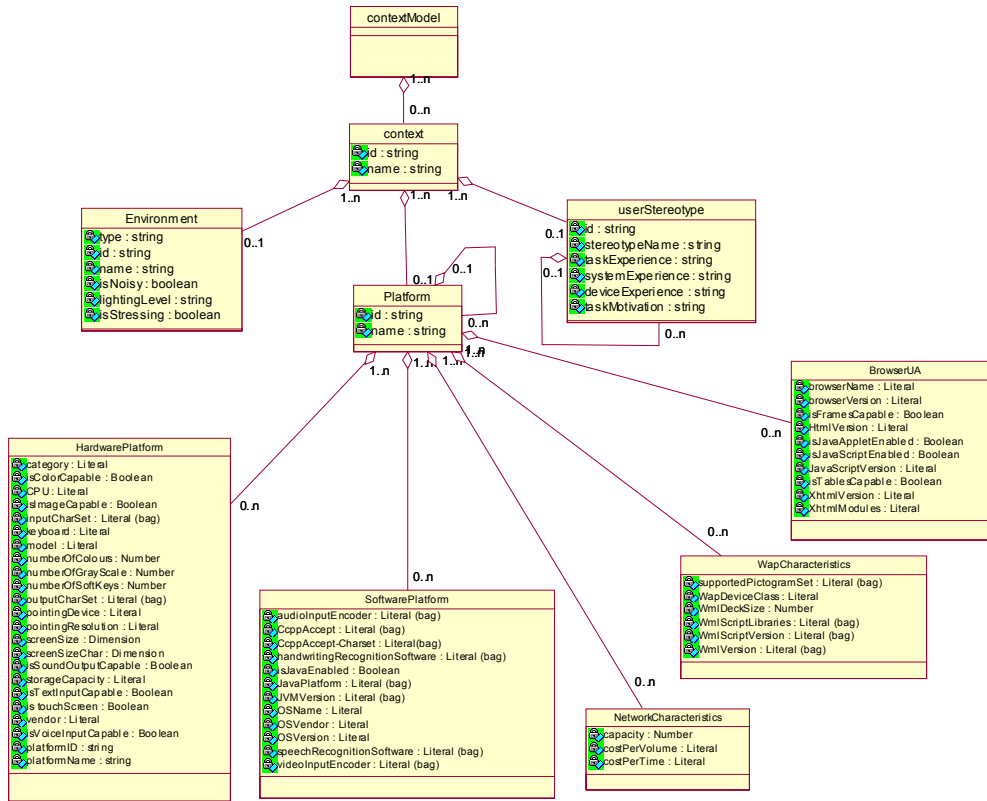
Figure 3-15 Context model

3.2.6 Inter-Model Relationships

Model integration is a well-known issue in transformation driven development of UI [Puer99]. Rather than proposing a collection of unrelated models and model elements, our proposal provides a designer with a set of pre-defined relationships allowing a mapping of elements from heterogeneous models and viewpoints (Fig. 3-16). This can be useful, for instance, for enabling the derivation of the system architecture (mappings between domain and CUI/AUI models), for traceability in the development cycle (reification, abstraction and translation), for addressing context sensitive issues (has context), for dialog control issues, for improving the preciseness of model derivation heuristics.
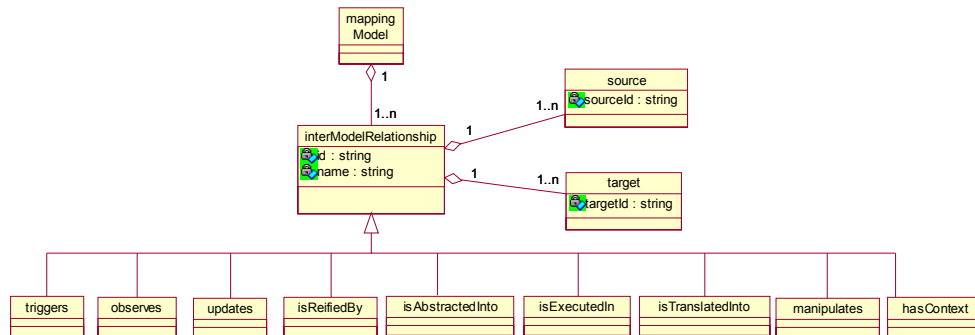
Figure 3-16 Inter-Model Mappings

3.2.6.a Mappings between the domain models and the UI models

Several relationships can be defined to explicit the relationships between the domain model and the UI models (both abstract and concrete):

- *Observes* is a mapping defined between an interaction object and a domain model concept (either an attribute, or an output parameter of a method). This mapping may be interpreted as follows: the content of a UI object must be synchronized when
    - A mapped attribute is modified. The new state resulting from this modification should be presented on the UI (the notion of view could be of interest here).
    - A mapped method is executed. Its output parameters are displayed on the UI.
- *Updates* is a mapping defined between an interaction object and a domain model concept (specifically, an attribute). "Updates" describes the situation where the attribute of an object in the domain model must be synchronized with the content of a UI object.
- *Triggers* is a mapping defined between an interaction object and a domain model concept (specifically, an operation). This mapping describes that a UI object is able to trigger a method from the domain model.

3.2.6.b Mappings to ensure the traceability of the development cycle

Our ontology is designed to be integrated in a framework where models are transformed into other models (see Chapter 4). This framework defines several

types of transformations in order to achieve multi-path development of user interfaces. *Traceability mappings* are helpful for keeping a trace of the execution of the transformations. For instance it may be interesting to know which concrete object reifies which abstract object, or vice versa, which abstract object is an abstraction of which concrete object.

- *Is Executed In* maps a task to an interaction object (a container or an individual component) allowing its execution. This relationship is notably useful for deriving a dialog control component, for ensuring that all tasks are supported appropriately by the system.
- *Is Reified By* indicates that a concrete object is the reification of an abstract one through a reification transformation.
- *Is Abstracted Into* indicates that an abstract object is the reification of a concrete one through an abstraction transformation.
- *Is Adapted Into* indicates that an interaction object (abstract or concrete) is adapted into another one as a result of an adaptation transformation.

3.2.6.c   Other mappings

Other useful mappings are:

- *Manipulates* maps a task to a domain concept. It may be an attribute, a set of attributes, a class (or an object), or a set of classes (or a set of objects). This relationship is useful when it comes to find the most appropriate interaction object to support a specific task.
- *Has Context* maps any model element to one or several contexts of use.

3.3 Abstract Syntax: graphs as underlying formalism

The *abstract syntax* is defined as the hidden structure of a language, its mathematical background [Meyer90]. Our abstract syntax takes the form of a, so called, *"enriched" directed graph*. That is to say an identified, labeled, typed, constrained graph. A graph structure naturally describes a set of concepts and their relationships, it is strongly correlated to the concept of ontology [John01]. Graph structures are appropriate when the number of relationships among the concepts of an ontology become too large to represent them with another mathematical structure (e.g., lists, trees, sets). As argued by [Sowa92] graphs are logically precise (Req. 4: *Formality*), humanly readable (Req. 3), and computationally tractable (Req. 5: *machine readable*). They have been used, for instance, to represent artifacts like code structures, system requirements, expert knowledge, causal systems, probabilistic systems, social structures. Additionally, extensive collections of algorithms for their manipulation are provided in the scientific literature.

In this section, we use the method introduced in [Mens99] to progressively consolidate our enriched graph structures that is the foundation of our ontology.

3.3.1 General Definitions

Definition 1. A **graph** g is defined by a quadruple $(V_g, E_g, source_g, target_g)$ such that:

1. V_g is a finite set of vertices (or nodes);
2. E_g is a finite set of edges (or arcs);
3. $source_g : E \rightarrow V$, is an injective function that assigns a source to each edge of E;
4. $target_g : E \rightarrow V$, is an injective function that assigns a target to each edge of E.

Definition 2. g is said to **directed** iff
$$\forall e \in E_g, \exists! v_i, v_j \in V_g \mid source(e) = v_i \wedge target(e) = v_j$$

Notation 1. Implicit graph reference. The notation $[SetName]_g$ (e.g. V_g) or $[FunctionName]_g$ (e.g. $source_g$) will be replaced by $[SetName]$ (e.g. V) or $[FunctionName]$ (e.g. $source$) if no confusion is possible (i.e. only one graph is concerned).

Notation 2. Graph component or element. The expression graph component or element refers undiscernibly to vertex or edges.

Notation 3. Implicit function reference. Let x stand for a graph component, $[FunctionName](x)$ denotes a function applied to x. In case of ambiguity : $[FunctionName]_v(x) = \sigma_{v \in V}[FunctionName](x)$ and $[FunctionName]_e(x) = \sigma_{e \in E}[FunctionName](x)$

3.3.2   Category Theory and Graphs Morphisms

Category theory is a generalized mathematical theory of structures. One of its goals is to reveal the universal properties of structures of a given kind via their relationships with one another [Marq97].

A category describes a set of objects that have an identical mathematical structure, and for which there exists morphisms between those objects and preserving this structure [Fokk92]. The major benefit of working with categories is that any property established for a category can established for any structure of this category.

Graphs are objects of a category of graphs with morphisms as structure preserving mappings between them.

Definition 3. Let $g = (V_g, E_g, target_g, source_g)$ and $h = (V_h, E_h, target_h, source_h)$ be two Graphs; a **graph morphism** from g to h is a pair $m = (m_v, m_e)$ of mappings $m_v : V_g \rightarrow V_h, m_e : E_g \rightarrow E_h$, such that:

1. $\forall e \in E_g, source_h(m_e) = m_v(source_g(e))$ (source nodes are preserved);
2. $\forall e \in E_g, target_h(m_e) = m_v(target_h(e))$ (target nodes are preserved).

Other properties of interest of graphs morphisms are :

Definition 4. Interesting **graphs morphisms properties:**
1. If m_v and m_e are injective (resp. surjective) m is injective (resp. surjective).
2. If m is injective and surjective (i.e. bijective), m is said to be isomorphic (written $m : G \cong H$ or simply $G \cong H$).
3. If m_v, m_e are total functions, m is said to be a total graph morphism. Otherwise m is said to be a partial graph morphism.

Thanks to morphisms, our initial graph definition (definition 1) will be extended with several features (i.e., identifies, label, type, constraints) while being sure to benefit of all theoretical results provided for the graph category. All features are then consolidated into a single graph definition to form the mathematical basis of our language. Such a way to proceed is found in [Mens99]

3.3.3 Identified Graphs

An identification function is introduced in order to univocally identify each node or edge of a graph. This function is useful as it allows differentiating instances of a same node that would be considered identical without this identifier.

Definition 5. Let $L = $ *(NodeId, EdgeId)* be a pair of disjoint and finite sets of predefined labels. g is said to be a **(I)-graph** iff g is a tuple (*g, Id*) such that:
1. g is a graph (see definition1);
2. *Id* is a pair of bijective functions, $Id = (Id_v, Id_e)$ where $Id_v : V \rightarrow NodeId$ and $Id_e : E \rightarrow EdgeId$.

Definition 6. Let g and h be two (I)-Graphs; Let m be a pair $m = (m_v, m_e)$ of mappings $m_v : V_g \rightarrow V_h, m_e : E_g \rightarrow E_h$; m is an **identifier preserving (I)-Graph morphism** if:

1. $\forall e \in E_g, source_g(m_e) = m_v(source_g(e))$ (source nodes are preserved);
2. $\forall e \in E_g, target_h(m_e) = m_v(target_g(e))$ (target nodes are preserved);
3. $Id_v(g) = Id_v(g) \circ m_g$ (nodes Id are preserved);
4. $Id_e(g) = Id_e(g) \circ m_g$ (edges Id are preserved).

From definition 5 and 6, it can be said that (I)-Graph is a category with (I)-Graphs as objects and identifier preserving morphisms as morphisms.

Note that Id_v and Id_e are bijective functions. Two nodes or edges cannot share the same identifier and for each identifier is univocally mapped onto an identifier. In mathematical term this can be expressed as follows:

$$\forall x, y \in (V \cup E), Id(x) = Id(y) \Rightarrow x = y \quad (Id \text{ is injective}).$$
$$\forall y \in NodeId \cup EdgeId, \exists x \in (V \cup E) \,|\, Id(x) = y \quad (Id \text{ is a surjection}).$$

3.3.4   Labeled Graphs

A label attached to each node and edge is introduced in order to label graph components with a name.

Definition 7. Let $L =$ *(NodeLabel, EdgeLabel)* be a pair of disjoint and finite sets of predefined labels. g is said to be a **(L)-graph** iff g is a tuple (*g, Label*) such that:

1. *g* is a graph (see definition 1) ;
2. *Label* is a pair of functions, *Label* = ($Label_v, Label_e$) where $Label_v : V \rightarrow NodeLabel$ and $Label_e : E \rightarrow EdgeLabel$.

Definition 8. Let g and h be two (L)-Graphs; Let m be a pair $m = (m_v, m_e)$ of mappings $m_v : V_g \to V_h, m_e : E_g \to E_h$; m is an **label preserving (L)-Graph morphism** if:

1. $\forall e \in E_g, source_g(m_e) = m_v(source_g(e))$ (source nodes are preserved);
2. $\forall e \in E_g, target_h(m_e) = m_v(target_g(e))$ (target nodes are preserved);
3. $Label_v(g) = Label_v(g) \circ m_g$ (node labels are preserved);
4. $Label_e(g) = Label_e(g) \circ m_g$ (edge labels are preserved).

From definition 7 and 8, it can be deduced that (L)-Graph is a category with (L)-Graphs as objects and label preserving morphisms as morphisms.

An important discussion on the nature of labeling functions is to be made. Indeed, the property of this function varies following the level of abstraction on which it is defined.

When our graph structure is exploited to describe a meta-model, a labeling functions $Label_v$ and $Label_e$ is totally bijective. This property can be mathematically expressed as follows:

$\forall x, y \in (V \cup E), Label(x) = Label(y) \Rightarrow x = y$ ($Label$ is injective).
$\forall y \in NodeLabel \cup EdgeLabel, \exists x \in (V \cup E) \mid Label(x) = y$ ($Label$ is a surjection).

This means that each graph component is univocally associated with a label and that each label is associated with a graph component. At this level identification and labeling functions are partly redundant.

But our graph language is supposed to describe meta-types as well as their instances (these instances being UI models). In this case the labeling functions $Label_v$ and $Label_e$ are only partial functions. This means that two UI model elements may share a same label.

Another important remark to be made is that the label is not used to specify a graph component type. An additional typing mechanism is introduced for this purpose.

3.3.5 Constrained Graphs

Constraining functions that operate on nodes or edges allow us to attach to any node or edge an arbitrary number of constraints. Constraints can consist in the expression of cardinality constraints, restrictions on the domain or the co-domain of certain functions, etc. It is proposed to express these constraints with first order logic expressions.

Definition 9. Let $C = $ *(NodeConstraint, EdgeConstraint)* be a pair of disjoint and finite sets of node constraints and edge constraints. g is said to be a **(C)-graph** iif g is a tuple (g, Co) such that:

1. g is a graph (see definition 1);
2. Co is a pair of surjective functions, $Co = (Co_v, Co_e)$ where $Co_v : V \rightarrow NodeConstraint$ and $Co_e : E \rightarrow EdgeConstraint$.

Definition 10. Let g and h be two (C)-Graphs; Let m be a pair $m = (m_v, m_e)$ of mappings $m_v : V_g \rightarrow V_h, m_e : E_g \rightarrow E_h$; m is an **constraint preserving (C)-Graph morphism** if:

1. $\forall e \in E_g, source_h(m_e) = m_v(source_g(e))$ (source nodes are preserved);
2. $\forall e \in E_g, target_h(m_e) = m_v(target_g(e))$ (target nodes are preserved);
3. $Co_v(g) = Co_v(h) \circ m_g$ (nodes constraints are preserved);
4. $Co_e(g) = Co_e(g) \circ m_g$ (edges constraints are preserved).

From definition 9 and 10, it can be deduced that a (C)-Graph is a category with (C)-graphs as objects and constraint preserving morphisms as morphisms.

3.3.6 Typed Graphs

Typing allows classifying nodes and edges by attaching types to them. Attaching several nodes (or edges) to the same types indicates a commonality in terms of properties between these nodes (or edges).

Definition 11. Let $TY = (NodeType, EdgeType)$ be a pair of disjoint and finite sets of predefined types. g is said to be a **(TY)-graph** iff g is a pair (g ,Ty) such that :

1. g = is a graph (see definition 1);
2. Ty is a pair of total functions attaching a type to each node and edge of the graph. $Type = (Ty_v, Ty_e)$ where $Ty_v : V \rightarrow NodeType$ and $Ty_e : E \rightarrow EdgeType$.

Definition 12. Let g and h be two (TY)-Graphs; Let m be a pair $m = (m_v, m_e)$ of mappings $m_v : V_g \rightarrow V_h, m_e : E_g \rightarrow E_h$; m is an **type preserving (TY)-Graph morphism** if:

1. $\forall e \in E_g, source_h(m_e) = m_v(source_g(e))$ (source nodes are preserved);
2. $\forall e \in E_g, target_h(m_e) = m_v(target_g(e))$ (target nodes are preserved);
3. $Ty_v(g) = Ty_v(g) \circ m_g$ (nodes types are preserved);
4. $Ty_e(g) = Ty_e(g) \circ m_g$ (edges types are preserved).

From definition 11 and 12, it can be deduced that a (L)-Graph is a category with (L)-graphs as objects and type preserving morphisms as morphisms.

The typing functions introduced here are total. This means that for all graph component there is a corresponding type. A same type may be assigned to several elements. A type may have no graph component of its type. This is mathematically expressed as follows:

$$\forall x \in (V \cup E), \exists y \in NodeType \cup EdgeType \mid Type(x) = y$$

3.3.7 Identified, Labeled, Constrained and Typed graph

All features defined above can be consolidated in a single graph category called (Identified, Labeled, Constrained, Typed)-Graphs (in short: (I,L,C,TY)-Graphs). Note that this consolidation could be modularized that is to say that features presented above can be consolidated "a la carte".

Definition 13. g is an **(Identified,Labelled,Constrained,Typed)-graph** iff:
1. g is a graph (see definition 1)
2. g is an identified graph (see definition 6)
3. g is a labeled graph (see definition 8)
4. g is a constrained graph (see definition 10)
5. g is a typed graph (see definition 12).

Definition 14. Let g and h be two (I,L,C,TY)-Graphs; Let m be a pair $m = (m_v, m_e)$ of mappings $m_v : V_g \rightarrow V_h, m_e : E_g \rightarrow E_h$; m is an **identifier, label, constraint, and type preserving (I,L,C,TY)-Graph morphism** iff:
1. m is a graph morphism (definition 5)
2. m is an identifier preserving morphism (definition 7)
3. m is a label preserving morphism (definition 9)
4. m is a constraint preserving morphism (definition 11)
5. m is a type preserving morphism (definition 13).

From definition 13 and 14, it can be deduced that (I,L,C,TY)-Graph is a category with (I,L,C,TY)-graph as objects and (I,L,C,TY)-Graph morphism as morphisms.

This consolidation has the advantage of being modular. This means that features presented above can be consolidated in an "a la carte" way to form other categories.

3.3.8 An Improved Typing Function

We want to have a better control on the typing mechanism. Graph types are introduced for this purpose ([Monta96,Corra96,Heck96]). Graph types contain all "type information" that is used to type the model level.

Types that are returned by the functions Ty_v and Ty_e (see definition 11) belong to two type sets ($NodeType$, $EdgeType$). These sets contain possible types.
The main idea with graph types is to replace type sets by graphs. In order to support this, the typing mechanism of definition 11 has to be slightly adapted.

> **Definition 15.** Let $Type = (NodeType, EdgeType)$ be a pair of disjoint and finite sets of types. Let TG be a fixed (L,C)-graph (TG is called a *type graph*). g is said to be a **(I,L,C,TY) TG-Typed graph** iff g is a pair (g, Ty^{TG}) where:
> 1. g is a (I,L,C,TY,N)-graph (see definition 16).
> 2. $Ty^{TG} : g \rightarrow TG$ such that *type* is a total (L,C)-graph morphism.

The above definition asserts that there must be a correspondence between, on the one hand, node and edge type at the model level and, on the second hand, node and edge labels at the meta-level. Furthermore, constraints defined on labels in TG are applicable to types in g. This situation is expressed in Fig. 3-17.
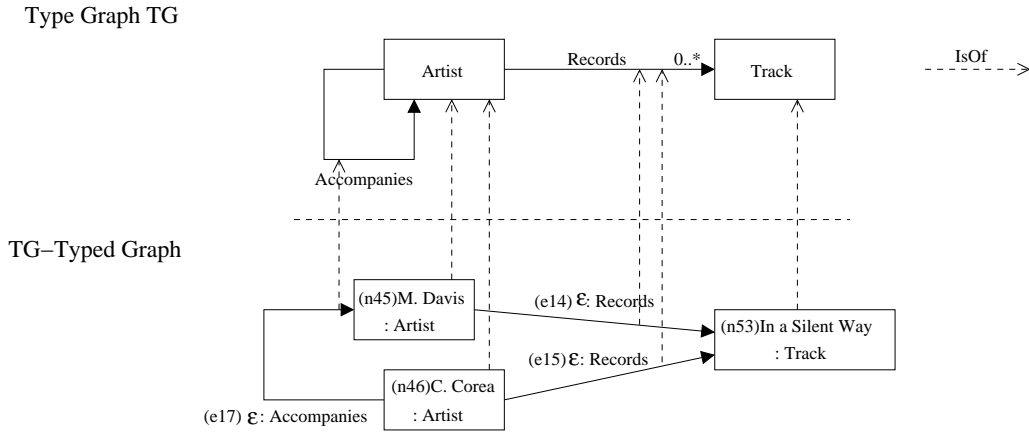
Figure 3-17 Typed Graph and its Graph Type

In addition the following graph morphisms can be defined:

> **Definition 16.** Let g and h be two (I,L,C,TY) TG-Typed Graphs; $m : g \rightarrow h$ is a (I,L,C,TY) **TG-Type preserving graph morphism** iff:
> 1. f is a (I,L,C,TY) graph morphism (see definition 17)
> 2. $\forall x \in dom(m) : type(h) \circ f = type(g)$.

From definition 15 and 16 it can be said that (L,C)-Graph is a category with (L,C)-graphs as objects and nesting preserving morphisms as morphisms.

From definition 15 and 16, it can be asserted that constraints defined in a type graph TG can also constrain the corresponding TG-Typed graph. For instance, a cardinality constraint on an edge between two types in a TG graph is effective on the TG-Typed graph. This could be expressed mathematically as follows:

$$\forall v \in V, ifTy(v) = "tutorial" \Rightarrow \exists E' \subseteq E \wedge$$
$$E' = \{e \in E \mid Label(e) = "isGivenBy" \wedge source(e) = v\} \wedge (1 \leq |E'| \leq 3)$$

In the above expression we define a cardinality constraint between a node representing an entity labeled "tutorial" and another entity labeled "speaker". The expressed constraint says that a tutorial cannot be given by more than three speakers.

Other constraints can limit the domain or the co-domain of source and target functions in order to avoid or force the use of certain type of edges with certain type of nodes. For instance an edge with the label "Is Husband Of" can only occur between two nodes with label "man" and "woman" (not the case anymore in Belgium). This example is mathematically expressed as follows:

$$\forall e \in E, label(e) = "isHusbandOf", \exists v_1, v_2 \in V \mid source(e) = v_1 \wedge$$
$$target(e) = v_2 \Rightarrow Label(v_1) = "man" \wedge Label(v_e) = "woman"$$

The reader may have noticed that examples of constraints have been defined on labels and not on types. Indeed, these examples are expressed at the concept level. They will be enforced at the model level. As labels at the concept level are types at the model level it is normal to express constraints on labels at the meta-level. In the second example, the "translation" of the constraint at the model level gives:

$$\forall e \in E, Type(e) = "isHusbandOf", \exists v_1, v_2 \in V \mid source(e) \wedge$$
$$target(e) = v_2 \Rightarrow Type(v_1) = "man" \wedge Type(v_2) = "woman"$$

In order, to simplify the expression of type graphs, types can be structured into partial orders. Organizing nodes and edges of the type graph into a partial order (see definition 18) presents the advantage of propagating constraints i.e., constraints applicable to one type can be directly inherited by all subtypes of this type.

Definition 17. A (L,C)-type graph TG is said to be $(\leq_v, \leq_e)$**-ordered graph** if $(\text{NodeLabel}, \leq_v)$ and $(\text{EdgeLabel}, \leq_e)$ are partial order.

Definition 18. The set of *NodeLabel* (see definition 8) is a **partial order** if $\exists \leq_v \in E$, such that:

1. Reflexivity: $\forall nodelabel \in NodeLabel \Rightarrow nodelabel \leq_v nodelabel$
2. Antisymmetry:
   $\forall nodelabel_i, nodelabel_j \in NodeLabel, if nodelabel_i \leq_v nodelabel_j$
   $\wedge nodelabel_j \leq_v nodelabel_i \Rightarrow nodelabel_i = nodelabel_j$
3. Transitivity:
   $\forall nodelabel_i, nodelabel_j, nodelabel_k \in NodeLabel, if\ nodelabel_i \leq_v$
   $nodelabel_j \wedge nodelabel_j \leq_v nodelabel_k \Rightarrow nodelabel_i \leq_v nodelabel_k$

Definition 19. The set of *EdgeLabel* (see definition 8) is a **partial order** if $\exists \leq_e \in E$, such that:

1. Reflexivity: $\forall edgelabel \in EdgeLabel \Rightarrow edgelabel \leq_v n$
2. Antisymmetry:
   $\forall edgelabel_i, edgelabel_j \in EdgeLabel, if edgelabel_i \leq_v edgelabel_j \wedge$
   $edgelabel_j \leq_v edgelabel_i \Rightarrow edgelabel_i = edgelabel_j$
3. Transitivity:
   $\forall edgelabel_i, edgelabel_j, edgelabel_k \in EdgeLabel, if edgelabel_i \leq_v$
   $edgelabel_j \wedge edgelabel_j \leq_v edgelabel_k \Rightarrow edgelabel_i \leq_v edgelabel_k$

As said above the definition of type graphs can be exploited to propagate constraints among types. Such a propagation mechanism is expressed in definition 20.

Definition 20. If T is partial-ordered type graph then the following **inheritance mechanisms** must be defined:

1. $\forall v, w \in V_t : \textit{if vlabel}(v) \leq_v \textit{vlabel}(w)$ then

   $\textit{vconstraints}(w) \subseteq \textit{vconstraints}(v)$ (constraints of supertypes are inherited from subtypes)

2. $\forall e \in E_t : \forall s_1, s_2, t_1, t_2 \in V_t : \textit{if source}(e) = s_1 \wedge \textit{target}(e) = t_1 \wedge$

   $\textit{vlabel}(s_2) \leq \textit{vlabel}(s_1) \wedge \textit{vlabel}(t_2) \leq_v \textit{vlabel}(t_1) \Rightarrow$

   $\exists f \in E_t \mid \textit{source}(f) = s_2 \wedge \textit{target}(f) = t_2 \wedge$

   $\textit{elabel}(f) = \textit{elabel}(e) \wedge \textit{econstraints}(f) = \textit{econstraints}(e)$

   (edge constraints between supertypes are inherited by edges among subtypes).

3.4 Concrete Syntax: a visual and textual syntax

A concrete syntax is an external appearance of a language. Describing a concrete syntax of a language consists of describing the allowed sentences of this language.

Historically, concrete syntaxes for formal languages have been expressed textually. Extended Backus-Naur Form grammars (EBNF), for instance, have been extensively used for this purpose [ISO96]. Fig. 3-18 shows a possible grammar for our language.

```
<uiSpecification>::={<uiModel>}
 <uiModel>::={<node>}| {<edge>}
<node>::= <nodeId> | <name> | <type> | {<attribute>} |{<constraint>}
<edge>::= <edgeId> | <label> | <type> | {<constraint>} | sourceNode | targetNode
<nodeId>::=<id>
<edgeId>::=<id>
<id>::= <char> | {<char>}
<name>::= <char> | {<char>}
<char> ::= a|b|c|d|e|f|g|h|i|k|…|8|9|0|…
<attribute> ::= <name> | value
<value>::=  <char> | {<char>}
<type>::= [listOfTypes]
<edge>::= <type> | <attribute> |  <sourceNodeId> | <targetNodeId>
<sourceNodeId>::=<id>
<targetNodeId>::=<id>
<constraint>::=  …
```

Figure 3-18 An excerpt of a BNF grammar of our language

We propose two types of syntax for our language: a visual one and a textual one. The visual syntax consists of boxes and arrows, a somewhat classic representation for a graphical structure. This visual syntax is mainly used in this work as an expression means for the transformation rules that are going to be developed in Chap. 4. The textual syntax is an XML based language, called UsiXML, its main use is to serve as exchange format between various applications exploiting our UI specification models at all development stages.

3.4.1   Visual syntax

A simple visual syntax is proposed based on the conceptual schemas described above. This syntax is taken from Attributed Graph Grammar tool (AGG) [Ehri99], a generic tool for specifying and executing graph transformations. This visual notation is illustrated in Fig. 3-19. A node is represented as a box. Node types are indicated on the upper part of this box, attribute-value couples are listed in the upper part of the box. Edges are represented as directional arrows. Their type appears as their label.
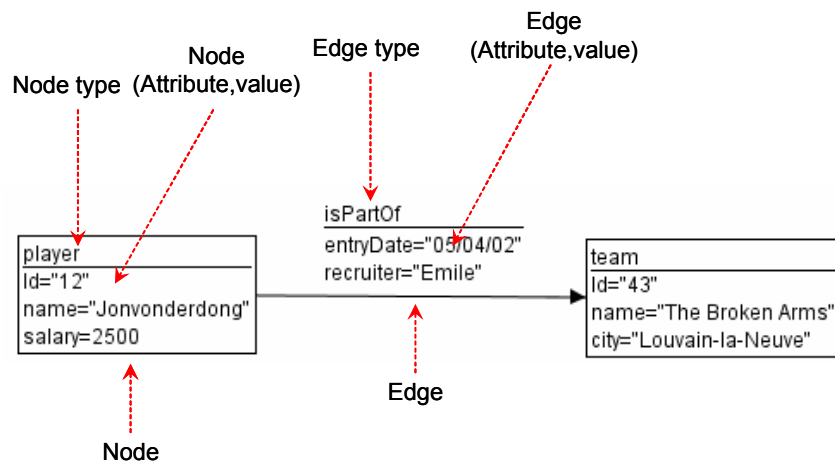
Figure 3-19 Visual syntax for expressing a host graph

3.4.2   UsiXML: textual syntax

XML stands for eXtensible Markup Language. XML is a subset of SGML, like HTML. An XML document is a textual document describing a set of data (not information!) with a tree-like structure. The aim of this format is to define a standard for exchanging data between heterogeneous applications (Req.22: *Support for tool interoperability*).

Unlike HTML, XML allows its users to define customized language elements. That's why XML is said extensible (Req. 10*: Ontological extendibility*). Even if XML, and markup languages in the large, was designed to be read by machines (Req. 5*: Machine-readable*) the fact that the data is conveyed with its underlying structure facilitates its legibility by a human agent (Req. 3*: Human readable*).

A definition of elements valid for a certain class of XML documents may be gathered in a Document Type Definition (DTD) or in a Schema [W3C01]. XML Schemas were introduced after DTD's and have the two main advantages of allowing a stronger typing mechanism, and a definition of inheritance relationships.

The conceptual schemas presented in Sec. 3.2 have been translated into an XML schema. The main difficulty in working with XML schemas is that, like DTDs, they only allow a description of tree-structured documents. A difficulty emerges as our abstract syntax consists of graphs. To mimic this graph structure we introduce two main types of XML elements: those that describe concepts called *elements nodes* (classes in conceptual schemas), those that describe concept relationships called *element relationship* (association classes or associations in conceptual schemas).

Element nodes are described with an XML element tag. XML attributes are used to describe the element's attributes (e.g., Fig. 3-20).

```
<task Id="T21" name="Select-Arrival-Date" type="interactive"
importance="5" actionType="select" actionItem="element"/>
```

Figure 3-20 a UsiXML element

Element relationships are all described using the same structure. An element allows designating the relationship name. A tag source and a tag target enable a specification of the source and target of the relationship (e.g., Fig. 3-21).

```
<graphicalTransition transitionType= "open" transitionEffect= "boxIn ">
  <source sourceId="Window1"/>
  <target targetId="Window2"/>
</graphicalTransition>
```

Figure 3-21 a UsiXML relationship

It is important to note that several relationships may be defined implicitly in UsiXML taking advantage of an XML document structure.

```
<window id="window1" name="Book-a-Flight" ... >
  <button id="button1" name="start-search".../>
  <button id="button2" name="cancel-search".../>
</window>
```

Figure 3-22 Implicit relationships in UsiXML

The relationships representing a structural decomposition are represented using the tag embedding structure (i.e., tree structure) of XML. These relationships are task decomposition and interaction objects decomposition relationships. From Fig 3-22, a graphicalContainment relationship between Window1 and Button1 may be inferred from the embedding of the second element in the first one.

The adjacency relationships (abstract, graphical and auditory) are represented by the sequence of elements in a specification. In Fig. 3-22, a relationship of graphicalAdjacency may be inferred between Button1 and Button2 from the ordering of declaration of these elements with Window1.

3.5 Conclusion

This chapter has presented an ontology for the specification of UIs.

For this purpose it introduced the following concepts:

- *Viewpoints* materializing different "concerns" on the UI system (Req. 6: *Ontological separation of concern*). Four viewpoints have been introduced, motivated, and extensively defined: A *final UI* viewpoint is the implementation of a UI system it can be seen from the code level or from the rendering level (i.e., its appearance); A *concrete UI* viewpoint is a description of a UI that is, as independent as possible, of any reference to implementation details (i.e., toolkit). An *abstract UI* has been defined as a description of the UI that is, as independent as possible, of any reference to the modalities for which a UI is designed (e.g., graphical interaction, vocal interaction). A *Task and Domain* viewpoint concerns a representation of UI systems in terms of tasks to be carried out by a user in interaction with the system along with the domain-oriented concepts as they are required by these tasks to be performed.
- *UI models* have been exposed thanks to conceptual schemas expressed in UML (Req. 1: *Ontological explicitness*). UI models gather abstractions of interest in the development of a UI system. Some of the UI models are transversal to all viewpoints. A context model describes the context for which a set of models, a model or a part of a model is specified for. An inter-model relationship allows relating different models (Req. 8: *Ontological homogeneity*) either if they belong to different viewpoints (Req. 20: *Traceability*) or to a same viewpoint.

In Sec. 3.3, a mathematical formalism has been presented as an abstract syntax for representing the conceptual schemas of Sec. 3.2., and their instance (Req. 4: *Formality,* Req. 8*: Ontological homogeneity*). This formalism consists in "directed, identified, labeled, constrained and typed graphs". We relied on the theory of categories and the morphism construct to progressively enrich an initial definition of a directed graph with the desired features. In this way, any theoretical result proven for the category of graphs can be applied to the graph construct exposed in this chapter.

Finally, in Sec. 3.4., as any ontology needs a concretization to be manipulated in the real world, we defined two different, yet semantically equivalent, concrete

syntaxes. These two syntaxes reflect the conceptual structure introduced in Sec. 3.2. while respecting the graph-based mathematical notation introduced in Sec. 3.3. A graphical syntax relies on boxes and arrows to express concepts in the scope of our language and their relationships. A textual syntax, called UsiXML, is based on an XML schema definition (Req 5: *machine readable*, Req. 22: *Support for tool interoperability*).

Chapter 4 Multi-Path Development of User Interfaces

4.1 Introduction

Chapter 2 concluded with an enumeration of observations on the state of the art in transformational development of UIs. From these observations a set of shortcomings outlined deficiencies in the approaches described so far in the literature. A set of requirements was finally elicited to address these shortcomings.

This chapter addresses the methodological shortcomings and requirements of Chapter 2 by defining a flexible methodological framework for achieving transformational development of UIs.

Chapter 3 introduced an ontology for describing UIs according to various viewpoints. This chapter introduces the principles of multi-path development of UI by integrating the viewpoints in a development methodology where (1) each viewpoint becomes a development stage (2) each transition between viewpoints is a development step (3) each development step is realized through a series of transformations defined in a transformation catalog.

In such a framework, three types of transformations can be identified: model-to-model transformations, model-to-code transformations (for generating UI code from models), and finally, code-to-model transformations (for extracting a model from UI code).

This dissertation is focused on model-to-model transformations. Model-to-code transformations are supported by techniques that have already benefited of a lot of attention in the literature [Czar00]. Although we also propose, a model-to-code tool (Sec. 4.8.5), it is simply an adaptation of existing model-to-code solutions to our proper case. Code-to-model transformations are supported by specific techniques that would necessitate a dissertation in itself. We rely on [Boui04] to perform this transformation.

Section 4-2 presents the reference framework that is used to illustrate the multi-path development of user interfaces. This framework introduces the concepts of development path, development step and development sub-step.

Section 4.3 introduces the language that is used to perform development steps of our framework i.e., graph rewriting and graph grammars. Graph rewriting and graph grammars are introduced and motivated in Sec. 4.3.2. An articulation of graph grammars with our reference framework is discussed in Sec. 4.3.3. A series of examples is then provided along with the presentation of the syntax used to represent graph rewriting rules (Sec. 4.3.4). Finally, the strategy for the application of rewriting rules is discussed in Sec. 4.3.5.

Section 4.4 illustrates a specific path i.e., forward engineering by decomposing this path into steps, and then steps into sub-steps. Each sub-step is discussed and illustrated with one or several graph transformation rules that enable the realization of this sub-step.

Section 4.5 discusses the application of graph transformation to another development path: reverse engineering.

Section 4.6 addresses a third type of development path, namely context of use adaptation. A transformation step has been defined for transforming each viewpoint into a viewpoint of the same type but adapted to new constraints imposed by a change in the intended context of use of the UI to build.

Section 4.7 presents a series of tools that have been developed around the concept of multi-path development of UIs.

Section 4.8 concludes by, notably, discussing our solution to the light of the requirements identified in Chapter 2.

4.2 Reference Development Framework

Chapter 3 introduced a concept of *UI viewpoint* i.e., task and domain, abstract user interface, concrete user interface, final user interface. Viewpoints are not only conceptual perspectives on a UI but also possible stages in a development process. A framework introduced by [Calv03], integrates viewpoints into a development process perspective (left part of Fig. 4-1). This framework defines transitions between different viewpoints. Theses transitions are called *development steps* (each occurrence of a numbered arrow of Fig. 4-1).

A **development step** (hereafter referred as step) is a transformation process of an instance of a source viewpoint into another instance of a target viewpoint where source and target viewpoints types are directly adjacent in the development process.
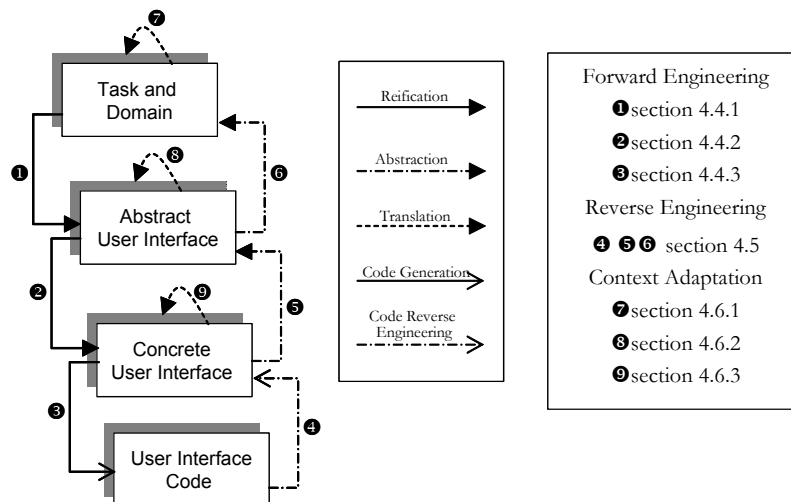
Figure 4-1 Transformation between viewpoints (left, mid.) & chapter reading map (right)

Development steps may be categorized as follows:

- **Reification** (1,2 in Fig. 4-1) is a transformation of a high-level requirement into a form that is appropriate for low-level analysis or design.
- **Abstraction** (5,6 in Fig. 4-1) is a transformation of a low level specification into a high-level specification

- ▪ **Translation** (7,8,9 in Fig. 4-1) is a transformation of a UI specification to adapt this specification to the constraints imposed by a new context of use. The context of use is defined after [Thev01] as a triple of the form *(e, p, u)* where *e* is a possible or actual environments considered for a software system, *p* is a possible or actual target platform, *u* is a user stereotype.
- ▪ **Code generation** (3 in Fig. 4-1) is a process of transforming a concrete UI model into a compilable or interpretable code.
- ▪ **Code reverse engineering** (4 in Fig. 4-1) is the inverse process of code generation i.e., it retrieves a concrete UI specification from a coded artifact.

Development steps may be combined to form development paths.

A **development path** (hereafter referred as path) represents a realization of a development activity in a particular project context. It is characterized by an initial viewpoint and a final viewpoint that is the goal of the development activity. A development path is represented by an archetypal composition of development steps.

Different types of *development paths* can be identified:

- ▪ **Forward engineering (**or **requirement derivation)** is "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation" [Chik90, Byrn92]. In this dissertation forward engineering can be viewed as a composition of *reifications and code generation* enabling a transformation of a high-level viewpoint into a lower level viewpoint.
- ▪ **Reverse engineering** is "the process of analyzing a subject system to (i)identify the system's components and their interrelationships and (ii)create representations of the system in another form or a higher level of abstraction" [Chik90, Byrn92]. In this dissertation reverse engineering can be seen as 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 the process of adapting a UI specification for another context from the one it was designed for. Context adaptation can be obtained from a *translation* of a UI model at any level.

Other development paths like:

- **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 can be 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** is a term coined by [Luo95]. It refers to a situation where a developer starts a development by a specification of the UI (no task or concept specification is priory built). Several contributions have shown that, in reality, a development cycle is rarely sequential and even rarely begins by a task and domain specification. The 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 iteration 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.

- **Leapfrog development** refers to the situation where an intermediary viewpoint is bypassed in the transformation process. In our framework for instance, it might not be needed to define an AUI if only one modality is targeted.

Development steps may be decomposed into *development sub-steps.*

A **development sub-step** (hereafter referred as sub-step) represents the realization of a particular concern in the achievement of a development step.

Some of these activities have been identified by [Luo95]. It can consist, for instance, of the selection of concrete interaction objects, the definition of the navigation, the definition of the container structure. This chapter proposes a set of sub-steps associated to each development step. The definition of development sub-steps may depend on the designer's practice, the organization rules, the type of artifact that is built, etc.

4.3 A Language for Specifying UI Models Transformation: conditional graph rewriting

4.3.1 Introduction

Model-to-model transformations have received a lot of attention in the recent literature [Varr02b, Mell03, Agra03]. The profusion of works on model-to-model transformation is mainly due to the Object Management Group (OMG) proposal on Model Driven Architecture [Mill02, OMG04].

Several techniques have been proposed in the literature to perform model-to-model transformation as required by our general framework (see Fig. 4-1). They have been surveyed in [Gerb02,Send03,Czar03]. The most relevant ones are:

- *Imperative languages* provide a mean to perform model transformation:

    - Text-processing languages like Perl or Awk are popular to perform small text transformation. These languages 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 meta-models. Jamda [Booc99], UMLAUT [Ho99], dMof [dMof02], Univers@lis [Univ99].

- *Relational approaches* [Ake03, Gerb02] rely on the declaration 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 models.

- *XSLT* [Clar99] 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 [Gerb02] 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* specification [OMG03b] provides us with 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). These are 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 elements, CWM does not provide us with a predefined language to specify the way these elements are transformed one to another.

- *Graph rewriting* has been used for many years to represent complex transformation systems. Graph rewriting is based on a pattern matching mechanism that selects a sub-graph in a graph structure and applies to this sub-graph any type of transformation (adding, deleting, or modifying a node or an edge). Graph rewriting rules may be gathered along with the graph on which they apply to form a, so called, *graph grammar*. Graph grammars have been applied in the software engineering field for representing, notably: software refactoring [Mens03], software evolution [Heck02], multi-agent system modeling [Depk02], modeling language formalization [Varr02a]. In the context of UI development, two approaches make an interesting use of graph rewriting rules: [Freu92] defines primitives for the manipulation of task models while [Sucr97,Sucr98] defines state based automaton where state transitions are operated with transformation rules.

Graph rewriting and graph grammars have been selected in the context of this dissertation to represent the various types of development steps populating our framework (see Fig. 4-1). The main reasons for this choice are that graph grammars:

- are rather declarative (they are based on graph patterns expression) and provide an appealing graphical syntax which does not exclude the use of a textual one (Req. 12: *Methodological explicitness,* Req. 17: *Methodological extendibility*).

- are based on a formally defined execution semantics based notably on pushout theory, for which many proofs have been provided (i.e., completeness [Habe01]; confluence [Heck02b]). (Req. 14: *Methodological formality,* Req. 15: *Executability,* Req. 19: *Predicatbility*).

- allow to describe transformations with the same vocabulary as specification models in a very consistent manner and for all development steps (Req. 18: *Methodological homogeneity*).

- provide extensions (i.e., conditional graph rewriting, typed graph rewriting) to check important properties of the artifact that is produced after a transformation (Req. 21: *Correctness*).

- offer modularity by allowing the fragmentation of complex transformation heuristics into small, independent chunks. The fact that graph rewritings have no-side effects facilitates this modularization (Req. 16. *Methodological separation of concern*).

4.3.2   Graph Rewriting and Graph Grammars: an overview

In Chapter 3, we have introduced a formalism that enables a representation of a UI specification using a graph structure. For the reasons developed above, *graph rewriting* systems appeared a natural choice to perform "high level manipulation" of graph structures i.e., reification, abstraction, and translation (see Sec. 4-2). We explain in this section the type of approach adopted in this work.

4.3.2.a   An introduction to graph grammars

Graph grammars provide us with an intuitive formalism for manipulating graph structures. A graph grammar is a set of graph rewriting rules (called in this work *graph transformation rules*), a graph to transform (called *host graph* or *initial graph*) and

a set of parameters (called *embed*) defining how to apply the rules on the host graph.

An algebraic approach to graph grammars has been invented by [Ehri73,79]. This approach generalizes, from strings to graphs, context-free grammars as introduced by Chomsky [Chom56]. This approach is called algebraic because it considers graphs as a special kind of algebra and defines a *gluing* operation of graphs as an algebraic construction called *pushout* in the category of graphs and total graph morphisms [Corra97].

The main advantage of the algebraic approach is that it enables proofs for a general category of graphs (it is based on *category* theory). Consequently, it is possible to apply those proofs to a large body of structures belonging to a general category of graphs.

After [Löwe93], a *graph transformation rule* (LHS,K,RHS) may be defined as a set of three graphs LHS, K, RHS. LHS is the Left Hand Side of the rule. It expresses a graph pattern that, if it matches in the host graph, will be modified to result in another graph called *resultant graph*. A LHS may be seen as a condition under which a transformation rule is applicable. RHS is the Right Hand Side of a rule. K, called *gluing graph*, is a sub-graph of LHS or RHS. It has two roles: (1) representing what is preserved during the rule application (2) showing where added elements are attached during the rule execution.

Definition 21. The **application of a rule** r to a graph G consists in the following steps:
1. *Find* an occurrence of LHS in G (this occurrence is called a *match*). If several occurrences exist, choose one non-deterministically.
2. *Remove* the part of G which corresponds to (LHS – K)
3. *Add* RHS – K to the result of last step
4. *Embed* RHS – K into G – (L – K) as it is given by the corresponding relation between RHS – K and K

The application of the four steps presented above is common to all algebraic approaches described in the literature. Two main transformation approaches have to been introduced in the literature [Roze97]: the Double Pushout Approach (DPO) and the Single Pushout Approach (SPO). The double pushout approach is the first approach presented in the literature. The single pushout approach [Löwe93] is a simplification of the first technique. The main difference between

the two techniques is that the first one makes an explicit use of an intermediary graph construct K called *gluing graph* from which two total graph morphisms may be defined $l : K \rightarrow LHS$ and $r : K \rightarrow RHS$. The SPO (illustrated by Fig. 4-2) defines a direct relation between a LHS and its corresponding RHS, this relation consists of a partial graph morphism $r : LHS \rightarrow RHS$ (see definition 4). In this dissertation we decided to use a SPO because the results obtained for the DPO are valid for the SPO as this latter has been proved a generalization of the first. Furthermore, the SPO is more intuitive than the DPO.

Definition 22. A **transformation rule** $r : LHS \rightarrow RHS$ is a partial graph morphism (see definition 4) from *LHS* to *RHS*
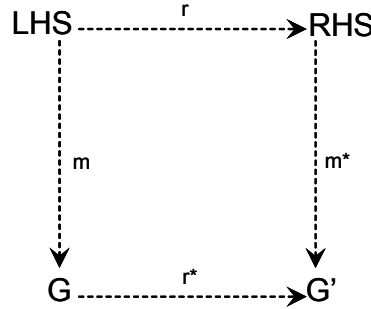
Figure 4-2 A rule application with the simple pushout approach

From definition 22, a direct transformation from G to G' may be defined as follows:

Definition 23. A $G \Rightarrow_{r,m} G'$ is a **direct transformation** such that $\exists$ total graph morphism $m : LHS \rightarrow G$ that forms a *pushout* ($r*$ and $m*$) with $r : LHS \rightarrow RHS$.

Intuitively, a *pushout* can be understood as a construction that allows to glue one part of a structure to another part of a structure. It may be understood as follows:

Definition 24. A **pushout** (Fig. 4-2) of two graph morphisms $m : LHS \rightarrow G$ and $r : LHS \rightarrow RHS$ is a triple $(G', m* : RHS \rightarrow G', r* : G \rightarrow G')$ such that:
1. G is a graph (see definition 1), $m*$ and $r*$ are graph morphisms (see definition 4).
2. $m* \circ r = r* \circ m$ (*Commutativity. This* property guarantees the existence of the pushout)
3. $\forall$ graph E, $\forall$ graph morphism $h_1 : RHS \rightarrow E$, $h_2 : G \rightarrow E$ and

---

$h_1 \circ r = h_2 \circ m : \exists!$ graph morphism $h : G' \to E$ such that $h \circ m^* = h_1$ and $h \circ r^* = h_2$ (*Universality.* This property guarantees that G' is minimal. For any other graph for which the commutative property holds there should be only one graph morphism between G' and this graph).
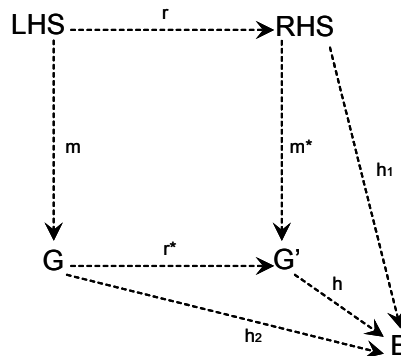
Figure 4-3 Pushout construction applied to graph morphisms

Transformations can be combined to form transformation systems:

Definition 25. A **transformation system** is an arbitrary group of transformation rules $r_1, r_2, r_3, ..., r_n$ indexed by their name.

From this definition *a graph grammar* may be defined as:

Definition 26. A **graph grammar** G can be defined as a tuple (G, R) where G is a *host graph* (or *initial graph*) and R is a transformation system (see definition 25).

An "*embed*" is an important parameter in the application of a graph transformation rule. Specifying removals and additions of elements in the resulting graph is not enough. Indeed, one has to specify what happens with edges coming and going from a node that is altered during an execution of a transformation. Are they left dangling? Are they systematically erased? Different types of *embed* have been studied in the literature. We adopt a simple and conservative approach regarding this problem:

- If a node in LHS has a correspondence in RHS, all edges adjacent to this node are preserved.
- If a node in LHS has no correspondence in RHS, all edges adjacent to this node are erased.

4.3.2.b   Conditional graph rewriting

Conditional graph grammars have been discussed very early in the literature. Conditions may be positive, Positive Application Conditions (PAC), or negative, Negative Application Condition (NAC). They may be also differentiated depending on the moment they are checked. The following categories of application conditions may be listed:

- *Positive application pre-conditions* are assertions on the host graph that have to hold true before the application of a rule. Positive application conditions were already mentioned in the seminal paper on graph grammars in 1969 [Pfalz69] and extended in [Ehri86, Habe96]. Most of positive application conditions can be expressed within the LHS of a rule itself. Nevertheless, when a condition falls far from the scope of a LHS it may be useful to express such condition as a separate structure.
- *Negative application pre-conditions*, also called forbidden contexts, are assertions that have to hold false before the application of a rule. Negative application conditions were mentioned very early in the literature in [Montanari70] and extended in [Ehrig86, Habe96].
- *Positive and Negative application post-conditions* are assertions that have to hold respectively true or false after the application of a rule. If not verified, the rule application is cancelled. Theoretical foundation of this technique can be found in [Heck95].

Conditional graph rewriting significantly enhances the expressivity of transformation rules. It is important to note that our implementation uses positive application condition to verify the *consistency* of our grammars. That is to say, verify that each graph produced by a transformation constitutes a legal sentence of the target vocabulary. Our target vocabulary is defined by typed graphs (as explained in Chapter 3). After each application of a rule the transformation engine checks if the resultant graph is well compliant with the meta-language defined in the graph of type. This process is referred in the literature as *typed graph transformation.*

Conditional graph rewriting entails a redefinition of our description of a rule application:

Definition 27. The application of a rule (with pre- and post-conditions) r to a graph G consists in the following steps:

1. *Find* an occurrence of LHS in G (this occurrence is called a *match*). If several occurrences exist, choose one non-deterministically.
2. *Check* preconditions of both type PAC and NAC. If not verified, then *skip*
3. *Remove* the part of G which corresponds to (LHS – K)
4. *Add* RHS – K to the result of last step
5. *Embed* RHS – K into G – (L – K) as it is given by the corresponding relation between RHS – K and K
6. *Check* postconditions of both type PAC (and notably that the resulting graph is properly typed) and NAC. If not verified, then *undo* the transformation rule

4.3.3 Graph Grammars and the Reference Framework

Graph transformations are used to perform viewpoint-to-viewpoint transformations i.e., reifications, abstractions and translations (see Fig. 4-1). We transform a UI specification with a set of transformation rules taken from a transformation catalog. Transformation rules have a common meta-model with UI specification models. Indeed, a *rule term* (i.e., a NAC, a LHS, or a RHS) may be seen as a fragment of specification. We preserve the consistency of transformed artifact as the resultant UI specification is checked upon its meta-model. Transformation rules resulting in a non-consistent resulting graph are not applied. Our transformations are *type preserving*.
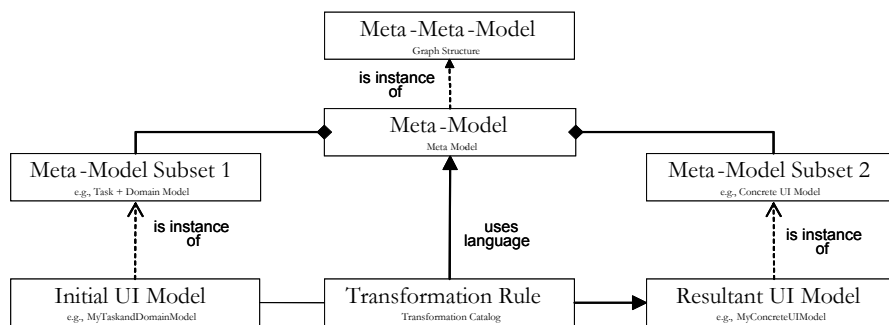
Figure 4-4 Type preserving UI model transformations

Fig. 4-4 shows how graph transformation articulates with the methodological concepts introduced in Sec. 4.2. A development path is composed of development steps. The latter being decomposed into development sub-steps. A development sub-step is realized by one (and only one) transformation system and a transformation system is realized by a set of graph transformation rules.

Methodological World

Graph Transformation World
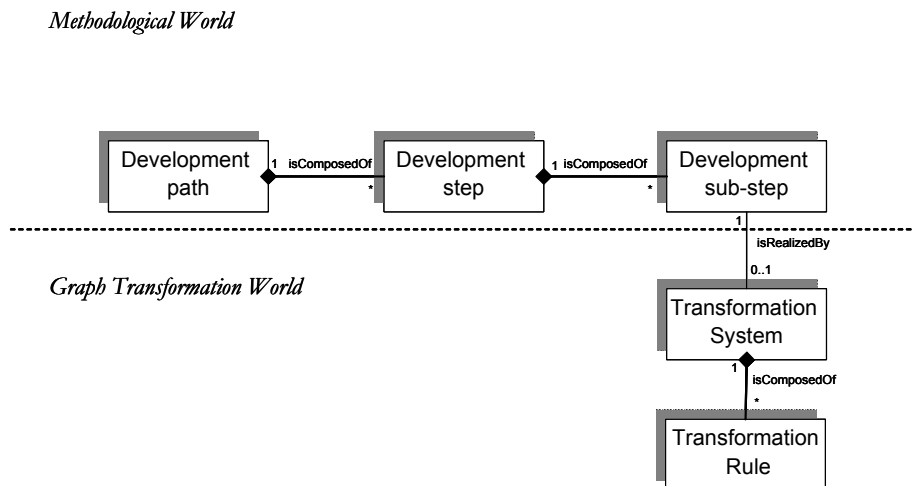
Figure 4-5 Articulation of graph transformations with transformational development of UIs

In the remainder of this chapter, we illustrate three important path types i.e., forward engineering, reverse engineering, and adaptation to the context of use. An example for each development step, and sub-step is provided. All examples use the graphical formalism of the graph transformation tool AGG [Ehri99] presented hereafter.

4.3.4   Concrete Syntax for Transformation Rules

4.3.4.a   Visual syntax for transformation rules

This section provides several examples of basic transformations in order to allow a reader that is unfamiliar with the graph transformation techniques to better understand the more complex transformations of Sec. 4.4, 4.5, and 4.6. We use simple node types (inspired from football) across the different examples. Each example shows a particular feature on the type of transformation defined in this dissertation.

▪ Node creation

The simplest rule that can operate on a graph is a node creation. Fig. 4-6 represents such a rule. Note the emptiness of the left hand side providing that no condition is necessary to create the node described in the right hand side.

LHS   RHS

player
Id="12"
name="Jovonderdong"
salary="1000"

Ø   ::=
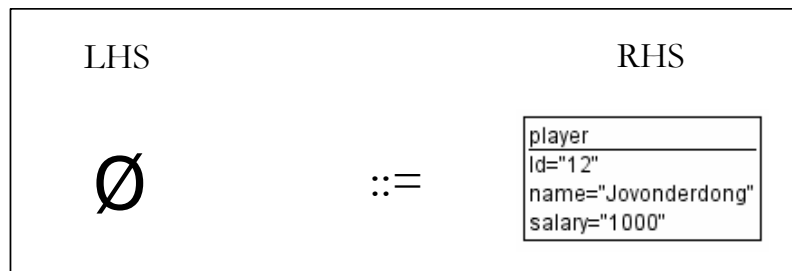
Figure 4-6 Creation of a node with attributes

▪ Node modification (identified instance)

Fig. 4-7 shows a rule selecting a specific node on the base of its id attribute and assigns to this node a specific attribute value. In this case graph transformation rules can not be considered as a pattern language as they make an explicit reference to one graph element in the host graph. Note that even if the node subject to the transformation is selected on the base of its id, it is necessary to indicate explicitly the gluing conditions (i.e., figures before the node type). Indeed the system does not know the semantics of the id (e.g., its uniqueness).

LHS   RHS

1:player
Id="12"

::=

1:player
Id="12"
salary="1500"
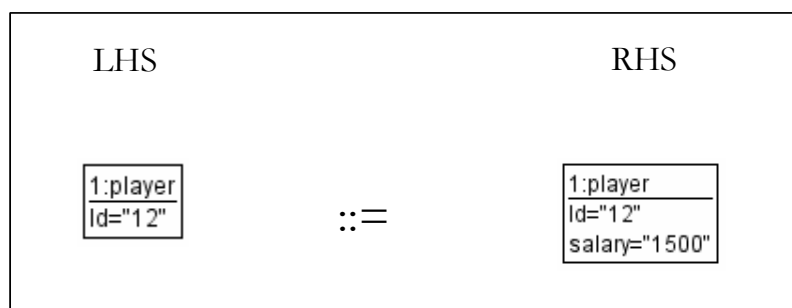
Figure 4-7 Node modification (identified instance)

- Node modification (unidentified instance)

Fig. 4-8 shows a rule that could be expressed as follows: "for all players that played the match on the 04/06/04, align their salary to 2000".
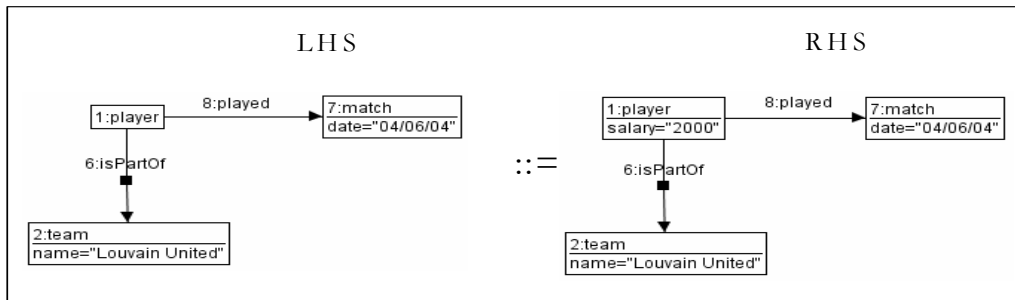
Figure 4-8 Node modification unidentified instance

- Negative application condition (1)

A negative application condition could be added to the preceding rule (Fig. 4-9). This negative application condition transforms the meaning of the rule into: "for all players that played the match on the 04/06/04, raise their salary to 2000 unless they played the match of the 10/10/03" (this last match was a very bad one!).
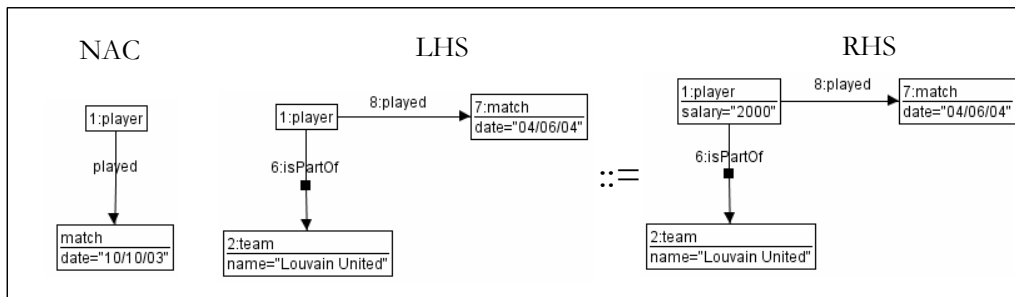
Figure 4-9 Negative application condition

- Negative application condition for iterative execution of rules (2)

If we want to be able to define rules that detect patterns on a graph structure and make appropriate modifications depending on the presence of this pattern, we have to let the system search iteratively for the left hand side. Consequently, there is a risk that the pattern matching algorithm will match several times on the same instances leading to an infinite looping of the execution of the rule. For this purpose a special negative application condition has to be introduced. "NAC2" in

Fig. 4-10, is such an example. It says that the rule should not be applied if the salary of the player equals already "2000".
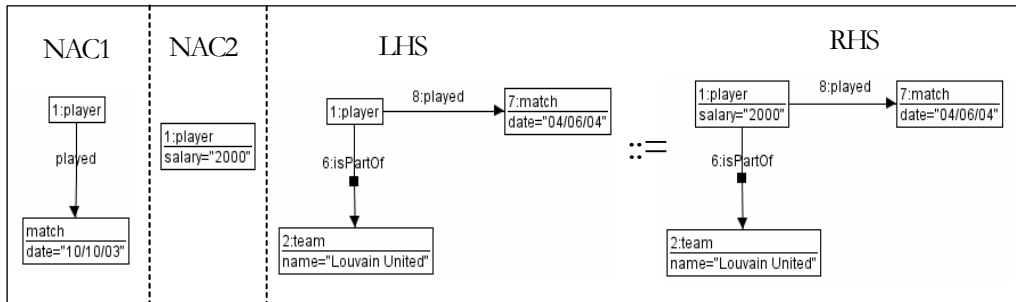
Figure 4-10 Negative Application Condition (2)

- Rule with variable and variable condition as positive application condition

Fig. 4-11 could be expressed as follows: "raise by 500 the salary of all players that played the match of the 04/06/04 only if their salary was inferior to 3000". This rule illustrates two different mechanisms. A first one consists in the use of a variable in the left hand side, this variable is incremented by a constant in the right hand side ("x:=x+500"). A second one consists in the use of a positive application condition that compares the value of a variable with a constant (note that x could have been compared with another variable).
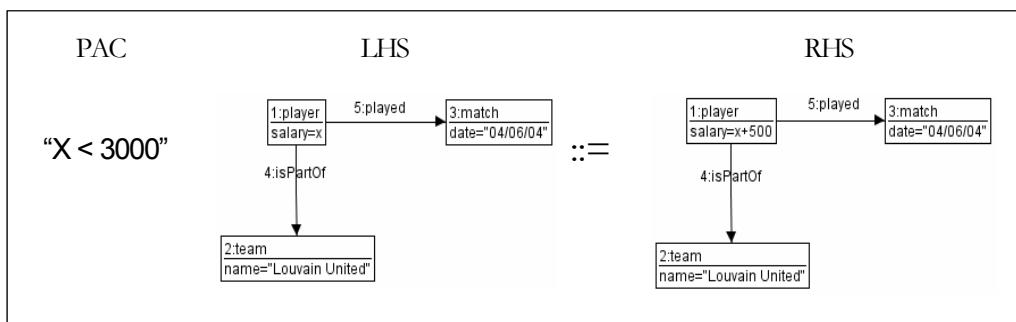
Figure 4-11 Rule with variable and positive application condition

- Transfer of an attribute value

Fig. 4-12 illustrates a very altruistic rule, which may be expressed as follows: "If two players of a same team are friends and one earns more than the other, then align their salaries". Here the value of a variable is transferred from one node (the richest player) to another one (the poor friend).
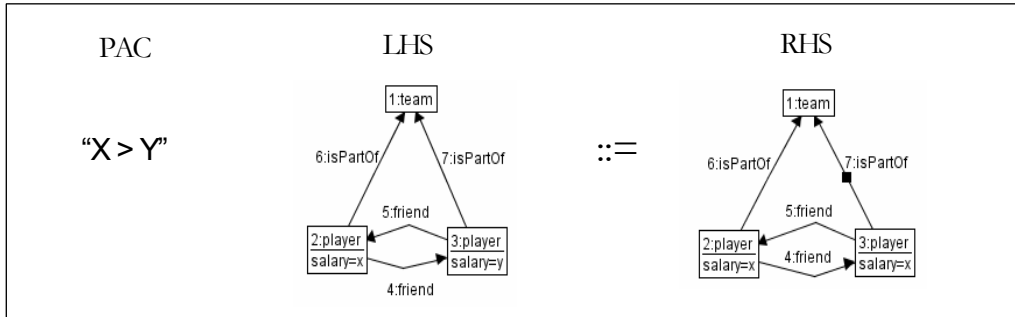
Figure 4-12 Transfer of an attribute value

- Edge creation

Fig. 4-13 illustrates a rule that could be expressed as follows "All players of Louvain United with a salary greater than 3000 should be assigned to the match of the 04/06/04" (It will be a tough match !)
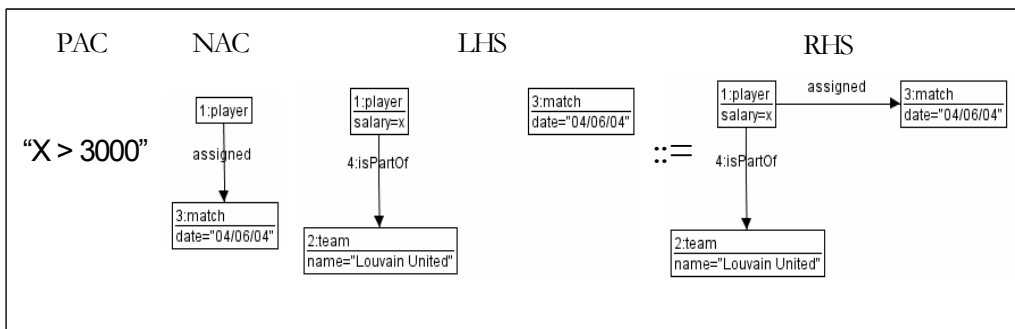
Figure 4-13 Edge creation

- Node deletion

Fig. 4-14 shows the most delicate operations of all: node deletion. Indeed, the problem with node deletion is that they raise the question of dangling edges (see discussion in Sec. 4.3.5). We adopt a very clear policy regarding this problem: all edges pointing to or originating from a deleted node should be erased. In other words, no dangling edges are allowed.
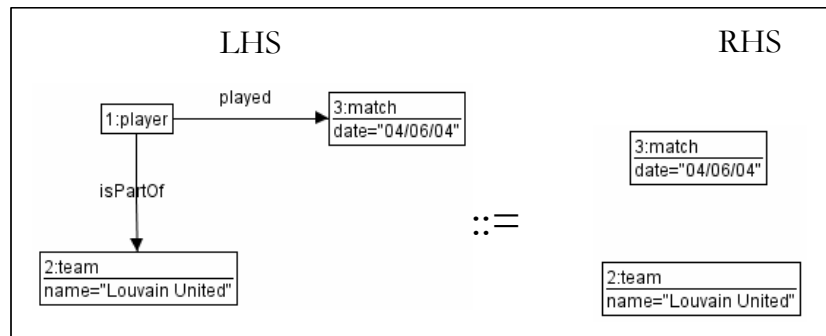
Figure 4-14 Node deletion

4.3.4.b Textual syntax

A textual expression of the transformation rules has been embedded in UsiXML (see Sec. 3.4.2). This textual syntax allows us to store rules and constitute transformation catalogs. Fig. 4-16 provides an example of the textual syntax used for a rule. Note that, the implicitness of the adjacency relationship cannot be transposed for rules as it raises ambiguities in the interpretation of rules (see discussion in Sec. 3.4.2).

Figure 4-15 Textual syntax for expressing transformation rules

4.3.5 Application Strategy of Transformation Systems

A transformation system is composed of several rules. This raises the problem of how to apply those rules while guaranteeing important properties: confluence and termination.

An application strategy of a graph grammar is defined as the order in which transformation rules are applied to an initial graph. Rules can be applied concurrently, in an order independent manner, or in a controlled sequential way. The rule application strategy raises the problem of the determinism of a grammar i.e., its ability to produce one and only one resultant graph. This property is also called *confluence*. The confluence property has been proved for a particular type of grammars where transformation rules were shown *parallely independent* [Löwe93]. Very intuitively, *confluence* can be proved for a parallel execution if one can demonstrate that transformation rules in a grammar do not interfere with each other. That is to say that no rule deletes or introduces nodes that are needed by another one to match.

In the context of this dissertation, the property of parallel independence is almost never possible to possible to assess. This observation is a consequence of the intrinsic nature of the process applied to an initial specification model. Indeed, our transformation systems realize an incremental consolidation of an initial specification. A transformation system relies, in most cases, on the information (i.e., specification chunks) generated by a preceding application of another transformation system.

Consequently, transformation systems proposed in this work must be controlled with a special technique called *Programmed Graph Rewriting* [Schü97], a generalization of ordered rewrite systems introduced in [Bunk82]. This techniques uses graph rewriting rules as process units that may be composed arbitrarily using a set of pre-defined operators (e.g., sequences, parallel sequences, loop structures, test) as so to obtain a desired algorithmic behavior.

Our application strategy is represented in Fig. 4-16. A development step is externally initiated (e.g., in response to a context change, by a designer's decision, by any external entity). Then the first transformation system is executed, when it is terminated, the second one is applied and so forth until the last transformation system terminates. A similar progression is applied into transformation systems. This trivial application strategy solves the problem of confluence.
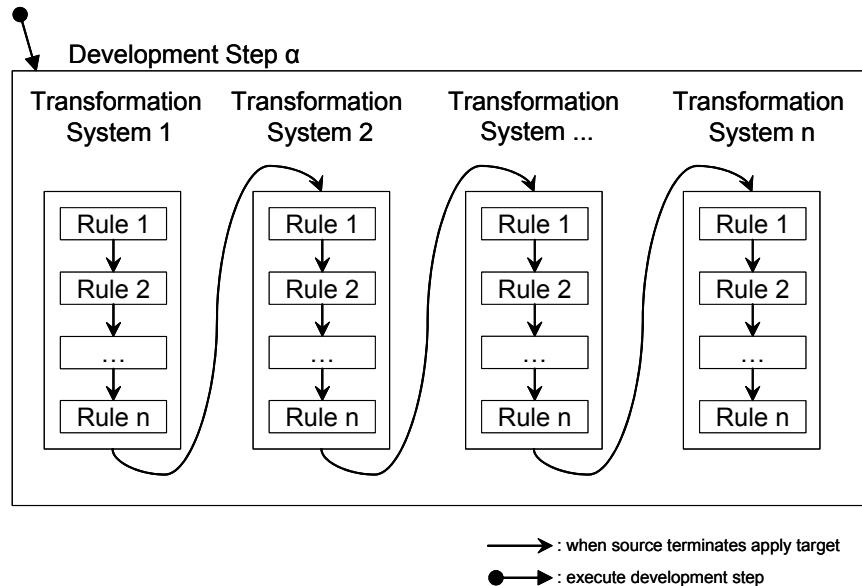
Figure 4-16 An ordered application strategy

Considering the application strategy of Fig. 4-16, it can be said that the termination of a development path can be guaranteed if each of its development steps terminates. A development step terminates if each of its sub-steps terminates. (i.e., its associated transformation systems). A transformation system terminates if each of its composing rules terminates.

A rule terminates when no more match can be found in the resultant graph. Note that a problem of infinite looping may arise, especially when dealing with non-deleting grammars. Indeed, a rule application consists at first of searching, non deterministically, a match into a host graph. It is more likely that a LHS will match several times on the same sub-graph if no precaution is taken. To solve this problem, a technique would be to tag already matched sub-graph, an alternative technique consists of replicating part of a right hand side in the negative application condition. This last technique was adopted in this work (see example in Fig. 4-10).

4.4 Forward Engineering

Forward engineering can be seen as a sequence of progressive refinements applied on a high level specification in order to obtain an application code or a lower level specification. [Czar00] identifies several types of refinements. The following ones are relevant in our context:

- *Decomposition* consists of refining a high level concept into a set of lower level concepts. For instance, a task is mapped onto a set of interaction objects.
- *Choice of representation* associate a representation with a higher level concept, for instance a couple (task, domain concept) is mapped onto a particular abstract interaction object.
- *Specialization* states that a more general abstraction is transformed into a more concrete one, being more specific for a particular context of use. For instance, an abstract interaction object equipped with an input facet is transformed into an input field for the graphical modality.
- *Concretization* involves adding more detail to a concept. For instance, adding style attributes (e.g., color, border style) to a concrete user interface specification.

Many of the transformations illustrated bellow can be assigned to one or several of these types.

Historically, model-driven methodologies have focused primarily on the derivation of a widget-dependent specification or, even, straightforwardly code (see Chap. 2) from a data model, a form of a domain model restricted to only data structures. When such methods and tools had to be used for producing several UIs for different computing platforms, it turned out that there was no support and no identification of the common parts, thus resulting on restarting the whole methodological process from scratch. Toolkit-independent models were introduced [Paus92] to tackle this problem. These abstractions became the goal viewpoint of forward engineering processes. In the meantime, it was realized that domain model (in its various forms) was not expressive enough to describe the wide variety of user tasks and, more important, their logical and temporal relationship. Indeed, a domain model is only able to express pattern tasks with a predefined semantics like create, read, update, delete, search, etc. The only way to

specify temporal relationships between these tasks was by using pre-conditions on domain methods, a very indirect way to describe the interaction of a person with a system! Task models were introduced in UI engineering [John92] to enable a rich expression of user's tasks. It is now acknowledged that task and domain models should be used in parallel as a starting point of a forward engineering path. That is the option that has been taken in the illustrations provided in this section. Note, however, that our approach can accommodate with a definition of rules relying solely on a task or a domain model. Coverage of older heuristics is thus also possible.

As shown in Fig. 4-17, 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 user interface, which is then transformed into a concrete user interface model. The concrete user interface model is then used to generate UI code. A forward engineering development path is detailed hereafter by decomposing it into development steps, and sub-steps.
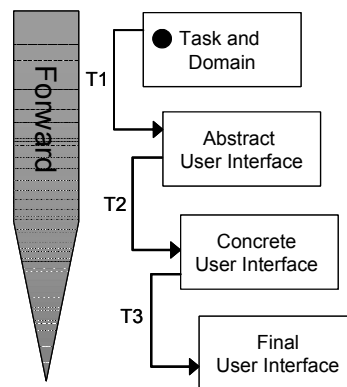
Figure 4-17 Forward development of UIs

4.4.1   Step: From Task & Domain to Abstract User Interface

Step T1 (see 4-17) concerns the derivation of an AUI from models at the T&D viewpoint (e.g., a task, a domain or task and domain model). This development step may involve development sub-steps illustrated in Fig. 4-18.

STEP : From Task & Domain to AUI
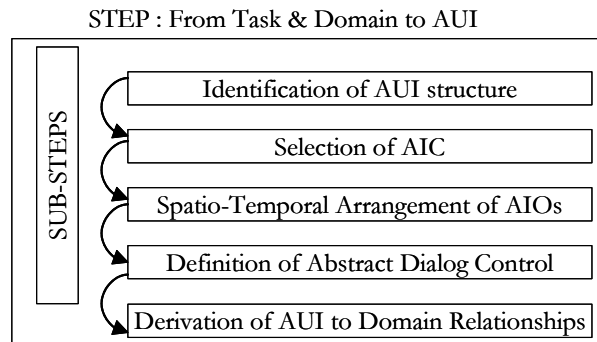
Figure 4-18 Development step : from Task & Domain to AUI

To illustrate this step, a decomposition into five sub-steps arranged by logical order is proposed. This decomposition illustrates the separation of concern principle applied to transformation processes (Req. 16).

First, abstract containers and abstract individual components (i.e., groups of abstract objects) are identified; then individual elements specification are refined and, finally, arranged into the previously identified containers. This completes the design of the abstract presentation. Then, the dialog is added thanks to two phases: a definition of abstract dialog control and the derivation of AUI to domain relationships.

This shows that the decomposition of this step into sub-steps can follow a logical order that is principle-based, here in a top-down approach.

4.4.1.a   Sub-step: Identification of Abstract UI structure

It consists of the definition of groups of abstract interaction objects. 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 simple heuristics like "for each group of tasks, 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 proposed by [Pate99] or information flow between tasks in TRIDENT method proposed by [Boba95c]). We propose an example based on Teallach methodology [Grif99] that creates an AUI structure, a transposition of a task hierarchical structure.

Example 1 is a transformation system composed of two rules (Rule 4-1, 4-2) enabling the creation of a simple hierarchical structure containing abstract individual components and abstract containers.

Rule 4-1: For each leaf task of a task tree, create an Abstract Individual Component. For each task, parent of a leaf task, create an Abstract Container. Link the abstract container and the Abstract Individual Element by a containment relationship.
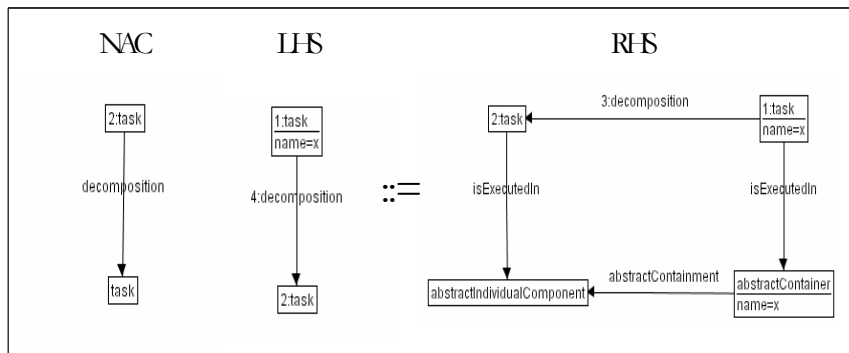
Rule 4-1 Creation of abstract individual components derived from task model leaves

Rule 4-2: Create an Abstract Container structure parallel to the task decomposition structure.
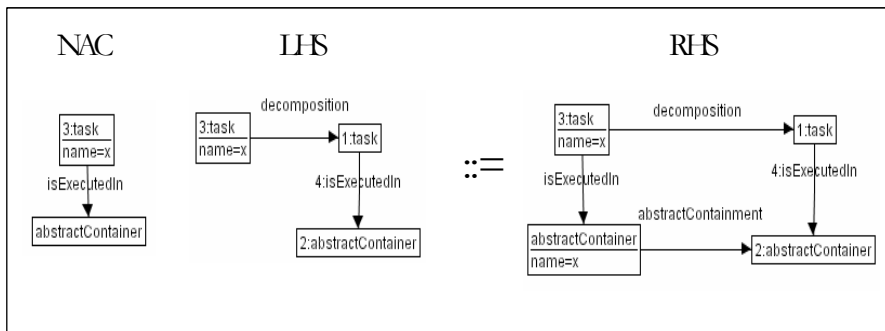
Rule 4-2 Creation of abstract containers derived from task model structure

4.4.1.b   Sub-step: Selection of abstract individual component

The current sub-step merges the information contained in the task model and in the domain model to produce the specification of AICs. An AIC specification is provided by the identification of its facets. As so, "responsibilities" of each AIC

are identified independently of the task and domain models (Req. 13: *Methodological flexibility*). To perform this transformation, several elements may be taken into consideration: action types, action items, task types, data types of domain attributes, domain of value of domain concepts, enumerated domains, structure of the domain model (e.g., inheritance, aggregations). It is hard to compare this sub-step with the literature. The level of abstraction of interactors we propose at this level is present only in [Grif99]. Consequently, we propose our own heuristic.

Leaf tasks of a task mode are described with an **actionType** and an *itemType* indicating a generic action and a generic object on which an action is being performed (see Sec 3.2.1). A **manipulates** relationship add information on the domain concepts that a task manipulates (**itemType** being very generic).

Table 4-1, provides us with a systematic expression of possible mappings of task types to AIC facet types. The left column presents a set of meaningful combination of values for task **actionType** and task **actionItem**. The right column shows corresponding AIC facets with a refined expression of the **actionType** and **actionItem** at the AIC level, depending on the type of domain concept that is manipulated (an attribute, a collection of attributes, an object or a collection of objects).

Task [actionType] + [actionItem]	AIC Facet type + [actionType] + [actionItem]
[Start/go] + [Operation]	[Control]
[Stop/exit] + [Operation]	[Control]
[Start/Go] + [Container]	[Navigation]
[Stop/exit] + [Container]	[Navigation]
[Select] + [Element]	[Input] + ([Select] + [Attribute Value] OR [Select] [Object])
[Select] + [Collection]	[Input] + ([Select] [Attribute Value Set] OR [Choose] [Object Set])
[Create] + [Element]	[Input] + ([Create] [Attribute Value] OR [Create] [Object])
[Create] + [Collection]	[Input] + ([Create] [Attribute Value Set] OR [Create] [Object Set])
[Delete] + [Element]	[Input] + ([Delete] [Attribute Value] OR [Delete] [Object])
[Delete] + [Collection]	[Input] + ([Delete] [Attribute Value Set] OR [Delete] [Object Set])
[Modify] + [Element]	[Input] + ([Update] [Attribute Value] OR [Update] [Object])
[Modify] + [Collection]	[Input] + ([Update] [Attribute Value Set] OR [Update] [Object Set])

[View] + [Element]	[Output] + ([View] [Attribute Value] OR [view] [object])
[View] + [Collection]	[Output] + ([View] [Attribute Value Set] OR [View] [Object Set])
[Monitor] + [Element]	[Output] + ([Monitor] [Attribute Value] OR [Monitor] [Object])
[Monitor] + [Collection]	[Output] + ([Monitor] [Attribute Value Set] OR [Monitor] [Object Set])
[Move] + [Element]	[Input] + ([Move] [Attribute] OR [Move] [Object])
[Move] + [Collection]	[Input] + ([Move] [Attribute Value Set] OR [Move] [Object Set])
[Duplicate] + [Element]	[Input] + ([Duplicate] [Attribute] OR [Duplicate] [Object])
[Duplicate] + [Collection]	[Input] + ([Duplicate] [Attribute Value Set] OR [Duplicate] [Object Set])

Table 4-1 Association of task action types with AUI facets

Example 2 is composed of Rule 4-3. It exploits information on task action types to attach appropriate facets to corresponding abstract individual components.

Rule 4-3: for each abstract individual element mapped onto a task such that the tasks nature consists of the activation of a method and this task is mapped onto a class, assign to the abstract individual component an action facet that activates the mapped method.
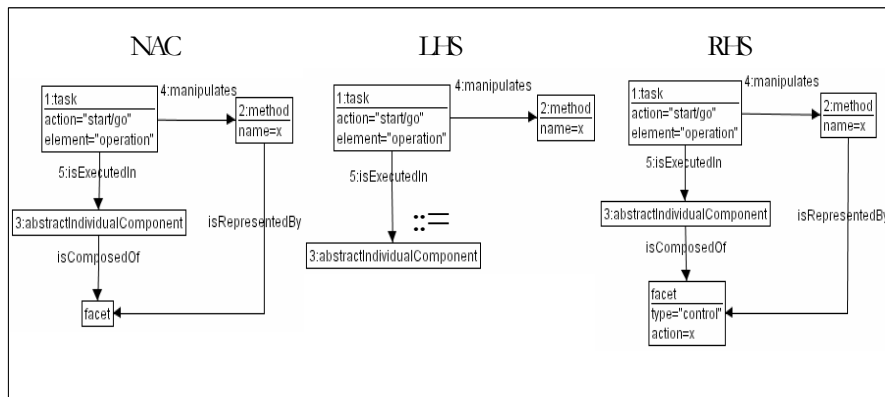
Rule 4-3 Creation of a facet for an abstract individual component derived from task action type

4.4.1.c   Sub-step: Spatio-temporal arrangement of abstract interaction objects

The principle of this sub-step is to exploit a task model structure to derive information on a spatio-temporal arrangement of elements populating an AUI. Spatio-temporal arrangement is not to be confused with a mechanism for controlling the locus of control of the UI (this latter concept is referred, in Chap. 3, to *dialog control)*. Spatio-temporal relationships only allow a specification of layout constraints between AIOs.

A task or domain model contains little information for a precise spatio-temporal specification. It cannot be said, for instance, that an abstract container partially overlaps another one, or that two AICs are "right aligned" on the simple basis that the tasks these AIOs represent are in such or such temporal relationship.

It is not amazing that the problem of layout derivation from task and domain model have been left in the shadow by the literature. Three solutions are proposed to face the problem of layout definition: (1) some heavy (and hard-coded) assumptions are done on the way elements should be organized topologically. Teresa [Pate03] tool, for instance, calculates layout on the basis of a built-in algorithm (2) presentation templates are at disposal of the designer to customize layout structure of the UI [Lonc96] (3) a designer is able, through a specific tool, to re-shuffle the layout by hand in the model itself [Boda95b]. Of course in any case a designer is always able to edit and reshuffle a UI at the code level (i.e., final UI). A problem with this solution is that modifications done in the code, potentially, endanger the consistency of the UI models with the code that has been generated.

Nonetheless, we consider that the order in which tasks are specified may reflect designer's intent about the ordering of elements allowing the realization of a task. An **abstractAdjacence** relationship expresses the existence of an Allen relationship (with no further detail) between two AIOs. This relationship might be either specified latter on by hand, by the designer, or left "as is" and transformed into a concrete relationship

Example 3 is composed of Rule 4-4. It places abstract individual components in precedence relationship (with **abstractAdjacency**) based on the fact that the tasks they represent are sequentially ordered. To perform a complete arrangement, every type of task temporal relationship has to be covered by a rule.

Rule 4-4: for every couple of AIC mapped onto sister tasks that are sequential ">>", create a relationship of type "abstractAdjacence" between these AIOs.
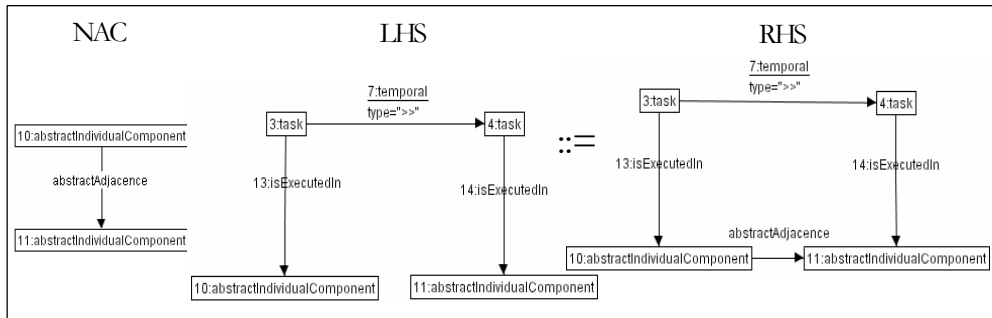
Rule 4-4 A sequentialisation of abstract individual component derived from task temporal relationships

4.4.1.d   Sub-step: Definition of Abstract Dialog Control

A task model defines temporal constraints between tasks. These constraints have been expressed in term of pre- and post- conditions in Sec. 3.2.1. Tasks have been mapped onto abstract containers and abstract individual components in a preceding sub-step (Sec. 4.4.1.a). The present sub-step transposes task temporal constraints to the AUI.

The dialog control expresses the locus of control (i.e., availability) for initiating the dialog in a UI. Dialog control consists of controlling certain states of the user interface in order to enforce temporal constraints imposed between the elements of the interface. Dialog control allows answering to the following question: when is such interaction object available or not?

As for task temporal operators, the abstract dialog control proposed here is based on an implicit mechanism of pre- and post- conditions underlying temporal constraints. Fig. 4-19 illustrates a dialog specification for an AUI. Facet types are represented as icons (i.e., a pen for input, a machine for control on machine initiative, a user working on a machine for control on user initiative). Abstract containers are represented as boxes. Temporal constraints are defined between these elements (see Sec. 3.2.1 for an explanation on temporal operators).

This representation allows a determination of the locus of control of elements i.e., to determine when they are available and when they are not. For instance it can be said that AC.2 cannot be available until AC.1 is "terminated". And AC.1 will be

terminated iff AIC.11 and AIC.12 and AIC.13 will be terminated and so on. Now an essential question is: what terminates an AIC exactly ? For a control AIC, the output of the method that is executed can determine if the AIC is terminated or not. But what terminates an input AIC ? This question is impossible to answer at this level as the concrete object that will reify this AIC is not known. It might be an auditoryInput or a textComponent. Their termination events might be, for instance: a blank for a period of 2 sec. for the auditory element, a click outside the focus of the object or a "tab key" press for the graphical component.
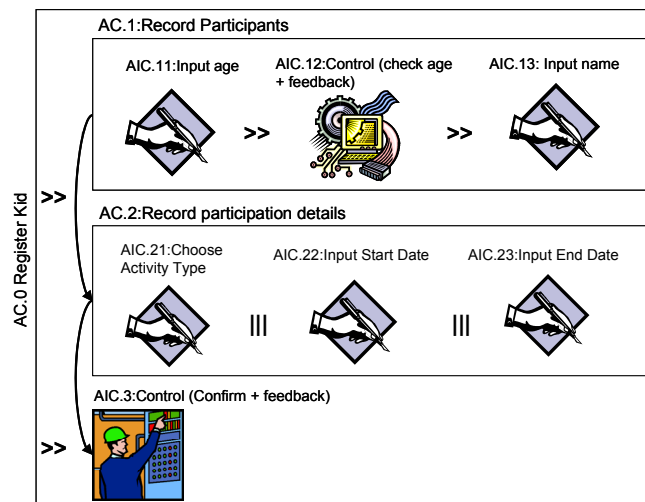
Figure 4-19 Abstract Dialog

Example 4: is composed of Rule 4-5. It consists of a transposition of task temporal relationship to abstract dialog control relationships.

Rule 4-5: for each couple of sister tasks mapped onto AICs, define a dialog control relationship between these AIC that has the same semantic as the temporal relationship.
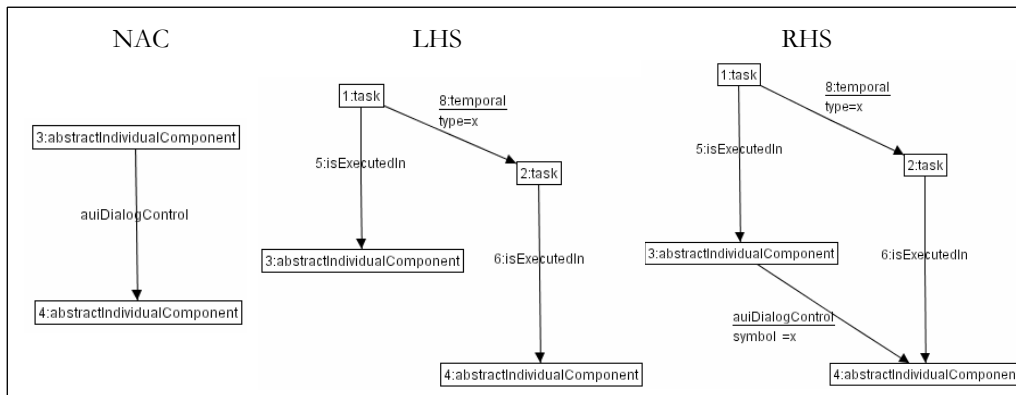
Rule 4-5 Abstract Dialog Derivation from Task Model

4.4.1.e   Sub-step: Derivation of AUI to domain mappings

Manipulates relationship has been introduced in Sec. 3.2.6. The information contained in this relationship may be the basis of a refinement expressing mappings between a UI model (at abstract or concrete level) and domain model. Three different heuristics may operate in order to achieve this goal: (1) tasks realizing an input on a domain value allow a derivation of an updates relationship, (2) a view task or a monitor task may allow a derivation of an observes relationship, and (3) a task having an operation as item allows a derivation of a triggers relationship. This example shows a heuristic that is more complex than merely mapping tasks or domain to UI elements.

Example 5 is composed of one single rule (Rule 4-6) and derives triggers relationship between an abstract individual component and a domain concept.

Rule 4-6: for each task that manipulates a method, the AIC that represents this task triggers the method.
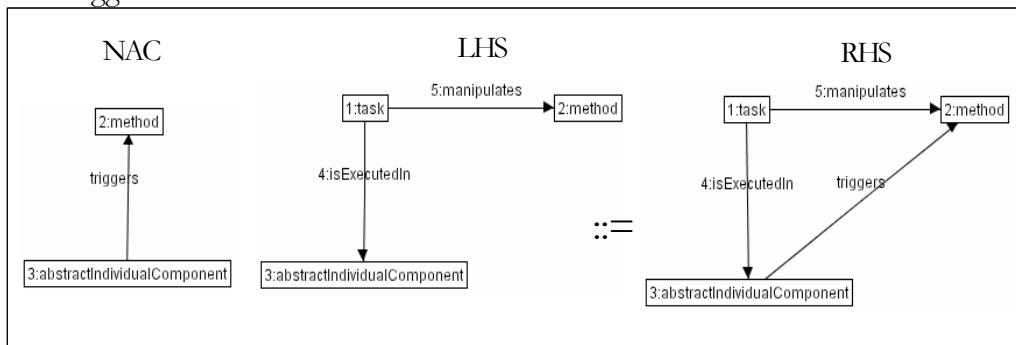
Rule 4-6 Deriving triggering relationships from task domain mappings

Example 6 is composed of Rule 4-7 and Rule 4-8. It sheds a new light on information passing relationship (see Sec. 3.2.1).

Rule 4-7: if two sister tasks manipulate a same attribute and are temporally constrained with a "sequence with information passing" relationship, each of these tasks being mapped onto an AIC, then the AIC that is mapped with the first task updates the attribute manipulated by the tasks. The second AIC observes this attribute.
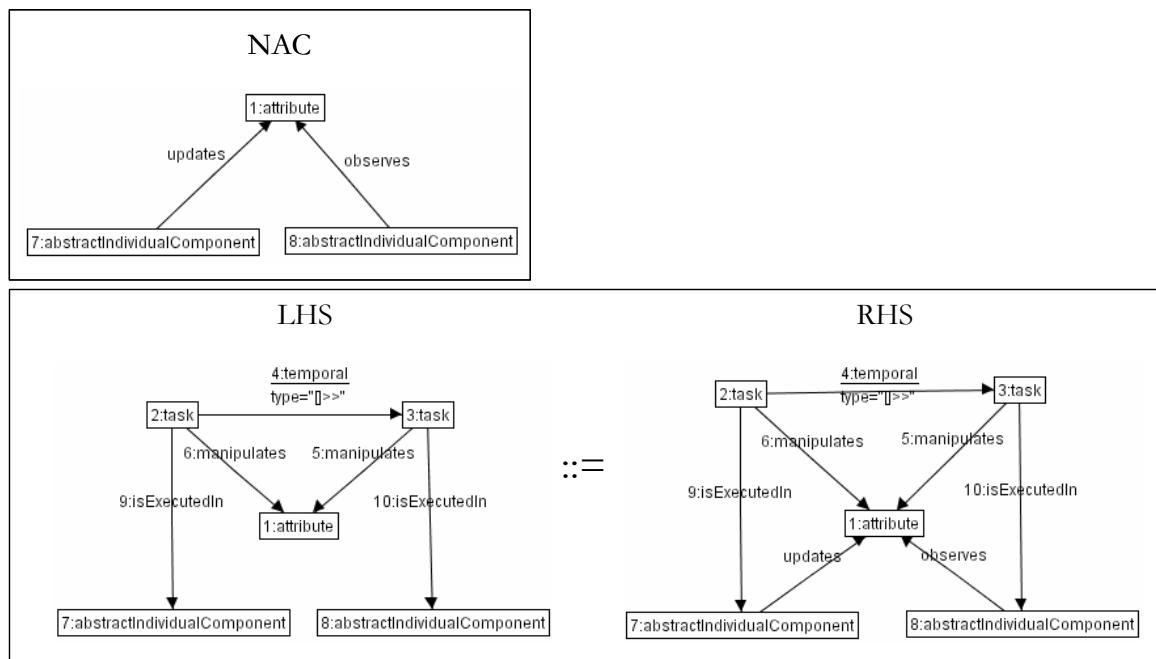
Rule 4-7 Derivation of Updates and observes structure on the base of a task relationship of sequential information passing

Rule 4-8: if two sister tasks manipulate a same attribute and are temporally constrained with a "concurrent information passing" relationship, and each of these tasks is mapped onto an AIC, then both AIC observe and update the attribute that is manipulated by the tasks.
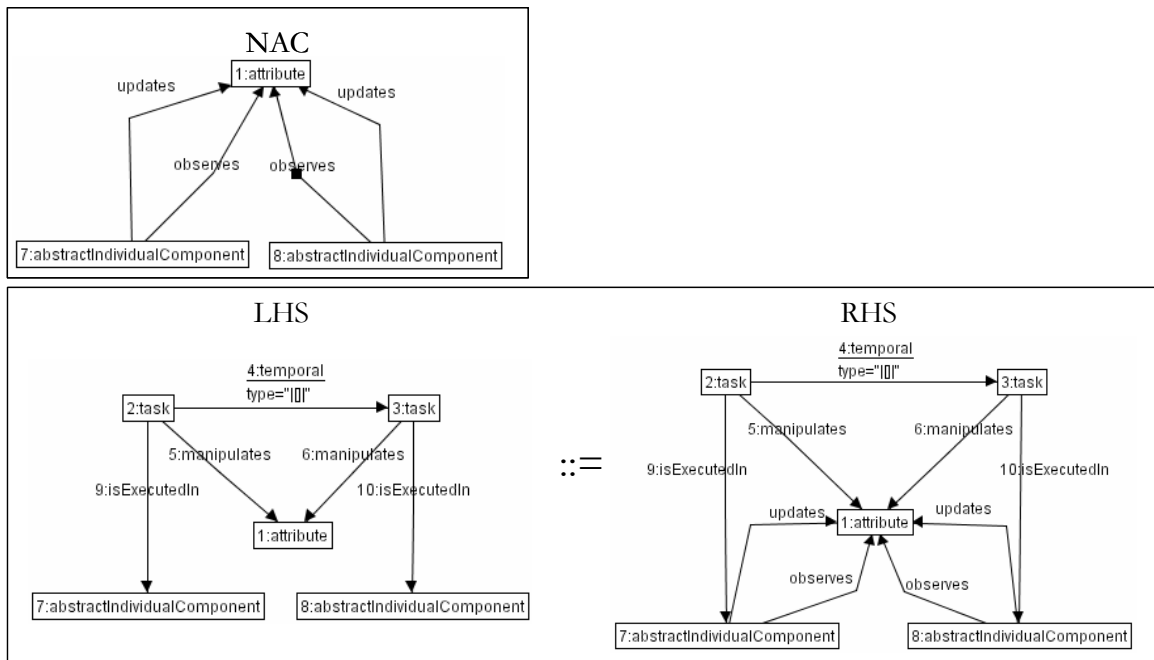
Rule 4-8 Derivation of Updates and Observes structure on the basis of a task relationship of concurrent information passing

4.4.2   Step: From Abstract User Interface to Concrete User Interface

Step T2 (see Fig. 4-1) consists of generating a concrete user interface from an abstract user interface. This development step may involve the development sub-steps illustrated in Fig. 4-20.
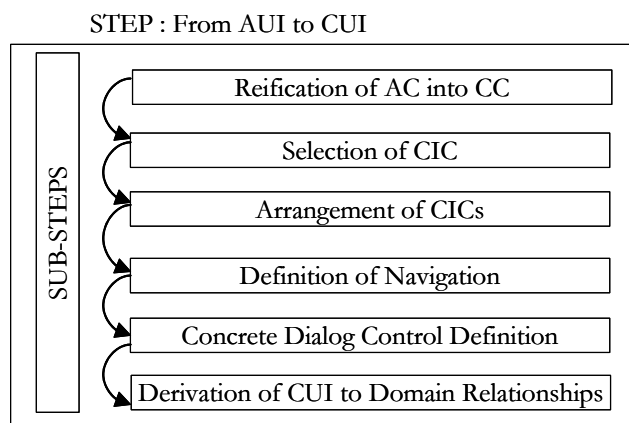
Figure 4-20 Development Step : from AUI to CUI

We decomposed this step into six sub-steps that are arranged by logical order. First, concrete containers (i.e., windows, boxes for the graphical modality) are identified; then the individual elements are derived and arranged into the previously identified containers. This completes the design of the presentation. After that, the dialog is added thanks to three phases: a definition of potential navigation, of the control of this navigation, and of other behaviors. Finally CUI to domain relationships are established by transposing these relationships from the AUI model.

Again, this shows that a decomposition of the current step into sub-steps can follow a logical order that is principle-based.

One can equally imagine a bottom-up process where individual elements are first identified. Then, these objects are assembled together in larger elements, along with their navigation. This approach largely contrasts with approaches where presentation and dialog work hand in hand (e.g., in programming by demonstration). In this type of approach, a combined approach is adopted, but it is very hard to expand the design knowledge used without largely affecting the rest of the design knowledge. This does not satisfy Req. 16: *methodological separation of concern*.

4.4.2.a Sub-step: Reification of abstract containers into concrete containers.

An abstract container can be reified into different types of concrete containers. Variables influencing this transformation are, notably: modality (graphical and auditory are supported by our conceptual framework), context of use (e.g., containers for a cell phone will not be the same that for a wall display), interaction style (e.g., direct manipulation, menu selection, forms, command language, natural language), designer's preference. A major difficulty of this step lies 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 leaf group of AIC in the AUI hierarchy. A maximal solution would be to group all abstract individual components and all abstract containers into one single concrete container (e.g., one window). Several window identification algorithms have been proposed in the literature. [Jans93] uses the concept of view on entity relationship schema to identify windows (the principle being one window per defined view); [Balz93] and

[Puer94] exploit a class diagram structure (i.e., generalization for [Balz93], generalization and aggregation in [Puer94]). [Boda95c] relies on a task centered representation mixed with a specification of task input and output flows (activity chaining graphs) to identify windows and window transitions. [Pate00] and [Luyt03] rely solely on task model structure (hierarchical decomposition and temporal relationships) to identify windows and window transitions. Here again all these approaches propose algorithms tightly coupled with the representation they manipulate. In the following example we externalize a simple heuristic based on Teallach derivation rules [Griff99]. In the original rule, the level "leaf-1" of task tree was assigned to a window. We exploit here the AUI hierarchical decomposition for window identification to apply a similar heuristic. An advantage of our approach is that each proposed algorithm may be modified, tested, and refined.

Example 7 is a transformation system composed of rule 4-9 and rule 4-10. This system transforms into windows, 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 4-9: 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.
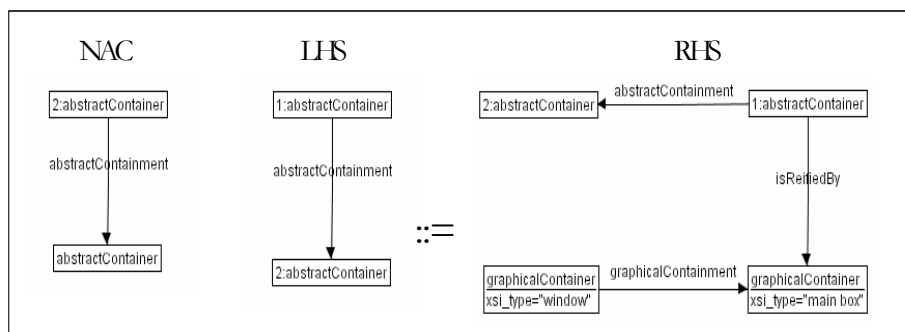
Rule 4-9 A creation of windows derived from containment relationships at the abstract level

Rule 4-10**:** each abstract container contained into an abstract container that was reified into a window is transformed into an horizontal box and embedded into the window.
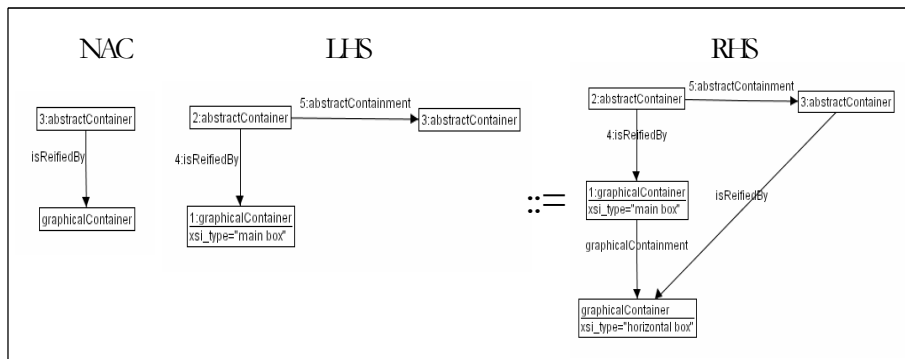
Rule 4-10 A generation of window structure derived from containment relationship at the abstract level

4.4.2.b   Sub-step: Selection of concrete individual components.

Functionalities of abstract individual components 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.

The selection of concrete individual components can be assimilated to the perennial problem of "widget selection" (see Chapter 2). The major difference in our approach is that we introduce an intermediary level between task&domain and concrete widget selection (i.e., the abstract level that synthesizes functionality of interactors). For instance, MacIda [Peto93] proposes an exploitation of the characteristics of domain concepts. Rules like the following may be found in this method: "For each attribute in the domain model generate an input field whose label is the name of the attribute". In [Vand97] the widget selection problem was extensively treated as 238 widget selection rules were provided. Until now it is the most extensive catalog of rules on the topic. Our method is able to cover all of these rules.

Example 8 is composed of Rule 4-11. It creates an editable text component (i.e., a textbox) to reify an AIO with an input facet.

Rule 4-11: 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).
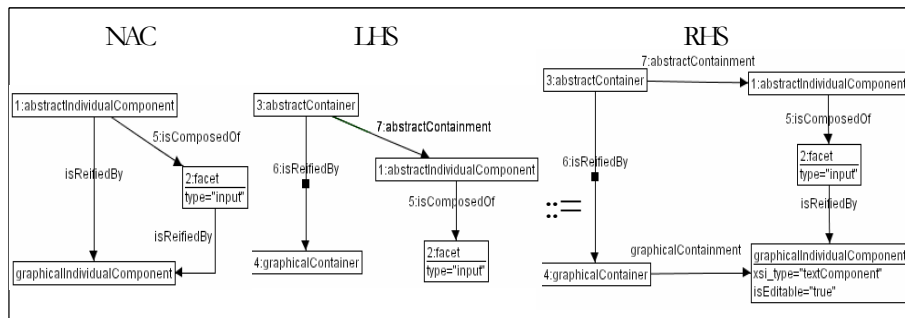
Rule 4-11 Creation of an editable text component (i.e., an input field) derived from facets type of abstract components

4.4.2.c   Sub-step: Arrangement of concrete individual component.

If specified, Allen relationships between abstract interaction objects may be interpreted in order to provide concrete layout information (see Sec. 3.2.3 for more details). It was discussed in Sec. 4.4.1.c that task and domain models provide very poor information on abstract layout. Abstract layout being defined with abstractAdjacency relationship, we rely on a concreteAdjacency (specialized into graphicalAdjacency and auditoryAdjacency) to specify a layout structure at the CUI level. Combined with our boxing system, this relationship allows us to define unambiguous layout specifications. Fig. 4-21 shows a layout that is enabled by these concepts. For this example, the specification would be:

graphicalContainement: MainWindow contains Box1, Box 1 contains Box 11 and Box 12, Box 12 contains Box 121 and Box 122, Box 121 contains menuItem 1, 2, 3, etc.

graphicalAdjacency (excerpt): Box11 adjacent Box11, Box121 adjacent Box122., menuItem1 adjacent menuItem2, menuItem2 adjacent menuItem3,…
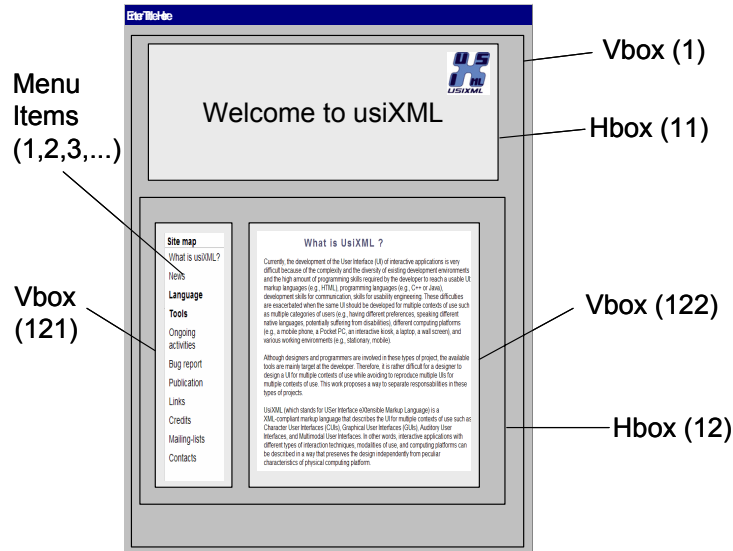
Figure 4-21 Possible layout using concreteAdjacency and a box embedding system.

Example 9 is composed of Rule 4-12. This example transforms an AUI into a concrete model for the graphical modality. It chains concrete individual components according to abstract individual component sequencing.

Rule 4-12: for each couple of abstract individual components related by an "abstractAjacency" relationship and reified into concrete individual components, generate a "concreteAdjacency" relationship between the concrete individual components.
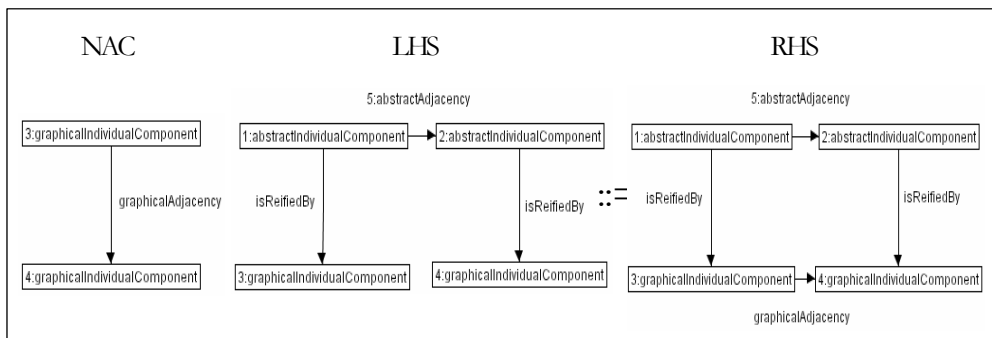
Rule 4-12 A placement of graphical individual components derived from spatio-temporal relationships at the abstract level

4.4.2.d Sub-step: Definition of navigation

Navigation is defined by a set of transitions between containers populating a UI. The reader has probably noticed that no navigation has been defined at the abstract level. Navigation is indeed not an abstract concept as it necessitates a partitioning into concrete containers. Navigation is only a side effect of reification of abstract containers into concrete containers. An all embracing UI on a wall display will require a little navigation in comparison of a cell phone. Ad hoc navigation objects may be created for this purpose (e.g., a menu bar, a tabbed dialog box).

Example 10 is composed of Rule 4-13. It generates a button to enable a navigation between two windows.

Rule 4-13: for each container related to another container belonging to different windows, and their respective abstract container being related by a "is before relationship", generate a navigation button in source container pointing to the window of target container.
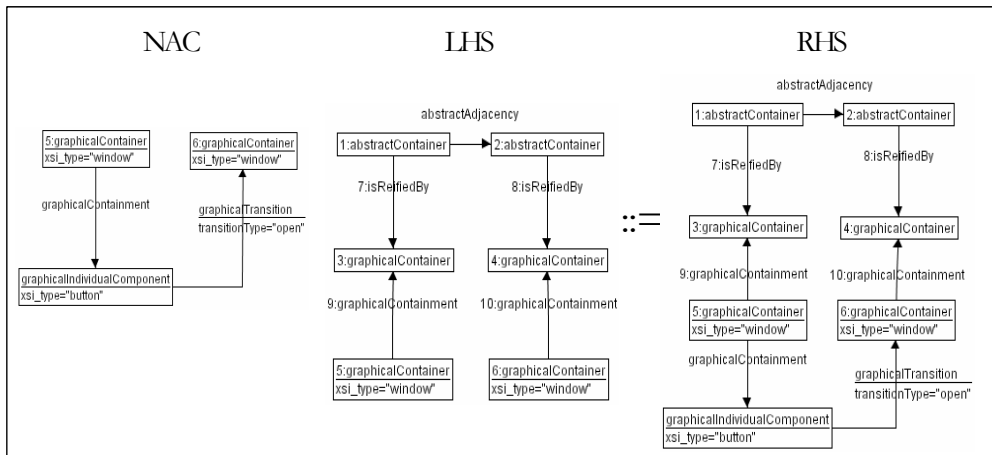
Rule 4-13 A window navigation definition derived from container adjacency relationships

4.4.2.e Sub-step: Concrete Dialog Control Definition

This sub-step consists of a simple transposition of abstract dialog relationships in the concrete world.

Example 11 is composed of Rule 4-14 transposes a dialog control relationship between two containers.

Rule 4-14: for each couple of abstract container with a dialog control relationship, transpose this relationship to the couple of concrete containers that reify them.
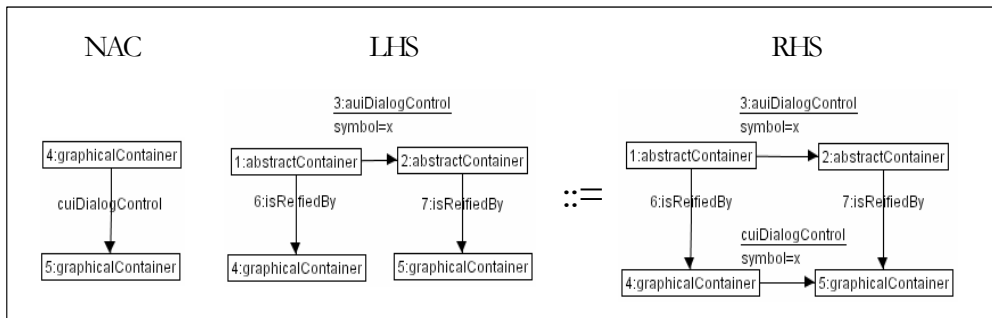
Rule 4-14 Derivation of the concrete dialog from abstract dialog

4.4.2.f   Sub-step: Derivation of CUI to domain relationships

This step consists of a transposition of "AUI to domain relationships" to the concrete level. A simple transitivity property between a domain concept, an abstract concept and a concrete concept is assumed.

Example 12 is composed of rule Rule 4-15. It transposes an "updates" relationship from an AIC to the CIC that reifies it.

Rule 4-15: for each AIC updating a domain concept, if a CIC reifies this AIC then the CIC updates this same domain concept.
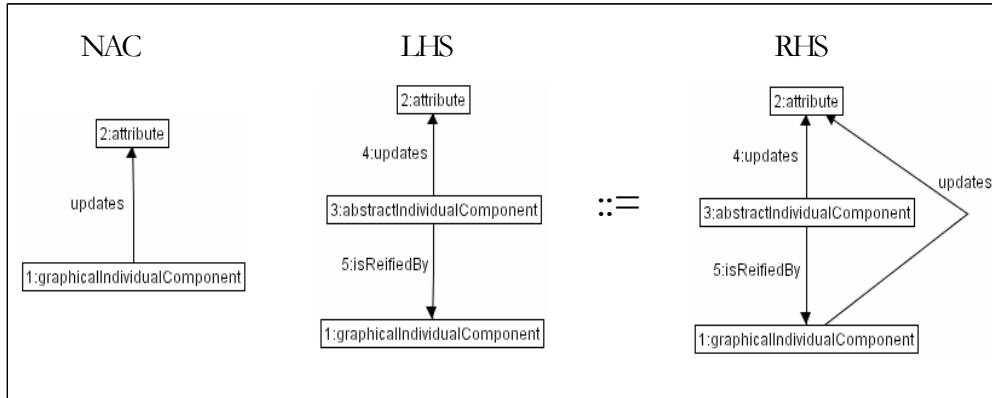
Rule 4-15 Transposition of update relationship

4.4.3 From Concrete User Interface to Code

Step T3 consists in code generation from a CUI. Code generation techniques for UIs is a very well known topic. [Czar00] presents a state-of-the art of model to code techniques (e.g., visitor-based approach and template based approach). Scientific results for this transformation have been shown in systems issued from research like: Janus [Balze95], Trident [Boda95b], Modi-D [Puer97] or from commercial world e.g., Genova [Geno04] or Oliva Nova [Moli02]. The present work does not particularly contribute to this area although several tools have been developed to provide code generation support from the concrete user interface level.

4.5 Reverse Engineering

As shown in Fig. 4-22, the starting point of UI reverse engineering is the user interface code. This code is analyzed and transformed into a higher level representation i.e., a concrete user interface. From this CUI model, an AUI and, finally, a task and domain model are retrieved.
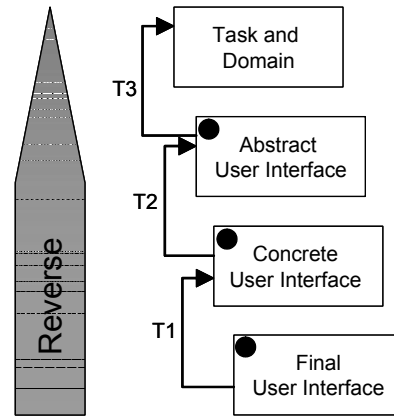
Figure 4-22 Reverse Transformational Development of UIs

Step T1 (see Fig. 4-22) consists of retrieving a concrete UI model from UI code or appearance. A state of the art in reverse engineering of UIs expressed according to the IEEE Terminology [Chik90] can be found in [Bouil04].

Transition T1 is notably supported by ReversiXML (formerly called Rutabaga [Boui04]). ReversiXML is an on-line tool functioning as a module of an Apache server. It takes as input a static HTML page, a configuration file containing a set of user defined options, and produces a UI at concrete and/or abstract level. The target language that is used by this tool is UsiXML (see Sec. 3.3.2). T1 step is a tedious operation since it may require as many "concretizers" as existing languages. Of course, if one restricts to, let us say markup-languages, some reuse can be considered. For other families of language (e.g., Lisp, Prolog, Python, Caml, C++, C#), a separate concretizer may be needed each time. This represents a serious development effort.

Transition T2 (see Fig. 4-22) consists of deriving an abstract UI specification from a concrete one. This derivation is relatively trivial because (1) the source model holds more information than the target model (2) there is a very smooth conceptual continuum between these two levels. Nevertheless, several development sub-steps may be identified: abstraction of CIO into AIO, abstraction of layout relationships, abstraction of navigation, abstraction of dialog, etc. We provide hereafter an example of such sub-step. In addition this transition is illustrated by our case study in Sec. 5.2.5.

Example 13 is composed of Rule 4-16. It consists of obtaining an abstract individual component equipped with an input facet.

Rule 4-16: for each editable graphical individual component, create an abstract individual component equipped with an input facet.
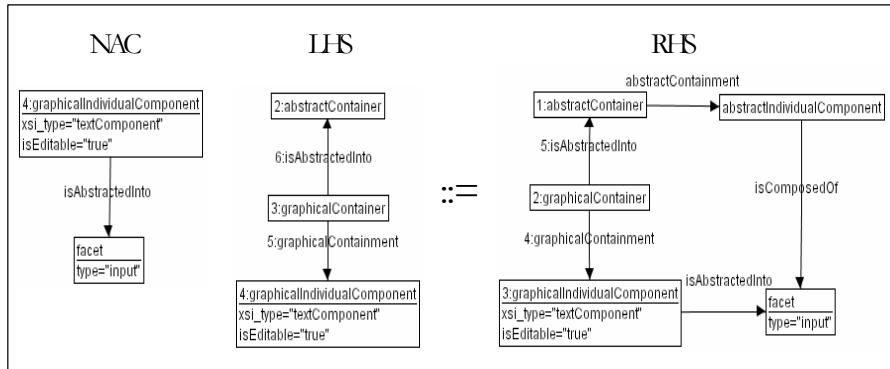
Rule 4-16 Creation of a facet at the abstract level derived from a type analysis of graphical individual components

Transition T3 (see Fig. 4-22) is the derivation of a task and concept specification from an abstract UI. A conceptual gap between AUI level and task and domain level being large, little information can be extracted from an AUI model to retrieve a task or domain specification.

Example 14 is composed of Rule 4-17. This example derives information on task action type from the abstract user interface level.

Rule 4-17: for each abstract individual component equipped with a navigation facet create a task with action type "start/go" on an item of type "element".
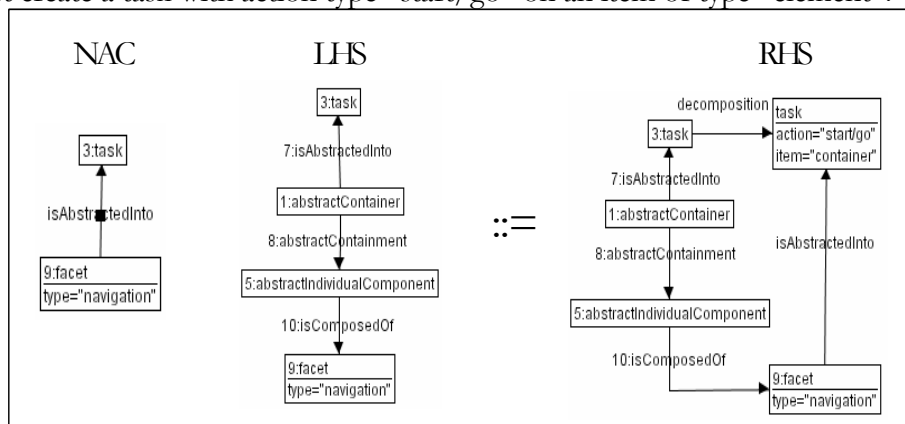
Rule 4-17 Definition of task action types derived from an analysis of facets at the abstract level

4.6 Adaptation to context change
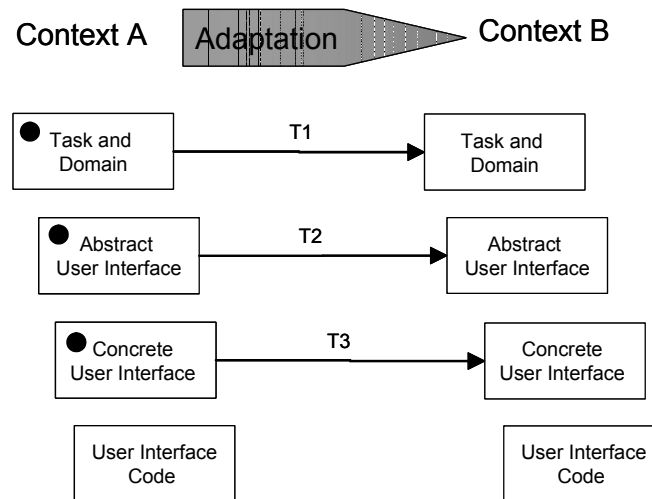
Figure 4-23 Context adaptation at different levels our framework

Context adaptation (illustrated in Fig. 4-23) covers model transformations adapting a viewpoint to another context of use. This adaptation may be done by the application of translations to models belonging to any viewpoint.

4.6.1   Step: From Task & Domain to Task & Domain

We propose one development sub-step type to exemplify adaptation at T1 level (see Fig. 4-23): transformation of a task model.

4.6.1.a   Sub-step: Transformation of a task model

Transformation of a task model can be useful to adapt a task specification to various categories of users, to various 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). A task model may also be 'filtered' according to various criteria (e.g., erase all tasks manipulating a video stream). Transformation rules may be defined to adapt a task specification to these constraints.

Example 15 is a transformation system composed of Rule 4-18 and Rule 4-19. A task hierarchy is "flattened" to allow an (expert) user to perform all tasks at the same time (i.e., concurrently).

Rule 4-18: (1) erases each intermediary task (i.e., non-leaf and non-root tasks). (2) attaches every leaf task to the root.
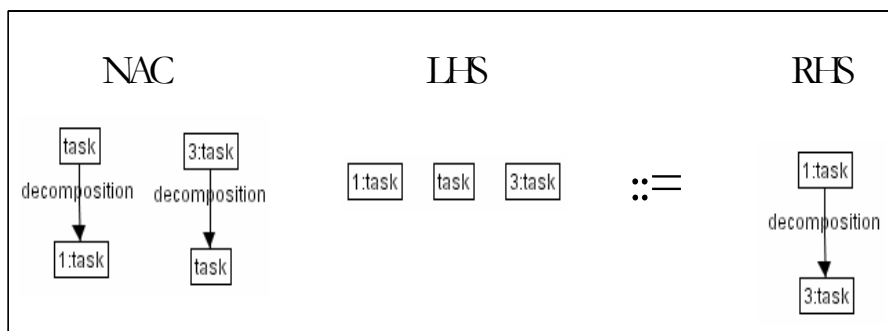
Rule 4-18 Flattening of a task tree structure

Rule 4-19**:** for each sister tasks change their temporal relationship into concurrent.
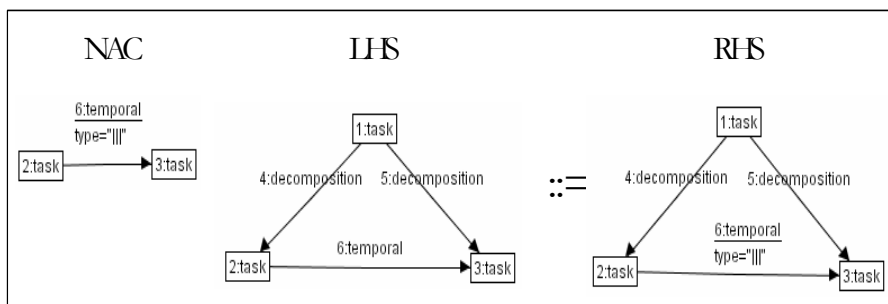
Rule 4-19 Transforming all temporal relationship to concurrent

4.6.2 Step: From Abstract User Interface to Abstract User Interface

Adaptation at the abstract level concerns abstract container reshuffling and abstract individual component modification (e.g., facet modification, facet splitting, facet merging). An example of abstract individual component modification.

4.6.2.a   Sub-step: 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 16 is a transformation system containing Rule 4-20 and Rule 4-21. 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.

Rule 4-20: for each pair of abstract individual component mapped onto concurrent tasks, transfer all facets of the abstract individual component that is mapped onto the task target of the concurrency relationship, to the other abstract individual component.
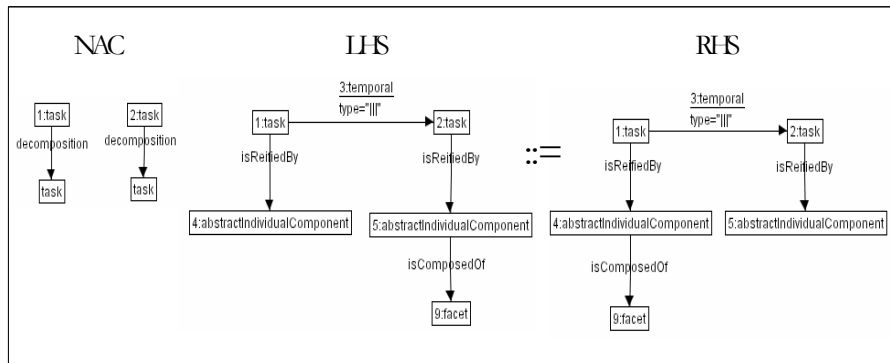
Rule 4-20 A merging of facets of abstract individual components

Rule 4-21: erase all abstract individual components that have no facets left.
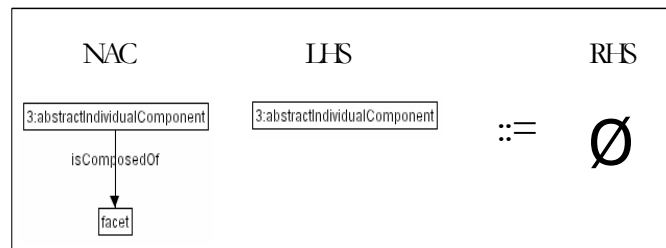
Rule 4-21 Erasing abstract individual components with no facets left

4.6.3   Step: From Concrete User Interface to Concrete User Interface

Adaptation at the concrete level is illustrated by several development sub-steps: container type modification (called concrete container re-formation), modification of the types of concrete individual components (called concrete individual components re-selection), layout modification (layout re-shuffling), or navigation re-definition. Examples for these first three adaptation types are given hereafter.

4.6.3.a   Sub-step: Concrete container re-formation

Concrete container Re-Formation may cover situations like container type transformation (e.g., a window is transformed into a tabbed dialog box), container system modification (e.g., a system of windows is merged into a single window).

Example 17 is a transformation system composed of Rule 4-22, Rule 4-23, Rule 4-25. This transformation adapts a window into a tabbed dialog box and transfer the window content into several "tabbed items".

Rule 4-22: each window is selected and mapped onto a newly created tabbed dialog box.
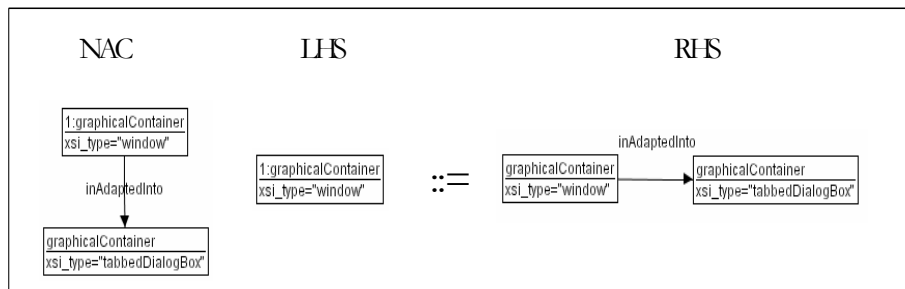
Rule 4-22 Initializing of the adaptation process by creating graphical component to adapt into

Rule 4-23: transfers every first level box of the window to adapt it into a tabbed item composing a tabbed dialog box.
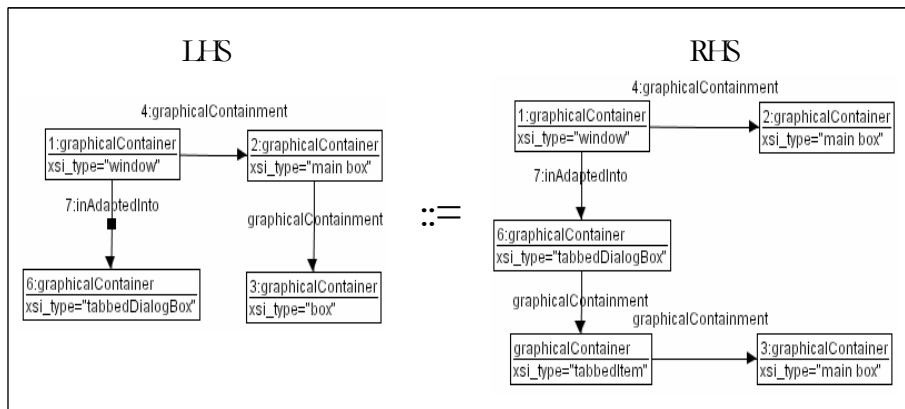
Rule 4-23 Creation of a tabbed item and transfer of the content of the adapted window

Rule 4-25**:** cleans up the specification of remaining empty main boxes.
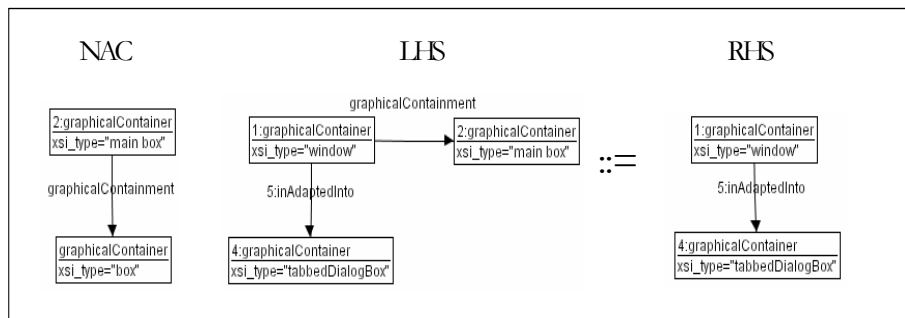
Rule 4-24 Deletion of unnecessary containers

4.6.3.b Sub-step: Concrete individual component re-selection

Re-selection transformations adapt individual component into other individual components. This covers individual component merging or slitting, or replacement.

Example 18 is composed of Rule 4-25. 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.

Rule 4-25: 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 a content has already been transferred).
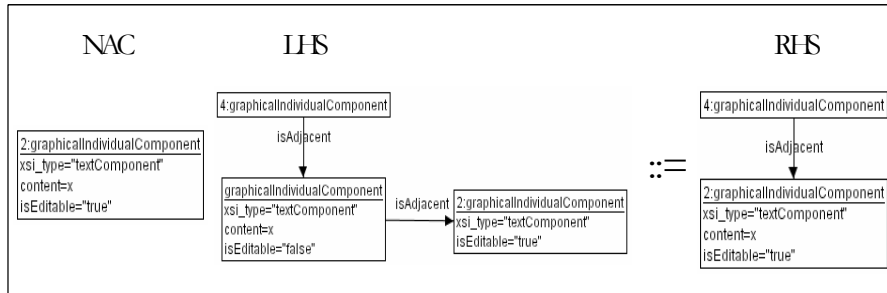
Rule 4-25 Merging of a non-editable text component (e.g., a label) and an editable text component (e.g., an input field) into one single editable text component

4.6.3.c   Sub-step: Layout re-shuffling

A layout at the concrete level is specified with horizontal and vertical boxes. An elements contained into a box may be glued to an edge of this box. Any transformation modifying this structuring is categorized as layout reshuffling transformation.

Example 19 is composed of Rule 4-26. It squeezes all boxes in order to "verticalize" its layout.

Rule 4-26: each box is transformed into a vertical box and every individual component is glued to left.
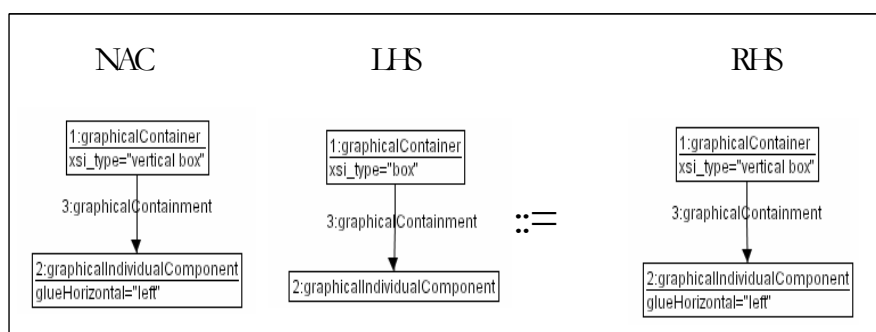
Rule 4-26 Squeezing of a layout structure to display vertically

4.7 Tool Support

An identification of tools required to the development of UIs adopting a model-driven approach has been discussed in [Szek96, Schl96, Puert97]:

- **Model editors** assist a designer in constructing the models. These tools consist in syntax editors, form based tools, or visual builders. Some model editors maintain a textual specification consistent with a graphical representation.
- **Design critics** provide a designer with quality assessment facilities. Models capturing explicit properties of the artifact are an ideal representation to perform evaluation.
- **Design assistants** help a designer in refining modeling artifacts. These tools propose knowledge bases represented as rules (most of them are production rules).
- **Implementation tools** translate a specification into a representation that can be used by a compiler, an interpreter or an interface builder.

We add to this list:
- **Transformation tools** provide support to the designer to edit, store and execute model transformation rules.
- **Reverse engineering tools** extract a modeling artifact from a coded representation.

Several tools have been exploited or developed in the context of this dissertation. They all play a certain role in making multi-path development a reality. We have classified these tools according to the classification framework presented above. More details on tools is provided in Annex 1.

Tool	Functional Coverage	Credits
GrafiXML [Usix04]	Graphical model editor: CUI (high fidelity), context model + Textual model editor: all UsiXML models + Code generation: XHTML 1.0, Java Swing	In collaboration with Benjamin Michotte
VisiXML [Usix04]	Graphical model editor: CUI (mid fidelity)	In collaboration with Manuel Van Sluys
FlashiXML	Flash Renderer	In collaboration

		with Youri Vanden Berghe
IdealXML[Usix04]	Model editor: Task & Domain, AUI, inter-model relationships	In collaboration with Francisco Montero
ReversiXML[Boui03]	Reverse engineering: from HTML 4.0 to CUI and/or AUI	Laurent Bouillon
TransformiXML API/GUI [Limb04]	Model transformation: from any UsiXML model to any UsiXML model	In collaboration with Victor Jaquero and Benjamin Michotte
AGG Transformation	General purpose tool for graph transformation	Olga Runge, TU Berlin

Table 4-2 Tools to support our approach

4.8 Conclusion

In this chapter, a multi-path development method based on graph transformation has been introduced, defined and illustrated.

This development method decomposes any development activity in a succession of development steps that consist of the transformation of the artifact(s) in the scope of a development stage (here referred as *viewpoint*) into other development artifacts. In this context, a development path is defined as an archetypal composition of development steps. We identified three typical development paths: forward engineering, reverse engineering, and context (of use) adaptation. 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 (Req. 13: *Methodological flexibility*).

To address the requirement of methodological separation of concern (Req. 16), development steps, have been further decomposed into development sub-steps. A development sub-step realizes one 'concern' of the transformation process e.g., definition of the dialog control, definition of the navigation, choice of interactors type.

To enable the expression (Req. 12: *Methodological explicitness*) and the execution (Req. 15: *Executability*) of the development steps. Each sub-step populating a step may be associated with a so-called transformation system, itself decomposed into transformation rules. Transformation systems and transformation rules are conditional graph rewriting rules sequentially composed into grammars. As so, transformations can be uniformly and consistently applied through all possible development paths (Req. 18: *Methodological homogeneity*). This application is based on a rigorous execution semantics provided by the graph transformation literature (Req. 14: *Methodological formality,* Req. 13: *Executability,* Req. 17: *Predicatbility*). Transformation rules and transformation systems may be stored in a textual format to enable their capitalization in a sort of development library (Req. 23: *Methodological Reuse,* Req. 22: *Tool Interoperability*).

Transformation systems and transformation sub-steps proposed in this chapter are only one possibility of realizing different development paths. Our methodology allows the introduction of new development sub-steps and/or new transformation systems for realizing sub-steps (Req. 17: *methodological extendibility*).

Allowing a designer to define her own transformation heuristics raises the problem of the correctness (Req. 21) of our method. Correctness is a relative notion depending on the context in which it is addressed. Two types of correctness may be considered: Syntactic (structural) correctness and semantic correctness [Varr02b]. Syntactic correctness stipulates that for any well-formed source model, any transformation rule produces a well-formed target model. Syntactic correctness is guaranteed by construction within our framework by the fact that all our transformations are type preserving. 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. A graph of types can also be complemented with the expression of specific consistency constraints inexpressible within the graph of types. Object Constraint Language (OCL) is used for this purpose in [Agra03], pre- and post-condition with graph patterns are used in [Akeh03]. This approach is compatible with ours. Semantic correctness stipulates a semantic adequacy between a source and a target model. In our context, semantic correctness proving is very hard to consider as by definition the domain of discourse of source model and target model are different. Furthermore, a designer being allowed to define her own transformation rules, a correctness proof would have to be instantiated for each newly defined rule.

Traceability (Req. 20) has been defined in Chapter 2 as the "degree to which a relationship can be established between two or more products (i.e., here models) of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another" [IEEE90]. A set of inter-model relationships have been introduced in Chapter 3 to enable the expressing of relationships of elements across viewpoints. In sec 4-4, 4-5, 4-6, we have seen that these relationships ensured traceability of the application of transformations i.e., it is possible to say, using these relationships, which model element is derived from another one. Although our solution meets the desired requirement, it could be regretted that these relationships have to be part of the expression of the rule it self. It could indeed be imagined to produce these traceability relationships automatically.

A collection of tools has been outlined in Sec. 4-7. These tools materialize our approach and show how each viewpoint of our framework can be edited (Req. 12 *Methodological flexibility*) and transformed. The existence of this collection of tools contributes to the requirement of tool interoperability (Req. 22: *Support for tool interoperability*).

Practically, the mere fact of decomposing a transformational development process into steps and sub-steps enables an identification of weaknesses of certain models in terms of expressivity. As the accuracy in the expression of transformation grew, some models revealed to need enrichment to allow their exploitation for derivation means e.g., the task model had to be enriched with various concepts for instance to describe the intrinsic nature of a task, the domain model needed a better expression on the nature of the domain of attributes. Some other concepts needed a more precise interpretation to exploit them in a transformation process e.g., the information passing.

On the other hand, some elements of a UI specification were shown very difficult to derive. A notable case is layout derivation. Two solutions can be considered to overcome this problem. A first one consists of adopting a lot of hypotheses on the UI structure. We used for instance in Sec. 4.4.1.c a simple heuristic to sequentialize widgets derived from the relationships between tasks they supported. This method, however fully automatic, gave arguable results in terms of both usability and aesthetic. A second solution would be to allow a designer to visually reshuffle models in a graphical editor. This method is very seducing but has the major drawback of endangering the consistency of carefully built, or derived, models. Two solutions may be combined to solve this problem: limit the designer's activity to tasks having no consequence on the model consistency, design specific algorithms (using probably graph grammars) to re-enforce consistency between models.

While layout generation proved to be very hard to realize automatically, other aspects of UI construction might also need the intervention of a designer in the refinement. Transformation driven development of UI have suffered from a lack of flexibility in their methodological stances. This does not refer only to the multiple entry point and exit point mentioned in the shortcomings of Chapter 2 but also to the little possibilities that are proposed to a designer to edit a model at any stage in the transformation process and enrich it manually. This is the underlying idea of the tool collection presented in Sec. 4.7. We believe that this essential feature if achieved while keeping a good level of consistency could benefit substantially to the acceptance of transformational approaches.

Chapter 5   Case Studies

5.1 Introduction

This chapter applies multi-path development of user interface to two different case studies. The two cases are progressive in terms of complexity. Their presentation relies on a series of illustrations showing how artifacts are progressively transformed according to various development sub-steps, steps, and paths.

The process adopted to develop the case studies of this chapter consists of: (1) Building initial models. Such models have been edited with their associated editing tool. For instance, IdealXML [Mont04] has been used to edit the task and domain model. (2) Editing and debugging of rules within the AGG graphical environment. Most of the rules have been elicited prior to realizing these case studies by a theoretical analysis of development sub-steps as illustrated in Chapter 4. (3) Importing initial models into the AGG graphical environment. (4) Selecting a transformation set and firing the rules contained in this set. (5) Exporting resulting models from AGG to UsiXML and illustration.

To facilitate the understanding and the continuous reading of the case studies, only a significant portion of transformation rules is provided for most sub-steps. The remaining rules can be defined by analogy to rules previously defined in the same set.

The first case study is devoted to the development of an opinion polling system, a reasonable scaled example of a typical information system. The development scenario is the following: a forward engineering path is applied from a definition of the task and domain viewpoint to produce both an AUI and a CUI. The CUI is

reshuffled by hand in our GrafiXML graphical editor. As these modifications are important and may endanger the consistency with the AUI, a reverse engineering path is applied to this modified CUI in order to recover a consistent AUI.

The second case study is devoted to the development of a virtual travel agent inspired from the FIPA [FIPA00] example since it is considered as a benchmarking case in information systems. The development scenario differs from the first case study: from a task and domain viewpoint, an AUI is derived. From this AUI, three different CUIs are forward engineered. Two CUIs specify two 2D graphical user interfaces. The first one is targeted to a context for a desktop computing platform allowing the realization of all tasks into one single window. The second CUI is targeted to a small display context that is typical for a Personal Digital Assistant (PDA). Elements are distributed as a set of small screens with a navigation scheme between. The third CUI is devoted to a derivation of an auditory interface. After this, the initial task model is pruned according to a heuristics that selects only important tasks so as to produce a new AUI and a new CUI.

5.2 Case Study 1: a Virtual Polling System

This case study applies a transformational approach in order to develop a UI aiming at collecting opinions of users. The scenario proposed for this case study is (Fig. 5-1): from initial Task & Domain models, an AUI is produced (T1) from which a CUI is derived (T2). After this, the CUI is manually reshuffled in a CUI graphical editor and reversed engineered to another AUI to address the round-trip problem (T3). The CUI can be rendered thanks to any UsiXML-compliant rendering engine.
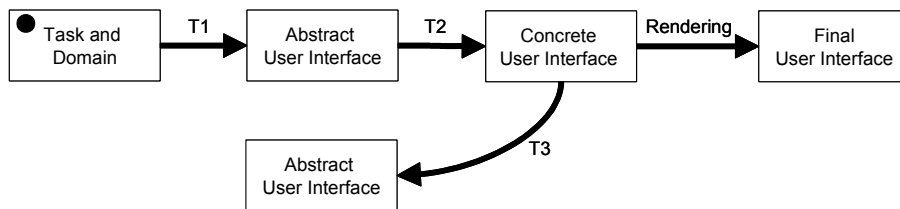
Figure 5-1 Development scenario for case study 1

5.2.1 Initial Representation

Fig. 5-2 illustrates the domain model of our UI as produced by a software engineer. A participant participates to a questionnaire. A questionnaire is made of several questions. A question is attached to a series of answers.
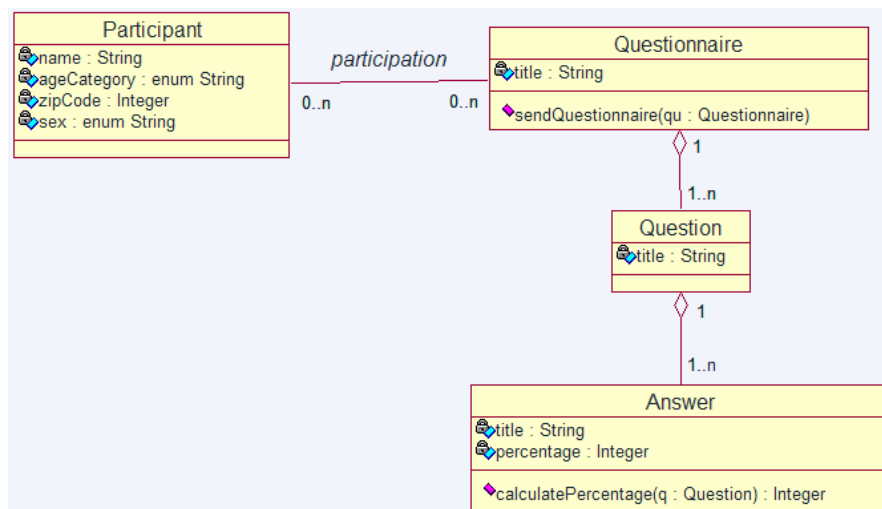
Figure 5-2 Class diagram for an opinion poll system

Fig 5-3. proposes the UsiXML specifications corresponding to the domain model. Lines 36 to 65 define the four classes of our diagram, the remaining elements defining the relationships between the domain classes. Lines 38 to 42 show the definition of an attribute with an enumerated domain, which is difficult with UML. Lines 51 to 53 show the definition of a method with its parameters.

```
35    <domainModel id="domainModelCS2" name="domainModel">
36      <domainClass id="DC1" name="Participant">
37        <attribute id="A1DC1" name="name" attributeDataType="String" attributeCardMin="1" attributeCardMax="1"/>
38        <attribute id="A2DC1" name="ageCategory" attributeDataType="String" attributeCardMin="1" attributeCardMax="1">
39          <enumeratedValue name="18-35"/>
40          <enumeratedValue name="35-45"/>
41          <enumeratedValue name="45+"/>
42        </attribute>
43        <attribute id="A3DC1" name="zipCode" attributeDataType="Integer" attributeCardMin="1" attributeCardMax="1"/>
44        <attribute id="A4DC1" name="sex" attributeDataType="String" attributeCardMin="1" attributeCardMax="1">
45          <enumeratedValue name="Male"/>
46          <enumeratedValue name="Female"/>
47        </attribute>
48      </domainClass>
49      <domainClass id="DC2" name="Questionnaire">
50        <attribute id="A1DC2" name="title" attributeDataType="String" attributeCardMin="1" attributeCardMax="1"/>
51        <method id="M1DC2" name="sendQuestionnaire">
52          <param id="P1M1DC2" name="qu" dataType="Questionnaire" paramType="input"/>
53        </method>
54      </domainClass>
55      <domainClass id="DC3" name="Question">
56        <attribute id="A1DC3" name="title" attributeDataType="Name" attributeCardMin="1" attributeCardMax="1"/>
57      </domainClass>
58      <domainClass id="DC4" name="Answer">
59        <attribute id="A1DC4" name="title" attributeDataType="String" attributeCardMin="1" attributeCardMax="1"/>
60        <attribute id="A2DC4" name="percentage" attributeDataType="Integer" attributeCardMin="0" attributeCardMax="1"/>
61        <method id="M1DC4" name="calculatePercentage">
62          <param id="P1M1DC4" name="q" dataType="Question" paramType="input"/>
63          <param id="P2M1DC4" dataType="Integer" paramType="output"/>
64        </method>
65      </domainClass>
66      <adHoc id="DA1" name="participation" roleACardMin="0" roleBCardMin="0" roleACardMax="n" roleBCardMax="n">
67        <source sourceId="DC1"/>
68        <target targetId="DC2"/>
69      </adHoc>
70      <aggregation id="DA2" roleACardMin="1" roleBCardMin="1" roleACardMax="n" roleBCardMax="1">
71        <source sourceId="DC2"/>
72        <target targetId="DC3"/>
73      </aggregation>
74      <aggregation id="DA3" roleACardMin="1" roleBCardMin="1" roleACardMax="n" roleBCardMax="1">
75        <source sourceId="DC3"/>
76        <target targetId="DC4"/>
77      </aggregation>
78    </domainModel>
```

Figure 5-3 Domain Model in UsiXML

Fig. 5-4 depicts a CTT representation of the task model envisioned for the future system. The root task consists of participating to an opinion poll. In order to do this, the user has to provide the system with personal data. After that, the user iteratively answers some questions. Answering a question is composed of a system task showing the title of the question and of an interactive task consisting in

selecting one answer among several proposed ones. Once the questions are answered, the questionnaire is sent back to its initiator. All temporal relationships are **enabling** which means that the source task has to terminate before the target task can be initiated.
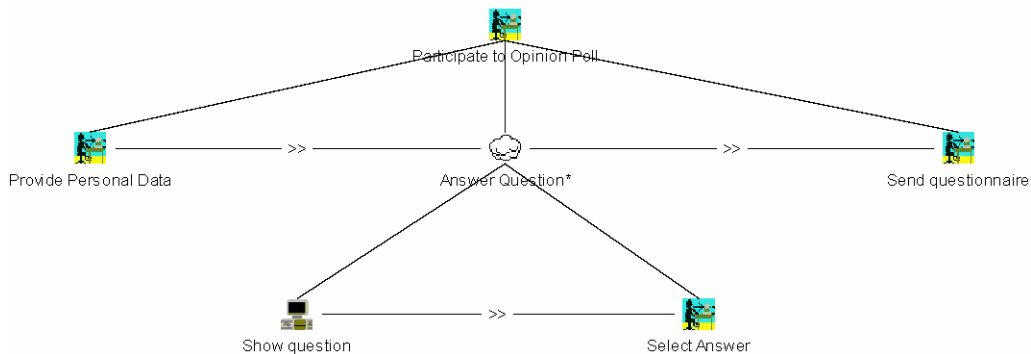
Figure 5-4 Task Model for an Opinion Poll System

Fig. 5-5 presents the UsiXML specifications corresponding to the task model. Lines 8 to 15 define tasks and their hierarchical structure, while lines 16 to 31 define the temporal relationships between these tasks.

Figure 5-5 Task model expressed in UsiXML

Fig. 5-6 depicts manipulates relationships between the task and the domain model as dashed arrows. Provide Personal Data is mapped onto Participant class. Show Question is mapped onto the attribute title of class Question. The task Select Answer is mapped onto the attribute title of the class Answer. And finally, the task Send Questionnaire is mapped onto the method sendQuestionnaire of the class Questionnaire.
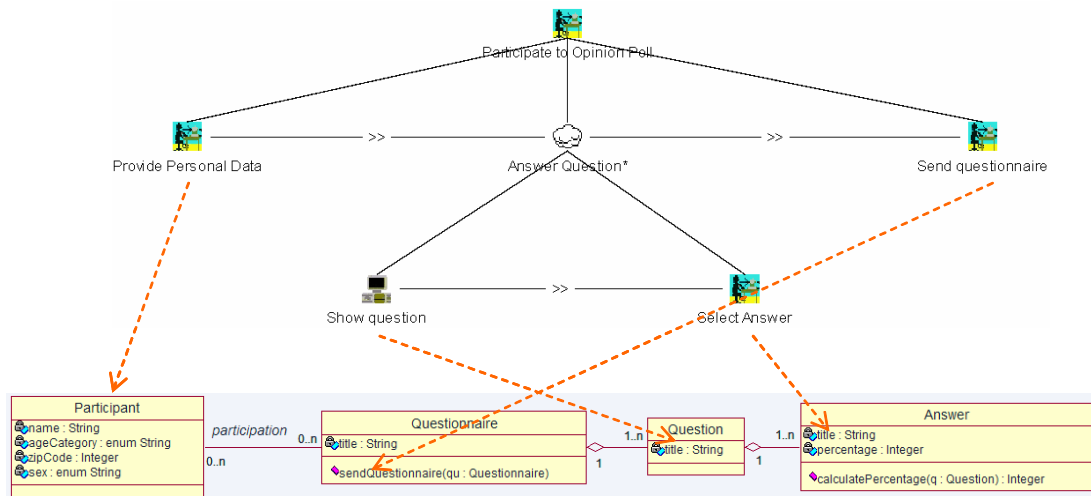
Figure 5-6 Mappings between a task model and a domain model

The UsiXML specifications of the relationships between the task model and the domain model are reproduced in Fig. 5-7, these relationships are made on the base of the id attribute of mapped elements.


```
80    <mappingModel id="MappingDomainCS2" name="mappingDomain">
81      <manipulates id="MA1">
82        <source sourceId="T1"/>
83        <target targetId="DC1"/>
84      </manipulates>
85      <manipulates id="MA2">
86        <source sourceId="T22"/>
87        <target targetId="A1DC3"/>
88      </manipulates>
89      <manipulates id="MA3">
90        <source sourceId="T23"/>
91        <target targetId="A1DC4"/>
92      </manipulates>
93      <manipulates id="MA4">
94        <source sourceId="T3"/>
95        <target targetId="M1DC2"/>
96      </manipulates>
97    </mappingModel>
```

Figure 5-7 Task to Domain Mapping in UsiXML

Unfortunately, the initial task representation is not precise enough to perform transformations. Indeed the task Provide Personal Data is an interactive task consisting in creating instances of Participant. In reality this task will consist in providing a value for each attribute of Participant. This could mean that the task model is not detailed up to the required level of decomposition. Therefore, rule 5-1 is applied to the task model and decomposes Provide Personal Data into four new sub-tasks, each of them manipulating an attribute of Participant. These new sub-tasks have the same type as their mother task. Note the way they are named using a post-condition on their name attribute.
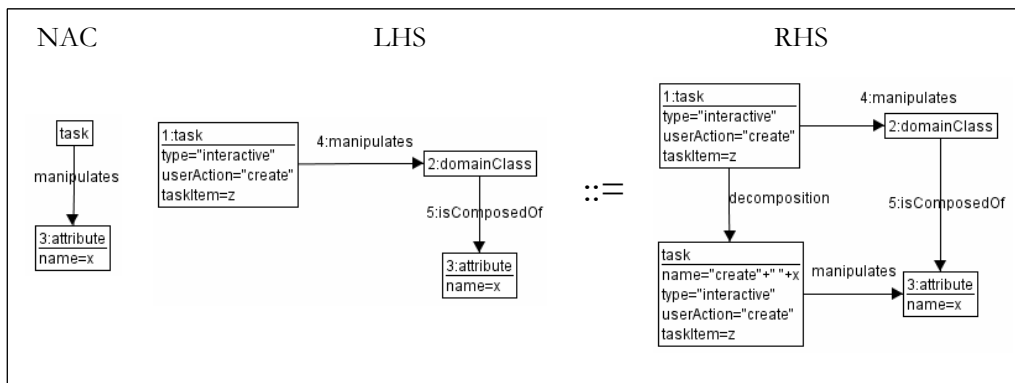
Rule 5-1 Consolidation of the task model

Consequently to the execution of this rule, four new tasks are created: create name, create zipCode, create ageCategory, create sex. "Create" is a very general action type. In the case of ageCategory and sex, create can be specialized into select because ageCategory and sex both hold an enumerated domain. This could be done by hand or performed automatically with the Rule 5-2. Rule 5-3 provides a default temporal relationship (set to enabling) when two sister tasks have no temporal relationships. The resulting task specification is provided in Fig. 5-8.
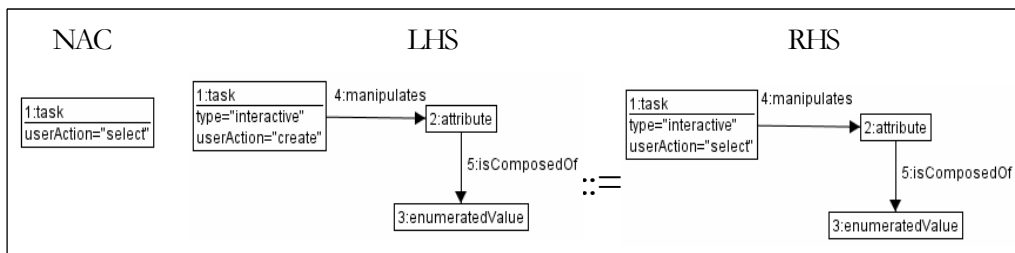
Rule 5-2 Specializing user action
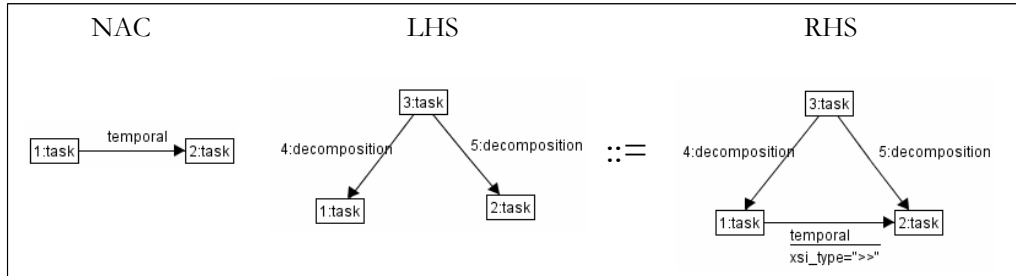
Rule 5-3 Enabling as default temporal relationships between two sister tasks
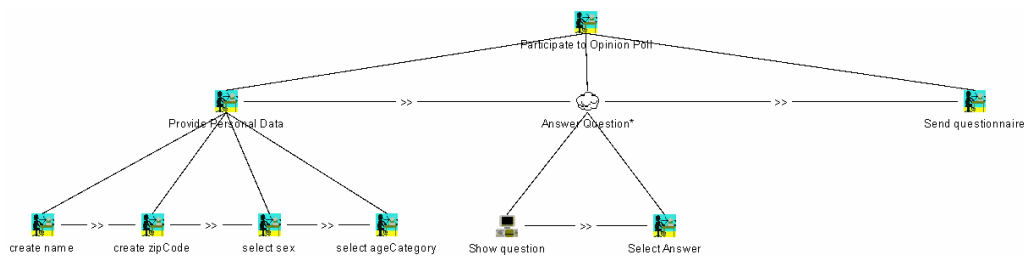
Figure 5-8 Refined Task Model for a Virtual Polling System

5.2.2 Transformation to an Abstract User Interface

From these initial representations, a transformation process is initiated to obtain an abstract user interface model.

5.2.2.a Identification of abstract UI structure

The identification of AUI structure is ensured by applying Rule 5-4, Rule 5-5, Rule 5-6, Rule 5-7, and Rule 5-8. These rules essentially recreate the task model structure by a hierarchical decomposition of abstract containers and abstract individual components.
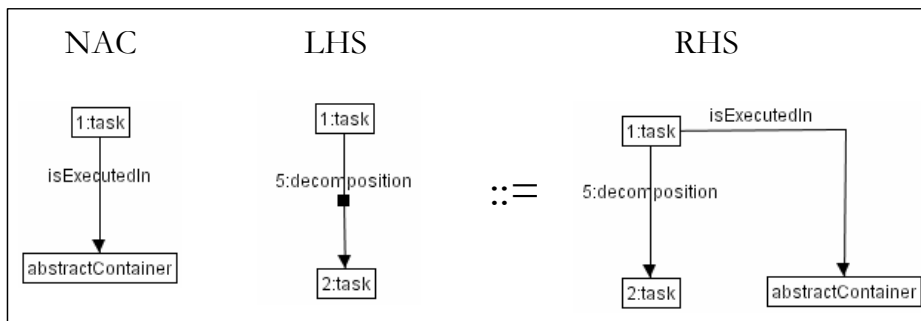
Rule 5-4 Create an AC for task that has task children
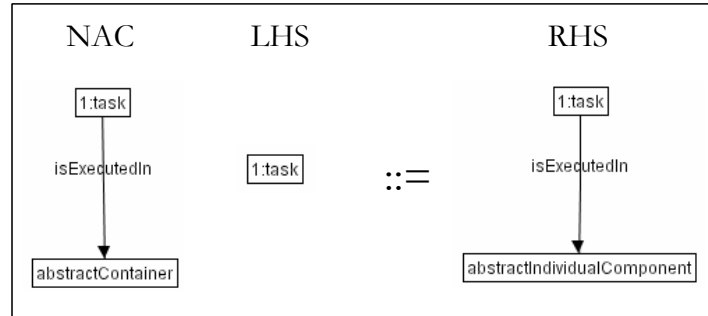
Rule 5-5 Create an AIC for leaf tasks
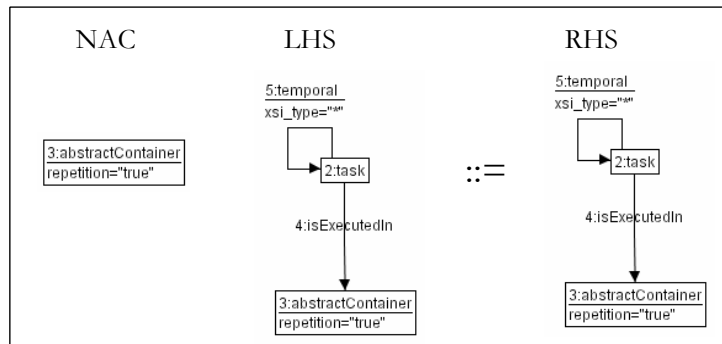
Rule 5-6 Iterative tasks are mapped onto repetitive AC
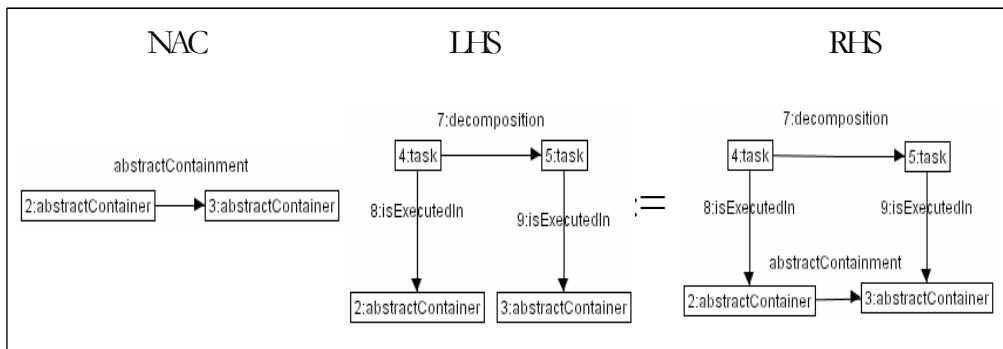
Rule 5-7 Reconstruct containment relationships between AC
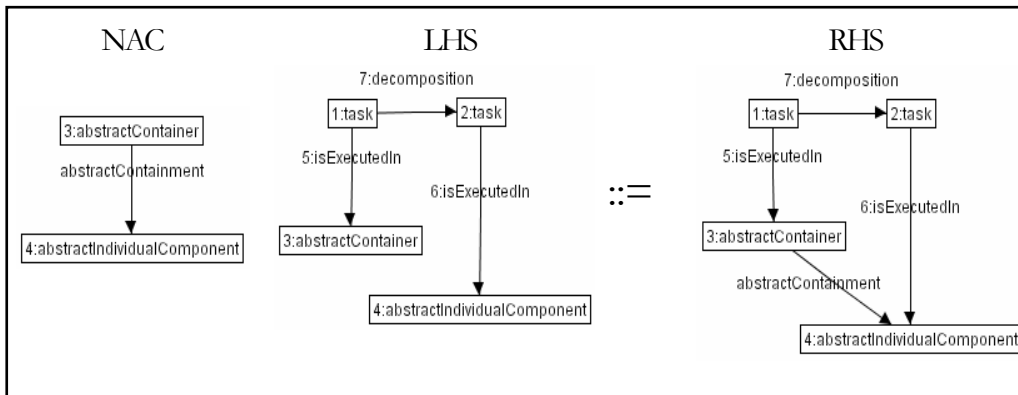
Rule 5-8 Reconstruct containment relationships between AIC

Fig. 5-9 shows how this sub-step is executed: dashed squares depict tasks grouping in abstract containers, dashed circles depict abstract individual components. As expressed in Sec. 4-4, more complex heuristics can be deployed to perform this transformation. Note that the designer can also want to define abstract containers by hand.
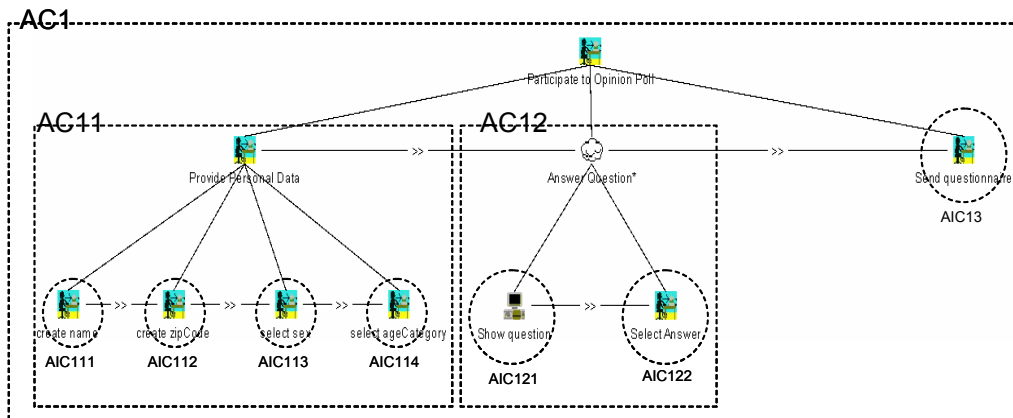
Figure 5-9 Mapping between a task model and an abstract UI

5.2.2.b   Selection of AIC

Each AIC can be equipped with facets describing its main purpose/functionality. As explained in Chapter 4, these facets are derived from the combination of the task model, the domain model, and the mappings between them. The mappings between the task and the domain models have been described above. We illustrate

some of the rules applicable to the present case study. From these mappings it can be derived that:

- AICs **create name** and **create zipCode** are equipped with an input facet of type "create attribute value".
- AICs **select sex** and **select ageCategory** are equipped with an input facet of type "select attribute value". The enumerated values associated to the attribute are transferred as selection value of the facet from the domain model.
- AIC **Show Question** is equipped with an output facet of type "output attribute value" (i.e., the question title).
- AIC **Select Answer** is equipped with an input facet of type "select attribute value". It is also set to repetitive as the amount of instances of answer is only known at run-time: no enumerated values are provided nor attribute instances.
- AIC **Send Questionnaire** is equipped with a facet control that references the name of the method on which it is associated, here **sendQuestionnaire**
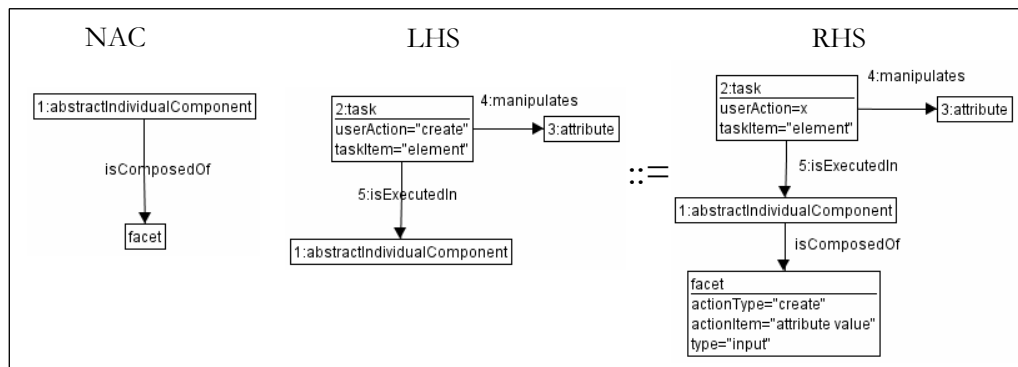
Rule 5-9 Create an input facet to AICs that realize creation tasks

5.2.2.c Spatio-Temporal arrangement of abstract interaction objects

We apply Rule 4-4 (reproduced as Rule 5-10), Rule 5-11, Rule 5-12, Rule 5-13. These rules reveal how implementing hierarchical rules in AGG could be repetitive: one rule should be introduced for each possible couple with AC and AIC as elements, that is a total of four rules.
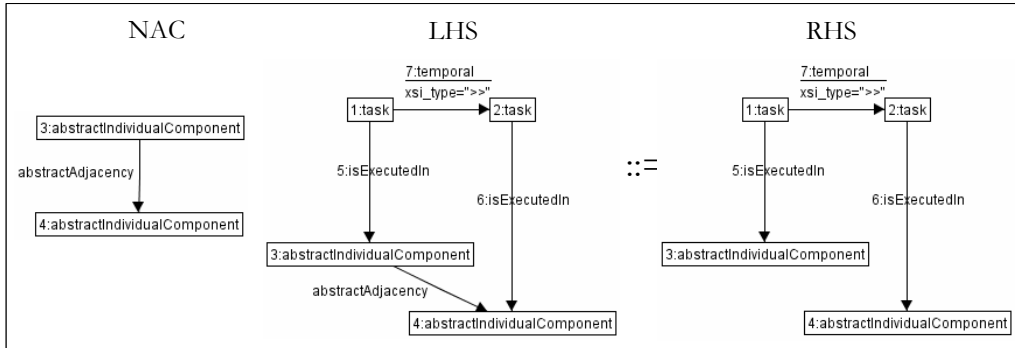
Rule 5-10 Deriving abstract adjacency for <AIC,AIC> couple
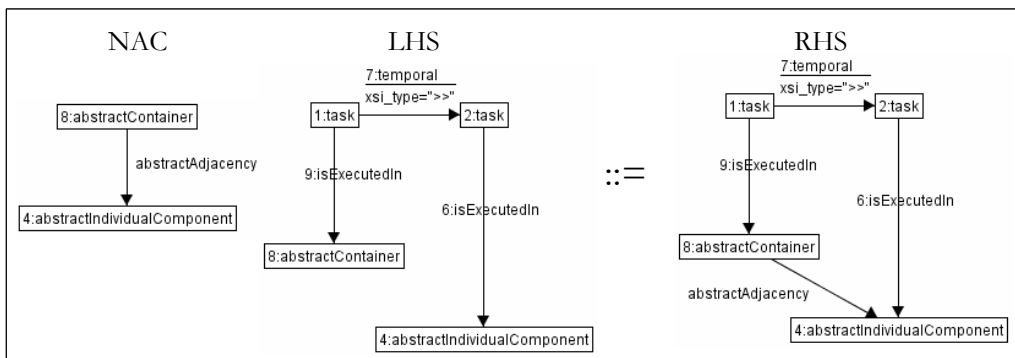
Rule 5-11 Deriving abstract adjacency for <AC,AIC> couple
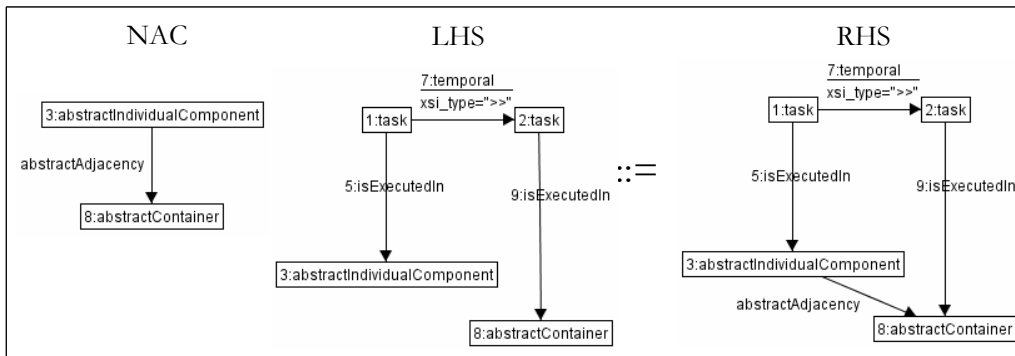
Rule 5-12 Deriving abstract adjacency for <AIC,AC> couple
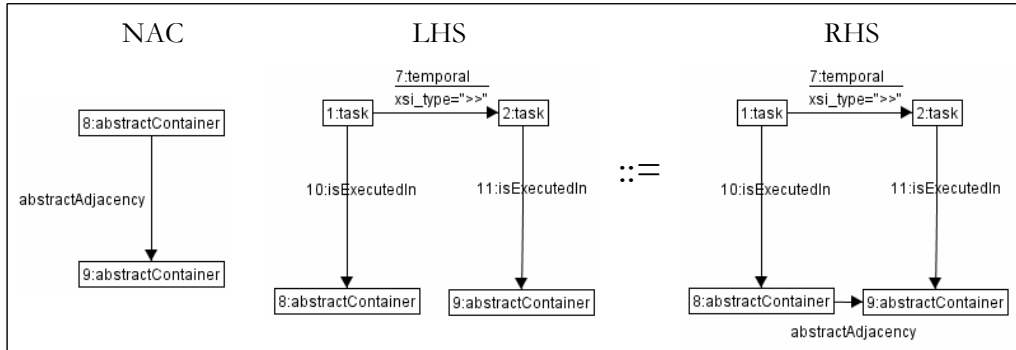
Rule 5-13 Deriving abstract adjacency for <AC,AC> couple

5.2.2.d   Definition of abstract dialog control

We apply Rule 4-5 and the like to realize this sub-step. Similarly to the previous step, a rule is defined for each combination of couple with AC and AIC as elements.

5.2.2.e   Derivation of AUI to domain mappings

Rule 4-6 is one of the rules applied in this sub-step. Rule 5-14 is another rule that is applicable to our case. It creates an **updates** relationship between the input facet of an AIC and the attribute manipulated by its associated task.
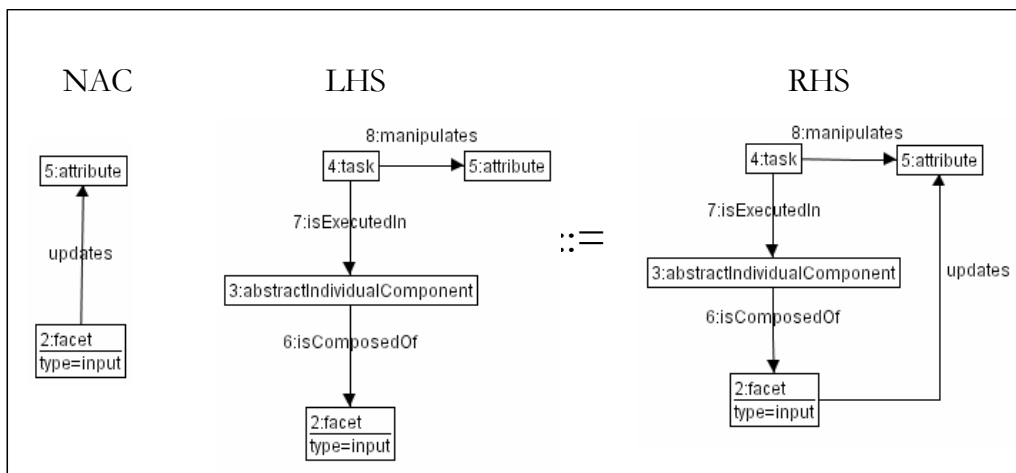
Rule 5-14 Derivation of the updates relationship for an input facet

5.2.2.f   Resulting specification

Figs. 5-15 and 5-16 respectively present the results of the above sub-steps and their corresponding UsiXML specifications. Lines 3 to 36 develop the AIO decomposition. Lines 38 to 61 represent relationships of abstract dialog control.
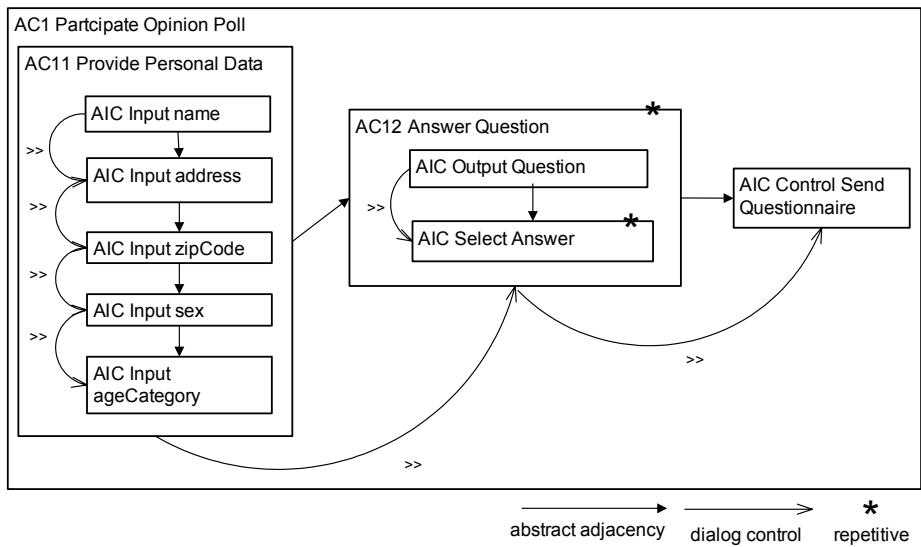
Rule 5-15 Representation of the AUI model for a Virtual Polling System

```
2    <auiModel name="CS1" id="CS1">
3      <abstractContainer id="AC1" name="Participate to Opinion Polll">
4   -     <abstractContainer id="AC11" name="Provide Personal Data">
5          <abstractIndividualComponent id="AIC111" name="create name">
6            <input id="FA1111" name="create name" actionType="create" actionItem="attribute value" datatype="String"/>
7          </abstractIndividualComponent>
8          <abstractIndividualComponent id="AIC112" name="create ageCategory">
9            <input id="FA1121" name="select AgeCategory" actionType="select" actionItem="attribute value" datatype="String">
10             <selectionValue name="18-35"/>
11             <selectionValue name="35-45"/>
12             <selectionValue name="45+"/>
13           </input>
14         </abstractIndividualComponent>
15         <abstractIndividualComponent id="AIC113" name="create zipCode">
16           <input id="FA1131" name="create zipCode" actionType="create" actionItem="attribute value" datatype="String"/>
17         </abstractIndividualComponent>
18         <abstractIndividualComponent id="AIC114" name="create sex">
19           <input id="FA1141" name="select sex" actionType="select" actionItem="attribute value" datatype="String">
20             <selectionValue name="female"/>
21             <selectionValue name="male"/>
22           </input>
23         </abstractIndividualComponent>
24       </abstractContainer>
25       <abstractContainer id="AC12" name="answerPoll" isRepetitive="true">
26         <abstractIndividualComponent id="AIC111" name="Output Question">
27           <output id="FA1111" name="Output Question" actionType="view" actionItem="attribute value"/>
28         </abstractIndividualComponent>
29         <abstractIndividualComponent id="AIC111" name="Select  Answer" isRepetitive="true">
30           <input id="FA1111" name="Select Answer" actionType="select" actionItem="attribute" datatype="String"/>
31         </abstractIndividualComponent>
32       </abstractContainer>
33       <abstractIndividualComponent id="AIC13" name="Send Questionnaire">
34         <input id="FA131" name="Send Questionnaire" actionType="start" actionItem="operation" action="sendQuestionnaire"/>
35       </abstractIndividualComponent>
36     </abstractContainer>
38     <auiDialogControl id="AR1" symbol=">>">
39       <source sourceId="AIC111"/>
40       <target targetId="AIC112"/>
41     </auiDialogControl>
42     <auiDialogControl id="AR2" symbol=">>">
43       <source sourceId="AIC112"/>
44       <target targetId="AIC113"/>
45     </auiDialogControl>
46     <auiDialogControl id="AR3" symbol=">>">
47       <source sourceId="AIC113"/>
48       <target targetId="AIC114"/>
49     </auiDialogControl>
50     <auiDialogControl id="AR4" symbol=">>">
51       <source sourceId="AC11"/>
52       <target targetId="AC12"/>
53     </auiDialogControl>
54     <auiDialogControl id="AR5" symbol=">>">
55       <source sourceId="AIC121"/>
56       <target targetId="AIC122"/>
57     </auiDialogControl>
58     <auiDialogControl id="AR6" symbol=">>">
59       <source sourceId="AC12"/>
60       <target targetId="AIC13"/>
61     </auiDialogControl>
62   </auiModel>
63
```

Figure 5-10 UsiXML code for AUI

5.2.3   From Abstract User Interface to Concrete User Interface

5.2.3.a   Reification of AC into CC

The simple heuristic solving the current problem consists of representing all tasks into one single window. This solution is often referred to as the "maximal window selection" [Vand94]. Other window identification schemes found in [Van94] can be equally defined. Rule 4-9 and 4-10 are applied to realize this sub-step. Each abstract container becomes a box except the top level that becomes a window ("maximal window" solution).

A variant of this rule is used in Sec. 5.3.4.a such that each abstract container becomes a window.

5.2.3.b   Selection of CICs

This sub-step involves the highest number of rules of all transformation sets as the different combinations of facet types, data types, cardinalities,…, are numerous. Table 5-1 provides the subset of rules applied in this case study. The designer can choose among the different alternatives provided by these rules.

Abstract Interaction Component	Facet Specification	Information to take into account	Possible Concrete Interaction Component
"create name" and "create zipCode"	Create attribute value	Data type, domain characteristics	A box with a label and an input field (Rule 5-16)
"select sex and select ageCategory"	Select attribute value + selection values known	Data type, domain characteristics, selection values	A dropdown list, a group of option buttons (Rule 5-17 and Rule 5-18)
"Show Questionnaire"	Output (value unknown)	Attribute, data type, domain characteristics	A label
"Select Answer"	Select attribute value + repetitive (selection values not known)	Data type, domain characteristics	A dropdown list, a group of option buttons
"Send Questionnaire"	Control	Feedback	A button

Table 5-1 correspondence between AIO types and CIO types
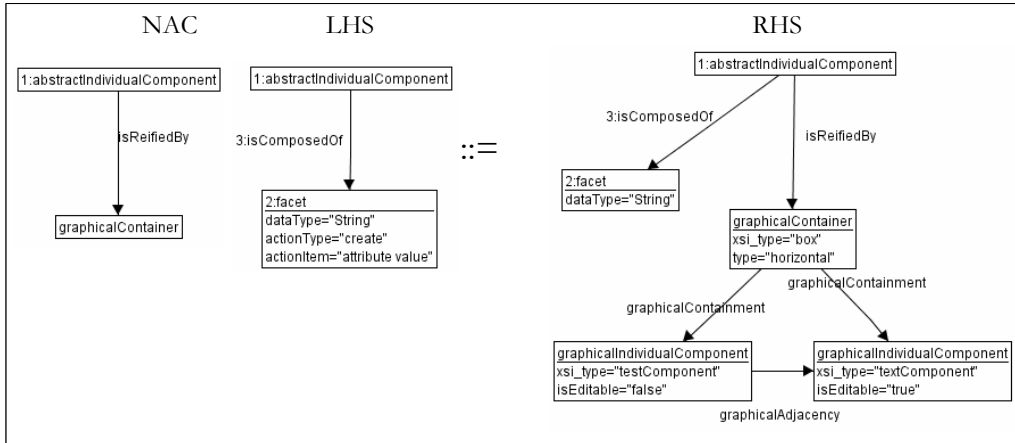
Rule 5-16 Creation of an input CIO from an input AIO
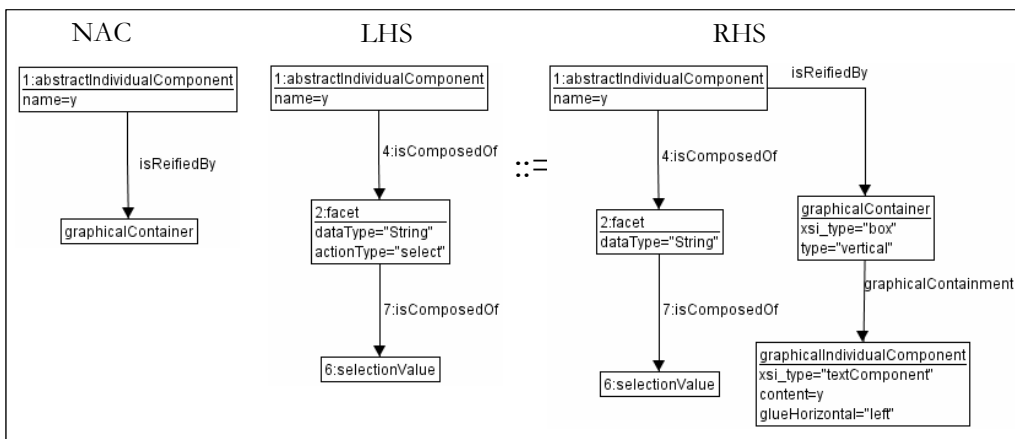
Rule 5-17 Creation of a box to hold radio button
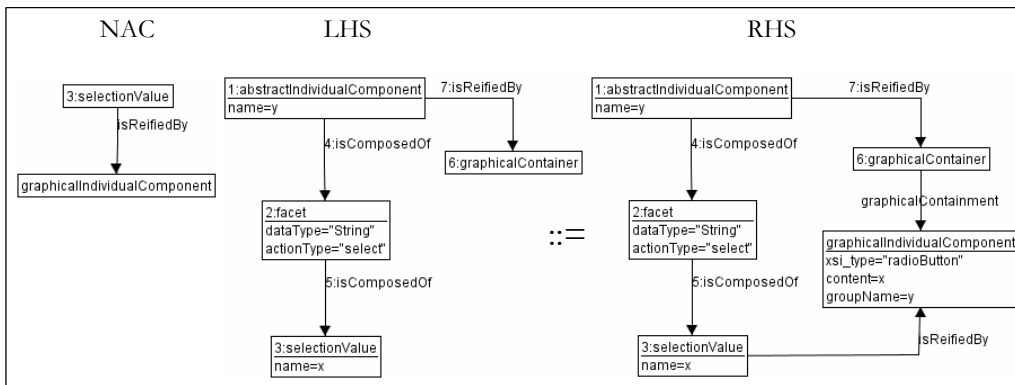
Rule 5-18 Creation of one radio button per selection value

5.2.3.c CIC placement

Rule 4-12 is mainly used to perform CIC placement. Like for abstract arrangement, a duplication of rules is necessary for covering combinations of couples with CC and CIC as elements.

5.2.3.d Navigation definition

Navigation specifies how the visibility property of CCs is set and, consequently, defines transitions between them. Since all elements are presented simultaneously into the same window, there is no particular need to define a sophisticated navigation scheme, which is the choice adopted here. However, some schemes can be added later on depending on the level of sophistication the designer want to add. Navigation definition is applied in Sec. 5.3.4.b.

5.2.3.e Concrete dialog control definition

Rule 4-14 is mainly used to perform CIC placement.

5.2.3.f Derivation of CUI to domain relationships

Rules like Rule 4-15 achieve the transposition of inter-model relationships.

5.2.3.g Resulting specification

The resulting specifications are obtained by realizing the above development sub-steps. Fig. 5-11 presents a mock-up of the graphical UI. Red rectangles denote invisible boxes, that are boxes whose visibility property has been set to "invisible" thanks to rules involved in the navigation definition (Section 5.2.3.d). Grey rounded rectangles denote visible boxes. The content of **Answer Question** will be a table containing questions and answers. The contents of this table will be determined at run-time when instances of questions and answers are known.

5. Case Studies

```
1   <cuiModel id="test_16-cui" name="test-cui">
2       <window ID="1" name="Main Window" icon="" content="participate to opinion poll" isVisible="true">
3           <box ID="2" name="Main Box" type="horizontal">
4               <box ID="3" name="Provide Personal Data" content="Provide Personal Date" type="vertical">
5                   <box ID="4" name="create name"  type="horizontal">
6                       <textComponent ID="9" name="create name" content="create name" isEditable="false"/>
7                       <textComponent ID="10"  name="create name" isEditable = "true"/>
8                   </box>
9                   <box ID="5" name="create zipCode" type="horizontal">
10                      <textComponent ID="11" name="create zipCode" content="create zipCode" isEditable="false"/>
11                      <textComponent ID="12"  name="create zipCode" isEditable = "true"/>
12                  </box>
13                  <box ID="6" name="select sex"  type="horizontal">
14                      <textComponent ID="13" content="select sex" name="select sex" />
15                      <radioButton ID="14" name="Female"  content="Female" groupName="select sex" glueHorizontal="right" />
16                      <radioButton ID="15" name="Male"  content="Male" groupName="select sex" glueHorizontal="right" />
17                  </box>
18                  <box ID="7" name="select ageCategory"  type="horizontal">
19                      <textComponent ID="17" content="select ageCategory" name="select ageCategory"/>
20                      <radioButton ID="16" name="18-35"  content="18-35" groupName="select ageCategory" glueHorizontal="right" />
21                      <radioButton ID="18" name="35-45"  content="35-45" groupName="select ageCategory" glueHorizontal="right" />
22                      <radioButton ID="19" name="45+"  content="45+" groupName="select ageCategory" glueHorizontal="right" />
23                  </box>
24              </box>
25              <box ID="8" name="Answer Question" content="Answer Question"  type="vertical">
26                  <table ID="20" name="Answer Question"  repetition="true">
27                      <textComponent ID="21"  name="Show Question"/>
28                          <table ID="22" name="Select Answer"  content="Select Answer" repetition="true">
29                              <radioButton ID="23" name="Select Answer" />
30                          </table>
31                  </table>
32              </box>
33              <button ID="24" name="Send Questionnaire"  content="Send Questionnaire"/>
34          </box>
35      </window>
36  </cuiModel>
```

Figure 5-11 UsiXML code for CUI level

Figure 5-12 CUI appearance for Virtual Polling System

5.2.4　Graphical Reshuffling of the CUI

A satisfactory layout is hard to derive in a systematic manner [Vand94]. No method surveyed in Chapter 2 allows a graphical reshuffling of the UI layout before being transformed to a final UI (FUI). With a UsiXML-compliant visual editing tool, such thing becomes possible. Fig. 5-13 shows a modification (i.e., an insertion of a box containing a label and a text editing zone) of the CUI presented in Fig. 5-12.

Consistency problems could arise when the CUI level is modified. If an element is withdrawn, the AIO it reifies (and transitively the tasks it supports) are no longer in correspondence and should be consequently erased. If an element is added, an element at the AUI should be added. Placement and dialog control relationships should be regenerated in both cases. If an attribute of an object is modified then AUI and CUI stay consistent. In our development scenario, the CUI is simply reverse engineered to a new AUI, rebuilt from scratch. Although solving the round-trip problem is beyond the scope of this dissertation, we demonstrate that a development scenario could be imagined on top of the concerned levels so as to address it. The means to address this problem are therefore already present. This is highlighted in the next section that reverse engineers the reshuffled UI layout.
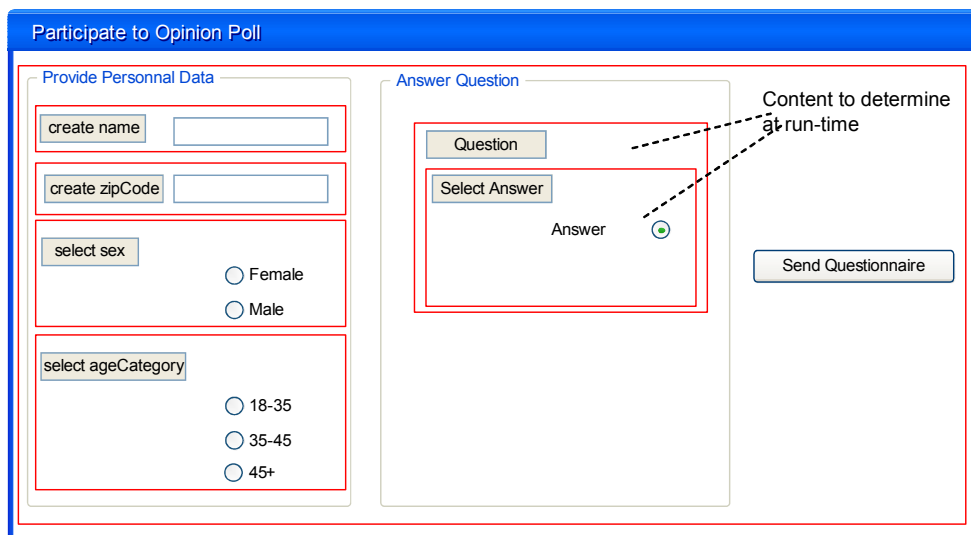
Figure 5-13 Reshuffled CUI

5.2.5 Reverse Engineering the AUI

After reshuffling the CUI manually, we define Rule 5-19, Rule 5-20, Rule 5-21, and Rule 5-22 to reconstruct the AUI structure from scratch.
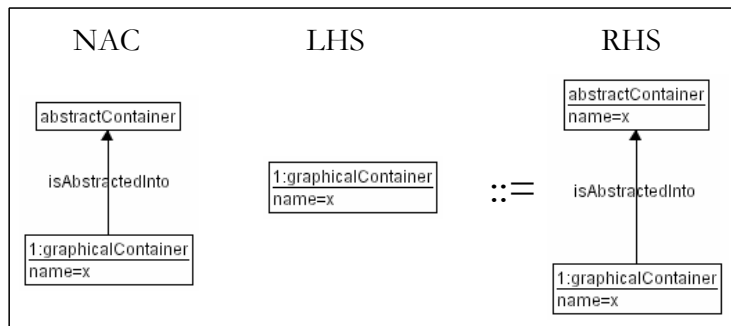
Rule 5-19 Creating an AC for each graphical container
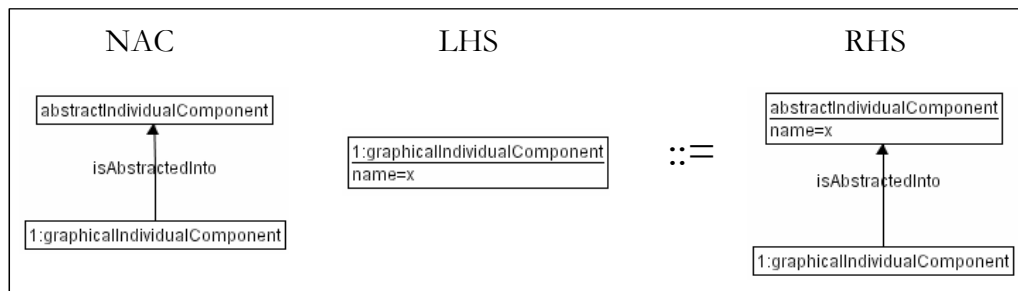
Rule 5-20 Creating an AIC for each graphical individual component
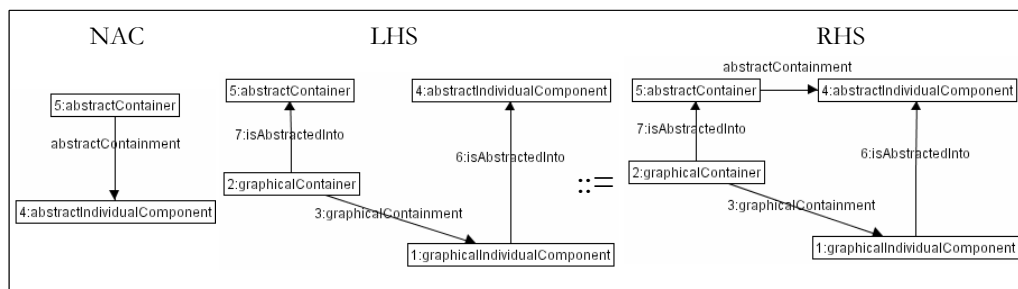
Rule 5-21 Recreating abstract containment relationships between ACs and AICs
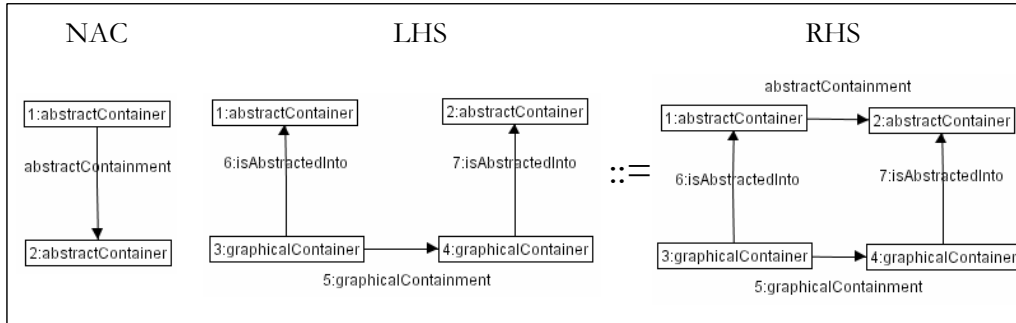
Rule 5-22 Recreating abstract containment relationships between ACs

After this, AICs facet types have to be recovered from the CIC specifications. This problem gives birth to many different rules. Here again, the restriction that "only one right hand side can be specified at a time", i.e., no disjunction allowed, turns out to be a true inconvenient. Indeed many different patterns in the CUI entail creating similar elements at the abstract level. Table 5-2 shows associations between CIC types and AIC facet types applicable to this case study. The column "special action" indicates some additional details for the rule representing this association.

CIC	AIC Facet Type
Button	Control AIC (Rule 5-23)
Radio button	Input for group of radio button. Transfer of selection values
text components if editable (i.e., input field)	Input (See Rule 5-24)
text components if not-editable (i.e., labels)	Output

Table 5-2 CICs and their associated facet types
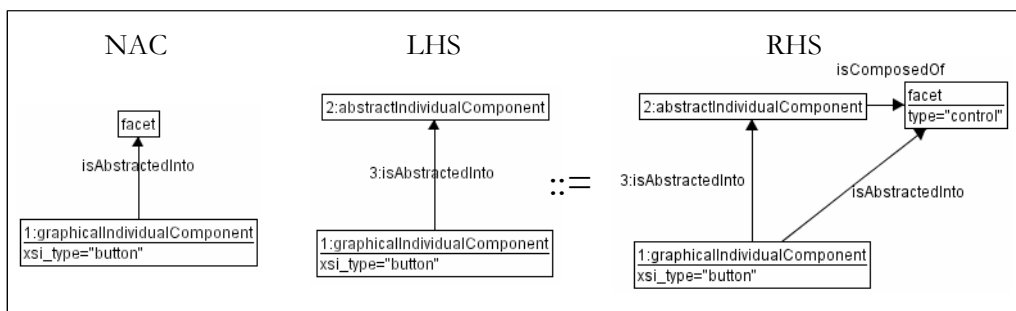
Rule 5-23 Reverse engineering of a button

LHS

```
                        6:abstractContainer

   abstractContainment                    abstractContainment

              15:isAbstractedInto
  abstractIndividualComponent           abstractIndividualComponent

                    5:graphicalContainer

   isAbstractedInto                           isAbstractedInto
         11:graphicalContainment      12:graphicalContainment

  1:graphicalIndividualComponent      2:graphicalIndividualComponent
  xsi_type="textComponent"     16:graphicalAdjacency   xsi_type="textComponent"
  isEditable="false"                    isEditable="true"
```

RHS

```
                        6:abstractContainer

        15:isAbstractedInto       abstractContainment

  11:graphicalContainment                  12:graphicalContainment
            5:graphicalContainer

                 abstractIndividualComponent

                        isComposedOf
   isAbstractedInto                          isAbstractedInto
                     facet
                     type="input"

  1:graphicalIndividualComponent      2:graphicalIndividualComponent
  xsi_type="textComponent"     16:graphicalAdjacency   xsi_type="textComponent"
  isEditable="false"                    isEditable="true"
```

Rule 5-24 RHS to reverse engineer a box containing a label and an input field

Rule 5-24 performs reverse engineering of a complex structure i.e., a box containing a label that is adjacent to an input field to a corresponding structure at the AUI level, thus achieving some form of design recovery.

Rule 5-25 and Rule 5-26 are rules used to reverse engineer, respectively, abstract adjacency relationship and abstract dialog control. Similarly to forward engineering, a rule should be defined to cover any possible combinations.
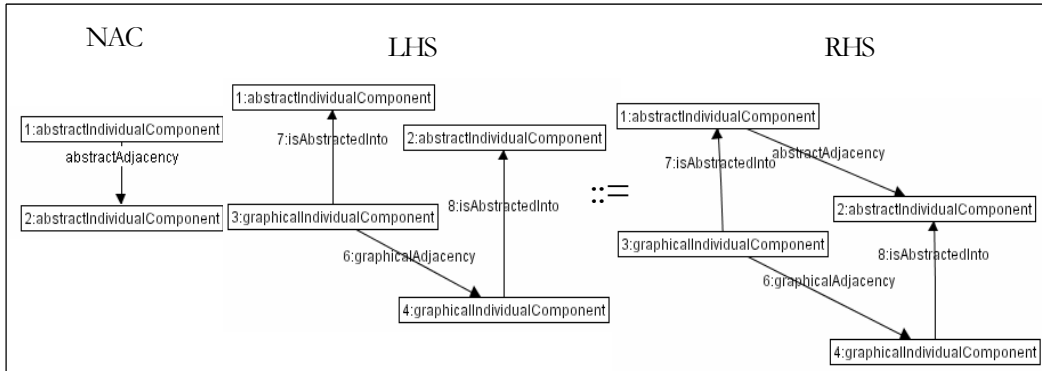
Rule 5-25 Reverse engineering abstract adjacency
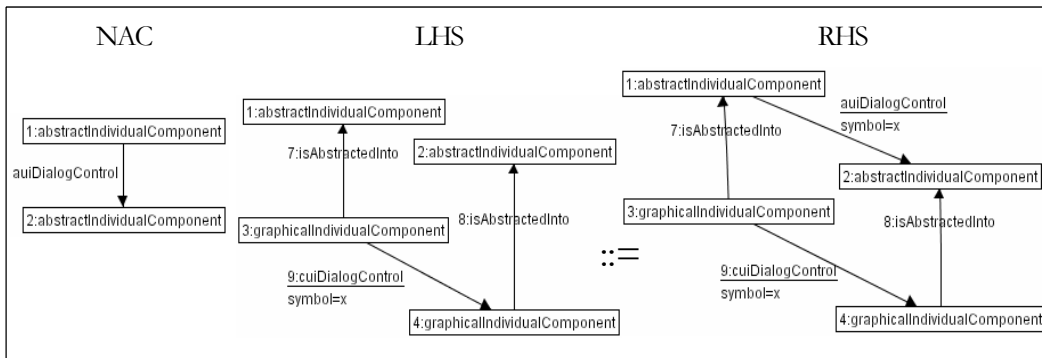
Rule 5-26 Reverse engineering AUI dialog control

5.2.6   Resulting Specification

The resulting abstract specification is illustrated by Fig. 5-14.
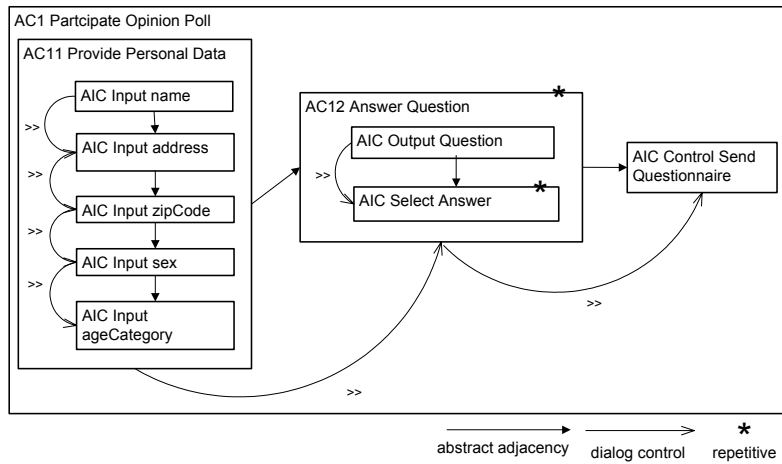
Figure 5-14 AUI model after reverse engineering process

5.3 Case Study 2: a Virtual Travel Agent

The second case study considered in this chapter is inspired by the benchmark specification provided by the FIPA [FIPA00]. This specification describes a Personal Travel Agent (PTA). The main function of this system is to allow a user searching, booking, and paying a flight, a rental car, or a hotel room.

The development scenario of this case study starts with an initial Task & Domain viewpoint (Fig. 5-14). From this viewpoint, an AUI (T1) is produced. Three different transformations are applied to the AUI to derive a graphical 2D CUI for a desktop PC i.e., a normal display (T2), a graphical 2D CUI for a small display (T3), and an auditory CUI (T4). Then the initial Task and Domain viewpoint is translated (T5) for a new context of use that is considered different and complex enough to complete the transformation from an adapted AUI instead of the same initial AUI. This is reflected by some changes in the task and domain models. From this adaptation, a new AUI is derived (T6) and a CUI for a normal display (T7) model is translated. This development scenario shows that transformations can be applied whenever and wherever they are considered appropriate. In this case, two different AUIs are derived from two task and domain models sharing the same initial point and several CUIs are derived from the first AUI. Again, only the newly introduced constructs are detailed.
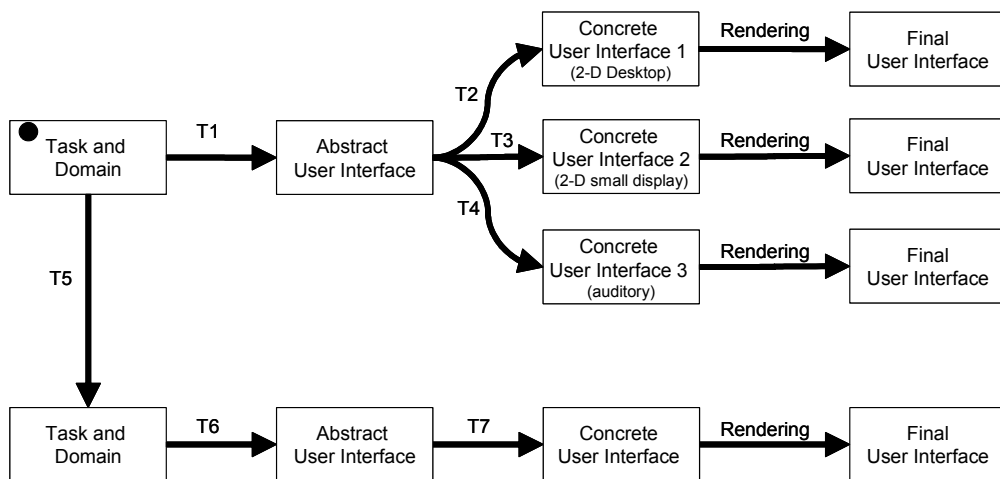
Figure 5-15 Development scenario for case study 2

220

5.3.1   Initial Representations
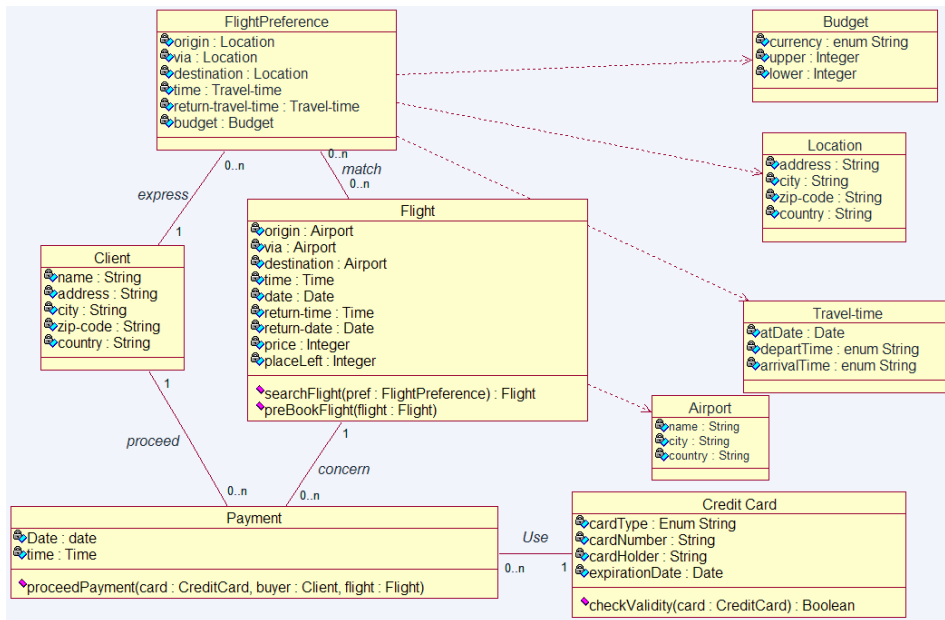
Figure 5-16 Class diagram for a PTA

The class diagram (Fig. 5-16) involves 9 classes. Flight characterizes an airplane flight with its origin, destination, time, date, etc. Client describes client's characteristics. FlightPreference describes the preferences of a client in terms of destination, schedules, and budget. Payment gathers information related to a flight payment. CreditCard provides information on credit cards, the only payment mean considered in our system. Budget, Location, Travel-Time and Airport are classes that are used as data types by FlightPreference and Flight classes.
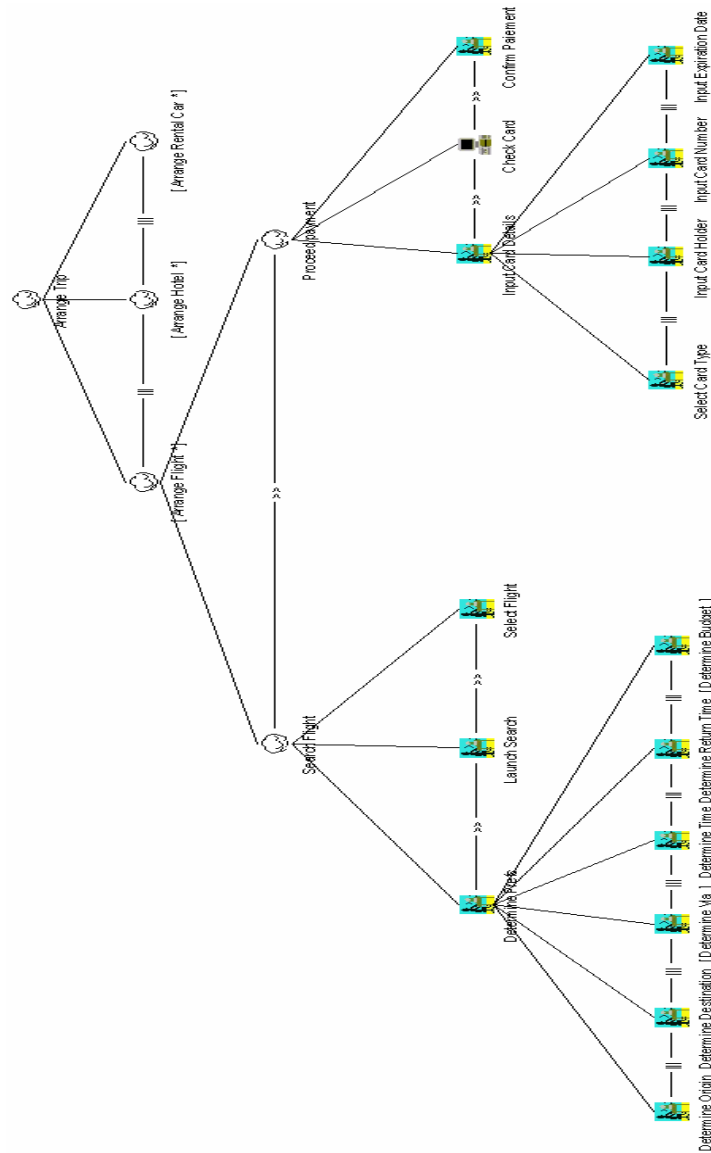
Figure 5-17 Task model for the PTA

The task model represented in Fig. 5-17 with the CTT notation can be described as follows: the user may either arrange a flight, a hotel room, or a rental car. This case study focuses on the first possibility. To arrange a flight, the user has first to search for a flight and then to pay for this flight. To search for a flight, the user has to determine her preferences, to launch the search and to select a flight among the results returned by the system. After selecting a flight, the user proceeds to final payment. For this purpose, the user inputs the details concerning her credit

card whose validity is verified by a system task. The user then confirms her payment.

The UsiXML specifications corresponding to this case study are substantially larger than those of the first case study. For this reason, the relationships between the domain model and the task model are summed up in Table 5-3. A dotted notation of the form Class.attribute denotes attributes.

Task	Domain concept
Determine origin (create element)	FlightPreference.origin
Determine Destination (create element)	FlightPreference.destination
Determine Via (create element)	FlightPreference.via
Determine Time (create element)	FlightPreference.time
Determine Budget (create element)	FlightPreference.budget
Launch Search (start operation)	FlightPreference.searchFlight( )
Select Flight (select element)	Output parameter of FlightPreference.searchFlight( )
Select Card Type (select element)	CreditCard.cardType
Input Card Holder (select element)	CreditCard.cardNumber
Input Card Number (select element)	CreditCard.cardHolder
Input Expiration Date (select element)	CreditCard.expirationDate
CheckCard (start operation)	CreditCard.checkValidity( )
Confirm Payment (start operation)	Payment.proceedPayment( )

Table 5-3 Mappings between the Task and the Domain for the PTA

Like for the first case study, before initiating the transformation process, we have to make sure that the decomposition level is appropriate. From the Task and Domain viewpoint introduced above, it is observed that all create tasks are well mapped onto corresponding attributes. This has been identified in the first case study as an appropriate level of decomposition. To support attributes attached to complex types, Rule 5-27 creates a sub-task for each attribute composing the complex data type. For example, the attribute origin is of type Airport ; Rule 5-28 therefore creates three sub-tasks to input the three attributes of the data type: name, city, and country.
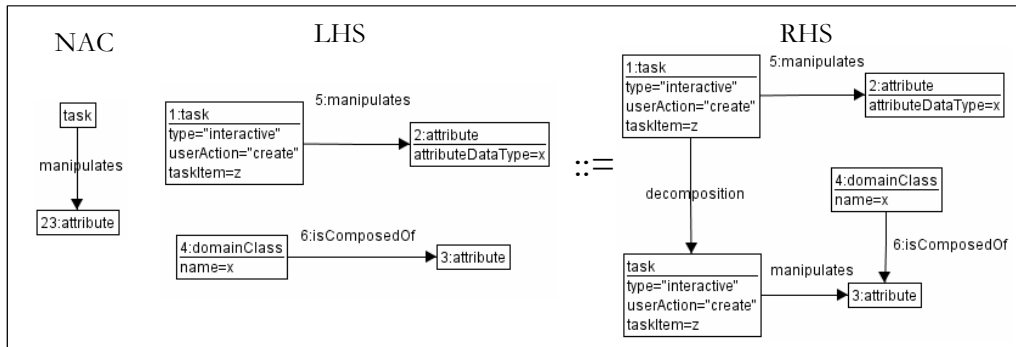
Rule 5-27 Creating sub-tasks for tasks manipulating complex types

5.3.2 Derivation of the AUI

The derivation of the AUI (result in Fig. 5-18) defined for the first case study can now be reused with two noticeable differences:

- The inclusion of a **select** task manipulating the output parameter of a method. This enables the user to perform a selection of the flights returned by the **searchFlight** method. As for **Answer.title** in the first case study, the exact amount of items in the selection is known only at run-time. Like in this latter case, this type of configuration gives birth to a repetitive AUI with a **input selection** facet.

- The inclusion of a system task of **start operation** type. This system task is not a inherent part of the UI but may be represented by a feedback object. Later on, one can decide whether this feedback object should be incorporated in the same containers (e.g., a label providing a feedback) or not (e.g., a pop-up window provides the feedback). In any case, the system task is translated into a AIC with a control facet.
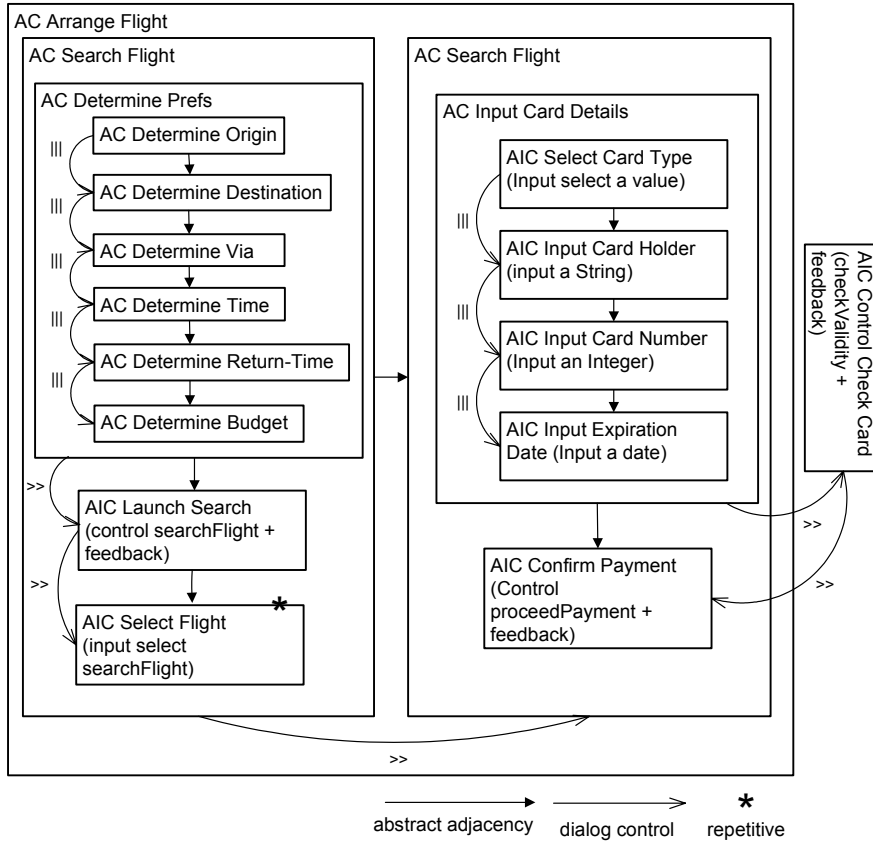
Figure 5-18 AUI for Virtual Travel Agent

5.3.3   Derivation of CUI for desktop

The derivation of the CUI for the desktop computing platform (Fig. 5-19) is based on the same principles than those we for the first case study. In this case, we define combination boxes ("combo boxes") for performing a selection when the number of items to select is greater than three. Any AUI part operating on a Date data type is mapped onto boxes containing four CICs: a label, a drop-down list for the day, a drop-down list for the month, and a drop-down list for the year. Note that a mapping of a date to a date picker at the CUI level can be defined alternatively.

Figure 5-19 CUI of the PTA for a desktop application

5.3.4   Derivation of CUI for small display

This derivation seeks to adapt the previously defined AUI for a display that is smaller than for a desktop. Only a few elements of previously defined transformation systems have to be re-defined to obtain a UI that is more appropriate for this constrained display. These transformations are detailed in the next subsections.

5.3.4.a Reification of AC into CC

Rule 5-28 states that all ACs at all decomposition levels are mapped onto windows to reduce the screen density of information manipulated in each window. More sophisticated rules for graceful degradation of UIs for small displays are discussed in [Flor04].
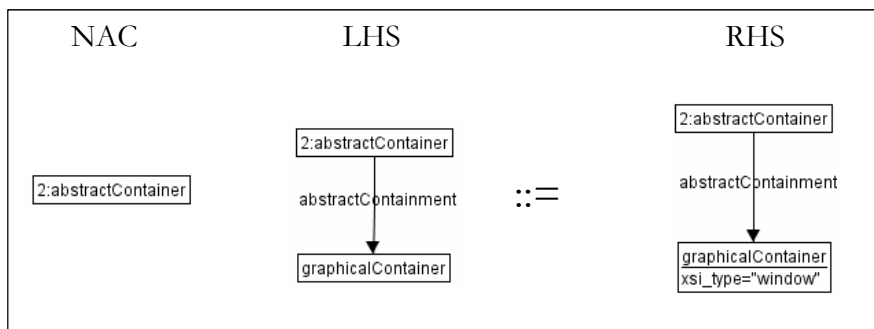
Rule 5-28 Every AC gives birth to a window
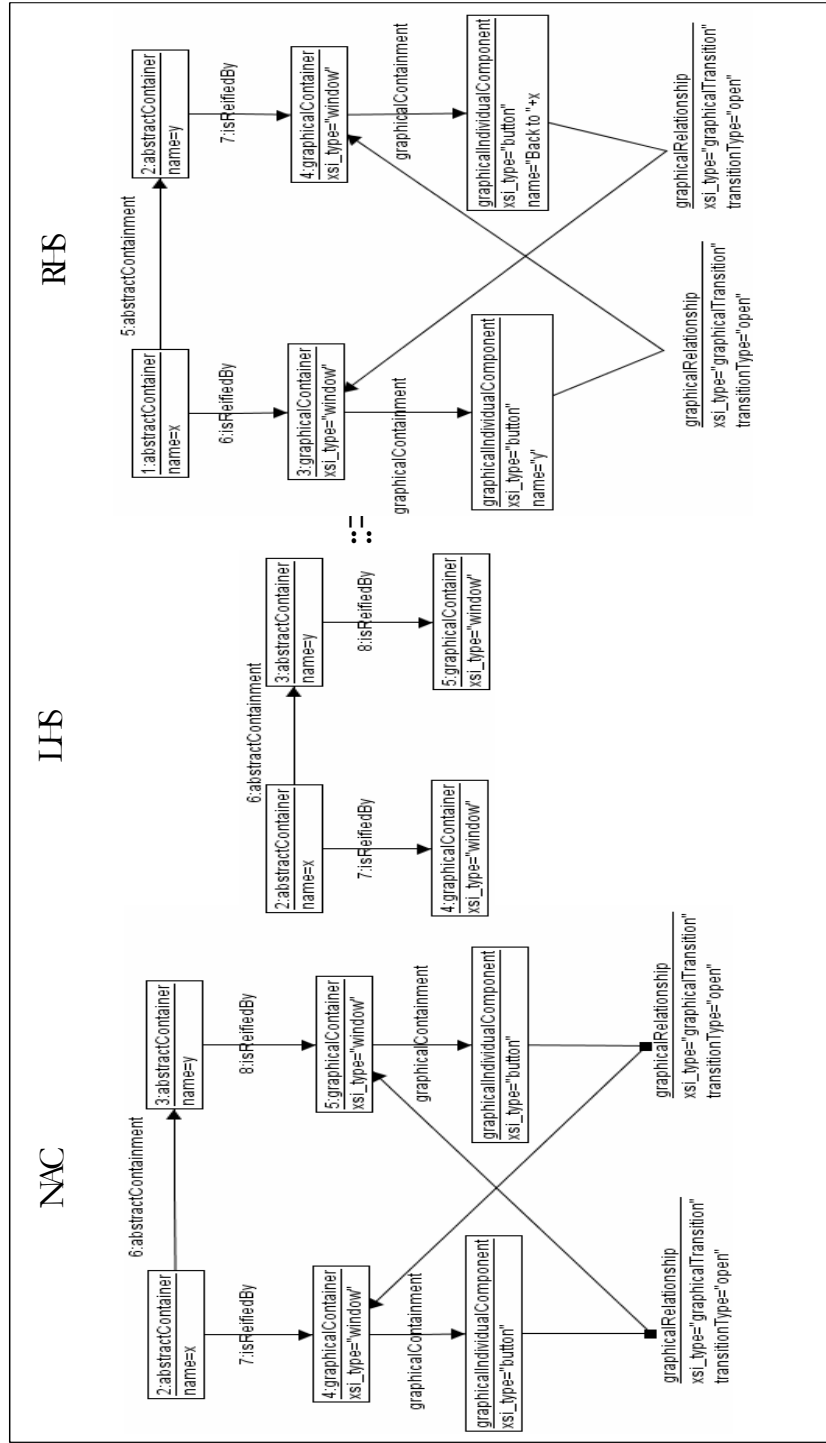
Rule 5-29 Generation of the navigation for small display

5.3.4.b   Navigation definition

Since a new type of display is considered, this step is different from the first case study and the first part of the current case study. The navigation defined here enables a user to navigate: (1) from any parent window to its children and (2) from any child to its respective parent window (Rule 5-29).

5.3.4.c   Resulting specification

The resulting specification of this step is illustrated in Fig. 5-20.
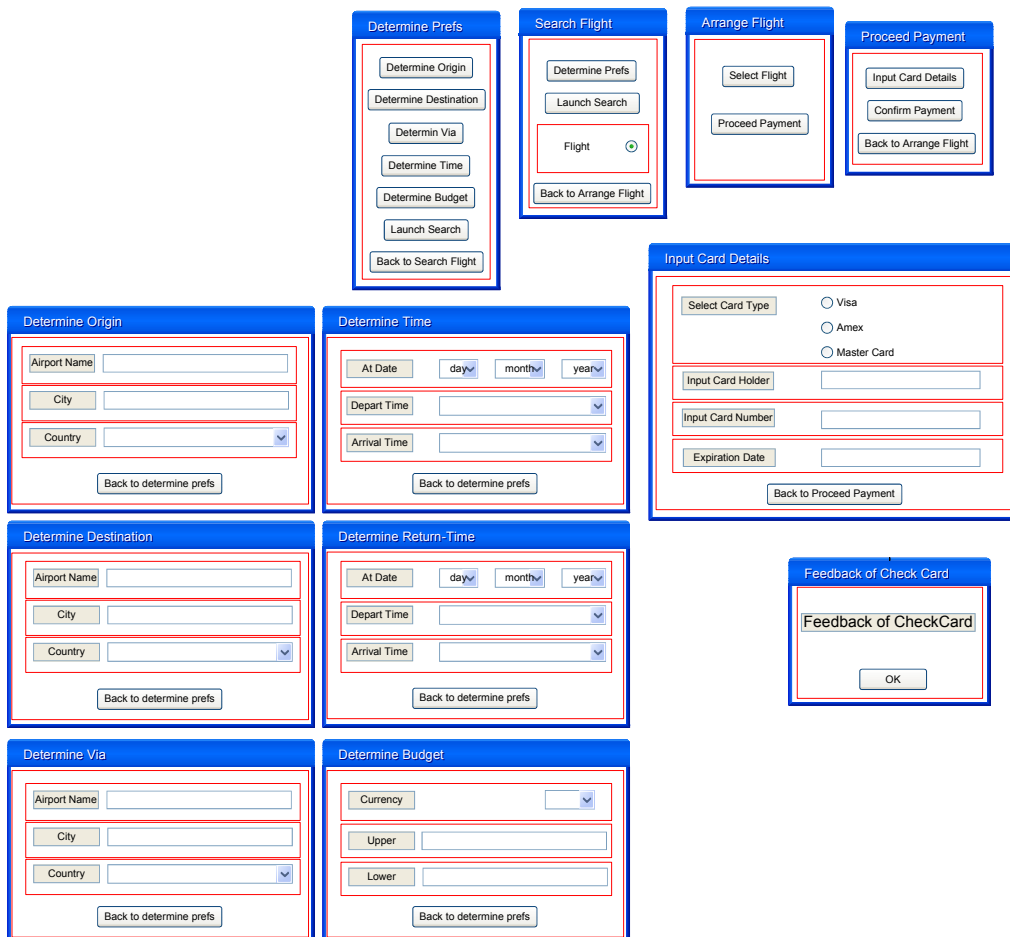
Figure 5-20 CUI of the PTA for a small display

5.3.5   Derivation of Auditory Interface

There is not a wide range of auditory CIOs in our ontology. Auditory containers gather auditory individual components. Auditory individual components are auditory input components and auditory output components. These elements allow us building auditory UIs as a succession of questions and answers.

Sequentially applying the different development sub-steps defined in Chapter 4 is not strictly necessary as the complexity of executing this development step is reasonable.

The reification of AC into CC consists of creating an auditory container for each AC. Auditory UIs being one-dimensional, per definition the structuring of interactors in auditory containers has only an importance for navigation definition i.e., when a possibility is offered to the user to initiate one part of the interface or another.

The selection of auditory individual components is the richest operation in the derivation of an auditory CUI. This operation consists of selecting an appropriate combination of output and input auditory components. Rule 5-30 operates such a complex transformation by deriving, from an input AIC, what could be called a "block" of speech dialogue defining a "question-answer-cancel" composite element. As it may be observed in this rule, inter-model relationships with the domain model are exploited to directly derive CUI to domain relationship. This violation of the methodological separation of concern is justified by the fact that elements of the domain model are used in the definition of the created auditory components e.g., the name of the mapped domain attribute becomes part of the output of an auditory element.

Our derivation rules are illustrated on a sub-tree of the case study. Fig. 5-21 proposes an illustration of the dialogue between the user and the vocal system.

NAC

LHS

::=

RHS

1:abstractIndividualComponent
actionType="create"
actionItem="attribute value"

isReifiedBy

auditoryContainer

2:attribute
name=x

3:updates

1:abstractIndividualComponent
actionType="create"
actionItem="attribute value"

observes

2:attribute
name=x

3:updates

1:abstractIndividualComponent

isReifiedBy

auditoryContainer

updates

auditoryContainment

auditoryOutput
prosody="invite"
content="Please insert the "+x

auditoryAdjacency

auditoryInput
ellapsedTime=3

auditoryAdjacency

auditoryContainment

auditoryContainment

auditoryOutput
content="you have entered"

auditoryAdjacency

auditoryContainment

auditoryContainment

auditoryOutput

auditoryAdjacency

auditoryTransition
transitionType="open"

auditoryContainment

auditoryOutput
prosody="invite"
content="if you want to restart your input, say yes"

auditoryAdjacency

auditoryInput
ellapsedTime=2

auditoryAdjacency

Rule 5-30 Transforming create AIC into complex input auditory components

Please insert the [cardType]

[Visa]

You have entered

[Visa]

If you want to restart your input, say

yes

[yes]

Please insert the [cardType]

[Amex]

You have entered

[Amex]

If you want to restart your input, say

yes

[-]

Please insert the [CardHolder]

…
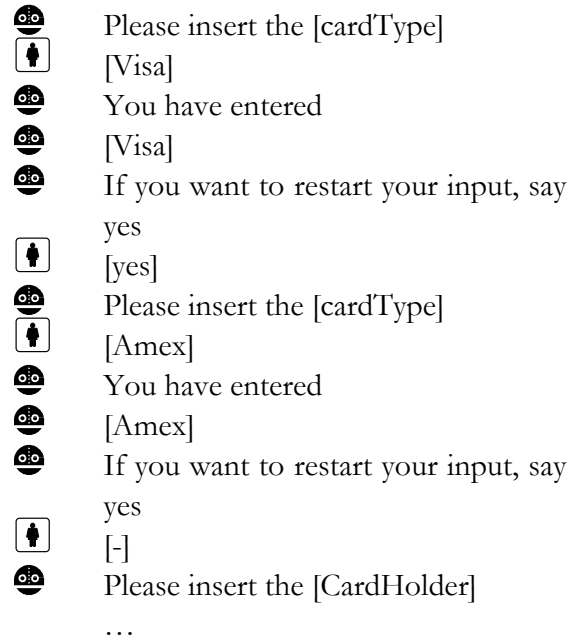
Figure 5-21 Instance of a dialog between a user and a vocal system

5.3.6 Translation of the Task Model and Forward Engineering the CUI

As described in the scenario of Fig. 5-15, after producing the three UI described above, the designer wants to regenerate a UI from a slightly altered version of the task model. For this purpose, the designer prunes this task model according to the task **importance** which is an attribute of the task evaluated on a scale of 1 to 5. Rule 5-31, Rule 5-32, Rule 5-33, Rule 5-34 realize a pruning of the task model presented in Fig. 5-16.
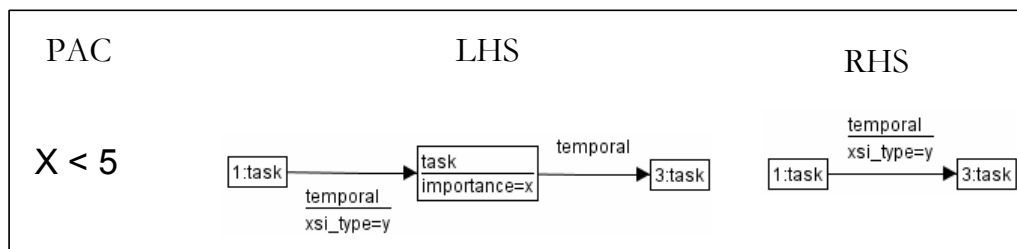
Rule 5-31 Erasing unimportant tasks while reconstructing temporal relationships
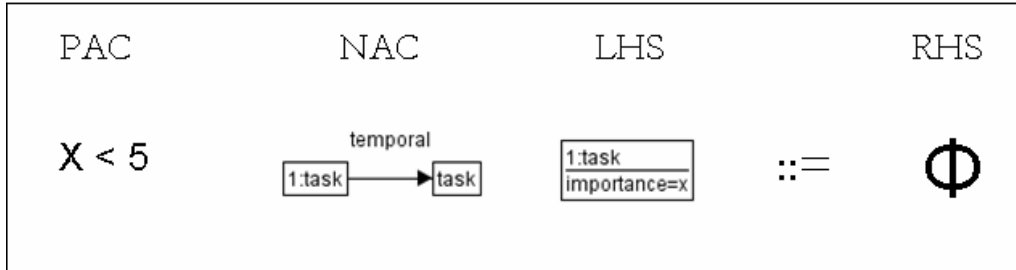
Rule 5-32 Erasing unimportant task that have no successor sister
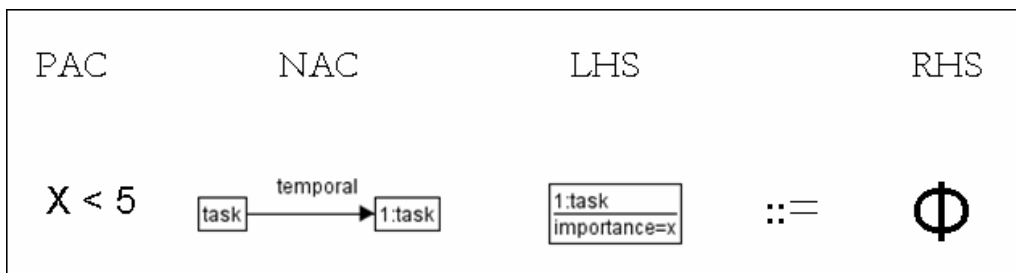
Rule 5-33 Erasing unimportant tasks that have no predecessor sister
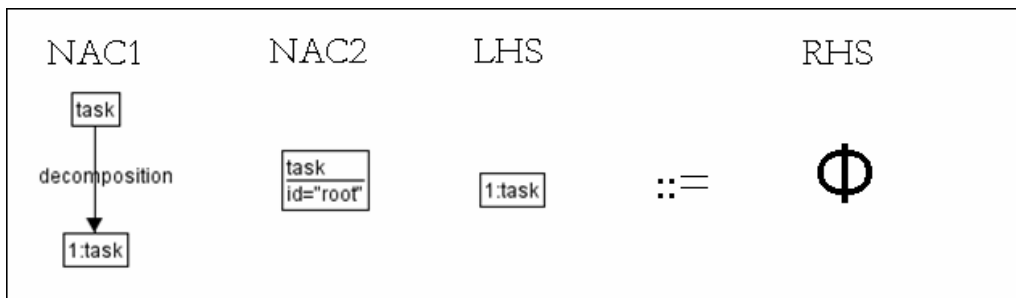
Rule 5-34 erasing all tasks that have no father except the root

The task model resulting from the application of these four rules is showed in Fig. 5-22. From this task model it is possible to initiate a development process similar to the one described at Sec. 5.3.2 and 5.3.3.
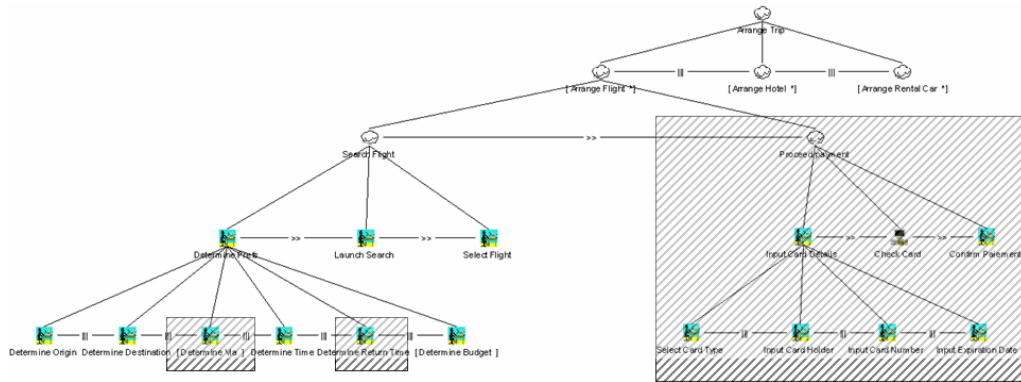
Figure 5-22 Task model of Fig. 5-16 after being pruned

5.4 Conclusion

The two case studies presented in this chapter show how multi-path development applies to low to mid-complex examples of both graphical and auditory interfaces.

To solve these case studies we have followed the following procedure: (1) Building initial models. Such models have been edited with their associated editing tool. (2) Editing and debugging of rules within the AGG graphical environment. Most of these rules have been elicited prior to realizing these case studies by a theoretical analysis of development sub-steps as illustrated in Chapter 4. (3) Importing initial models into the AGG graphical environment. (4) Selecting a transformation set and firing the rules contained in this set. (5) Exporting resulting models from AGG to UsiXML and illustration.

This process led us to deduce the following conclusions regarding the strengths and weaknesses of our method.

Our case studies showed the **feasibility** of developing a UI in a principled-based and rigorous manner relying on explicit transformation catalogs at any time. The diversity of development paths that have been presented highlight the possibility of manipulating UI related artifacts according to different development scenarios and pave the way to consider multiple other alternatives. In particular, new development scenarios can be developed by refinement (e.g. a more elaborated scenario), by composition (e.g., a new scenario by composing several existing scenarios), by transformation (e.g., a newly defined scenario by deriving other forms of scenarios from existing ones) or by reusing. The reuse of transformations has been illustrated when transformation systems have been straightforwardly reused from one case study to another one. As so, we avoid *ad hoc* development catalogs and enable a capitalization of transformations in a consolidated approach while trying to avoid the proliferation of scenarios that are close to each other.

The difficulty and weaknesses we encountered while realizing these case studies are the following:

Lack of expressivity of models. As it was observed in Chapter 4, the mere fact of decomposing a transformational development process into steps and sub-steps enables an identification of weaknesses of certain models in terms of expressivity.

As the preciseness in the expression of transformation grows, some models revealed to need enrichment to allow their exploitation for derivation means e.g., the task model had to be enriched with various concepts. For instance, to describe the intrinsic nature of a task, the domain model needed a better expression on the nature of the domain of attributes.

Inherent complexity of certain sub-steps. The complexity of some sub-steps (for instance, those involving a definition of the layout) relies in the multiple criteria to be considered simultaneously and the high number of possible design options. This complexity is not diminished by our method. Only the elements underlying design options are made explicit, and their performance is enabled in a formal manner.

Difficulty in finding an appropriate level of generalization when defining rules remains difficult. Conditional graph rewriting offers expressions having no side effect i.e., a rule only affect parts of the graph defined in its scope. Nonetheless, a rule may always have a "wider" scope than planned by its designer. It therefore affects unexpected graph elements. On the other hand, defining very precise rules entails defining a collection of rules for realizing a transformation that could be obtained with the application of one single and more generic rule. An automatic recognition of sets of rules able to be synthesized in one rule would be desirable in this case. This problem is an illustration of the *rule composition* issue raised in the literature.

Lack of disjunction in the rule expression. There is a redundancy in the expression of certain rules. Indeed, certain sets of rules operate a similar modification to the initial graph (i.e., they have a similar RHS) but have slightly different application conditions (i.e., their LHS). Imagine a rule applying a transformation to all AICs manipulating String or Integer data. To handle this case, two distinct rules should be defined. As so, we think that the possibility of using disjunction in the application conditions of rules could help us to decrease the number of rules and the risk of inconsistencies in the application of the method.

Difficulty to get meta-information on the graph. During the execution of certain sub-steps, it turned out that the state of the specification could have been, advantageously, complemented by meta-information on the specification itself. *Meta-information* is information that is processed by externally analyzing the state of the specification. For instance, a meta-information could consist of counting the

number of edges pointing or starting from a specific node. This meta-information could help us to define meta-heuristics that are rules able to select an appropriate transformation rule.

Difficulty in ordering rules within transformation systems. It happens that two rules of a same transformation system apply to similar graph nodes. These rules are referred to in the literature as a *critical pair*. In this case, the ordering of rules has an impact on the graph resulting from the transformation system. *Critical pair analysis* is an algorithmic analysis technique operating on graph grammars and identifying conflicting rule couples. This technique is available in the AGG environment. Nonetheless, once these pairs are identified, it remains tedious to modify or re-arrange conflicting rule couples.

Difficulty in ordering sub-steps within steps. In a similar manner to rules, it is not an easy task to order sub-steps within a same step. Each sub-step, along with its associated transformation system, produces a graph presenting certain characteristic i.e., type of nodes and relationships produced during the execution of the sub-step. Arranging sub-steps such that the information produced by the previous sub-step will not be modified afterwards remains an undetermined activity. The help of a formal expression of pre- and post- condition of each sub-step would certainly improve this aspect.

Difficulty in implementing generalization in AGG environment. AGG as such does not support hierarchy. This hierarchy had to be mimicked in this environment by merging sub-types of certain nodes into a single node and using an attribute to differentiate the subtypes. This is a well known solution to people who want to translate UML generalizations into database relational schemas. Consequently, in some cases this entailed the definition of several rules when only one could be defined. Nonetheless, the lack of hierarchy in AGG did not reduce the expressivity of the transformation system themselves. Some of them are just verbose.

Difficulty in coordinating tools around an evolving ontology. The method that is proposed involves the collaboration of many tools. On the other hand, our ontology, after its first expression, one year ago, has evolved rapidly. This is due notably to the feedback received from the first users of UsiXML. Any change applied to the ontology entails the adaptation of several tools resulting in a lot of development effort and, also, delays in the support of modifications brought to the ontology. Coordinating tools in such context is not an easy task.

Difficulty in generating identifiers. By definition, a rule does not generate the identifiers of the created elements as it applies to a pattern. Adding the identifier at the export of the specification has been the choice we made to solve.

Chapter 6   Conclusion

6.1 Context of This Work

Transformational development is one of the answers provided by the Software Engineering (SE) community to tackle the problem of building software in a systematic and principle-based way.

Transformational development in SE defines the development of software as a progressive refinement of abstract models into concrete models, until program code [Somm99]. This transformational development relies on catalogs of transformations able to (semi-)automatically perform model-to-model and model-to-code transformations.

Transformational development of user-interfaces (TDUI) specializes principles of transformational development in the context of UI development. By analogy with transformational development in SE, it defines the development of user interface systems as a successive application of transformations to an initial representation. This generally implies a progressive refinement of an abstract model into a concrete model, until program (here UI) code, or vice versa.

Since the mid-nineties, numerous engineering methods have been proposed to support TDUI (see Chapter 2). Most of them are concentrated on deriving UI code from abstract models, others are focused on recovering a model from a UI implementation. A more recent trend gave birth to methods dedicated to the adaptation of a UI system to multiple contexts of use, as many variations of these contexts have been observed.

6.2 Content of This Dissertation

The state of the art of Chapter 2 reveals a series of shortcomings in existing approaches for achieving TDUI. These shortcomings delineated our problem space. These shortcomings lead us to conclude that TDUI can be improved along several dimensions.

For this purpose, this dissertation proposes (1) an ontological framework based on an explicit and rigorous representation of concepts relevant to UI development (2) a methodological framework based on the ontological framework previously introduced. This methodological framework introduces a new paradigm for UI development called **multi-path development of UIs** that is characterized by the following principles:

- *Transformation driven*: a development method is composed of development stages. A development step is a transition from one stage to another one. Development steps rely on explicit and rigorous transformation catalogs.

- *Multiple-path:* the context of development projects may involve variable arrangements of development steps. A development path refers to a particular arrangement of steps. Multi-path development refers to the capacity of a method to accommodate to various development paths.

Chapter 3 presents an ontology for the specification of UIs.

Sec. 3-2 details the concepts and relationships in the scope of our ontology. Two essential artifacts were introduced to structure this ontology:

- *Viewpoints* materialize different "concerns" on the UI system. Four viewpoints have been introduced, motivated, and defined: A *final UI* viewpoint is the implementation of a UI system as it can be seen from the code level or from the rendering level (i.e., its appearance); A *concrete UI* viewpoint is a description of a UI which is, as independent as possible, of any reference to implementation details (i.e., toolkit). An *abstract UI* has been defined as a description of the UI that is as independent as possible of any reference to the modalities for which a UI is designed (e.g., graphical interaction, vocal interaction). A *Task and Domain* viewpoint concerns a representation of UI systems in terms of tasks to be carried out by a user in interaction with the

system along with the domain-oriented concepts as they are required by these tasks to be performed.

- *UI models* have been exposed thanks to conceptual schemas expressed in UML. UI models gather concepts of interest in the development of a UI system. Some of the UI models are transversal to all viewpoints: a context model describes the context for which a set of models, a model or a part of a model is specified for. An inter-model relationship allows a designer to relate different models across or inside viewpoints.

In Sec. 3.3, a mathematical formalism for representing our ontology is motivated and presented. This formalism consists in "directed, identified, labeled, constrained and typed graphs" and can be considered as the abstract syntax of our ontology. Sec. 3.4. illustrates two different, yet semantically equivalent, concrete syntaxes for our ontology. These two syntaxes reflect the conceptual structure introduced in Sec. 3.2. while respecting the graph-based mathematical notation introduced in Sec. 3.3. A graphical syntax relies on boxes and arrows to express concepts in the scope of our language and their relationships. An XML compliant syntax, called UsiXML, relies on XML schemas to enable a textual representation of any concepts presented in Sec. 3.2.

As a result of Chapter 3, any UI specification model and viewpoint is represented under the form of a large graph. Chapter 4 introduced a methodology for manipulating this graph structure to support TDUI.

Chapter 4 presents a development method for achieving multi-path development of UIs.

This development method decomposes any development activity (i.e, a development scenario) in a series of *development steps* consisting in the transformation of the artifact(s) in the scope of a *development stage* (here referred as *viewpoint*) into other development artifacts. In this context, a *development path* is defined as an archetypal composition of development steps. We identified three typical development paths: forward engineering, reverse engineering, and context (of use) adaptation.

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.

Development steps have been further decomposed into *development sub-steps*. A development sub-step realizes one 'concern' of the transformation process at a time. For instance, the definition of the dialog control, the definition of the navigation, or the selection of appropriate interactors.

To enable an expression and an execution of the development steps, each sub-step populating a step may be associated with a so-called *transformation system*, itself decomposed into *transformation rules*.

Transformation systems and transformation rules are *conditional graph rewriting* rules sequentially composed into grammars. Conditional graph rewriting and graph grammars are advantageous in our context as they propose a declarative syntax, an reasonable computational power, a formally defined execution semantics, an appealing graphical syntax, a high degree of modularity, and last but not least, they perfectly integrate with our ontological framework as transformation rules are composed of fragments of specifications (i.e., patterns).

Transformation rules and transformation systems may be stored in a textual format to enable their capitalization in a sort of transformation catalogs called *development library*.

A collection of tools has been introduced in Sec. 4-7 and annex 1. These tools materialize our approach and show how each viewpoint of our framework can be edited and transformed. The existence of this collection of tools contributes to the requirement of tool interoperability.

6.3 Validation

6.3.1 External Validation

External validation is realized by the application of our method on case studies. The main goal of these case studies is to show the feasibility i.e., the capability to solve the problems raised by the presented case studies.

The two case studies presented in this chapter show how multi-path development applies to low to mid-complex examples of both graphical and auditory interfaces.

Our case studies showed the **feasibility** of developing a UI in a principled-based and rigorous manner relying on explicit transformation catalogs at any time. The diversity of development paths that have been presented highlights the possibility of manipulating UI related artifacts according to different development scenarios and paves the way to consider multiple other alternatives. In particular, new development scenarios can be developed by refinement (e.g. a more elaborated scenario), by composition (e.g., a new scenario by composing several existing scenarios), by transformation (e.g., a newly defined scenario by deriving other forms of scenarios from existing ones) or by reusing. The reuse of transformations has been illustrated when transformation systems have been straightforwardly reused from one case study to another one. As so, we avoid ad hoc development catalogs and enable a capitalization of transformations in a consolidated approach while trying to avoid the proliferation of scenarios that are close to each other.

6.3.2 Internal Validation

The internal validation of a methodology consists in assessing its characteristics against a set of selected criteria. The relevant criteria, called requirements, for our methodology have been elicited and motivated after the state of the art of Chapter 2. This section proposes a discussion for each of these requirements.

6.3.2.a Ontological Requirements

Requirement 1: Ontological explicitness – states that our ontology should be defined externally to any methodology manipulating it and in an explicit way that facilitates its dissemination and manipulation among stakeholders. (Motivation: Short. 1).

Discussion: Ontological explicitness has been fully achieved. An ontology for UI specification has been presented in Chapter 3. This ontology has been defined independently of any process manipulating it. Any external agent is able to learn our ontology, access its inner concepts, structure and logic.

Requirement 2: Expressivity – means that a conceptual framework should provide enough details to address problems that motivated the elicitation of its constituent concepts. In our context models should, at least, provide enough details to allow an implementation of the system it describes. This essential requirement is not fulfilled by many formal methods, for instance those focusing on verifying state properties of the system that is being built.

Discussion: The expressivity of our ontology can be assessed with several arguments:

▪ Concepts at the CUI provide enough details to enable the generation of a final UI for several toolkits, including HTML, XHTML, Flash DRK6, Java AWT, Java Swing, to name a few.

▪ Concepts at AUI provide enough details to enable the generation of a CUI for several modalities. This has been namely illustrated in Chapter 4 and in the second case study of Chapter 5.

▪ The expressivity of the domain model is the one of UML class/object diagrams (although several concepts were added to enable UI derivation). It is out of the scope of this dissertation to discussion of the expressivity of such notation.

▪ The expressivity of our task model outweighs the one of CTT, the reference formalism that was chosen to represent user's tasks. This formalism has proven unsatisfactory for several reasons: the expression of leaf task is not precise (or constrained) enough to enable the derivation of precise UI specifications, the connection to the domain model remain somewhat vague (they are only done using a reference to a textual list of objects), the LOTOS temporal relationships give rise to interpretation arguments between experts. Yet, CTT remains the most popular notation in TDUI for user task specification.

▪ Inter-model relationships increase dramatically the expressivity of each model taken individually.

Lack of expressivity could appear in the future along with the application of our method for UI types or development context for which it was not thought for.

Requirement 3: Human readable – means that the provided ontology should be proposed in a format that enables its legibility by a human agent.

Discussion: The assessment of this requirement is somewhat tempered. Indeed, our graphical syntax has been proved efficient for specifying rules. The examples of Chapter 4 and 5 show the appropriateness of this formalism for human use. On the other hand when it comes to visualize models, graphical notation becomes hard to manage as the number of nodes and relationships grows. That is why tools presented in Chapter 4 use UML, CTT, WYSIWYG UI representations instead of a collection of nodes and edges. Property sheets allow us to provide and visualize complementary details of a specification. On the other hand, the XML syntax, called UsiXML, is not intended for direct human usage. UsiXML allows a grasp on the structure of specification models (e.g., for visualizing hierarchy of elements) but is very hard to read when it comes to interpret relationships (the reader has to search the Id of the nodes in relation in the specification).

Requirement 4: Formality – states that models are expressed in such a level of accuracy that it enables automatic reasoning on their properties. (Motivation: Short. 2, 4).

Discussion: The formality of our ontology relies in its abstract syntax i.e., a mathematical graph structure (i.e., directed, identified, labeled, constrained, typed graphs). This graph structure is built by a progressive consolidation of an initial simple graph category thanks to graph morphisms. This allows us to benefit from any theoretical result proved for a simple graph category.

Requirement 5: Machine readable – states that the proposed ontology should be legible by a machine.

Discussion: This requirement is completely met by the definition of an XML syntax enabling the expression of the concepts of our ontology and in compliance with the abstract syntax defined for this ontology. The collection of tools that manipulate UsiXML format is an evidence of the machine readability of this syntax.

Requirement 6: Ontological separation of concern – states that models should differentiate aspects of the problem at hand [Parna72,Dijk76]. Models defined in our methodology should capture and, segregate, different levels of abstractions.

Discussion: The concepts of viewpoint and UI model allow a segregation of the concepts of our ontology into different abstraction layers. The CUI viewpoint is the toolkit independent level, the AUI level is the modality independent level, the Task and Domain viewpoint is the computation independent level.

Requirement 7: Verifiability of specification – is defined as: "the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases" [Meye97]. Applied to specification, verifiability refers to the possibility of checking easily desirable properties (e.g., consistency, usability criteria).

Discussion: This requirement is facilitated by formality and explicitness. Verifiability has not been addressed, per se, in the context of this dissertation. The fact that our ontology has been defined explicitly and formally facilitates its verification.

Requirement 8: Ontological homogeneity – refers to the property for a set of concepts of being defined using a common syntax. All models concepts should be described in a single formalism that facilitates their integration and processing.

Discussion: This requirement is met by the definition of all our concepts within a single abstract syntax. This requirement has been a major motivation for choosing graphs as a representational structure.

Requirement 9: Reuse of specifications – refers to the possibility of reusing whole or a part of a specification for another system. The proposed framework should facilitate reusing specifications.

Discussion: The fact that any specification can be exchanged using an XML syntax facilitates reuse of specifications. The ability of transforming theses specifications with a set of transformation rules increase the possibilities for reusing the specifications. Yet, this dissertation has not addressed the problem of providing meta-descriptors for indexing specification fragments. The use of specification chunks as a pattern language should be addressed in our future works.

Requirement 10: Ontological extendibility – refers to the ease of adapting a conceptual structure to the occurrence of newly elicited concepts. HCI is a vast area covering the definition of multiple types of interfaces, interaction techniques, and interaction contexts. A specification language should be equipped with

extension mechanisms to allow its evolution in parallel with the artifact it seeks to model. This property is particularly relevant in the domain of TDUI as new widgets, interaction devices, techniques and styles are constantly appearing.

Discussion: Extendibility is facilitated by several characteristics of our ontology:

- Modularity of our framework. Each model of our framework is defined independently of the others (application of separation of concerns). Extending our methodology to other models is possible by simply defining a conceptual schema for this model and translating it into our concrete syntax. New inter-model relationships can be defined to relate this newly introduced artifact with the rest of our ontology. New rules may then be defined to take this new model into account in the transformation process.

- Structuring of models. Each model is based on partial orders of node types and edge types. This clear structuring facilitates the introduction of new concepts while not endangering the existing structure of models, thus implying some ontological stability. For instance, CIO types are sub-typed into graphical CIO and auditory CIO, CUI relationships are partitioned in graphical relationship and auditory relationship. Adding 3D CIOs and 3D relationships would consist in adding a type for CIO and a type for CUI relationships.

Requirement 11: Standards – states that the expression means that the rules used to represent our ontology should rely on well accepted standards in the software engineering community.

Discussion: The expression of the conceptual schemas relies on UML class diagrams. Our textual concrete syntax relies on XML schemas. The advantages of these respective techniques remain valid in our approach.
Methodological Requirements

Requirement 12: Methodological explicitness – states that the constituent steps of our methodology should be defined in a way that facilitates the comprehension of its internal logic and its application.

Discussion: Methodological explicitness is guaranteed by several factors:
Ontological explicitness is a pre-requisite of methodological explicitness.
Decomposition of development paths into development steps, and sub-steps.

Existence of a well-defined syntax for expressing methodological steps.

Requirement 13: Methodological flexibility – refers to the ability to initiate the development from any development stage (i.e., multiple entry points) and to terminate it at any development stage (i.e., multiple exit points).

Discussion: Methodological flexibility has been demonstrated in Chapter 4 and 5. It is improved by several factors:
- Ontological separation of concerns.
- Our ontology was designed to allow an exploitation of models independently of the definition of other models. For instance it is possible to initiate development from any model, single or combined with others, and to terminate it similarly with an equivalent degree of freedom.

Requirement 14: Methodological formality – states that development steps should be expressed in such a level of accuracy that it enables an unambiguous interpretation of the process they describe.

Discussion: the level of formality of our methodology is tantamount to the one of conditional graph rewriting and graph grammars. Therefore no room is left for another interpretation of development steps and sub-steps that is not allowed by this formalism.

Requirement 15: Executability – states that development steps should be expressed in such a level of accuracy that it is possible to execute them by an automaton.

Discussion: Using conditional graph rewriting and graph grammars enables the executability of our transformation rules. The fact that each development sub-step is coupled with one transformation system ensures the executability of each transformation step.

6.3.2.b Methodological Requirements

Requirement 16: Methodological separation of concerns. – refers to a partitioning of methodological steps according to the process types they realize.

Discussion: Methodological separation of concerns is one property that is satisfied by the underlying concepts of our methodology: viewpoint, development stage, development path, development step and development sub-step. These methodological concepts have been dissociated from the concepts realizing them i.e., transformation systems and transformation rules.

Requirement 17: Methodological extendibility – refers to the ability left to the designer to extend the development steps proposed in a methodology.

Discussion: Transformation systems and transformation sub-steps proposed in Chapter 4 and 5 are only possibilities of realizing different development paths. Our methodology allows the introduction of new development sub-steps and/or new transformation systems for realizing sub-steps, thus encouraging the exploration of alternatives for each sub-step.

Requirement 18: Methodological homogeneity – refers to the property of methodological steps of being defined using a common syntax. All transformation steps should be described in a single formalism that facilitates their understanding and processing.

Discussion: This requirement is met for any model-to-model transformation. It could be regretted that, in EHCI, graph grammars have not yet been used in the model-to-code and code-to-model steps.

Requirement 19: Predictability – refers to the possibility provided by a methodology to foretell the result of the application of development steps.

Discussion: Predictability is positively impacted by several elements.
- The execution of rules relies on an explicit and formal execution semantics.
- The application strategy of grammars defined in Sec. 4.3.5 ensures the confluence of our grammars.
- The designer is at any time able to access and modify the transformation systems and sub-steps definitions.
- Methodological steps may be applied step by step.

Consequently, our development process is totally transparent to the designer. The only barrier to predictability remains the knowledge prerequisites from the designer's side.

Requirement 20: Traceability – is defined [IEEE90] as the "degree to which a relationship can be established between two or more products (i.e., here models) of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another".

Discussion: A set of inter-model relationships have been introduced in Chapter 3 to enable the expression of relationships of elements across viewpoints. In sec 4-4, 4-5, 4-6, these relationships have been used to ensure traceability of the application of transformations i.e., it is possible to say, using these relationships, which model element is derived from another one. Our solution meets the desired requirement, although it can be regretted that these relationships have to be part of the rule expression itself. These traceability relationships could be produced automatically.

Requirement 21: Correctness – can be defined as the ability of a software to perform their exact tasks [Meye97]. In the context transformational development, correctness can be defined as the adequacy of an artifact A with respect to the other artifact(s) B such that B is the source artifact that was used to derive A.

Discussion: Correctness is a relative notion depending on the context in which it is addressed. Two types of correctness can be considered: Syntactic (structural) correctness and semantic correctness [Varro02b]. Syntactic correctness states that for any well-formed source model, any transformation rule produces a well-formed target model. Syntactic correctness is guaranteed by construction within our framework by the fact that all our transformations are type preserving. 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. A graph of types may also be accompanied with the expression of specific consistency constraints inexpressible within the graph of types. Object Constraint Language (OCL) is used for this purpose in [Agra03], pre- and post-condition with graph patterns are used in [Akeh03]. Semantic correctness states that a semantic adequacy between a source and a target model (this corresponds to the definition given for Req. 21). In our context, proving semantic correctness is hard as, by definition, the domain of discourse of source model and target model are different. Furthermore, a designer is allowed to define her own transformation rules, a correctness proof would have to be instantiated for each newly defined rule.

Requirement 22: Support for tool interoperability – Tool interoperability refers to the possibility of reusing the output provided by a tool into another tool. Our method should foster interoperability of tools working on specification models e.g., editors, critiquing tools, code generators, interpreters.

Discussion: Support for tool interoperability is positively impacted by a common UI description language that is shared among tools (UsiXML) and that the coverage of UsiXML is large enough to accommodate multiple tools. When new concepts need to be introduced in our ontology, the support of new tolls can be maintained by relying on ontological extendibility.

Requirement 23: Methodological reuse – refers to the possibility in a methodology to capitalize on the knowledge defined by designers to perform development steps and re-using this knowledge for other developments.

Discussion: The reuse of transformations has been illustrated by the fact that transformation systems can be reused from one case study to another one. When no possibilities exist for reusing a transformation sub-step, the methodological extendibility enables the definition of new transformations.

6.4 Summary of Contributions

The contributions of this work can be summarized depending on the type of audience it might affect:

- The intended audience of this dissertation is the research community and those persons responsible for development methodologies in organizations. This dissertation provides a mean of **expression**, **structuring**, and **execution** of a TDUI realizing various development scenarios so as to support multi-path development of user interfaces.

  - The **expression** of transformation catalogs realizing TDUI relies on an explicit and formal definition of concepts partitioned in different viewpoints, each maintaining a particular insight on UI systems. Transformations themselves are formally expressed with conditional graph rewriting and graph grammars. This expression enables and exchange of transformation catalogs among the research community and thus fosters incremental research and development efforts.

– The **structuring** of the development process is ensured by the introduction of the following concepts: development path, development step, development sub-step. Both the underlying concepts and their articulation together structure TDUI so as to enable a composition of development steps into relevant development paths.

– The **execution** of transformation catalogs realizing TDUI is defined by the execution semantics provided for programmed graph rewriting. Development sub-steps are associated with development systems which in turn are composed of transformation rules.

Although not initially intended, this dissertation is also beneficial for other communities such as:

- HCI designers and developers:

  – **Expression** of design knowledge. The design knowledge that is tacitly and implicitly maintained in the head of designers is made explicit, thus identifying potential gaps.
  – **Formalization** of design knowledge and models used to design UIs. This allows identifying potential conflicts, contradictions and underspecified aspects of the design knowledge used to design UIs in the everyday practice.
  – **Communication** of design knowledge and models. Once explicit the design knowledge can be communicated and consolidated among designers. This fosters consistency in the development efforts across development projects in an organization.

- Software engineering community. Those interested in transformational approaches in software engineering will find in this dissertation an **application** of this paradigm to a specific problem domain.

- Graph transformation community. The graph transformation community consolidates its theoretical fundamentals since the late sixties. This dissertation provides a significant **application** of these fundamentals to a domain that remains unprecedented.

6.5 Future works

A lot of things remain to be done around the framework presented in this dissertation. We point out the following things as the most interesting issues for us:

Extend the ontology to other types of UI: the ontology that was proposed only covers concrete user interfaces for 2-D graphical interfaces and auditory user interface. One could consider extensions to 3-D UIs, virtual reality, mixed reality and tangible interfaces. The extension to multi-modal UIs (i.e. UIs where modalities are intertwined) could also be considered.

Extend the ontology to other types of model and concepts: Models in the scope of our ontology have been defined in a modular manner. Right now, the most desirable model extension we'd like to do is to consider UML workflow models for replacing CTT models. Workflow models present many advantages with respect to CTT and offer a very appropriate notation for collaborative applications. Another model that could be consolidated is the context model. For instance, a context model for multi-surface and distributed user interface could be taken into account [Lach04].

Define high-level building blocks for supporting design. As we dispose of a language for expressing a wide range of concepts describing UIs, an idea could be to define high-level building blocks that would overweight the level of individual concepts themselves. Like proposed by [Fowl01] with his analysis patterns [Fowl96], it is possible with our textual syntax to store UI descriptions corresponding to commonly found UI across various business areas. These "patterns" could be defined at different levels of our ontology. At domain level: reusing existing OO patterns could even be considered. At task level: Molina [Moli03] identified several task-based patterns that could be expressed using our syntax. More complex patterns could couple domain patterns and task patterns. At abstract and concrete UI levels, specification chunks could describe domain specific UI parts like a form for a registration, or a payment window. This raises the problem of indexing these specification chunks, retrieving them and assembling them in a meaningful manner.

Extend methodological steps definition. The decomposition of the development process proposed in this dissertation is probably not adequate for all

development situations. New sub-steps could be identified depending on the application type and the development context.

Expand the flexibility of development steps. This may be done by introducing richer control structures to pilot the application of rules (e.g., loops, conditional structures), and by introducing human intervention for controlling methodological steps during their execution.

Extend catalogs of transformations. This dissertation has shown that for a same sub-step definition alternate transformation systems could be applied. Our transformation catalogs are certainly not exhaustive and could be enriched with new transformation systems.

Extend formal foundations to improve the articulation of sub-steps. Considering the difficulties to order sub-steps within steps, it would be worth to define a formal foundation of the expression of sub-step pre and post conditions. Such formal expression could rely on the theory of abstract proving that automatically generates a valid post condition by applying a transformation on a precondition, thus allowing a checking of a property of *correctness by construction* [Abra87].

Validate catalogs of transformation by human factors experts. This may be done by: (1) empirical testing of transformation catalogs by human factors expert conducting studies to discover, introduce or modify transformation rules within a same transformation system or across (2) usability testing of UIs produced by our application of the TDUI paradigm (3) external validation of our method on more case studies with varying parameters (4) consider the incorporation of usability properties in the transformation process itself.

Hide the complexity. It might be argued that conditional graph rewriting and graph grammars require, from the designer's side, a substantial knowledge in formal methods. Interesting works could lead to a decrease in complexity in the expression of transformation rules. For instance, generating traceability links automatically could be considered.

Embed transformation systems in run-time scenarios. The transformation process proposed in this work was imagined at design time. Nonetheless, no elements of our methodology prevent to consider the application of transformation systems at run-time. This might be considered for scenarios such

as dynamic (re-)allocation of tasks, dynamic adaptation to changes in the context of use, partial or total migration, and distributed UIs.

Thanks for reading this document or at least this last page !

References

A

[Abra87]
Abramsky S., and Hankin C., Abstract Interpretation of Declarative Languages. Ellis-Horwood, Chichester, 1987.

[Abra99]
Abrams M., Phanouriou C., Batongbacal A. L., Williams S., and Shuster J., *UIML: An Appliance-Independent XML User Interface Language*, in Mendelzon A. (Ed.), Proceedings of 8th International World-Wide Web Conference WWW'8 (Toronto, May 11-14, 1999), Elsevier Science Publishers, Amsterdam, 1999. Available online: http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html.

[Abri98]
Abrial J.-R., The B-Book: Assigning Programs to Meanings, Cambridge
University Press, Cambridge, 1996.

[Abow91]
Abowd G., *Formal Aspects of Human-Computer Interaction*, PhD thesis, University of Oxford, 1991.

[Agra03]
Agrawal A., Karsai G., and Ledeczi A., *An End-to-end Domain-Driven Software Development Framework*, in Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, OOPSLA'03 (Anaheim, October 26-30), ACM Press, New York, 2003, pp. 8–15.

[Akeh03]
Akehurst, D., Kent S., and Patrascoiu O., *A Relational Approach to Defining and Implementing Transformations in Metamodels,* in Software and Systems Modeling 2(4), 2003, pp. 215–239. Available on-line: http:// www .cs.kent.ac.uk/pubs/2003/1764, 2003.

[Alle83]
Allen J. F., *Maintaining Knowledge about Temporal Intervals*, in Communication of ACM, 26, 1983, pp. 123-154.

[Anne00]
Annett, J., Cunningham, D., and Mathias-Jones, P. *A method for measuring team skills*, in Ergonomics, 43, 2000, pp. 1076–1094.

[Anne67]
Annett J., and Duncan K., *Task analysis and training design* in Occupational Psychology, 41, 1967, pp. 211-227.

[Aren91]
Arens Y., Miller L., and Sondheimer N. K., Presentation Design Using an Integrated Knowledge Base, Addison-Wesley, Reading, 1991, pp. 241-258.

[Azev00]
Azevedo P., Merrick R., and Roberts D., *OVID to AUIML - user-oriented interface modelling* in Nunes N. (Ed.), *Proceedings* of 1st International Workshop "Towards a UML Profile for Interactive Systems Development" TUPIS'00 (York, October 2-3, 2000), 2000. Accessible at http://math.uma.pt/tupis00/submissions/azevedoroberts/azevedoroberts.html.

B

References

[Baad98]

Baader F., Nipkow T., Term Rewritting and all that, Cambridge University Press, Cambridge, 1998.

[Bail03]

Bailey, B.P., and Konstan, J.A. Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design. Proc. of the ACM Conference on Human Factors in Computing Systems CHI'2003 (Fort Lauderdale, April 2003). ACM Press, New York, 2003, pp. 313-320

[Barc99]

Barclay P. J., Griffiths T. , McKirdy J., Paton N.W., Cooper R., and Kennedy J., *The teallach tool : Using models for flexible user interface design*, in Vanderdonckt J. (Ed.), Proceedings of CADUI'99, Kluwer Academic, 1999, pp. 139-157.

[Ball00]

Ball T., Colby Ch., Danielsen P., Jagadeesan L. J., Jagadeesan R., Läufer K., Matag P., and Rehor K., *SISL: Several interfaces, single logic*, Technical report, Loyola University, Chicago, January 6th, 2000.

[Balz76]

Balzer R., Goldman N., Wile, D.,   *On the Transformational Implementation approach to programming*, in Proceedings of the 2nd International Conference on Software Engineering ICSE '76 (San Francisco, California, United States), 1976, pp. 337-344.

[Balz93]

Balzert, H., Der JANUS-Dialogexperte: vom Fachkonzept zur Dialogstruktur, in Proceedings der GI-Fachtagung Softwaretechnik SoftwareTechnikTands'93*,* Dortmund, 1993, pp. 62-72.

[Balz95]

Balzert, H., *From OOA to GUI - The JANUS-System*, in Nordbyn, K., Helmersen, P.H., Gilmore, D.J., Arnesen, S.A. (Eds.), Proceedings of the Fifth IFIP TC13 Conference on Human-Computer Interaction *INTERACT'95* (Lillehammer, June 25-29, 1995), Chapman & Hall, London, 1995, pp. 319–324.

[Balz96]

Balzert H., Hofmann F., Kruschinski V. , and Miemann C., *The JANUS application development environment - generating more than the user interface*, in Vanderdonckt J. (Ed.), Proc. Of the 2nd Int. Workshop on Computer-Aided Design of User Interfaces CADUI'96 (Namur 5-7 June 1996), Namur University Press, Namur, 1996, pp. 183-206.

[Bare02]

Baresi, L., Heckel R., *Tutorial Notes on Foundations and Applications of Graph Transformation*, An introduction from a software engineering perspective, First Int. Conference on Graph Transformation ICGT'02 (7-12 september 2002, Barcelona, Spain), Electronic Notes in Computer Science, 2002.

[Bart88]

M.-F. BARTHET, *Logiciels interactifs et ergonomie*, Dunod Informatique, Paris, 1988

[Bart95]

Barthet M. F., *The DIANE method and its connection with MERISE method*, in Proc. of IEA World Conference '95 (Rio de Janeiro), 1995.

[Bart96]

Barthet, M.-F., and Tarby, J.-C.*, The Diane+ method*, in Vanderdonckt J. (Ed.), Computer-aided design of user interfaces, Presses Universitaires de Namur, Namur, 1996, pp. 95–120.

[Baum00]

Baumeister L. K., John B. E., and Byrne M. D., *A comparison of tools for building GOMS models tools for design*, in Proc. of ACM Confereence On Human Factors in Computing Systems CHI'2000, ACM Press, New York, 2000.

References

[Bear96]

Beard, D. V., Smith, D. K., and Denelsbeck, K. M., *QGOMS: A direct-manipulation tool for simple GOMS models*, in Proceedings of ACM Conference on Human Factors in Computing Systems CHI '96, ACM Press, New York, 1996, pp. 25–26.

[Bend83]

Bendas J.B., Design through transformation, in Proceedings of the 20th ACM IEEE conference on Design Automation Conference (Miami Beach, Florida, United States), IEEE Press, 1983, pp. 253-256.

[Bett02]

Bettin, J., *Measuring the Potential of Domain-Specific Modelling Techniques*, in Proceedings of the 2nd Domain-Specific Modelling Languages Workshop DSVL'2002, satellite event of OOPSLA 2002, Working Papers W-334, Helsinki School of Economics, 2002, pp. 39-44. Available on-line: http://www.cis.uab.edu/info/OOPSLA-DSVL2/Papers/Bettin.pdf

[Bhar95]

Bharat K. A., and Hudson S. E., Supporting distributed, concurent, one-way constraints in user interface applications in Proceedings of the 8th ACM Syposium on User Interface and Software Technology, ACM press, 1995, pp. 121-132.

[Boda94]

Bodart F. , Hennebert A. , Leheureux J. , Provot I. , Sacré B. , and Vanderdonckt J., *A model-based approach to presentation: A continuum from task analysis to prototype*, in Paternò F (Ed.), *Proceedings of 1st Eurographics Workshop on Design, Specification, Verification Od Interactive Systems DSV-IS'94 (Carrara, June 8-10, 1994)*, pages 25-39, Vienna, 1994. Eurographics Series.

[Boda94b]

Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Vanderdonckt, J., *Towards a Dynamic Strategy for Computer-Aided Visual Placement*, in Proc. of 2nd ACM Workshop on Advanced Visual Interfaces AVI'94 (Bari, 1-4 June 1994), T. Catarci, M.F. Costabile, S. Levialdi & G. Santucci (Eds.), ACM Press, New York, 1994, pp. 78-87.

[Boda95a]

Bodart F. , Hennebert A. , Leheureux J. , Provot I. , Vanderdonckt J. , and Zucchinetti G. , *Key activities for a development methodology of interactive applications*, in Benyon D. and Palanque P. (Eds.), Critical Issues in User Interface Systems Engineering (London), Springer-Verlag,1995, pp. 109-134.

[Boda95b]

Bodart F. , Hennebert A. , Lheureux J. , Provot I. , Sacré B. , and Vanderdonckt J., *Towards a systematic building of software architecture: The TRIDENT methodological guide* in Proceedings of 1st Eurographics Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'95 (Vienna), Springer Verlag, 1995.

[Boda95c]

Bodart F., Hennebert A.-M., Leheureux J.-M., Vanderdonckt J., *Computer-Aided Window Identification in TRIDENT*, in Nordbyn K., Helmersen P.H., Gilmore D.J.and Arnesen S.A. (Eds.), Proceedings of Fifth IFIP TC 13 International Conference on Human-Computer Interaction INTERACT 95 (Lillehammer, 27-29 June 1995), , Chapman & Hall, London, 1995, pp. 331-336.

[Boeh84]

Boehm W. , Gray T.E., and Seewaldt T., *Prototyping versus specifying: A multiproject experiment* in IEEE Transactions on Software Engineering, 10(3), 1984.

[Booc03]

Boocock, P., *The Jamda Project*, 6 May 2003. Available online: http://jamda.source forge.net/.

[Boms98a]

References

Bomsdorf B. and Szwillus G., From task to dialogue: Task-based user interface design, in *SIGCHI Bulletin 30*, 1998, pp. 40-42. Available online: http://www.acm.org/sigchi/bulletin/ 1998.4/szwillus.html.

[Boms99a]
Bomsdorf, B.,and Szwillus, G., *Tool support for task-based user interface design.*, SIGCHI Bulletin, 31(4), 1999, pp. 27–29. Available online: http://www.uni-paderborn.de/cs/ag-szwillus/chi99/ws/

[Boms99b]
Bomsdorf B., and Swillius G., *CMF a coherent modelling framework for task-based user interactive design*, in Vanderdonckt J., and Puerta A. (Eds*.),* Computer-Aided Design of User Interfaces II, Proceedings of the Third International Conference on Computer-Aided Design of User Interfaces CADUI 99 (Louvain-La-Neuve, Belgium, 21-23 October 1999), Springer Verlag, 1999, pp. 293-311.

[Boui04]
Bouillon, L., Vanderdonckt, J., and Chow, K.C., *Flexible Re-engineering of Web Sites* in Proceedings of 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, Portugal, 13-16 January 2004), ACM Press, New York, 2004, pp. 132-139.

[Breu97]
Breu R., Hinkel U., Hofmann C., Klein C., Paech B., Rumpe B., and Thurner V., *Towards a formalization of the unified modeling language*, in Proceedings of 11th European Conference on Object-Oriented Programming ECOOP'97 *(*Jyväskylä, Finland), Springer Verlag, LNCS, 1997.

[Brow97]
Brown, J., *Exploring Human-Computer Interaction and Software Engineering. Methodologies for the Creation of Interactive Software*, in *SIGCHI Bulletin* 29(1), 1997, pp. 32–35.

[Brow97]
Browne T., Dávila D., Rugaber S., and Stirewalt K., *Using declarative descriptions to model user interfaces with MASTERMIND*, in Paterno F., and Palanque P. (Eds.), Formal Methods in Human Computer Interaction, Springer-Verlag, Berlin, 1997.

[Bunk82],
Bunke, H, *Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation* in IEEE Pattern Analysis and Machine Intelligence, 4(6), November, 1982, pp. 574-582.

[Byrn92]
Byrne E. J., *A conceptual foundation for software re-engineering* in Proceedings of the Conference on Software Maintenance, IEEE Computer Society Press, November 1992 pages 216--235.

C

[Caet02]   Caetano, A., Goulart, N., Fonseca, M. and Jorge, J., *JavaS-ketchIt: Issues in Sketching the Look of User Interfaces*, Proc. of the 2002 AAAI Spring Symposium - Sketch Understanding (Palo Alto, March 2002), 2002, pp. 9-14.

[Calv01a]
Calvary G. , Coutaz J. , and Thevenin D., *Supporting Context Changes for Plastic User Interfaces : A Process and a Mechanism*, in Blandford A. , Vanderdonckt J. , and Gray P. (Eds.), Joint Proceedings of HCI'2001 and IHM'2001 (Lille,10-14 September 2001), Springer Verlag, London, 2001, pp. 349-363.

[Calv01b]
Calvary G. , Coutaz J. , and Thevenin D., *A unifying reference framework for the development of plastic user interfaces*, in Proceedings of IFIP WG 2.7 Conference on Engineering the User Interface EHCI'2001 (Toronto, May 11-13, 2001)., Chapman & Hall, London, 2001.

References

[Calv03]

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J., *A Unifying Reference Framework for Multi-Target User Interfaces* in Interacting with Computers, 15(3), June 2003, pp. 289–308.

[Card83]

Card S.K., Moran T.P., and Newell A., *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, New York, 1983.

[Cast02]

Castro J., Kolp M., and J. Mylopoulos, *Towards Requirements-Driven Information Systems Engineering: The Tropos Project*, in Information Systems, 27, Elsevier, Amsterdam, 2002.

[Crea99]

Crease, M., Gray, P., and Brewster, S.A. *Resource Sensitive Multi-Modal Widgets,* in Volume II of the Proceedings of INTERACT '99 (Edinburgh, UK), British Computer Society, 1999, pp. 21-22.

[Chatt99]

Chatty S., and Dewan P. (Eds.), *Engineering for Human-Computer Interaction*, Kluwer Academics, 1999.

[Chea81]

Cheatham T. E., Holloway G. H., and Townley J. A., *Program Refinement By Transformation*, in Proceedings of the 5th international conference on Software engineering International Conference on Software Engineering ICSE 81 (San Diego, California, United States), IEEE press, 1881, pp. 430-437.

[Chik90]

Chikofsky E.J. and Cross J.H., *Reverse Engineering and Design Recovery: A Taxonomy* in IEEE Software, 1(7), January 1990, pp. 13-17.

[Chom56]

Chomsky N., *Three models for the description of language* in IRE Transaction on Information Theory, 2, 1956, pp. 113-123.
Workshop on Source Code Analysis and Manipulation SCAM'01 (Florence, 10 Nov. 2001), IEEE Computer Society Press, Los Alamitos, 2001, pp. 168-178.

[Cons99]

Contantine, L., and Lockwood L., *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design,* Addison-Wesley, Reading, 1999.

[Cons03]

Constantine L. L., *Canonical Abstract Prototypes for Abstract Visual and Interaction* in Proceedings of the 10th International workshop on Design, Specification and Evaluation of Interactive Systems DSV-IS 2003 (june 11-13 2003, Funchal, Portugal), LNCS 2844, Springer Verlag, Berlin, 2003, pp. 1-15.

[Cord01]

Cordy J.R., Dean T.R., Malton A.J. and Schneider K.A., Software Engineering by Source Transformation - Ex-perience with TXL, Proc. of IEEE 1st Int.

[Corr97]

Corradini, Ehrig H. , Heckel R. , Korff M. , Löwe M. , Ribeiro L. , and Wagner A., *Algebraic approaches to graph transformation - part I: Single pushout approach and comparison with double pushout approach*, in Rozenberg G. (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations, World Scientific,1997, pp. 247-312.

[Cout87]

Coutaz J., PAC, *An Implementation Model for Dialog Design* in Proceedings of Interact `87 (Stuttgart, September 1987), 1987, pp. 431-436.

[Cout02]

References

Coutaz J., Limbourg Q., Loubna I., Paternò F., Santoro, C., and Vanderdonckt J., XML specification for User Interface Modeling Language, Cameleon Project Delivrable 1.3, 2002.

[Coye04]

Coyette A., Faulkner S., Kolp M., Limbourg Q., and Vanderdonckt, J., *SketchiXML: Towards a Multi-Agent Design Tool for Sketching User Interfaces Based on USIXML*, IAG, July 2004, submitted to publication.

[Clar99]

Clark J., *XSL: Transformations (XSLT). version 1.0 W3C recommendation*, Technical report, W3C, 1999. Available online: http://www.w3.org/TR/xslt.

[Czar00]

Czarnecki, K., and Eisenecker, U.W., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Reading, 2000.

[Czar03]

Czarnecki, K., and Helsen S., Classification of Model Transformation Approaches, in Online Proceedings of the OOPLSLA'03 workshop on Generative Techniques in the Context of Model Driven Architectures, 2003, avalaible at http://www.softmetaware.com/oopsla2003/mda-workshop.html.

D

[Depk02]

Depke, R., Heckel, R., and Küster, J.M., *Formal Agent-Oriented Modeling with UML and Graph Transformation*, in Science of Computer Programming, 44(2), August 2002, pp. 229–252.

[Diap90]

Diaper D., *Task analysis for knowledge descriptions (TAKD): The method and examples* in Diaper D. (Ed.), Task Analysis for Human-Computer Interaction, Ellis-Horwood, 1990, pp. 108-159.

[Dijk72]

Dijkstra, E. W., *The Humble Programmer*, in Communication of the ACM, 15(10), 1972, pp. 859-866.

[Dijk76]

Dijkstra, E. W., *The discipline of programming*, Prentice Hall, Engelwood Cliffs, NJ, 1976.

[Ditt00]

Dittmar A., *More precise descriptions of temporal relations within task models*, in Pallanque P. and Paternó F. (Eds.), Proc. of the 7th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'00 (Limerick, 5-6 June 2000), volume 1946 of *Lecture Notes in Computer Science*, pages 151-158, Berlin, 2000. Springer Verlag.

[Dix90]

Dix A. J., *Non-determinism as a paradigm for understanding the user interface* in Thimbleby H. W., and Harrison M. D. (Eds.), Formal Methods in Human-Computer Interaction, Cambridge University Press, Cambridge, pp. 97-127.

[Dix91]

Dix, A. J., *Formal methods for interactive systems*, Academic Press, London, 1991.

[Dix98]

Dix A., Finalay J., Abowd G., and Beale R., *Human-Computer Interaction*, , 2nd edition, Prentice Hall Europe, London, 1998.

[dMof02]

dMoF*, dMOF 1.1, An OMG Meta Object Facility Implementation*, The Corba Service Product Manager, University of Queensland. Available online: http://www.dstc.edu.au/Products/CORBA/MOF/, may 08 2002.

References

[Dsou99]

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

E

[Ecks98]

       R. Eckstein. *Java Swing.* O'Reilly, 1st edition, 1998.

[Ehri73]

       Ehrig H., Pfender M., and Schneider H. J., Graph Grammars and algebraic approach, in 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 197-180.

[Ehri79]

       Ehrig H., *Introduction to the algebraic theory of graph grammars - a survey*, in Proceedings International Workshop on Graph Grammars and their Applacation to Computer Science and Biology, Springer-Verlag, 1979, pp. 1-69.

[Ehri86]

       Ehrig H., and Habel A., *Graph grammars with application conditions* in
Rozenberg G., and Salomaa A. (Eds.), The Book of L, Springer-Verlag, 1986. pp. 87-100.

[Ehri99]

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

[Eise00]

       Eisenstein J. , Vanderdonckt J. , and Puerta A., *Adapting to mobile contexts with user-interface modeling*, in Proceedings of IEEE Workshop on Mobile Computing Systems and Applications WCSMA'2000 (Monterey, December 7-8, 2000), IEEE Computer Press, Los Alamitos, 2000, pp. 83-92.

[Eise01]

       Eisenstein J. , Vanderdonckt J. , and Puerta A. . *Applying model-based techniques to the development of UIs for mobile computers*, in Proceedings of ACM Conference on Intelligent User Interfaces IUI'2001 (Albuquerque, January 11-13, 2001), ACM Press, New York, 2001, pp. 69-76.

[Elwe95]

       Elwert T. and Schlungbaum T., *Modelling and generation of graphical user interfaces in the TADEUS approach*, in. Bastide R., and Palanque P. (Eds.), Design, Specification and Verification of Interactive Systems DSV-IS '95, Springer-Verlag, Wien, 1995, pp. 193-208.

[Engl99]

       Englebert V., Hainaut J.-L., DB-MAIN: A Next Generation Meta-CASE, in Information Systems Journal, 24(2), 1999, pp. 99-112.

F

[Feur97]

       R. Feur. *MFC Programming.* Addisson-Wesley Advanced Windows Series, 1997.

[Fipa00]

       Foundation for Physical Agents, Fipa Personnal Travel Assistance Specification, 2000. Available online: http://ww.fipa.org.

[Flor04]

       Florins M., and Gemo M., *A UA/PROF- based platform model for use in model-based approaches to user interface design*, Salamandre report, March 2004.

[Flor04b]

References

Florins, M., Vanderdonckt, J., Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems, in Proc. of 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, 13-16 January 2004), ACM Press, New York, 2004, pp. 140-147.

[Fokk92]

Fokkinga M.M,. *A gentle introduction to category theory: the calculational approach*, in Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics, University of Utrecht, September 1992, pp. 1-72.

[Fole91]

Foley J. , Kim W. , Kovacevic S. , and Murray K., GUIDE - an intelligent user interface design environment, in Sullivan J. and Tyler S. (Eds.), Architectures for Intelligent Interfaces: Elements and Prototypes, Addison-Wesley, 1991, pp. 339-384.

[Fole95]

Foley J., *History, results and bibliography of the user interface development environment (UIDE), an early model-based system for user interface design and implementation*, in Proceedings of DSV-IS'94, Springer Verlag, Vienna, 1995, pp. 3-14.

[Fowle96]

Fowler M., *Analysis Patterns*, 1st edition, Addisson-Wesley Professional, Reading, 1996.

[Foun00]

Fountain A. and Ferguson P., *Motif Reference Manual*, O'Reilly, 2nd edition, 2000.

[Flan99]

D. Flanagan. *Java Foundation Classes in a Nutshell*. O'Reilly, 1st edition, 1999.

[Freu92]

Freund, R., Haberstroh, B., and Stary, C., *Applying Graph Grammars for Task-Oriented User Interface Development,* in Koczkodaj W.W., Lauer, P. E., and Toptsis, A.A. (Eds.), Proceedings of 4th International Conference on Computing and Information ICCI'92 (Toronto, May 28-30, 1992). IEEE Computer Society Press, Los Alamitos, 1992, pp. 389–392.

G

[Gala93]

Galaxy Application Environment. Visix Software Inc., 11440 Commerce Park Drive, Reston (VA 22091), 1993.

[Gamb97]

Gamboa, F., Scapin D. L., *Editing MAD* task descriptions for specifying user interfaces, at both semantic and presentation levels*, in M. D. Harrison & J. C. Torres, (Eds.), Proceedings of Fourth International Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS '97, Springer-Verlag, Berlin, 1997, pp. 193–208.

[Gerb02]

Gerber, A., Lawley, M., Raymond, K., Steel, J., and Wood, A.: 2002, *Transformation: The Missing Link of MDA*, in Proceedings of the 1st International Conference on Graph Transformation ICGT'02 (Barcelona, October 7-12, 2002), Lecture Notes in Computer Science, Vol. 2505. Springer-Verlag, Berlin, pp. 90–105.

[Geno04]

Genova development Environment, Genera, Trondheim, Norway, 2004. Available online: http://www.genera.no.

[Gind00]

R. Ginda. Writing a mozilla application with and javascript. In *Proceedings of O'Reilly Open Source Software Convention (Monterey, July 19-20, 2000)*, 2000. Accessible at http://www.mozilla.org/docs/ora-oss2000/chatzilla/overview.html.

[Glad04]

References

Glade. Glade homepage, 2004. http://glade.gnome.org/.

[Goul85]

Gould J. D. and Lewis C., *Designing for Usability : Key principles and what designers think*, in Communication of the ACM, 28(3), 1985, pp. 300-311.

[Grae03]

Gaeremynck Y., Bergman L.D., and Lau T., *MORE for Less: Model Recovery from Visual Interfaces for Multi-Device Application Design* in Proceedings of the 8th ACM International Conference on Intelligent User Interfaces IUI'2003 (Miami, 12-15 Jan. 2003), ACM Press, New York, 2003, pp 69-76.

[Gram96]

Gram C. and Cockton G., *Design Principles for Interactive Software*, Chapman & Hall, London, 1996.

[Grah96]

Graham T., Damker H. , Morton C. , Telford E. , and Urnes T., *The clock methodology: Bridging the gap between user interface design and implementation*, Technical Report CS-96-04, York University (Canada), 1996.

[Griff01]

Griffiths, T., Barclay, P., Paton, N.W., McKirdy, J., Kennedy, J., Gray, P.D., Cooper, R., Goble, C., and Pinheiro da Silva, P., *Teallach: a Model-based User Interface Development Environment for Object Data-bases* in Interacting with Computers 14(1), pp. 31–68.

[Grif99]

Griffiths T.A., Barclay P.J., McKirdy J. , Paton N.W., Gray P.D., Kennedy J. , Cooper R. , Goble C.A., West A. , and Smyth M., *Teallach: A model-based user interface development environment for object databases*, in Paton N.W. and Griffiths T. (Eds.), Proceedings of User Interfaces to Data Intensive Systems UIDIS '99, IEEE Press, Los Alamitos, 1999, pages 86-96.

[Grif98]

Griffiths T. , McKirdy J. , Forrester G. , Paton N. , Kennedy J. , Barclay P. , Cooper R. , Goble C. , and Gray P., *Exploiting model-based techniques for user interfaces to database*, in Proceedings of Visual Database Systems (VDB '89), Chapman & Hall, London, 1998, pp. 21-46.

[Gree88]

Green T.R.G., Schiele F. , and Payne S.J., Formalisable Model of User Knowledge in Human-Computer Interaction, Academic Press, 1988, pp. 3-46.

[Grub93]

Gruber T. R., *A translation Approach to Portable Ontologies,* in Knowledge Acquisition, 5(2), 1993, pp. 199-220.

[GTK04]

The GTK home page, 2004. http://www.gtk.org.

H

[Habe96]

Habel A. , Heckel R. , and Taentzer G., *Graph grammars with negative application conditions* in Fundamenta Informaticae, 26(3), 1996.

[Habe01]

Habel A., and Plump D., Computational Completeness of Programming Languages Based on Graph Transformation, in Honsell F., and Miculan M. (Eds.), Foundations of Software Science and Computation Structures (FOSSACS) 2001, Lecture Notes in Computer Science 2030, Springer Verlag, 2001, pp. 230-245.

[Hack98]

References

Hackos, J. T., and Redish, J. C., *User and task analysis for interface design*, Wiley, New York, 1998.

[Hare87]
Harel D., *Statecharts: A visual formalism for complex systems* in Science of Computer Programming, 8, 1987, pp. 231-274.

[Harr90]
Harrison, M.D. and Thimbleby, H.W., editors, *Formal Methods in Human Computer Interaction*, Cambridge University Press, Cambridge, 1990.

[Hart90]
Hartson H., Siochi A., and Hix D., *The UAN: A user-oriented representation for direct manipulation interface design* in ACM Transactions on Interactive Systems, 8(3), 1990, pp. 181-203.

[Hart99]
Hartson, H.R. and Hix, D., *Toward Empirically Derived Methodologies and Tools for Human-Computer Interface Development* in International Journal of Man-Machine Studies, 31(4), 1999, pp. 477–494.

[Heck95]
Heckel R. and Wagner A., *Ensuring consistency of conditional graph grammars - a constructive approach*, in Lecture Notes in Theoretical Computer Science, Springer Verlag, 1995.

[Heck02]
Heckel, R., Mens, T. and Wermelinger, M. (eds.), *Proceedings of the Workshop on Software Evolution through Transformations: Toward Uniform Support throughout the Software Life-Cycle.* Electronic Notes in Theoretical Computer Science **72**(4). On-line: http://www1.elsevier.com/ gejng/31/29/23/127/49/show/Products/notes/index.htt, 2002.

[Heck02b]
Heckel, R., Küster, J.M., Taenzer, G., Confluence of Typed Attributed Graph Transformation Systems, in [Corr02], pp. 161-176.

[Hewe96]
Hewett T. T., Baecker R., Carey T., Gasen J. , Mantei M., Perlman G., Strong G., and Verplank W., *Curricula for human-computer interaction*, Technical Report 608920, ACM Special Interest Group on Computer-Human Interaction Curriculum Development, 1996.

[Ho99]
Ho, W.-M., Jézéquel, J.-M., Le Guennec, A., and Pennaneach, F, *UMLAUT: An Extensible UML Transformation Framework*, in Proceedings of the 14th IEEE International Conference on Automated Software Engineering ASE'99 (Cocoa Beach, October 12-15, 1999), IEEE Computer Society Press, Los Alamitos, 1999, pp. 275–279.

[Hong01]
Hong, J.I., Li, F.C., Lin, J., and Landay, J.A. End-User Perceptions of Formal and Informal Representations of Web Sites, Extended Abstracts of Proc. of ACM Conf. on Human Factors in Computing Systems CHI 2001 (Seattle, March 31-April 5, 2001). ACM Press, New York, 2001.

[Horr98]
Horrocks I., *Constructing the User Interface with Statecharts*, Addisson-Wesley, Harlow, 1998.

[Huds96]
S.E. Hudson and I. Smith. Ultra-lightweight constraints. In *Proceedings of the ACM Symposium on User Interface Software Technology*, pages 147-155, 1996.

I

[IEEE90]

References

IEEE society, *Glossary of Software Engineering Terminology*, IEEE Standard n° 610.12-1990, IEEE press, 1990.

[ISO96]

International Standard Organization, ISO EBNF standard, ISO/IEC 14977:1996(E) document, 1996.

J

[Jame91]

James M.G., *PRODUSER: PROcess for developing USER interfaces*, in J. Karat, editor, *Taking Software Design Seriously*, Academic Press, 1991.

[Jans93]

Janssen, C., Weisbecker, A., Ziegler, *Generating User Interfaces from Data Models and Dialogue Net Specifications*, in Ashlund, S., Mullet, K. Henderson, A., Hollnagel, E., White, T. (Eds.), Proceedings of the ACM Conference on Human Factors in Computing Systems InterCHI'93 (Amsterdam, April 14-19), ACM Press, New York, 1993, pp. 418-423.

[John84]

Johnson, P., Diaper, D., and Long, J., *Tasks, skill and knowledge: Task analysis for knowledge based descriptions* in Proceedings of First IFIP Conference on Human-Computer Interaction Interact '84, Elsevier Science Publishers, North-Holland, 1984, pp. 23–28.

[John88]

Johnson P., Johnson H., Waddington R., and Shouls A., *Task related knowledge structures: Analysis, modelling and application*, in Jones D.M., and Winder R. (Eds.), People and Computers IV (Manchester, 1988), Cambridge University Press, Cambridge, 1988.

[John89]

Johnson P., and Johnson H., *Knowledge analysis of task: Task analysis and specification for human-computer systems*, in Downton A. (Ed.), Engineering the Human-Computer Interface, McGraw-Hill, Maidenhead, 1989, p. 119-144.

[John90]

John B.E., *Extensions of GOMS analyses to expert performance requiring perception of dynamic visual and auditory information*, in Proceedings of ACM Conference on Human Factors in Computing Systems CHI'90 (Seattle, USA), ACM Press, New York, 1990, pp. 107-115.

[John92a]

Johnson P., *Human-Computer Interaction: Psychology, Task Analysis and Software Engineering*, McGraw-Hill, London, 1992.

[John92]

Johnson P., Markopoulos P., and Johnson H., *Task knowledge structures: A specification of user task models and interaction dialogues*, in Proceedings of Task Analysis in Human-Computer Interaction, 11th Int. Workshop on Informatics and Psychology (Schraeding, June 9-11), 1992.

[John96]

John B. E. and Kieras D. E., *The GOMS familly of user interfaces analysis techniques: Comparison and contrasts* in ACM Transactions on Computer-Human Interaction, 3(4), 1996, pp. 320-351.

[John01]

Johnson M., and Dampney, *On category theory as a (meta) ontology for information systems*, Proceedings of the international conference on Formal Ontology in Information Systems FOIS 01 (Ogunquit, Maine, USA), 2001, pp. 59-69.

K

[Kier95]

References

Kieras D.E., Wood S.D., Abotel K., and Hornof A., *GLEAN: A computer-based tool for rapid GOMS model usability evaluation of user interface designs*, in Proceedings of the ACM Symposium on User Interface Software and Technology UIST'95, ACM Press, New York, 1995, pp. 91-100.

[Kier99]

Kieras D,. *A guide to GOMS model usability evaluation using GOMSL and GLEAN3*, Technical report, University of Michigan, 1999.

[Kim93]

Kim, W.C. and Foley, J.D, *Providing High-level Control and Expert Assistance in the User Interface Presentation Design*, in Ashlund, S., Mullet, K. Henderson, A., Hollnagel, E., White, T. (Eds.), Proceedings of the ACM Conference on Human Factors in Computing Systems InterCHI'93 (Amsterdam, April 14-19), ACM Press, New York, 1993, pp. 430-437.

[Kirw92]

Kirwan, B., and Ainsworth, L. K., *A guide to task analysis*, Taylor & Francis, London, 1992.

[Kirw00]

Kirwan B.and Ainsworth L. K., *A Guide to Task Analysis*, Taylor and Francis, London, 2000.

[Kell92]

Kelly C., and Cogan L., *User modelling and user interface design*, in Harrisson M., Monk A., Diaper D. (Eds.), People and Computers, Cambridge University Press, Cambridge, 1992, pp. 227-246.

[Kepp03]

A. Kepple, J. Warmer, W. Bast, *MDA Explained - The Model Driven Architecture: Practice and Promise*, Addison Wesley, 2003.

[Kras88]

Krasner G., Pope S., *A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80* in Journal of Object Oriented Programming, August/September, 1988, pp. 26-49.

[Kusk02]

Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.-G., *An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation*, in Butler, M.J., Petre, L., Sere, K. (Eds.), Proceedings of 3rd International Conference on Integrated Formal Methods IFM'02 (Turku, May 15-18 2002), Lecture Notes in Computer Science, Vol. 2335. Springer-Verlag, Berlin, 2002, pp. 11–28.

L

[Lach04]

Lachenal C., Barralon N., Rey G., Coutaz J., I-AM, *A Middleware for Multi-surface, Multi-instrument, Multi-cursor interaction*, submitted for publication to UIST 04.

[Land96]

Landay, J.A., Interactive Sketching *for the Early Stages of User Interface Design,*. Ph.D. thesis, report #CMU-CS-96-201, Computer Science Department, Carnegie Mellon Uni-versity, Pittsburgh, December 1996.

[Larm01]

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

[Laur95]

Lauridsen, O., *Systematic methods for user interface design* in Engineering for Human-Computer Interaction, Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction EHCI 95(Yellowstone Park, USA, August

References

1995), Bass L. J., Unger, C. (Eds.), IFIP Conference Proceedings 45, Chapman & Hall, 1996, pp.169-188.

[Lee03]

Lee D.H., Ko H.I., and Sung M.Y., *Collaborative Multimedia Presentation Authoring in a 3D Spatio-Temporal Space*, in Proceedings of HCI '03 (Poenix Park, Kangwondo, February 10-13, 2003), 2003, pp. 575-580.

[Lim94]

Lim K. Y., and Long J., *The MUSE Method for Usability Engineering*, Cambridge Series on Human-Computer Interaction, Cambridge University Press, Cambridge (UK), 1994.

[Lim96]

Lim, K.Y.,and Long, J., *Structured task analysis: An instantiation of the MUSEmethod for usability engineering* in Interacting With Computers, 8(1), 1996, pp. 31–50.

[Limb00]

Limbourg Q., Vanderdonckt J., and Souchon N., *The task-dialog and task presentation mapping problem: Some preliminary results*, in Palanque P., and Paternó F. (Eds), Proc.Of the 7th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'00 (Limerick, Ireland, June 5-6 2000), . Springer Verlag, Berlin, 2000, pp. 227-246.

[Limb03a]

Limbourg, Q. and Vanderdonckt, J., *Comparing Task Models for User Interface Design*, in Diaper, D., Stanton, N. (Eds.), The Handbook of Task Analysis for Human-Computer Interaction, Lawrence Erlbaum Associates, Mahwah, pp. 135-154.

[Limb04]

Limbourg, Q. and Vanderdonckt, J., *Transformational Development of User Interfaces with Graph Transformations*, in Proceedings of 5th International Conference on Computer-Aided Design of User Interfaces CADUI'2004 (Madeira, January 14-16, 2004), Kluwer Academics Publishers, Dordrecht, 2004.

[Limb04b]

Limbourg, Q., Vanderdonckt, J., Michotte, B., and Bouillon, B.: 19 February 2004, *TOMATOXML, a General Purpose XML Compliant User Interface Description Language*, TOMATOXML V1.2.0. Working Paper n°105. Institut d'Administration et de Gestion (IAG), Louvain-la-Neuve.

[Limb04c]

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M., and Trevisan, D.:, USIXML: A User Interface Description Language for Context-Sensitive User Interfaces, in Luyten, K., Abrams, M., Limbourg, Q., Vanderdonckt, J. (Eds.), Proceedings of the ACM AVI'2004 Workshop Developing User Interfaces with XML: Advances on User Interface Description Languages UIXML'04 (Gallipoli, 2004), 2004. Available online http://www.edm.luc.ac.be/uixml2004/index.php?selected=program

[Lizk74]

Lizkov B. H., and Zilles S. N., *Programming with Abstract Data Types*, Computation Structure Group Memo n°99, MIT, Project MAC, Cambridge, 1974.

[Lonc96]

Lonczewski F., and Schreiber S., *The FUSE-system : An integrated user interface design environment*, in Vanderdonckt J. (Eds.), Proceedings of CADUI'96, Presses Universitaires de Namur, Namur, 1996, pp. 37-56.

[Löwe93]

Löwe M., *Algebraic approach to single-pushout graph transformation* in Theoretical Computer Science, Vol. 1, 1993, pp. 181-224.

[Lu98]

References

Lu, S., Paris, C., and Vander Linden, K., *Towards the automatic generation of task models from object oriented diagrams*, in Proceedings of Seventh IFIP Working Conference on Engineering for Human-Computer Interaction
(EHCI '98), Kluwer Academic Dordrecht, 1998.

[Luo93]

Luo P., Szekely P., and Neches R*., Management of interface design in HUMANOID*, In Proceedings of InterCHI'93, 1993, pp. 107-114.

[Luo94]

Luo P., *A human-computer collaboration paradigm for bridging design conceptualization and implementation*, in Paternó F. (Ed.), Design, Specification and Verification of Interactive Systems '94 (Heidelberg, 1994), Springer-Verlag, 1994, pp. 129-147.

[Luo95]

Luo, P., *A Human-Computer Collaboration Paradigm for Bridging Design Conceptualization and Implementation* in Paternò F. (Ed.), Interactive Systems: Design, Specification, and Verification, Proc. of the 1st Eurographics Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'94*,* (Bocca di Magra, June 8-10, 1994), Springer-Verlag, Berlin, pp. 129–147.

[Luyt04]

Luyten, K., Abrams, M., Limbourg, Q., Vanderdonckt, J., Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages" UIXML'04 (Gallipoli, May 25, 2004), Gallipoli, 2004.

M

[Macr04]

Macromedia Inc., *DevNet Resource Kit Volume 6*, 2004. Available online: http://www.macromedia.com/software/drk/productinfo/product_overview/volume6/

[Mahf95]

Mahfoudhi A., Abed M. , and Angue J.-C., *TOOD: Task object oriented description for ergonomic interfaces specification*, in Proceedings of IFA/IFIP/IFO/IEA Symposium on Analysis, Design and Evaluation of Man-Machine Systems (Cambridge, June 27-29, 1995), 1995.

[Mahf01]

Mahfoudhi, A., Abed, M., and Tabary,D., *From the formal specifications of user tasks to the automatic generation of the HCI specifications*, in Blandford A., Vanderdonckt J., & Gray P., (Eds.), People and computers XV, Springer, London, 2001, pp. 331–347.

[Mark97]

Markopoulos P., *A Compositional Model for the formal specification of user interface software*, PhD thesis, Departement of Computer Science, Queen Mary and Westfield College, University of London, 1997.

[Marq97]

Marquis J.-P., *Stanford encyclopedia of philosophy: Category theory*, 1997. Available online: http://plato.stanford.edu/entries/category-theory/.

[Mart98]

Martin J., and Odell J., *Object-Oriented Software Methods*, 2nd edition, Prentice Hall, Upper Saddle River, New Jersey, 1998.

[Mell03]

Mellor S. J., and Clark A. J. (Eds.), *Introduction to Model Driven-Development*, in IEEE Software 20(5), 2003, pp. 14-18.

[Mens99]

References

Mens T., *A Formal Foundation for Object-Oriented Software Evolution*, PhD thesis, Vrije Universiteit Brussel, 1999.

[Mens01]

Mens, T., Van Eetvelde, N., Janssens, D., and Demeyer, S., *Formalising Refactoring with Graph Transformations*, in Fundamenta Informaticae, 21, 2001, pp. 1001–1022.

[Merl94]

Merlo E.,. Gagné P.-Y, and Thiboutôt A., *Inference of Graphical AUIDL Specifications for the Reverse Engineering of User Interfaces*, in Proc. of IEEE Int. Conf. on Software Maintenance, IEEE Computer Society Press, Los Alamitos, 1994, pp. 80-88.

[Merl95]

Merlo. E., Gagné P.-Y., Girard J.-F., Kontogiannis K., Hendren L., Panagaden P., and De Mori R., *Reengineering User Interfaces*, in IEEE Software, 12(1), January 1995, pp. 64-73.

[Merr04]

Merriam-Webster. *Collegiate dictionnary online*, 2004. Available online: http://www.m-w.com.

[Meye97]

Meyer B., *Object-Oriented Software Construction*, Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1997.

[Meye90]

Meyer, B., Introduction to the Theory of Programming Languages, Prentice Hall, New York, 1990.

[Mill03]

Miller, J., Mukerij J., MDA Guide version 1.0.1, 2003. Available online : www.omg.org.

[Mitt04]

Mittelbach F., and Goosens M., *The Latex Companion*, Second Edition, Addisson-Wesley, 2004.

[Moli02]

Molina, P.J., Meliá, S., Pastor, O,.J*ust-UI: A User Interface Specification Model*, in: Kolski C., Vanderdonckt J. (Eds.), Computer-Aided Design of User Interfaces III, Kluwer Academic Publishers, Dordrecht, 2002, pp. 63-74.

[Moor96]

Moore M.M., *Rule-Based Detection for Reengineering User Interfaces*, in Proceedings of the 3rd IEEE Working Conf. on Reverse Engineering WCRE'96 (Monterey, 8-10 Nov. 1996), IEEE Press, Los Alamitos, 1996, pp. 42-49.

[Moor97]

Moore M.M., and Rugaber S., Using Knowledge Representation to Understand Interactive Systems, in Proc. of the 5th IEEE Int. Workshop on Program Comprehension IWPC'97 (Dearborn, 28-30 May 1997), IEEE Press, Los Alamitos, 1997, pp. 60-69.

[Mont70]

Montanari, U.G., *Separable Graphs, planar Graphs and Web Grammars*, in Inf. Contr., 16, 1970, pp. 243-267.

[Mont04]

Montero F., Limbourg Q., and Vanderdonckt J., Pattern-Based Design of User Interfaces based on USIXML, 2004, submitted for publication.

[Moli03]

Molina P. J., Belenguer J., and Pastor O., Describing Just-UI Concepts Using a Task Notation, in proceedings of DSV-IS 2003(Funchal, Madeira. June, 2003), Lecture Notes in Computer Science, Berlin, Springer Verlag, 2000.

References

[Mori03]

Mori, G., Paternò, F., and Santoro, C., Tool Support for Designing Nomadic Applications, in Proceedings of IUI 2003 (Miami, Florida, January 12-15), ACM Press, New York, 2003.

[Myer91]

Myers B.A., Separating application code from toolkits: Eliminating the spaghetti of call-backs ui builders, in Proceedings of the ACM Symposium on User Interface Software and Technology UIST'91, ACM Press, pp. 211-220, 1991.

[Myer92]

Myers B. A., and Rosson M. B., *Survey on user interface programming*, in Proceedings of the SIGCHI conference on Human factors in computing systems SIGCHI'92, 1992, pp. 195-202.

[Myer95a]

Myer B. A., *User Interface Software Tools* in ACM Transactions on Computer-Human Interaction, 2(1), March 1995, pp. 64-103.

[Myer00]

Myers B., Hudson S. , and Pausch R., *Past, present, future of user interface tools*, ACM Transactions on Computer-Human Interaction, 7(1), 2000, pp. 3-28.

N

[Nana95]

Nanard, J. and Nanard M., *Hypertext Design Environment and the Hypertext Design Process* in Communications of the ACM 38(8), August 1995, pp. 49–56.

[Nava01]

Navarre D., Planque P., Bastide R. , and Sy O., *A model-BAsed tool for interactive prototyping of highly interactive applications*, in Proc. Of the 12th IEEE Int. Workshop on Rapid System Prototyping (Monterey, USA), IEEE Press, Los Alamitos, 2001.

[Newm03]

Newman M. N., Lin J., Hong J. I., and Landay J. A., *DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice*, in Human-Computer Interaction, 18(3), 2003, pp. 259-324.

[Niel94]

Nielsen J., and Levy J., *Measuring usability: preference vs. performance*, Communications of the ACM, 37(4), April 1994, pp.66-75.

[Niga95]

Nigay L., Coutaz J., *A Generic Platform for Addressing the Multimodal Challenge*, in Proceedings of CHI'95, ACM Press, New York, 1995, pp. 98-105.

O

[Olle88]

Olle T.W., Hagelstein J., Macdonald I. G., Rolland C., Sol H.G., Van Assche F., Verijn-Stuart A., *Information Systems Methodologies, a framework for understanding*, Addisson-Wesley, 1988.

[Olse86]

Olsen D.R., *MIKE: The Menu Interaction Kontrol Environment* in ACM Transactions on Graphics 5(4), October 1986, pp. 318–344.

[Olse83]

Olsen D. R., *Syngraph, a graphical user interface generator*, in Computer Graphincs. Proceedings SIGGRAPH'83 (Detroit, USA), 1983, pp. 43-50.

[Olse98]

Olsen D. R., *Developing User Interfaces*, Morgan Kaufmann Publishers, San Francisco, 1998.

References

[OMG01]

Object Management Group, The Model Driven Architecture (MDA), draft, 9 July 2001. Available at http://www.omg.org/mda/specs.htm.

[OMG03a]

Object Management Group, The Unified Modelling Language specification, version 1.5, 01 March 2003. Available online: http://www.omg.org/technology/documents/formal/uml.htm

[OMG03b]

Object Management Group, *Common Warehouse Specification version 1.1.*, march 2003, Available online: http://www.omg.org/docs/formal/03-03-02 .pdf.

[Oust94]

J. Ousterhout, *Tcl and TK Toolkit*, Addisson Wesley, Reading, 1994.

P

[Pala94]

Palanque Ph., and Bastide R., *Petri net based design of user-driven interfaces using interactive cooperative object formalism*, in Paternò F. (Ed.), Proceedings of 1st Eurographics Workshop on Design, Specification and Verification of Interactive Systems, DSV-IS 94,Springer Verlag, 1994.

[Pala97]

Palanque P., Spécifications formelles et systèmes interactifs: Vers des systèmes fiables et utilisables. travail d'Habilitation à diriger des recherches. Technical report, Université de Toulouse 1, 1997.

[Pala97]

Palanque, P. and Paterno,F.(Eds.), *Formal Methods in Human Computer Interaction*, Springer-Verlag, London, 1997.

[Parn69]

Parnas, D. L., *On the Use of Transition Diagram in the Design of a User Interface for Interactive Computer System*, *in* Proceedings of the 24th National Conference, ACM press, New York, 1969, pp. 379-385.

[Parna72]

Parnas D.L., *On the Criteria to be used in Decomposing Systems into Modules* in Communication of the ACM, 15 (12), 1972.

[Part83]

Partsch, H. and Steinbruggen, R., *Program Transformation Systems*, ACM Computing Surveys 15(3), September 1983, pp. 199–236.

[Pate97]

Paternò F. , Mancini C. , and Meniconi S., *ConcurTaskTree: A diagrammatic notation for specifying task models*, in Howard S. , Hammond J. , and Lindgaard G. (Eds.), Proceedings of IFIP TC 13 International Conference on Human-Computer Interaction Interact'97 (Sydney, July 14-18, 1997), Kluwer Academic Publishers, Boston, 1997, pp. 362-369.

[Pate98]

Paternò F. , Santoro C. , and Tahmassebi S., *Formal model for cooperative tasks: Concepts and an application for en-route air traffic control*, in Markopoulos P., and Johnson P. (Eds.), Proc. Of 5th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'98 (Abingdon, June 3-5, 1998), Springer-Verlag, Vienna, 1998.

[Pate00]

Paternò, F., *Model-Based Design and Evaluation of Interactive Applications*, Springer-Verlag, Berlin, 2000.

[Pate02]

References

Paternò. F, and Santoro. C., *One model, many interfaces*, in Kolski C., and Vanderdonckt J. (Eds.), Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002 (Valenciennes, 15-17 May 2002), Kluwer Academics Publishers, Dordrecht, 2002, pp. 143-154.

[Pate03]

Paternò, F.,and Santoro, C., A Unified Method for Designing Interactive Systems Adaptable to Mobile and Stationary Platforms, Interacting with Computers, Elsevier, 15, 2003, pp. 349-366.

[Paga02]

Paganelli L., and Paternò F., *Automatic Reconstruction of the Underlying Interaction Design of Web Applications*, in Proc. of the 14th ACM Int. Conf. on Software Engineering and Knowledge Engineering SEKE'02 (Ischia, 15-19 July 2002), ACM Press, New York, pp. 439-445.

[Paus92]

Pausch, R., Conway, M. and DeLine, R. *Lessons Learned from SUIT, the Simple User Interface Toolkit Practice and Experience*, ACM Transactions on Information Systems, 10 (4), 1992, pp. 320–344.

[Payn86]

Payne S. J., and Green T. R. J.*, Task-action grammars: A model of the mental representation of task languages* in Human-Computer Interaction, 2(2), 1986, 93-133.

[Peto93]

Petoud I., and Pigneur Y.*, An Automatic and Visual, Approach for User Interface Design*, Engineering for Human-Computer Interaction, North Holland, Amsterdam, 1993, pp. 403-420.

[Pfal69]

Pfalz, J. L., Rozenberg, A., Web Grammars. Proceedings of the International joint Conference on Artificial intelligence, Washington, 1969, pp. 609-619.

[Pinh00]

Pinhero Da Silva P., *User interface declarative models and development environments: A survey* in Paternò F., and Palanque P., Interactive Systems. Design Specification, and Verification DSV-IS '00, LNCS, Springer Verlag, Amsterdam, 2000, pp. 207-226.

[Plim04]   Plimmer B., and Apperley M., *Interacting with Sketched Interface Designs: An Evaluation Study,*.Proc. of ACM Conf. on Human Factors in Computing Systems CHI'04 (Vienna, April 2004), ACM Press, New York, 2004.

[Puer94]

Puerta, A. R., Eriksson, H., Gennari, J. H., and Musen, M. A., Beyond Data Models for Automated User Interface Generation, in People and Computers IX, Proceedings of HCI'94, Cambridge University Press, Cambridge, 1994, pp 352-366.

[Puer96]

Puerta, A.R., *The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development*, in Vanderdonckt, J. (Ed.), Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'96 (Namur, June 5-7 1996), Presses Universitaires de Namur, Namur, 1996, pp. 19-36.

[Puer97]

Puerta, A.R., *A Model-Based Interface Development Environment*, in IEEE Software 14(4), 1997, pp. 41–47. Available online: http://www.arpuerta. com/pubs/ieee97.htm

[Puer98]

Puerta A.R., *Supporting user-centered design of adaptive user interfaces via interface models*, in First Annual Workshop On Real-Time Intelligent User Interfaces For Decision Support And Information Visualization, January 1998.

[Puer99]

273

References

Puerta, A. and Eisenstein, J., *Towards a General Computational Framework for Model-Based Interface Development Systems Model-Based Interfaces*, in Proceedings of 3rd International ACM Conference on Intelligent User Interfaces IUI'99 (Redondo Beach, 5-8 January 1999), ACM Press, New York, 1999, pp. 171–178.Available online: http://www.arpuerta.com/pubs/iui99.htm

[Puer01]

Puerta A. and Eisenstein J., *A representational basis for user interface transformations,* in Wiecha Ch., and Szekely P. (Eds.), Proceedings of CHI'2001 Workshop "Transforming the UI for Anyone, Anywhere - Enabling an Increased Variety of Users, Devices, and Tasks Through Interface Transformations" (Seattle, April 1-2, 2001), ACM Press, New York, 2001.

R

[Raml01]

El-Ramly M., Iglinski P., Stroulia E., Sorenson P., and Matichuk B., *Modeling the System-User Dialog Using Interaction Traces*, in Proc. of 8th IEEE Working Conf. on Reverse Engineering WCRE'2001 (Stuttgart, 5-7 Oct. 2001), IEEE CS. Press, Los Alamitos, 2001, pp. 208-217.

[Reen79]

Reenskaug T., *Models-Views-Controllers*, Technical note, Xerox PARC, December 1979. Available online: http://heim.ifi.uio.no/~trygver/mvc/index.html.

[Reis81]

Reisner P., *Formal grammar and human factors design of interactive graphics systems*, in IEEE transaction on software engineering, 7, 1981, pp. 229-240.

[Rens03]

Rensik, A. (Ed.): 2003, *Proceedings of the 1st Workshop on Model-Driven Architecture: Foundations and Applications MDAFA'03* (Enschede, June 26-27, 2003), CTIT Technical Report TR-CTIT-03-27, University of Twente, Twente, 2003. Available online: http://trese.cs.utwente.nl/mdafa2003/

[Ricc01]

F. Ricca, P. Tonella, and I.D. Baxter, "Restructuring Web Applications via Transformation Rules", Proc. of IEEE Workshop on Source Code Analysis and Manipulation SCAM'2001 (Florence, 5-9 Nov. 2001), IEEE Computer Soc. Press, Los Alamitos, 2001, pp. 150-160

[Robe98]

Roberts, D., Berry, D., Isensee, S. and Mullaly, J. Designing for the User with OVID: Bridging User Interface Design and Software Engineering. Macmillan Technical Publishing: Indianapolis, 1998.

[Rosz97]

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

[Ruga99]

Rugaber S., *A Tool Suite for Evolving Legacy Software*, in Proc. of IEEE Int. Conf. on Software Maintenance ICSM'99 (Oxford, 30 August-3 Sep. 1999), IEEE Comp. Society Press, Los Alamitos, 1999, pp. 33-39.

[Rumb91]

Rumbaugh J., *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

S

[Sacr95]

References

Sacré B. , Provot I. , and Vanderdonckt J., *Une description orientée-objet des objets interactifs abstraits utilisés en interface homme-machine*, Technical Report rapport de recherche IHM/Ergo/10, Institut d'Informatique, FUNDP Namur, 1992. Révision du 13 novembre 1995.

[Savi01]

Savidis, D. Akoumianakis, and C. Stephanidis. *The Unified User Interface Design Method*, chapter 21, pages 417-440. Lawrence Erlbaum Associates, Mahwah, 2001.

[Scap89]

Scapin D., and Pierret-Golbreich C., *Towards a method for task description: MAD*, in Berlinguet L., and Berthelette D. (Eds.), Proc. Of the Conf. Work with Display Units WWU'89, Elsevier Science Publishers, Amsterdam, 1989, pp. 27-34.

[Scap01]

Scapin, D. L.,and Bastien, J. M. C., *Analyse des tâches et aide ergonomique à la conception : l'approche MAD**, in Kolski C. (Ed.), Analyse et conception de l'IHM: Interaction homme-machine pour les systèmes d'information, Edition Hermes,
Paris, 2001, pp. 85–116.

[Schl96]

Schlungbaum E., *Model-based user interface software tools - current state of declarative models*, Technical Report GIT-GVU 96-30, Georgia Institute of Technology, 1996.

[Schl96]

 Schlungbaum E.and Elwert T., *Dialogue graphs - a formal and visual specification technique for dialogue modeling*, in Siddiqi J.I., Roast C.R. (Eds.), Proceedings of the BCS-Facs Workshop on Formal Aspects of the Human-Computer Interface, Springer-Verlag, London, 1996.

[Schl97]

Schlungbaum E., *Tutorial notes on model-based interface development environments, state of the art and challenges of further development*, presented at the Conference on Prototyping of User Interfaces: Basic, Techniques and Experience PB'97, 1997

[Schn98]

Schneiderman B., *Designing the User Interface. Strategies for Effective Human-Computer Interaction*, Third edition, Addisson-Wesley, Reading, 1998.

[Schr94]

Schreiber S., *Specification and generation of user interfaces with the BOSS system*, in Gornostaev J. et al, (Eds.), Proceedings East-West International Conference on Human-Computer Interaction EWHCI'94 (St. Petersburg, August 2-6, 1994), Springer, Moskau, 1994.

[Schü97]

Schürr, A., *Programmed Graph Replacement Systems*, in [Rosz97], pp. 479-546.

[Send03]

Sendall S., and Kozacynski W., *Model Transformation: The Heart and Soul of Model-Driven Software Development*, in IEEE Software, 20(5), 2003, pp. 42-45.

[Shep89]

Shepherd, A. (1989). Analysis and training in information technology Tasks. In D. Diaper (Ed.), Task analysis for human-computer interaction (pp. 15–55). Chichester, England: Ellis Horwood.

[Shep95]

Shepherd, A., *Task analysis as a framework for examining HCI Tasks*, in Monk A. and Gilbert N. (Eds.), Perspectives on HCI: Diverse approaches, Academic Press London, 1995, pp. 145–174.

[Sinh04]

Sinha R., *ROI of usability, A collection of links*, 2004. Available online: http://www.rashmisinha.com/useroi.html.

[Smit96]

References

Smith, M. J.,and O'Neill, J. E., *Beyond task analysis: Exploiting task models in application implementation*, in Proceedings of ACM Conference on Human Aspects in Computing Systems CHI '96, ACM Press, NewYork, 1996, pp. 263–264.

[Somm99]
I. Sommerville, Sofgtware Engineering, 5th edition, Addisson Wesley, 1999.

[Souc02]
Souchon N., Limbourg Q., and Vanderdonckt J., *Task modelling in multiple contexts of use*, In Proceedings of the 9th International Workshop on Design, Specification and Verification of Interactive Systems Workshop DSV-IS'02 (Rostock, June 12-14, 2002), LNCS, Springer Verlag, 2002.

[Sowa92]
Sowa J. F., *Conceptual Graphs Summary*, in Eklund P., Nagle T., Nagle J., and Gerholz L. (Eds.), Conceptual Structures: Current Research and Practice, Ellis Horwood, 1992, pp. 3-52.

[Stan95]
Standish Group, *Chaos Software Development Report*, 1995. Available online: www.projectsmart.co.uk/docs/chaos_report.pdf

[Stan98]
Stanton, N., and Young, M*., Is utility in the eye of the beholder? A study of ergonomics methods,* in Applied Ergonomics, 29(1), 1998, pp. 41–54.

[Stan04]
Stănciluescu, A., Limbourg, Q., Vanderdonckt, J., *Graful – modalitate de reprezentare a elementelor interfeţei cu utilizatorul*, Proc. of 1st National Conference on Computer-Human Interaction ROCHI'2004 (Bucharest, September 23-24, 2004), Ştefan Trăuşan-Matu, S., Pribeanu, C. (Eds.), Polytechnic University of Bucharest, Bucharest, 2004, to appear.

[Stir97]
Stirewalt K., *Automatic Generation of Interactive Systems from Declarative Models*, PhD thesis, Georgia Institute of Technology, 1997.

[Stir98]
Stirewalt R. E. K., *MDL: A language for bonding UI models*, in Vanderdonckt J. (Ed.), Proc. Of the 3rd Int. Workshop on Computer-Aided Design of User Interfaces (21-23 October 1999, Louvain-la-Neuve, Belgium), Luwer Academics, Dordrecht, 1998, pp. 159-170.

[Stir98]
Stirewalt R. E. K., and Rugaber S., *Automating UI generation by model composition*, in Proc. Of the 13th Conf. On Automated Software Engineering ASE'98 (Honolulu, 13-16 October 1998), IEEE Press, Los Alamitos, 1998.

[Stir00]
Stirewalt R. E. K., and Rugaber S., *The model-composition problem in user-interface generation*, in Automated Software Engineering, 7, April 2000, pp.101-124.

[Strou00]
Stroulia E., Thomson J., and Situ Q., *Constructing XML-speaking Wrappers for WEB Applications: Towards an Interoperating WEB*, in Proc. of IEEE 7th Working Conf. on Reverse Engineering WCRE'2000 (Brisbane, 23-25 Nov.2000), IEEE Computer Society Press, Los Alamitos, 2000, pp. 59-69.

[Strou03]
Stroulia E., El-Ramly M., Iglinski P., and. Sorenson P.G., *UI Reverse Engineering in Support of Interface Migration to the Web*, in Journal of Automated Software Engineering, 10(3), July 2003, pp. 271-301

[Sucr97]

References

Sucrow B., *Formal specification of human-computer interaction by graph grammars under consideration of information resources*, in, *Proceedings of the 1997 Automated Software Engineering Conference (ASE `97)*, IEEE Computer Society, 1997, pp. 28-35.

[Sucr98]

Sucrow, B., *On Integrating Software-Ergonomic Aspects in the Specification Process of Graphical User Interfaces* in Transactions of the SDPS Journal of Integrated Design & Process Science, 2(2), June 1998, pp. 32–42.

[Sumn97]

Sumner, T., Bonnardel, N., and Harstad Kallak, B., *The Cognitive Ergonomics of Knowledge-Based Design Support Systems*, in Proceedings of ACM Conference on Human Aspects in Computing Systems CHI'97 (Atlanta, March 22-27 1997), ACM Press, New York, 1997, pp. 83–90.

[Sutc89]

Sutcliffe, A. (1989). Task analysis, systems analysis and design: Symbiosis or synthesis? InteractingWith Computers, 1(1), 6–12.

[SUN04]

SUN Microsystems. Java fondation classes home page, 2004. Available online: http://www.java.sun.com/products/jfc.

[Sung02]

Sung M.Y., Lee D. H., Rho S.J., Rhee S.Y., Authoring Together in a 3D
Spatio-Temporal Space, in Proceedings on ACM Multimedia '02 Workshop of International Conference on Immersive Telepresence ITP (Juan-les-Pins,
France, December 6, 2002.

[Szek90]

P. Szekely. *Template-based mapping of application data to interactive displays*, in Proceedings of UIST'90, ACM Press, 1990, pp. 1-9.

[Szek92]

Szekely P. , Luo P. , and Neches R., *Facilitating the exporation of interface design alternatives: The HUMANOID model of interface design*, in Proceedings of SIGCHI'92, 1992, pp. 507-515.

[Szek96a]

Szekely P. , Sukaviriya P. , Castells J. , Muthukumarasamy J., and Salcher E., *Declarative interface models for user interface construction tools : The MASTERMIND approach*, in Engineering for Human-Computer Interaction, Chapman & Hall, London, 1996, pp 120-150.

[Szek96b]

Szekely P., *Retrospective and challenges for model-based interface development* in Vanderdonckt J. (Ed.), Proc. Of 2nd Int. Workshop on Computer-Aided Design of User Interfaces CADUI'96 (Namur, June 5-7, 1996), Presses Universitaires de Namur, Namur, 1996.

T

[Tarb93]

Tarby J.C.. *Gestion automatique du dialogue homme-machine à partir des spécifications conceptuelles.* Master's thesis, Université de Toulouse I, September 1993.

[Tarb96]

Tarby J-C, and M-F Barthet, *The DIANE+ method*, in Vanderdonckt J. (Ed.), Computer-Aided Design of User Interfaces, Proc. of the 1st Int. Workshop on Computer-Aided Design of User Interfaces CADUI'96 (Namur, 5-7 June 1996), Presses Universitaires de Namur, Namur, 1996, pp. 95-119.

[Taub90]

Tauber M. J., *ETAG: Extended task action grammar. a language for the description of the user's task language* in Diaper D. , Gilmore D. , Cockton G. , and Shackel B. (Eds.), Proc. of the

References

3rd IFIP TC 13 Conf. On Human Computer Interaction Interact '90 (Cambridge, 27-31 August 1990), Elsevier, Amsterdam, 1990, pp 163-168.

[Teor86]
Teory T.J., Yang D., Fry J.P., A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model, ACM computing surveys 18(2), june 1986, pp. 197-222.

[Thev99]
Thevenin D., and Coutaz J., *Plasticity of user interfaces: Framework and research agenda*, in Sasse A., and Johnson Ch. (Eds.), Proceedings of 7th IFIP TC 13 International Conference on Human-Computer Interaction Interact'99 (Edinburgh, August 30-September 3, 1999), IOS Press, London, 1999, pp. 110-117.

[Thim90]
Thimbleby H. W., *User Interface Design*, Addison-Wesley, 1990.

[Thev01]
Thevenin, D., *Adaptation en Interaction Homme-Machine: le cas de la Plasticité*, Ph.D. thesis, Université Joseph Fourrier, Grenoble, France, 2001. Available online: http://iihm.imag.fr/publs/2001.

[Trae99]
Traetteberg H., *Modeling work: Workflow and task modeling*, in Vanderdonckt J., and Puerta A.R. (Eds.), Proc. Of 3rd Int. Conf. On Computer-Aided Design of User Interfaces CADUI'99 (Louvain-la-Neuve, 21-23 October 1999), Kluwer Academics, Dordrecht, 1999, pp. *275*-280.

[Trev02]
Trevisan, D., Vanderdonckt, J., and Macq, B, *Analyzing Interaction in Augmented Reality Systems* in Pingali, G., Jain, R. (Eds.), Proceedings of ACM Multimedia 2002 International Workshop on Immersive Telepresence ITP'2002 (Juan Les Pins, 6 December 2002), ACM Press, New York, 2002, pp. 56–59.

[Tsib00]
Tsibidis G., Arvantitis T.N., and Baber C., *CHI 2000 proposal for the what, who, where, when, why and how of context-awareness,* in Proc. Of CHI'2000 Workshop on Context Awareness (The Hague, April 1-6, 2000), Research report 2000-18e.Atlanta, GVU Center, Georgia University of Technology, 2000.

U

[Unge96]
Unger, C., and Bass L. (Eds) *Engineering for HCI*, Kluwer Academics Publishers, 1996.

[Univ99]
Univers@lis:, France Telecom. Available online: http://universalis.elibel.tm.fr/ site/

[Usco01]
Uscom. *The tea widget set*, 2001. Available online: http://uscom.com/~isoft map .html

[UIML04]
UIML.Org. Homepage of the user interface markup language, 2004. www.uiml.org.

V

[Vand93]
Vanderdonckt, J. and Bodart, F., *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*, in Ashlund, S., Mullet, K. Henderson, A., Hollnagel, E., White, T. (Eds.), Proceedings of the ACM Conference on Human Factors in Computing Systems

References

InterCHI'93 (Amsterdam, April 14-19, 1993), ACM Press, New York, 1993, pp. 424-429.

[Vand93]
Vanderdonckt J., and Bodart F., *Encapsulating knowledge for intelligent automatic interaction objects selection* In Ashlund S., Mullet K., Henderson A., Hollnagel E., and White T. (Eds.), Proceedings of the ACM Conference on Human Factors in Computing Systems InterCHI'93 (Amsterdam, 24-29 April 1993), ACM Press pages, New York, 1993, pp. 424-429.

[Vand96]
Van der Veer G. C., Van der Lenting B. F., and Bergevoet B. A. J., *GTA: Groupware task analysis - modelling complexity* in Acta Psychologica, 91, 1996, pp. 297-322.

[Vand97]
Vanderdonckt J., *Conception Assistée de la Présentation D'une Interface Homme-Machine Ergonomique Pour Une Application de Gestion Hautement Interactive*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur, 1997.

[Vand98]
Vanderdonckt, J. (Ed.), *Proc. Of the 3rd Int. Workshop on Computer-Aided Design of User Interfaces CADUI'99 (21-23 October, Louvain-la,Neuve, Belgium)*, Kluwer Academics, Dordrecht, 1998.

[Varr02]
Varró, D., *A Formal Semantics of UML Statecharts by Model Transition Systems*, in Proceedings of the 1st International Conference on Graph Transformation ICGT'02 (Barcelona, October 7-12, 2002), Lecture Notes in Computer Science, Vol. 2505. Springer-Verlag, Berlin, pp. 378-392.

[Varr02b]
Varró, D., Varró, G., and Pataricza, A., *Designing the Automatic Transformation of Visual Languages*, in Science of Computer Programming, 44, 2002, pp. 205–227.

[Veer96]
van der Veer, G. C., Van der Lenting, B. F., and Bergevoet, B. A. J., *GTA: Groupware task analysis-modeling completity*, in Acta Psychological, 91, 1996, pp. 297–322.

W

[W3C99]
W3C consortium, *HTML 4.01 specification, W3C Recommendation,* 24 Dec. 1999. Available at http://www.w3.org/TR/REC-html40/

[W3C01]
W3C Consortium, *XML Schema Specification, W3C Recommendation*, 2 May 2001. Available at http://www.w3.org/XML/Schema#dev

[W3C04]
W3C consortium, *Voice Extensible Markup Language (VoiceXML) Version 2.0, W3C Recommendation,* 16 March 2004 available at http://www.w3.org/TR/voicexml20/

[W3C04b]
W3C consortium, *OWL Web Ontology Language Reference, W3C Recommendation,* 10 February 2004. Available at http://www.w3.org/TR/owl-ref/

[Wals99]
Walsh N., *Learning Perl/Tk. Graphical User Interfaces with Perl*, O'Reilly, 1999.

[Wegn97]
Wegner, P., *Why Interaction is more Powerful than Algorithms* in Communications of the ACM, 40(5), 1997, pp. 80-91.

[Weis93]

References

Weiser M., *Some computer science issues in ubiquitous computing*, in Communication of the ACM, 36(7), 1993, pp. 75-84.

[Weli98a]

Van Welie M. , Van der Veer G.C., and Eliens A., *An ontology for task world models*, in Markopoulos P., and Johnson P. (Eds.), Proc. Of 5th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'98 (Abingdon, June 3-5, 1998), pages 57-70, Vienna, 1998. Springer-Verlag.

[Weli98b]

van Welie, M., and van der Veer, G. C., *EUTERPE: Tools support for analyzing cooperative environments*, in Green T.R. G., Bannon L., Warren C. P.,and Buckkleys J. (Eds.), Cognition and co-operation, INRIA, Roquencourt, 1998, pp. 25–30.

[Wiec90]

Wiecha C et al., *ITS: A Tool for Radly Developinf Interactive Applications*, in ACM Transactions on Information Systems, 8(3), pp. 204-236, 1990.

[Wils96]

Wilson S., and Johnson P., *Bridging the generation gap: From tasks to user interface design*, in Vanderdonckt J. (Ed.), Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'96, Presses Universitaires de Namur, Namur, pp. 77-94, 1996.

X

[XVT96]

XVT, XVT Software, Inc., 4900 Pearl East Circle, Boulder, CO, 80301, USA, 1996.

Z

[Zand95]

Vander Zanden B., and Myers B. A., *Demonstrational and constraint-based techniques for oictorially specifying application objects and behaviours*, in ACM transactions on Computer-Human Interaction, 2(4), 1995, pp. 308-356.

[Zand96]

Vander Zanden B.T., and Venckus S.A., *An empirical study of constraint usage in graphical applications*, in Proceedings of the ACM Syposium on User Interface Software and Technology (UIST'96), 1996, pp. 136-146.

Annex: Tool Support

Attributed Graph Grammars tool

Requirements:
- Edition (including debugging) of transformation rules and models
- Execution of transformations
- Import and export function to and from UsiXML.

AGG (Attributed Graph Grammars tool) [Ehri99] is an multi-purpose graph transformation tool built in the "graph grammar group" at TU Berlin. It provides 1) a programming language enabling the specification of graph grammars 2) a customizable interpreter enabling graph transformations. AGG may be used in two different ways: through its Graphical User Interface (GUI) or through its Application Programming Interface (API). AGG can be considered as a genuine programming environment based on graph transformations. Fig. A-1 illustrates the GUI of AGG. Frame 1 is the grammar explorer. Frames 2, 3 and 4 enable to specify sub-graphs composing 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 a production will be applied is represented in Frame 5.

In the context of this dissertation, AGG was used as a **transformation editor** (including debugging functions) and **interpreter.** An import and an export function from and to (a preliminary version of) UsiXML models has been developed and described in [Stan04].
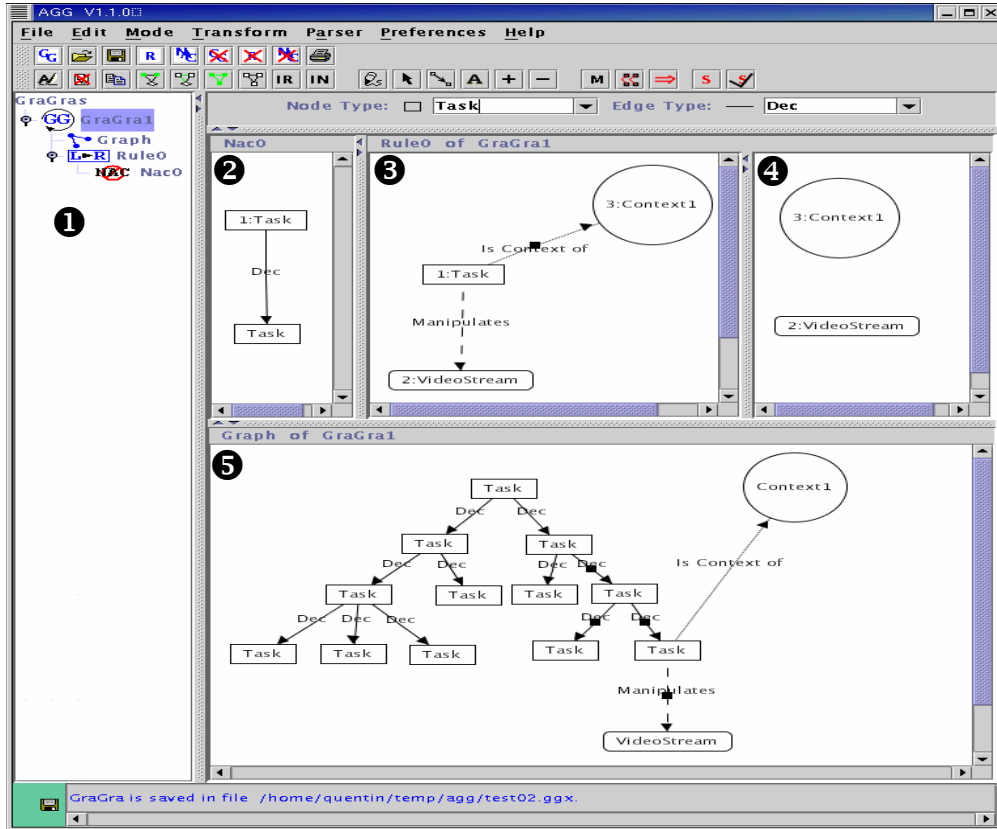
Figure A-1 AGG Graphical User Interface

TransformiXML API

Requirement:

- Interpretation of transformation rules from a UsiXML description of rules and host models.

Several Application Programming Interfaces are available to perform model-to-model transformations (e.g., DMOF [dMof02] or Univers@lis [univ99]). AGG API was selected due to our prior experience with its GUI version. Using AGG API as a transformation tool allows us to realize the following scenario (Fig. A-2): an initial model along with a set of rules expressed in UsiXML are transmitted to transformiXML API. UsiXML elements (models and rules) are parsed and transformed into AGG object types. Rules are successively applied to the models. The resulting specification, under the form of AGG objects, is parsed and transformed into UsiXML elements. This process is notably described in [Limb04].
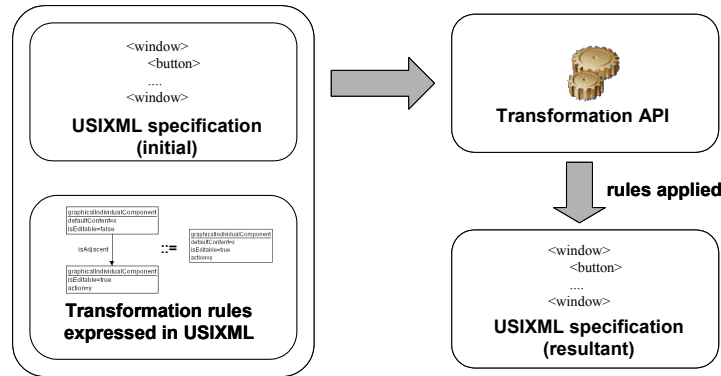
Figure A-2 Development process based on transformation application.

TransformiXML GUI

Requirements:

- Allow to manage a development library (a library containing a catalog of transformation rules)
- Allow to associate development sub-step with transformation systems
- Execute transformations

Fig A-3 presents a prototype of TransformiXML. The basic flow of tasks with transformiXML GUI is the following: a user chooses an input file containing models to transform. She, then, chooses a development path by selecting a starting point and a destination point (e.g., the viewpoint to obtain at the end of the process). Depending on the content of the input file some of the development paths may not be available. A tree allows the user to visualize the proposed development model (i.e., all the steps and sub-steps for a chosen path). The user can load another development model for the selected path. Now the task of the user consists in attaching one transformation system for each development sub-step. By clicking on a sub-step in the tree, a set of transformation systems realizing the chosen sub-step are displayed. A transformation system may be attached to the current sub-step by clicking "Attach to current sub-step". The user may also want to edit the rules either in an XML editor (the one of grafiXML, for instance) or in AGG environment. After attaching a transformation system for each rule in the development model, the user may apply the transformation either step by step or as a whole. The result of the transformation is then explicitly saved in a UsiXML file.
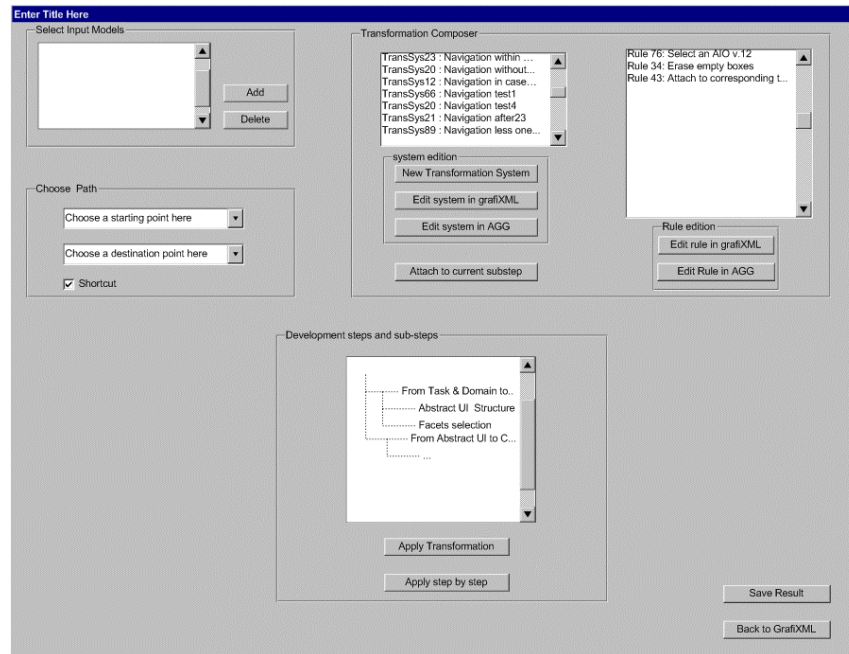
Figure A-3 TransformiXML

GrafiXML

Requirements:

- Edition in a WYSIWYG manner (What You See Is What You Get) of models at the CUI level (for graphical UIs).
- Import/ Export from and to UsiXML.

Editing a concrete UI in UsiXML directly can be considered as a tedious task, a specific editor called *GrafiXML* has been developed to face the development of CUI models. In this editor, the designer can draw in direct manipulation any graphical UI by directly placing CIOs and editing their properties in a property sheet (Fig. A-4). At any time, the designer may want to see the corresponding UsiXML specifications (Fig. A-5) and edit it. Selecting a UsiXML tag automatically displays possible values for this tag in a contextual menu. When the tag or the elements are modified, those changes are propagated to the graphical representation. In this way, a bidirectional mapping is maintained between a UI and its UsiXML specifications: each time a part is modified, the other one is updated accordingly.
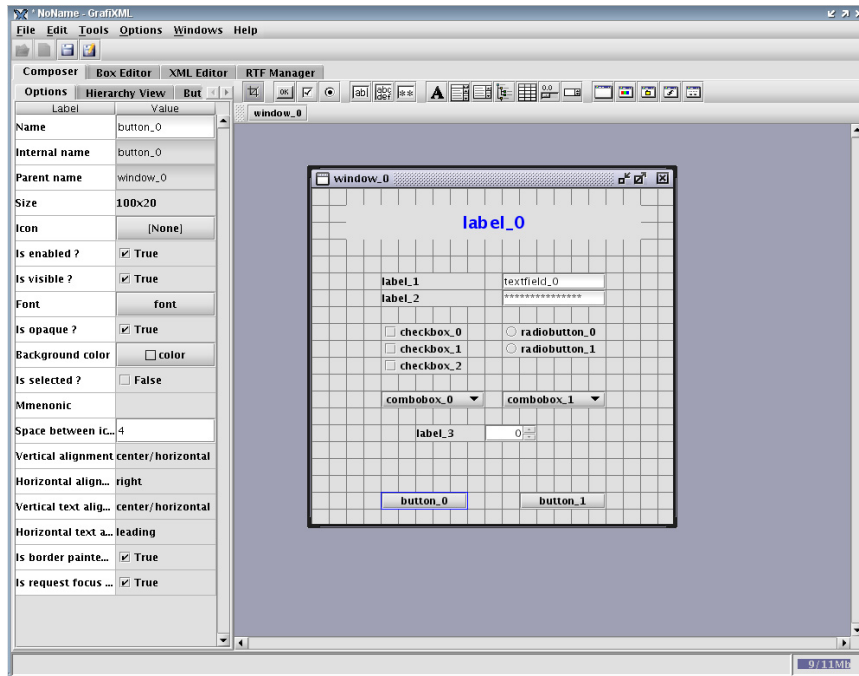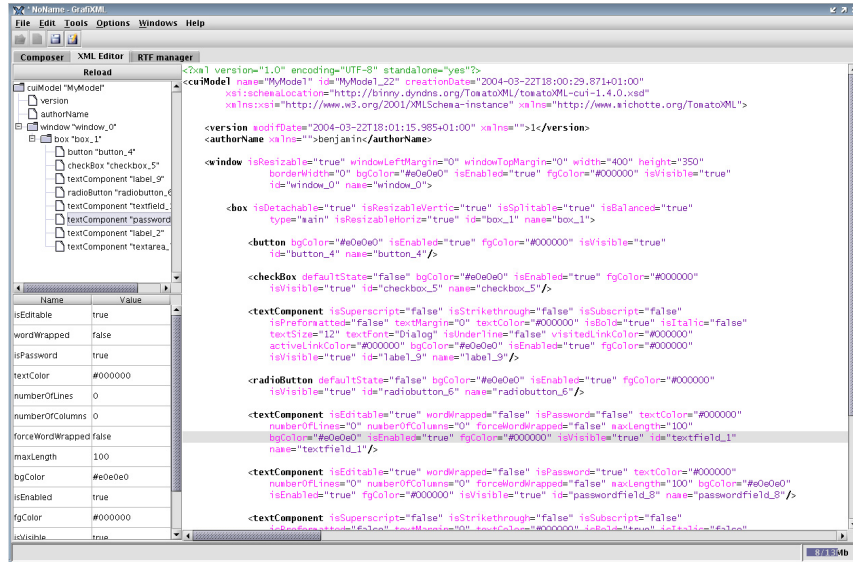
Figure A-4 GrafiXML main window

Figure A-5 UsiXML textual editor within GrafiXML tool

IdealXML

Requirement:
- Edition of Domain Model
- Edition of Task model
- Edition of AUI Model

IdealXML is described in [Mont04]. IdealXML enables to specify in a WYSIWYG manner the task model (Fig A-6 upper-left), the domain model (Fig 4-29 upper-right) and, the abstract user interface model (Fig A-6 lower-left).

The task model has the appearance of CTT notation introduced by [Pate99]. The domain model has the appearance of a class diagram. The AUI model has the form of a hierarchical structure of embedded boxes whose leafs are abstract individual components and their facets (specific icons have been designed to represent facet types).
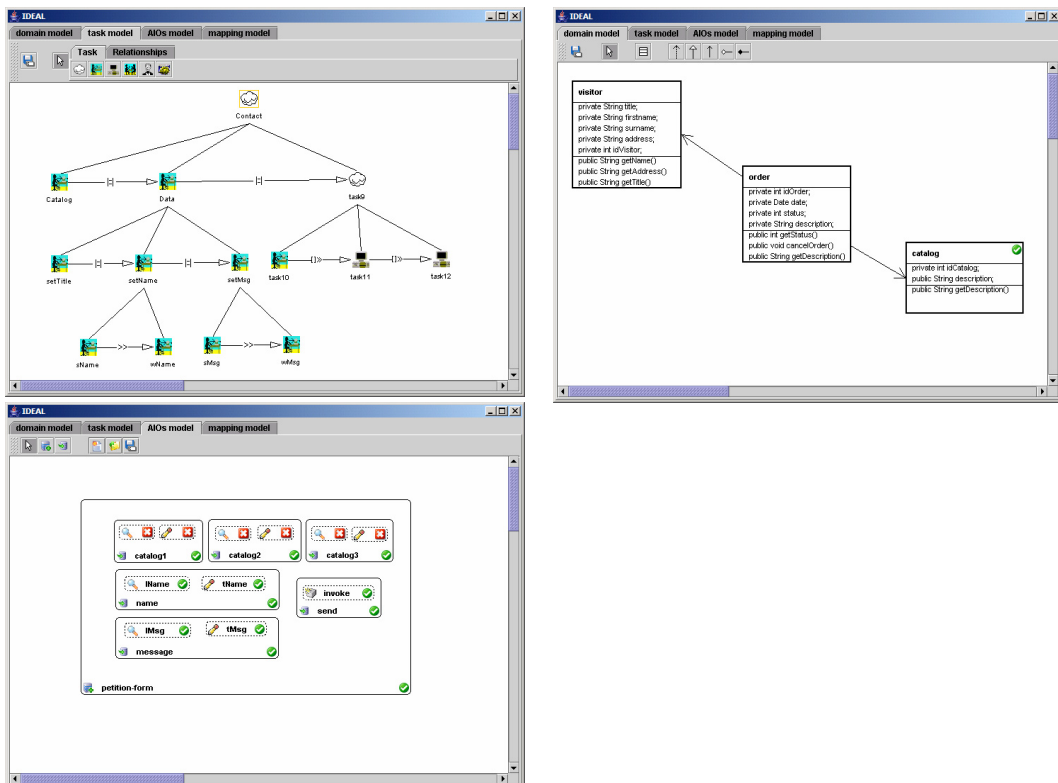

Figure A-6 Domain, Task, AUI model editors in IdealXML

ReversiXML

A specific tool, called ReversiXML (formerly called Rutabaga [Boui04]), automatically reverse engineers the presentation aspects from an existing HTML Web page at both the CUI and AUI levels. This tool allows developers to recover an existing UI so as to incorporate it again in the development process. In this case, a re-engineering can be obtained by combining two abstractions, one translation, and two reifications. This is particularly useful for the evolution of legacy systems.

Code Generators and Interpreters

Requirements:
- Generate "renderable" code in a high level language
- Render UsiXML

Two tools allow one to obtain a graphical rendering from a CUI specification. GrafiXML is equipped with an export module that allows a generation of XHTML code and Java Swing objects (see export menu in Fig. A-7). TransformiXML allows an interpretation of a CUI specification directly in flash. In this case a CUI may be assimilated to the final user interface.
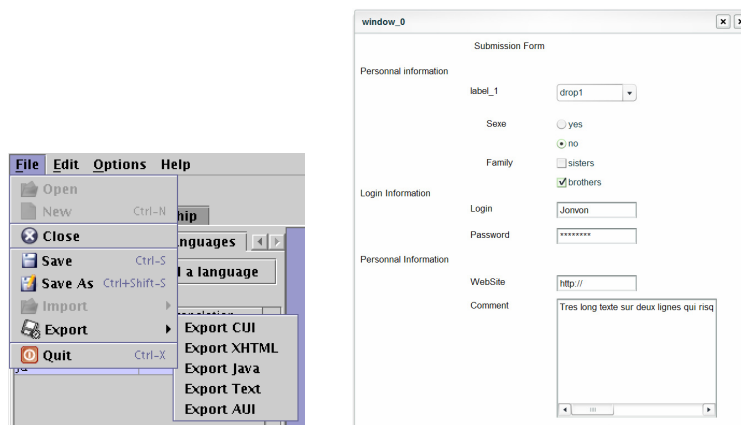

Figure A-7 GrafiXML export formats (left) and a UsiXML specification interpreted by FlashiXML (right)