Reverse Engineering of Web Pages based on Derivations and Transformations

Laurent Bouillon, Quentin Limbourg, Jean Vanderdonckt, and Benjamin Michotte
Université catholique de Louvain (UCL), School of Management (IAG),
Information Systems Unit (ISYS), Belgian Lab. of Computer-Human Interaction (BCHI)
Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium)
{bouillon, limbourg, vanderdonckt, michotte}@uclouvain.be

Abstract

The final user interface of an information system could be reverse engineered according to a Model-Driven Engineering perspective to multiple levels of abstraction, either independently of each other or simultaneously depending on the designer's goals: concrete user interface (which is independent of the user interface toolkit), abstract user interface (which is independent of any interaction modality), and "task and concepts" (which is independent of any particular implementation). To support the user interface reverse engineering from the code level to any model level, a set of derivation rules have been implemented in ReversiXML. To support the user interface reverse engineering from any model to any other level, graph grammars have been implemented in TransformiXML. A graph grammar consists of graph transformation rules, called productions, that accept as input a graph representation of the user interface to be reengineered, apply the transformation, and obtain a result that can be further exploited to re-create a new user interface.

1. Introduction

According the Model-Driven Engineering (MDE) [16], four paths exist for supporting forward and reverse engineering of computer-based systems: code to code (synonym of transcoding), code to model (for reverse engineering), model to code (for forward engineering), and model to model (for any step). According to this terminology, techniques for User Interface (UI) *forward engineering* typically support a development life cycle where the UI can be obtained from one or many abstract models (e.g., a task model, a domain model). Conversely, techniques for UI *reverse engineering* attempt to find back these abstract models by examining the UI in different ways (e.g., static or dynamic analysis). By combining reverse engineering and forward engineering into one sequence, *reengineering* [5] can be supported in general or *adapting* in particular when a UI needs to be adapted for another context of use that was not initially planned for (e.g., another computing platform). Although these different approaches (i.e., forward, reverse, reengineering, and adapting) can be combined in theory to incrementally build more sophisticated processes, we observe in practice that this is hard to achieve. To characterize this situation and prior to any further investigation, we define a set of *properties of interest* for UI reengineering:

- **Separation of concerns**: any reengineering system should be decomposed into a *reengineering logic* containing the knowledge required to perform the reengineering and a *reengineering engine* applying this knowledge with a clear separation between. The engine should be controlled by *reengineering parameters* that should be externally given (Fig. 1).

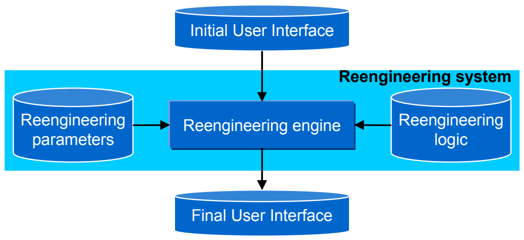


Fig. 1. Separation of concern in a reengineering system.

- **Flexible operationalization of reengineering logic**: any type of knowledge (e.g., rewriting rule [26], transformation [25], production rule [21]) should be specified explicitly in a way that remains external to the engine so that the engine remains autonomous when the logic changes. In this way, the logic can be expanded, modified when needed.
- **Flexible usage of reengineering engine**: the engine should rely on the knowledge contained in the logic to apply it in a way that is controlled by the reengineering parameters. In this way, the reengineering process can be made observable and traceable to the designer's eyes. If needed, the engine could be controlled by the designer, by a system agent or in a mixed-initiative way combining both.
- **Multi-level abstraction**: the system should produce a reengineered final UI from an initial UI so

as to reach multiple levels of abstraction on demand [18]. Typical levels include device independence (i.e. obtaining a UI model that is independent from any interaction device), computing platform independence (i.e., obtaining a UI model that is independent from any computing platform), and…

- **Modality independence**: the system should also produce a UI model that remains independent of any modality of interaction [17]. At this level, there should be no reference to the peculiar interaction techniques and modalities (e.g., character UI, graphical UI, vocal UI, and virtual UI)
- **Language neutrality**: a consequence of the platform independence is that, in principle, the system should be developed so as to accommodate UI programming in multiple languages whether they are imperative (e.g., Java, C++), declarative (e.g., Prolog or LISP), or markup (e.g., HTML, XML).

Several tools that have been developed so far to support UI forward engineering, reverse engineering, reengineering, or migration (i.e. porting a UI from one source computing platform such as the Web to another target computing platform such as a PocketPC or a mobile phone) suffer from one common shortcoming: the algorithms and the logic they use is often hardcoded in the system itself, thus leaving little or no room for supporting the above properties of interest. It is hard to modify the knowledge contained in the system nor it complex if not possible to control the process in a flexible way. Equally frequent is the system tightly coupled to one particular language (e.g., COBOL or C) or a family of them (e.g., HTML, DTHML). When the 'separation of concern' property is not satisfied, it is likely that others properties are not better satisfied or perhaps very partially.

In this paper, we would like to develop a series of techniques that addresses the above shortcomings by having a system satisfying the properties of interest by construction. For this purpose, we show in Section 2 how existing work has attempted to address this problem by reporting on some significant initiatives. In Section 3, a reference framework for UI reengineering is presented that characterizes how various techniques may address the properties of interest. Section 4 covers the 'code to model' part of the MDE approach by relying on derivation rules. Section 5 covers the 'model to model' part of the MDE approach by defining a technique based on graph grammars and graph transformation to support progressive transformation of a initial UI into a final one by applying a series of transformations. Each transformation, also called production, is defined by left hand-side, a right hand-side, and a negative application condition. Section 6 explains how two tools that have been developed to support the different parts of the MDE approach applied to UIs: REVERSIXML for derivation rules and TRANSFORMIXML, a tool that has been developed to support graph transformations applied on different models involved in the reference framework. Some examples of reverse engineering transformations are given that transforms the UI specifications one level at time or that skips one or multiple levels simultaneously. All levels are uniformly expressed in USIXML, a XML-compliant UI Description Language. Section 7 concludes the paper by summarizing the original aspects of the contribution.

2. Related work

First of all, we have to distinguish UI reverse engineering from data extraction from web pages [3,5]. The former is aimed at capturing the UI contained in a web page [4], while the latter is intended to extract [12], preferably automatically or semi-automatically, the data contained in a web page. In the first case, the ultimate goal is to recover a UI model consisting of both presentation and dialog ; in the second case, the ultimate goal is intended to recover a data or a domain model. Although the UI is a very practical way to look for data extraction, the problem of UI reverse engineering is probably easier than data reverse engineering since the HTML is more structured for the UI than it is for data. Several efficient techniques exist for data extraction (e.g. some representative examples include Lixto [14] and wrappers [9,12]). This paper is focusing of UI reverse engineering so as to reach the full UI reengineering cycle.

Preliminary work has already been conducted in the field of UI reengineering, especially to migrate character-based UIs (CUIs) to Graphical User Interfaces (GUIs). Another trend in the reengineering of UI is the migration of the UI to a platform that uses another modality. Most of the tools allow reverse engineering by constructing an abstract representation of the system UI that is further transformed to obtain a new UI.

MORPH [21,22,23] identifies basic user interaction tasks in legacy code by applying static program analysis techniques, including control flow analysis, data flow analysis, and pattern matching. The resulting model is then used to transform the detected abstractions in a graphical environment from a specific widget toolkit. 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 [27] 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.

MORE [13] produces applications that are device independent. A Platform Independent Application can be created either by a design tool or by abstracting a concrete UI thanks to the generalization process. Generalization is done by reverse engineering the UI 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 a particular device. Similarly to MORPH, operationalization and control are partial.

The reengineering process in TAMEX [15,29,30] allows one to produce HTML UIs composed of data contained in several other Web pages. The approach followed by TAMEX is based on the concept of task-specific mediation: information sources within an application domain are encapsulated within wrapper agents (data extraction) interacting with an intelligent intermediary agent, the mediator (aggregate data). 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.

REWEB's reengineering process [10] restructures Web applications in order to avoid their inevitable degradation. It uses a set of transformation rules aiming at the improvement of maintainability, usability and portability. It also restructures the design thanks to a web application model and by incorporating frame-based navigation. AUIDL is a User Interface Description Language used for reverse engineering a model of the UI so as to recreate it in another context of use [19,20].

WEBREVENGE [24] is a tool that analyzes Web site code in order to automatically reconstruct the underlying logical interaction design. Such a design is represented through task models that describe how activities should be performed to reach users' goals. WEBREVENGE is capable of recovering a task model from web pages, which is different from VAQUITA [3], which only recovers the concrete presentation model of a web page in XIML (www.ximl.org), without supporting the forward engineering part that should come afterward.

The idea of program transformation [25] has been extensively researched by various works, including TXL [6,7,8,28], although not particularly targeted at UIs. [25] is also another example of successfully applying program transformation to reengineer an existing system.

After analyzing these various approaches, we observe that the properties of interest are only partially supported in most existing work and there is room to introduce a flexible mechanism for UI transformation, based on graph grammars and transformations.

3. User interface re-engineering concepts

3.1. The Cameleon reference framework

The Cameleon Reference Framework [4] locates UI development steps for context-sensitive interactive applications. A *context* is defined as a triple of the form <*U*, *P*, *E*> where *U*, *P*, *E* respectively represent a user, a platform model, and a physical environment. A simplified version (Fig. 2) structures the development process for two contexts of use (here, *A* and *B*) into four levels of abstraction with respect to code:

1. **Task and concepts**: describe the various tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed.
2. **Abstract User Interface (AUI)**: a canonical expression of the renderings and manipulation of the domain concepts and functions in a way that is independent of the interaction modality (e.g., graphical, vocal or tactile). The elements used in the logical UI are abstractions of existing widgets.
3. **Concrete User Interface (CUI)**: concretizes a AUI into Concrete Interaction Objects (CIOs) so as to define widgets layout and interface navigation. This interface is now composed of existing UI widgets, but the widgets are independent of any particular toolkit.
4. **Final User Interface (FUI)**: The UI produced at the very last step of the reification process is supported by a multi-target development environment. A FUI is typically the UI code to be interpreted or compiled.

The downward arrows represent reification steps (forward engineering), from the more abstract to the operational interface. Reification is the transformation of a description (or of a set of descriptions) into a description (or a set of descriptions) whose level of abstraction is lower than that of the source one(s). In the multi-target reference framework, it is the inference process that covers the inference process from high-level abstract descriptions to run-time code. Upward arrows stand for abstraction steps. This process transforms any specifications into specifications at a higher level of abstraction. Here, abstraction is the elicitation of descriptions that are more abstract than the descriptions that serve as input to this process.

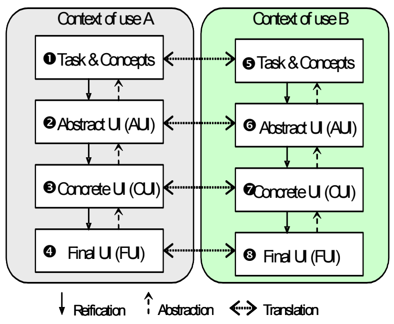


Figure 2. The Cameleon Reference Framework [4].

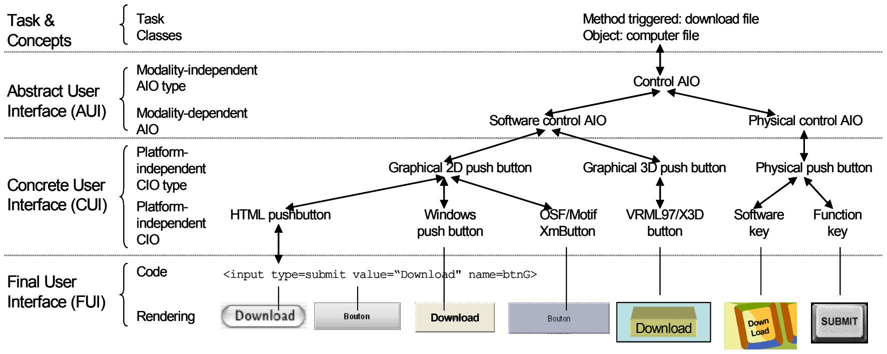


Figure 3. User interface expression in USIXML.

Finally, horizontal arrows correspond to the translation of the interface from one type of platform to another, or more generally, from one context to another. Not all steps should be achieved in a sequential ordering. Instead, locating **what** steps are performed, when, from which entry point and toward what subsequent step are important. In Fig. 2, transcoding tools start with a FUI for a source platform (❹) and transforms it into another FUI for a target platform (❽). Similarly, portability tools start with a CUI for a source platform (❸) and transforms it into another CUI for another platform (❼), that in turn leads to a new FUI for that platform (❽). To overcome shortcomings identified for these tools, there is a need to raise the level of abstraction by working at the AUI level. *UI Reverse Engineering* abstracts any initial FUI (❹) into concepts and relationships denoting a AUI (❷), which can then be translated into a new AUI (❻) by taking into account constraints and opportunities for the target platform. *UI Forward Engineering* then exploits this AUI (❻) to regenerate a new AUI adapted to this platform, by recomposing the CUI (❼) which in turn is reified in an executable FUI (❽).

3.2. The User Interface Description Language

To express a UI at any level of abstraction, as defined in Fig. 2, a User Interface Description Language (UIDL) has been defined and developed: UsiXML (USer Interface eXtensible Markup Language – http://www.usixml.org), a XML-compliant UIDL that covers all the aspects of Fig. 2 [17,18]. USIXML model collection (Fig. 7) is structured according to the four basic levels of abstractions defined in Fig. 2. Each level of Fig. 2 can be in turn further decomposed into sub-levels as illustrated in Fig. 3 [17]:

- At the FUI level, the rendering materializes how a particular UI coded in one language (markup, programming or declarative) is rendered depending on the UI toolkit, the window manager and the presentation manager. For example, a push button programmed in HTML at the code sub-level can be rendered differently, here on MacOS X and Java Swing. Therefore, the code sub-level is materialized onto the rendering sub-level.
- Since the CUI level is assumed to abstract the FUI independently of any computing platform, this level can be further decomposed into two sub-levels: platform-independent CIO and CIO type. For example, a HTML push-button belongs to the type "Graphical 2D push button". Other members of this category include a Windows push button and XmButton, the OSF/Motif counterpart.
- Since the AUI level is assumed to abstract the CUI independently of any modality of interaction, this level can be further decomposed into two sub-levels: modality-independent AIO and AIO type. For example, a software control (whether in 2D or in 3D) and a physical control (e.g., a physical button on a control panel or a function key) both belong to the category of control AIO.
- At the T&C level, a task of a certain type (here, download a file) is specified that naturally leads to AIO for controlling the downloading.

4. Derivation Rules

According to MDE [16], 'code to model' means that the code level (here, the FUI ❹) should be abstracted to any upper level (here, to CUI ❸, AUI ❷, and Task & Concepts ❶, respectively). If we want to achieve a certain level of independence with respect to the platform and the language, it is impossible to adopt a uniform approach for every type of FUI. This would require a semantic definition of nearly all programming and markup languages that could be ever used to develop a UI and a module capable of interpreting the code according to these semantics. This is far beyond the capabilities of what we want to reach. However, it is possible to express rules that consider the source language of the FUI so as to abstract it to any upper level. The abstraction can go until concrete, abstract, and task & concepts, independently or concurrently. This means that the reverse engineering can cope with multiple levels of abstraction simultaneously if needed.

For this purpose, we rely of *derivation rules* that derive UsiXML specifications at any level from the source code. The source code is of course varying from one platform to another, but the derivation rules do not fundamentally change. These derivation rules are represented as functions that can be interpreted both at design- and run-time, as they can be applied for every element. Fig. 4 reproduces a derivation rule that captures some aspect of the UI layout: Element.id represents the current element (to which the attribute is added). Each derivation rule examines language patterns of the source code and creates a corresponding structure in terms of a UI graph.

CheckAlignement(x,element)
$\forall$ x $\in$ T_s : x.IsInPath(center) $\rightarrow$ AddAttribute (element.id, "glueHorizontal, "middle")
$\forall$ x $\in$ T_s : x.IsInPath(div.align=center) $\rightarrow$ AddAttribute (element.id, "glueHorizontal", "middle")
$\forall$ x $\in$ T_s : x.IsInPath(div.align=right) $\rightarrow$ AddAttribute (element.id, "glueHorizontal", "right")
$\forall$ x $\in$ T_s : x.IsInPath(div.align=left) $\rightarrow$AddAttribute (element.id, "glueHorizontal", "left")
$\forall$ x $\in$ T_s : x.align=left$\rightarrow$AddAttribute (element.id, "glueHorizontal", "left")
$\forall$ x $\in$ T_s : x.align=right$\rightarrow$AddAttribute (element.id, "glueHorizontal", "right")
$\forall$ x $\in$ T_s : x.align=center$\rightarrow$AddAttribute (element.id, "glueHorizontal", "middle")
$\forall$ x $\in$ T_s : x.valign=bottom$\rightarrow$AddAttribute (element.id, "glueVertical", "bottom")
$\forall$ x $\in$ T_s : x.valign=top$\rightarrow$AddAttribute (element.id, "glueVertical", "top")
$\forall$ x $\in$ T_s : x.valign=middle$\rightarrow$AddAttribute (element.id, "glueVertical", "middle")
$\forall$ x $\in$ T_s : x.IsInPath(div.valign=middle)$\rightarrow$AddAttribute (element.id, "glueHorizontal", "middle")
$\forall$ x $\in$ T_s : x.IsInPath(div.valign=top)$\rightarrow$AddAttribute (element.id, "glueHorizontal", "top")
$\forall$ x $\in$ $T_{s:}$ x.IsInPath(div.valign=bottom) $\rightarrow$AddAttribute(element.id, "glueHorizontal","bottom")

Figure 4. Example of a derivation rule.

The set of derivation rules of Fig. 4 check if the element *x* is in the same path as an alignment modifier (div, center, align attributes) in a HTML page, and if it is the case, an attribute is added in the UsiXML specifications to reflect the horizontal position of the element. Fig. 5 shows a derivation rule for abstracting a "Submit" HTML pushbutton into a command button at the CUI level. This rule could be generalized for every push button in principle, but here the label clearly confirms the nature of the push button.

$\forall$ x $\in$ T_s : x = input $\wedge$ (x.type="button" $\vee$ x.type="submit" $\vee$ x.type="image" $\vee$ x.type="reset")
$\rightarrow$Addnode ("button", idbutton) where idbutton =$\sum$ node $\in$ T_t $\wedge$ AddAttribute (idbutton, "id", idbutton) $\wedge$ AddAttribute (idbutton, "name", idbutton) $\wedge$ AddAttribute (idbutton, "isVisible", "true")

Figure 5. Derivation rule for a "Submit" button.

Similar derivation rules exist for addressing the following families of problems (here for a Web page): detecting the UI structure, mirroring the UI structure in the specifications, recognition of UI layout, detection of widgets selected for input, output, and operations, identification of composition of elements that together create a group, identification of visual and auditive properties of a UI, identification of intra-page links, identification of extra-page links, etc.

For instance, Fig. 6 exemplifies how a sub-part of the Google home page can be reverse engineered at all levels thanks to those derivation rules: the "Google Search" submit button at the FUI level is abstracted into a push button belonging to a window at the CUI level. This can be in turn abstracted into an activator belonging to a container at the AUI level and into a "Launch Search" sub-task of the general "Search page" task. The ">>" indicates that a sequence is needed: the keywords need to be entered first, then "Launch Search" or "Launch Special Search" in any order ("|||" LOTOS operator).

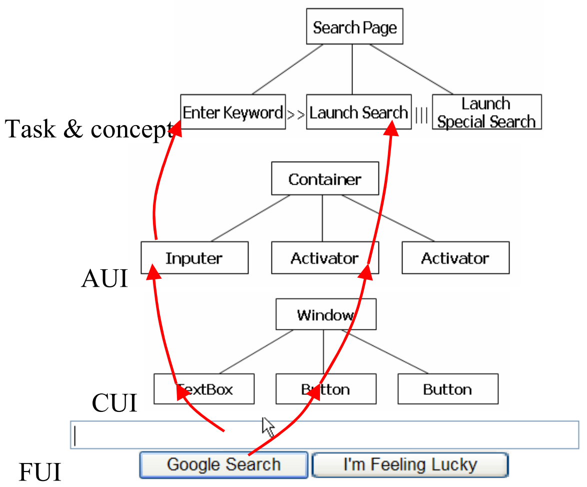


Figure 6. Reverse engineering of a FUI.

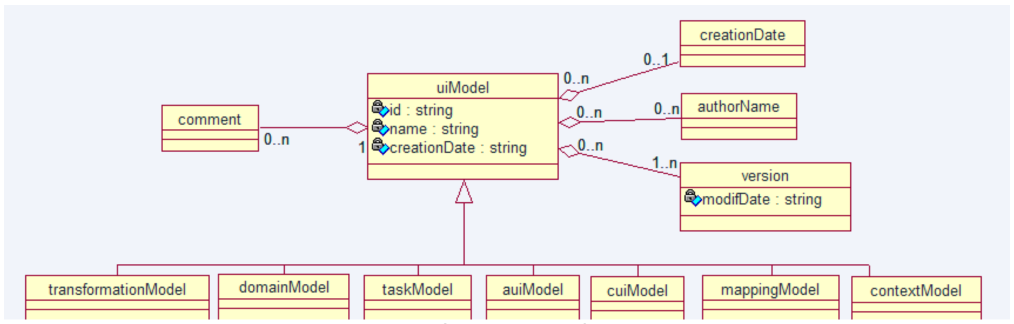


Figure 7. USIXML Model Collection.

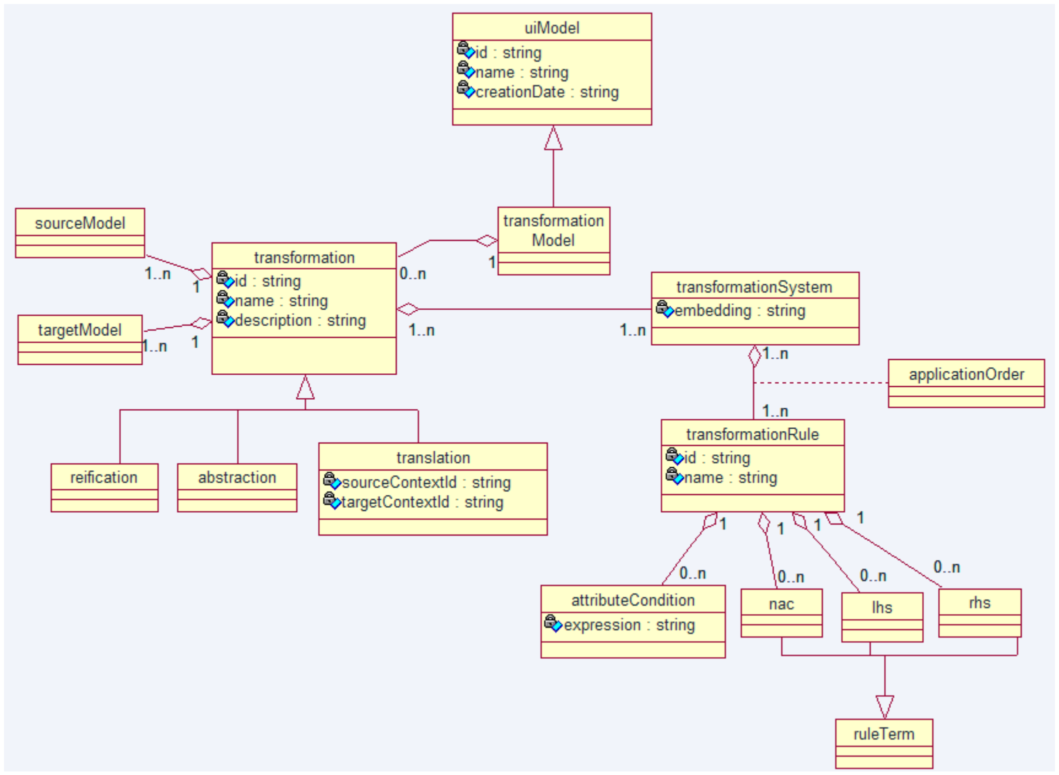


Figure 8. Transformation model as defined in UsiXML.

5. Graph grammars and transformations

5.1. Definitions of models

Thanks to derivation rules introduced in the previous Section 4, we can obtain UsiXML specifications of a FUI at any level of abstraction from the code level. In practice, it is possible either to derive all levels simultaneously from the FUI or to obtain only the CUI level and ask for further abstractions later on. According to MDE, this part is called 'model to model' transformation. To support them, Graph Transformation (GT) techniques [2,11,16] were chosen to formalize UsiXML, the language designed to support multi-path UI development, because it is (1) **Visual**: every element within a GT based language has a graphical syntax; (2) **Formal**: GT is based on a sound mathematical formalism (algebraic definition of graphs and category theory) and enables verifying formal properties on represented artifacts; (3) **Seamless**: it allows representing manipulated artifacts and rules within a single formalism. Furthermore, the formalism applies equally to all levels of abstraction of UsiXML.

Thanks to the four abstraction levels, it is possible to establish mappings between instances and objects found at the different levels and to develop transformations that find abstractions or reifications or combinations. For example, if a GUI needs to be virtualized, a series of abstractions is applied until the sub-level "Software control AIO" sub-level is reached. Then, a series of reification can be applied to come back to the FUI level to find out another object satisfying the same constraints, but in 3D. If the GUI needs to be transformed for a UI for augmented reality for instance, the next sub-level can be reached with an additional abstraction and so forth. The combinations of the transformations allow establishing development paths. To face multi-path development of UIs in general, UsiXML is equipped with a collection of basic UI models (i.e., domain model, task model, AUI model, CUI model, context model and mapping model) (Fig. 7) and a so-called *transformation model* (Fig. 8) that supports transformation between the different levels of abstraction. Beyond the AUI and CUI models, other models are defined:

- **taskModel**: is a model describing the interactive task as viewed by the end user interacting with the system. A task model represents a decomposition of tasks into sub-tasks linked with task relationships. Therefore, the decomposition relationship is the privileged relationship to express this hierarchy, while temporal relationships express the temporal constraints between sub-tasks of a same parent task.
- **domainModel**: is a description of the classes of objects manipulated by a user while interacting with a system.
- **mappingModel**: is a model containing a series of related mappings between models or elements of models. A mapping model gathers a set of inter-model relationships that are semantically related.
- **contextModel**: is a model describing the three aspects of a context of use in which a end user is carrying out an interactive task with a specific computing platform in a given surrounding environment. A context model consists of a user model, a platform model, and an environment model.
- **uiModel**: is the topmost superclass containing common features shared by all component models of a UI. A uiModel may consist of a list of component model sin any order and any number, such as task model, a domain model, an abstract UI model, a concrete UI model, mapping model, and context model. A UI model needs not include one of each model component. Moreover, there may be more than one of a particular kind of model component.

5.2. Definitions of model transformations

Transformations are specified using transformation systems. Transformation systems rely on the theory of graph grammars [2,11]. We first explain what a transformation system is and then illustrate how they may be used to specify UI model transformation. The proposed formalism to represent model-to-model transformation in UsiXML is graph transformations. UsiXML has been designed with an underlying graph structure. Consequently any graph transformation rule can be applied to a UsiXML specification. Graph transformations have been shown convenient and efficient for our present purpose in [25]. A transformation system is composed of several transformation rules. Technically, a rule is graph rewriting rule equipped with negative application conditions and attribute conditions. Fig. 6 illustrates how a transformation system applies to a UsiXML specification [17]: let *G* be a UsiXML specification, when 1) a Left Hand Side (*LHS*) matches into *G* and 2) a Negative Application Condition (*NAC*) does not matches into G (note that several NAC may be associated with a rule) 3) the *LHS* is replaced by a Right Hand Side (*RHS*). *G* is resultantly transformed into *G'*, a resultant UsiXML specification. All elements of *G* not covered by the match are considered as unchanged. All elements contained in the LHS and not contained in the RHS are considered as deleted (i.e., rules have destructive power). To add to the expressive power of transformation rules, variables may be associated to attributes within a LHS. Theses variables are initialized in the LHS, their value can be used to assign an attribute in the expression of the RHS (e.g., LHS : button.name:=x, RHS : task.name:=x). An expression may also be defined to compare a variable declared in the LHS with a constant or with another variable. This mechanism is called *attribute condition*.

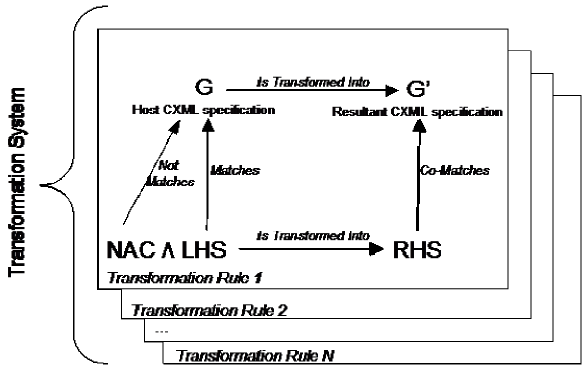


Figure 9. Transformation system in UsiXML.

A designer reverse engineers an HTML page with a tool (e.g., [3]) in order to obtain a CUI model. Transformation 1 (Fig. 10) is an abstraction that takes a button at the CUI level and abstracts it away into an abstract interaction object. The LHS selects every button and the method they activate and create a corresponding abstract interaction object equipped with a control facet mapped onto the method triggered by its corresponding concrete interaction object. Some behavioral specification is preserved at the abstract level. Specifying a behavior in UsiXML is also achieved through graph transformation. Rule 1 in transformation 1, in its LHS, embeds a fragment of transformation system.

```
<abstraction id="AB1" name = "AbstractButtonWIthControl" de-
scription = "this translation abstracts buttons into an AIO with an
activation facet"
    <transformationSystem id = "TR2" name="Transfo2"...>
    <transformationRule id = "rule1" name "abstractsBut">
    <lhs>
    <button ruleSpecificID="1" mapID="2">
    <behavior>
    <action>
    <transformationSystem>
    <transformationRule>
    <rhs>
    <method ruleSpecificID="3"
    mapID ="4" name=X />
    <isTriggeredBy isFired="true">
    <source sourceId="1">
    <target targetId="3">
    </isTriggeredBy>
    </rhs>
    </transformationRule>
    </transformationSystem>
    </action>
    </behaviour>
    </button>
    </lhs>

    <rhs>
    <abstractIndividualComponent ruleSpecificId="5">
    <control activatedMethod=X>
    </abstractIndividualComponent>
    <isAbstractedInto>
    <source sourceId="2"/>
    <target targetId="5"/>
    <isAbstractedInto>

    <button ruleSpecificId="1" mapID="2">
    <behavior>
    <transformationSystem>
    <transformationRule>
    <rhs>
    <method ruleSpecificID="3" mapID ="4"/>
    <isTriggeredBy isFired="true">
    <source sourceId="1">
    <target targetId="3">
    </isTriggeredBy>
    </rhs>
    </transformationRule>
    </transformationSystem>
    </behaviour>
    </button>
    </rhs>
    ...
    <nac.../>

    </transformationRule>
    </transformationSystem>
</abstraction>
```

Figure 10. Transformation 1: abstraction.

This may seem confusing at first sight but is very powerful at the end i.e., we use a mechanism transforming a UI behavioral specification into another one! In the RHS, one also see that a relationship is abstracted into has been created. This relationship ensures traceability of rule application and helps in maintaining coherence among different levels of abstraction. Fig. 11 shows a graphical representation of another transformation: select all boxes at the CUI level, set these boxes to “vertical”. All widgets contained in this box are then glued to the left of the box. Note the presence of a negative application condition that ensures that this rule will not be applied to an already formatted box. A general purpose tool for graph transformation called AGG (Attributed Graph Grammars) was used to specify this example. No proof the superiority of graphical formalism over textual ones, but at least the USIXML designer has the choice between both.

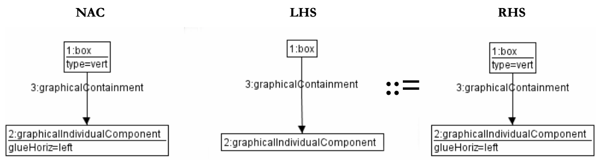


Figure 11. Graphical representation of a transformation.

Traceability (and reversibility as a side-effect) of model transformation is enabled thanks to a set of so-called *interModelMappings* (e.g., *isAbstractedInto*, *IsReifiedInto*, *isTranslatedInto*) allowing to relate model elements belonging to different models [17,18]. As so it is possible to keep a trace of the application of rules i.e., when a new element is created a mapping indicates of what element it is an abstraction, a reification, a translation, etc. Another advantage of using these mappings is to support multi-directional development is that they explicitly connect the various levels of our framework and realizes a seamless integration of the different models used to describe the system. Knowing the mappings of a model increases dramatically the understanding of the underlying structure of a UI. It enables to answer to questions like: what task a interaction object enables?, what domain object attributes are updated by what interaction object? what interaction object triggers what method? An environment called AGG (Attributed Graph Grammars tool) is used for this experiment. AGG consists of a genuine programming environment based on graph transformations. It provides 1) a programming language enabling the specification of graph grammars 2) a customizable interpreter enabling graph transformations. AGG was chosen because it allows the graphical expression of directed, typed and attributed graphs (for expressing specifications and rules).

More complex transformation rules could also be defined, such as, for instance: for each editable graphical individual component belonging to the CUI, create an abstract individual component equipped with an input facet at the AUI level (Fig. 12), for each abstract individual component equipped with a navigation facet create a task with action type “start/go” on an item of type “element” (Fig. 13). WEBREVENGE [24] contains such rules for recovering a task model (at the T&C level in the Cameleon framework [4]), but these rules are hardcoded in the tool, thus preventing the designer from applying any modification, generalization, etc.

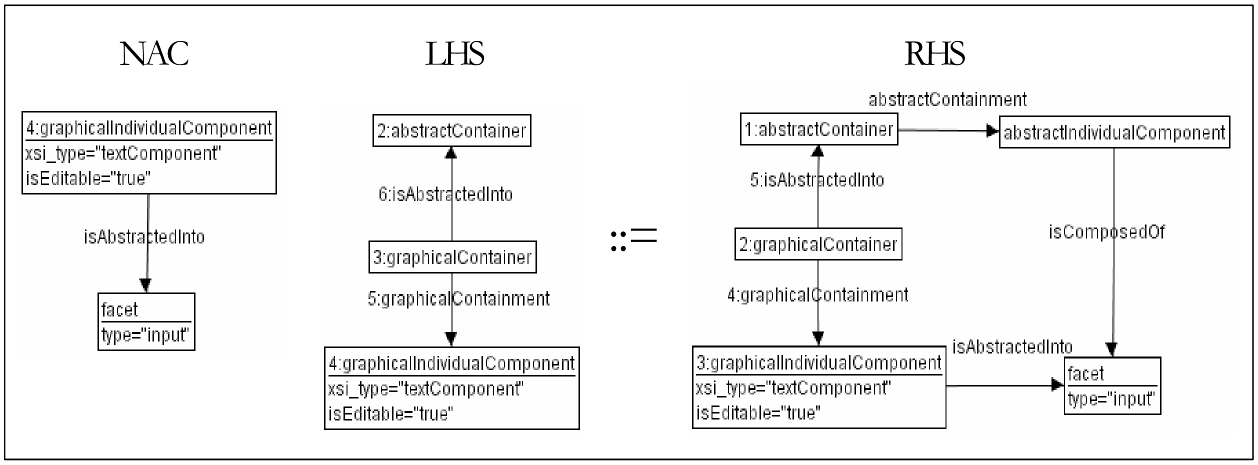


Figure 12. Transformation rule from CUI to AUI.

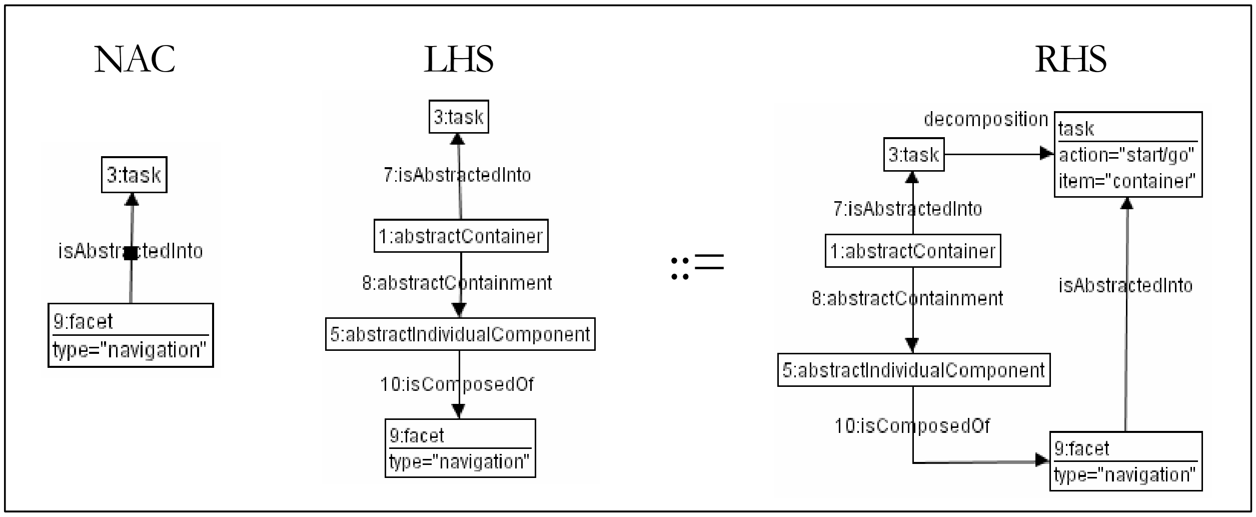
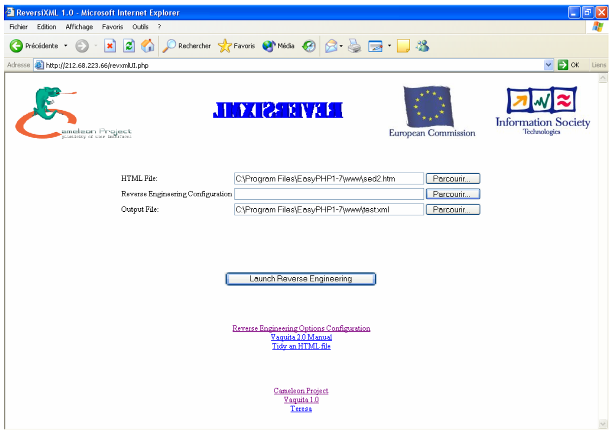


Figure 13. Transformation rule from AUI to Task & Concepts.

6. Software support for UI reverse engineering

To ensure the different operations required by UI re-engineering and reverse engineering as defined above, two applications have been developed: REVERSIXML applies derivation rules to reverse engineer a web page into CUI and AUI expressed in UsiXML. Contrarily to Vaquita [3], REVERSIXML contains a parsing engine that automatically processes all derivation rules so as to produce multiple specifications in UsiXML, as opposed to XIML (www.ximl.org). XIML contains only one level of abstraction and is presentation oriented. UIML [1] is another comparable language for specifying a UI, as opposed to UI markup languages such as (X)HTML, XAML (www.xaml.org).


ReversiXML 1.0 - Microsoft Internet Explorer

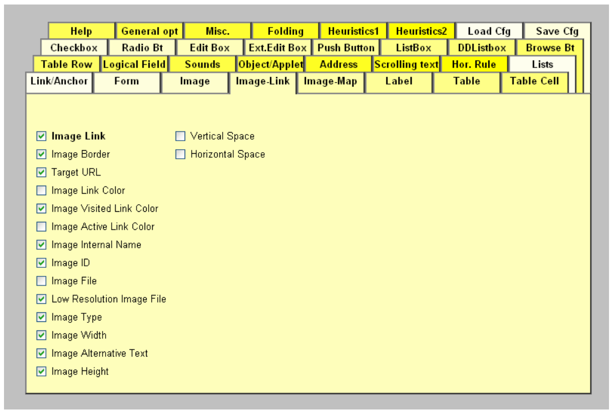
http://212.68.223.66/revxmlUI.php

HTML File: C:\Program Files\EasyPHP1-7\www\sed2.htm

Reverse Engineering Configuration

Output File: C:\Program Files\EasyPHP1-7\www\test.xml

Figure 14. Main screen of ReversiXML for HTML reverse engineering.



- ✓ **Image Link**
- ✓ Image Border
- ✓ Target URL
- ☐ Image Link Color
- ✓ Image Visited Link Color
- ☐ Image Active Link Color
- ✓ Image Internal Name
- ✓ Image ID
- ☐ Image File
- ✓ Low Resolution Image File
- ✓ Image Type
- ✓ Image Width
- ✓ Image Alternative Text
- ✓ Image Height
- ☐ Vertical Space
- ☐ Horizontal Space

Figure 15. Control Panel of Derivation rules.

In this way, REVERSIXML captures more information than Vaquita (e.g., the behavior) and at more levels of abstractions. This application (Fig. 14) is accessible at http://www.isys.ucl.ac.be/bchi/research/reversi/RevXMLUI.php. It is an on-line application developed in PHP V5.0 coupled with Tidy (for well-formed web pages). A control panel (Fig. 15) gives access to the selection of derivation rules to be processed. Such a configuration can be saved in a configuration file for future usage. The second application developed in TRANSFORMIXML: it is a Java-based application that performs the following steps: it takes a UsiXML file as input and a series of transformations to be applied. The current specifications in the UsiXML file are then transformed into their graph counterparts so as to perform the graph transformations. Once these transformations have been processed, the resulting graph is exported back in UsiXML in order to obtain the UsiXML specifications of the resulting UI. Since the transformations can be applied between any pair or level (except the code level), the tool could be considered as rather generic. The transformation engine used by TRANSFORMIXML is coming from the AGG tool.

7. Conclusion

A series of about 120 derivation rules belonging to different categories of reverse engineering have been incorporated in REVERSIXML to support full HTML reverse engineering. All tasg and attributes of HTML are supported by this set of rules. But it does not cover other markup or imperative languages that could be used in HTML pages, such JavaScript, Perl, Flash, etc. This allows properties of interested to be addressed to some extent. In particular, production rules can be added and modified at any time, they are accessible. Their application can be made observable, traceable, and controllable. Multiple levels of abstraction can be supported: FUI to CUI is supported by tools like [2] or others that directly converts Web pages into a XML-compliant language. Other productions can span as following: CUI to AUI, AUI to T&C. Not only these rules can be fired independently of each other (thus reaching one level at a time), but also simultaneously (thus reaching multiple levels at the same time). Once a Web page has been transformed into USIXML, these specifications can then be submitted to TRANSFORMIXML to apply productions on demand. Provided that a converter exists to transform an existing piece of code into its USIXML counterpart, the same productions can be applied similarly.

8. Acknowledgements

We gratefully acknowledge the support of the SIMILAR network of excellence, the European research task force creating human-machine interfaces similar to human-human communication (http://www.similar.cc), Sixth Framework Program (European Commission, FP6-IST1-2003-507609) and the Salamandre project, Région Wallonne, Initiatives III (Walloon Region, Convention n°001/4511). For more information on the specification language and tools, see http://www.usixml.org.

9. References

[1] M. Abrams, C. Phanouriou, A. Batongbacal and J. Shuster, UIML: An Appliance-Independent XML User Interface Language, *Proc. of 8th World Wide Web Conference WWW'8* (Toronto, 11-14 May 1999), Computer Networks, Vol. 31, No. 11-16, pp. 1695-1708.

[2] A. Agrawal, G. Karsai, and A. Ledeczi, "An End-to-end Domain-Driven Software Development Framework", *Comp. of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming OOPSLA'03* (Anaheim, 26-30 Oct. 2003), ACM Press, New York, pp. 8-15.

[3] L. Bouillon and J. Vanderdonckt, "Retargeting Web Pages to other Computing Platforms", *Proc. of IEEE 9th Working Conf. on Reverse Engineering WCRE'2002* (Richmond, 29 Oct.-1 Nov. 2002), IEEE Computer Society Press, Los Alamitos, 2002, pp. 339-348.

[4] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A Unifying Reference Framework for Multi-Target User Interfaces", *Interacting with Comp.*, Vol. 15, No. 3, June 2003, pp. 289-308.

[5] E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, Vol. 1, No. 7, January 1990, pp. 13-17.

[6] J.R. Cordy, "Hints on the Design of User Interface Language Features - Lessons from the Design of Turing", in *Languages for Developing User Interfaces*, Jones and Bartlett, Boston, 1992, pp. 329-340.

[7] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider, "Software Engineering by Source Transformation - Experience with TXL", *Proc. of IEEE 1st Int. Workshop on Source Code Analysis and Manipulation SCAM'01*

(Florence, 10 Nov. 2001), IEEE Computer Society Press, Los Alamitos, 2001, pp. 168-178.

[8] J.R. Cordy, "Generalized Selective XML Markup of Source Code Using Agile Parsing", *Proc. of IEEE 11th Int. Workshop on Program Comprehension IWPC'2003* (Portland, 10-11 May 2003), IEEE Computer Society Press, Los Alamitos, 2003, pp. 144-153.

[9] V. Crescenzi, G. Mecca, P. Merialdo, and P. Missier, "An Automatic Data Grabber for Large Web Sites", Proc. of the 30th Int. Conf. on Very Large Data Bases VLDB'2004 (Toronto, August 31-September 3, 2004), Morgan Kaufmann, 2004, pp. 1321-1324.

[10] G.A. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza, "An Approach for Reverse Engineering of Web-Based Applications", *Proc. of 8th IEEE Working Conference on Reverse Engineering WCRE'2001* (Stuttgart, 5-7 October 2001), IEEE Computer Society Press, Los Alamitos, 2001, pp. 231-240.

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

[12] D.W. Embley, D.M. Campbell, Y.S. Jiang, Y.-K. Ng, R.D. Smith, S.W. Liddle, and D.W. Quass, "A conceptual-modeling approach to extracting data from the web", *Proc. of the 17th Int. Conf. on Conceptual Modeling ER'98* (Singapore, November 16-19, 1998), Lecture Notes in Computer Science, Vol. 1507, Springer-Verlag, Berlin, 1998, pp. 78-91.

[13] Y. Gaeremynck, L.D. Bergman, and T. Lau, "MORE for Less: Model Recovery from Visual Interfaces for Multi-Device Application Design", *Proc. of the 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2003* (Miami, 12-15 Jan. 2003), ACM Press, NY, 2003, pp 69-76.

[14] G. Gottlob, Ch. Koch, R. Baumgartner, M. Herzog, and S. Flesca, "The Lixto Data Extraction Project - Back and Forth between Theory and Practice", *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems PODS'2004* (Paris, June 14-16, 2004), ACM Press, New York, 2004, pp. 1-12

[15] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, and B. Matichuk, "Modeling the System-User Dialog Using Interaction Traces", *Proc. of 8th IEEE Working Conf. on Reverse Engineering WCRE'2001* (Stuttgart, 5-7 Oct. 2001), IEEE CS. Press, Los Alam., 2001, pp. 208-217.

[16] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, "Transformation: The Missing Link of MDA", *Proc. of the 1st Int. Conf. on Graph Transformation ICGT'02* (Barcelona, 7-12 Oct. 2002), Lecture Notes in Computer Science, Vol. 2505, Springer-Verlag, Berlin, 2002, pp. 90-105.

[17] Q. Limbourg and J. Vanderdonckt, "UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence", *Engineering Advanced Web Applications*, Matera, M., Comai, S. (Eds.), Rinton Press, Paramus, 2004, pp. 325-338.

[18] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and Lopez, V. "UsiXML: a Language Supporting Multi-Path Development of User Interfaces, *Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS'2004* (Hamburg, July 11-13, 2004). Lecture Notes in Computer Science, Vol. 3425, Springer-Verlag, Berlin, 2005, pp. 207-228.

[19] E. Merlo, P.-Y. Gagné, and A. Thiboutôt, "Inference of Graphical AUIDL Specifications for the Reverse Engineering of User Interfaces", *Proc. of IEEE Int. Conf. on Soft-ware Maintenance* (19-23 September 1994), IEEE Computer Society Press, Los Alamitos, 1994, pp. 80-88.

[20] E. Merlo., P.-Y. Gagné, J.-F. Girard, K. Kontogiannis, L. Hendren, P. Panagaden, and R. De Mori, "Reengineering User Interfaces", *IEEE Software*, Vol. 12, No. 1, January 1995, pp. 64-73.

[21] M.M. Moore, "Representation Issues for Reengineering Interactive Systems", *ACM Computing Surveys*, Vol. 28, No. 4, December 1996. Article # 199.

[22] M.M. Moore, "Rule-Based Detection for Reengineering User Interfaces", *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering WCRE'96* (Monterey, 8-10 Nov. 1996), IEEE CS Press, Los Alam., 1996, pp. 42-49.

[23] M.M. Moore and S. Rugaber, "Using Knowledge Representation to Understand Interactive Systems", *Proc. of the 5th IEEE Int. Workshop on Program Comprehension IWPC'97* (Dearborn, 28-30 May 1997), IEEE Computer Society Press, Los Alamitos, 1997, pp. 60-69.

[24] L. Paganelli and F. Paternò, "Automatic Reconstruction of the Underlying Interaction Design of Web Applications, *Proc. of the 14th ACM Int. Conf. on Software Engineering and Knowledge Engineering SEKE'02* (Ischia, 15-19 July 2002), ACM Press, NY, pp. 439-445.

[25] H. Partsch and R. Steinbruggen, "Program Transformation Systems", *ACM Computing Surveys*, Vol. 15, No. 3, September 1983, pp. 199-236.

[26] 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.

[27] S. Rugaber, "A Tool Suite for Evolving Legacy Software", *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.

[28] K.A. Schneider and J.R. Cordy, "Abstract User Interfaces: a Model and Notation to Support Plasticity in Interactive Systems", *Proc. of Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2002* (Rostock, 12-14 June 2002), Lecture Notes in Computer Science, Vol. 2220, Springer-Verlag, Berlin, 2002, pp. 28-48.

[29] E. Stroulia, J. Thomson, and Q. Situ, "Constructing XML-speaking Wrappers for WEB Applications: Towards an Interoperating WEB", *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.

[30] E. Stroulia, M. El-Ramly, P. Iglinski, and P.G. Sorenson, "UI Reverse Engineering in Support of Interface Migration to the Web", *Journal of Automated Software Engineering*, Vol. 10, No. 3, July 2003, pp. 271-301.