View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by IntechOpen

We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Automatic Generation of User Interface Models and Prototypes from Domain and Use Case Models

António Miguel Rosado da Cruz and João Pascoal Faria
Instituto Politécnico de Viana do Castelo / Fac. Engenharia Univ. do Porto e INESC Porto
Portugal

1. Introduction

The development of interactive systems typically involves the separate design and development of disparate system components by different software developers. The user interface (UI) is the part of an interactive system through which a user can access the system functionality. User interface development is a complex task that typically involves the construction of prototypes and/or models. A prototype facilitates the communication with the stakeholders, especially with the end users, and allows for the validation of elicited requirements. Modelling is a well established way people take for dealing with complexity. A model allows one to focus on important properties of the system being modelled and abstract away from unimportant issues. Software models may capture relevant parts of the problem and solution domains and are typically used as a means for reasoning about the system properties and for communicating with the stakeholders.
The user interface tends to be viewed differently, depending on what community the UI designer belongs to. UI designers that are more identified with the Software Engineering (SE) community tend to highlight the system functionality issues, and how it encapsulates system behaviour to provide to the user. UI designers that are more identified with the Human-Computer Interaction (HCI) community tend to focus on user task analysis and the way the user shall work on the UI.
According to the HCI perspective, one of the concerns that shall be modelled is the user intended tasks on the interactive system, and this is made through the development of user task analysis. Typically, task analysis and modelling involve the development of goal and task hierarchies and the identification of objects and actions involved in each task (Dix et al., 1998). Besides this task model, a view of the UI relevant aspects of the system core structure and functionality may also be modelled, along with a UI presentation model, in order to complete the whole interactive system model.
In the SE community, a common practice is to build a Unified Modelling Language (UML) system model, comprising a domain model and a use case model, supplemented by a non-functional UI prototype, in the early stages of the software development process (Jacobson et al., 1999; Pressman, 2005). The domain model captures the main system's domain classes, its attributes, relations and, in some cases, its operations, through UML class diagrams. The use case model captures the main system functionalities from the user's point of view

Source: User Interfaces, Book edited by: Rita Mátrai,
ISBN 978-953-307-084-1, pp. 270, May 2010, INTECH, Croatia, downloaded from SCIYO.COM

www.intechopen.com

through UML use case diagrams and accompanying textual descriptions. The UI prototype is used to elicit and validate requirements with the stakeholders, and is typically not integrated with the system model. Also, the use case and domain models are typically ambiguous and incomplete, having most of the constraints and business rules specified in textual natural language, and preventing the automatic validation of its consistency. This kind of models is mainly used for abstracting away from system complexity, helping reasoning about the system and facilitating communication between the team members and with the stakeholders.

Model driven development (MDD) approaches, like Domain Specific Modelling – DSM – (Kelly & Tolvanen, 2008), or the OMG's Model Driven Architecture – MDA – (Kleppe et al., 2003), are based on the successive refinement of models and on the automatic generation of code and other sub-models, thus requiring the unambiguous definition of models.

After briefly surveying the current approaches to the automatic generation of UI models and prototypes, this chapter presents an approach for the automatic generation of form-based applications within a model-driven software development setting (Cruz & Faria, 2007). The approach proposed involves the iterative and incremental development of a domain model, and optionally a use case model, by the modeller, and the testing of an automatically generated executable prototype.

2. Current model-based approaches to user interface automatic generation

This section briefly surveys and compares the main current approaches for the automatic generation of user interface prototypes (UIP), or UI models (UIM), from non-UI system models, like domain or application structural models, use case or task models, and some kind of system behavioural models.

As stated before, typical methodologies for modelling interactive applications use disparate views, or submodels, to capture different aspects of the system (domain or application model, task model, dialogue model, abstract and concrete presentation models) (Pinheiro da Silva, 2000). Most of existing approaches to UI generation demand the specification of a UI model (see for example the approaches surveyed by Pinheiro da Silva (Pinheiro da Silva, 2000)).

2.1 The XIS approach

Few approaches found in the literature allow a model-to-model generation of a UIM/UIP within a MDD setting. ProjectIT and the XIS profile and approach (Silva et al., 2007; Silva & Videira, 2008; Silva, 2003) promote a vision that separates modelling of different system concerns into disparate sub-models, namely an Entities view, a Use Case view and a User Interface view.

A XIS-based model may follow a dummy or a smart approach. In the dummy approach, the entities view is composed only of a domain model, the use case view only defines an actors' hierarchy (actors view) and a user interface view (an abstract presentation model) must be fully specified comprising an Interaction Spaces View, which defines the abstract screens that serve as interface between the users and the system, and the Navigation Space View, which specifies the possible navigation flows between the defined interaction spaces.

A XIS-based model within the smart approach shall have the following sub-models:

- Entities View: Composed of a Domain View and a Business Entities View. The Domain View models the domain entities by using a UML class model with properly XIS-profile

stereotyped classes, attributes, associations, and enumerations. The Business Entities View is used to group together a set of domain entities, in a coarser granularity entity («XisBusinessEntity») that shall be manipulated in the context of a use case. A business entity must designate a master entity and a sequence of detail entities, or it must define an aggregation of other business entities.

- Use-Cases View: Subdivided in the Actors View, which defines the hierarchy of actors that can perform operations on the system, and the UseCases View, which identifies use cases and relates each actor with the use cases that it can perform. The UseCases View also associates each use case to the business entity on which the actors related to that use case can perform operations («XisOperatesOnAssociation»). This stereotype has a tagged-value, operations, which enables the definition of the set of allowed operations that must be subset of the operations identified in the business entities view for that business entity.

In the smart approach, XIS allows the generation of models from models - that is the case of the User-Interfaces View in the smart approach, although it is not yet available in the ProjectIT-Studio tool.

A XIS model may, then, be inputted to a model to code (M2C) generation process, made available in ProjectIT through templates. All model views in XIS are platform independent, and M2C scripts operate on XIS models. The XIS profile does not support OCL nor the full specification of operations' syntax. It only allows the declaration of operations' name, not its signature, nor semantics (body or pre-/post-conditions) (Saraiva & Silva, 2008; Silva et al., 2007).

2.2 The OO-Method approach

The OO-Method approach / Olivanova (Pastor & Molina, 2007; Pastor et al., 2004; Molina, 2004; Molina & Hernández, 2003) aims at producing a formal specification of a software system in an executable formal object-oriented language named OASIS. But, in order to avoid the complexity traditionally associated to the use of formal methods, the OOMethod only asks for the software engineer to graphically model a system at a conceptual level - the conceptual model –, which is then translated, through a set of modelling patterns provided by the method, to an OASIS specification – the execution model. The OO-Method starts, then, with the construction of a conceptual model, which is in turn composed of the following sub-models (Pastor et al., 1997; Pastor & Insfrán, 2003; Pastor & Molina, 2007):

- Object Model. Represented through a UML class diagram, capturing domain classes and classes associated to user roles. For each class, the object model captures information about its attributes, services (operations triggered by message events with the same name), derived attributes, constraints and relationships (aggregation and inheritance).
- Dynamic Model. Used to specify valid object lifecycles and interaction between objects. To specify valid object lifecycles, a state transition diagram is used per class, representing its valid states and the valid transitions between states. Transitions may have attached control or triggering conditions. Object interactions are represented by a (non-UML) interaction diagram for the whole system. Two types of interactions are possible: Triggers, which are services of objects that are automatically activated when a condition is satisfied; and, Global interactions, which are transactions involving services of different objects.

- Functional Model. Captures the semantics attached to any change of state, as a consequence of a service occurrence. For that, it is declaratively specified how each service changes the object state depending on the arguments of the involved service and the current object state. Nevertheless, for not demanding the knowledge of OASIS, the OO-Method provides a model where the software engineer only has to categorize every attribute among a predefined set of three categories and introduce the relevant information depending on the corresponding selected category (Pastor et al., 1997; Pastor & Insfrán, 2003).
- Presentation Model. The last step is to specify how users will interact with the system (Pastor & Insfrán, 2003). Just-UI adds to the OO-Method a Presentation Model that intends to capture the characteristics of the User Interface as they are conceived at conceptual level during the requirements elicitation phase of a system's development process (Molina et al., 2001; Molina & Hernández, 2003). The kind of information that is collected in the presentation model of the OO-Method is based on conceptual interface patterns based on Abstract Interaction Objects (AIO).

The abstract execution model is based on the concept of conceptual modelling patterns. The OlivaNova transformation engines provide a well-defined software representation of the conceptual modelling patterns in the solution space.

2.3 The ZOOM approach

The ZOOM approach to interactive systems modelling and development (Jia et al., 2005) provides a set of process, notations, and supporting tools that enable model-driven development. ZOOM, which stands for Z-based OO modelling notation, is an object-oriented (OO) extension to the formal specification language Z. ZOOM separates an application into three parts – structure, behaviour, and user-interface – and provides three separate, but related, notations to describe each of those parts: ZOOM for structural models; ZOOM-FSM for specifying behavioural models through finite state machines; and, ZOOM-UIDL, a user interface description language for UI models. ZOOM provides a Java-like textual syntax for structural and behavioural models and an XML-based language for the User-Interface model. Furthermore, ZOOM provides a graphical representation of models consistent with UML diagrams (Jia et al., 2007; Jia et al., 2005), enabling a graphical formal modelling of a software system.

An event-based framework integrates the different parts of a ZOOM model, enabling its validation and execution.

ZOOM may be used in a MDD setting by applying model “compilation” tools. These, are tools that enable the generation of a complete application from a ZOOM model, exposing its functional requirements through a UI generated from the UI model. The generated code must not only meet all functional requirements, but the generation process must address the choice of architecture, data structures and algorithms (Jia et al., 2005; Jia et al., 2007).

2.4 Other approaches

In (Martinez et al., 2002) a methodology for deriving UIs from early requirements existing in an organization’s business process model is presented. Martinez’s approach follows a set of heuristics for extracting use cases and actors from the business process model. Each use case’s normal and exceptional scenarios are then specified using message sequence charts enriched with UI related information. These UI enriched sequence diagrams are then used

for automatically generating application forms and state transition diagrams for the interface objects and control objects present in the sequence diagrams.
UI generation is also approached in (Elkoutbi et al., 2006) based on the identification of usage scenarios. Elkoutbi's approach starts from a system domain structural model with OCL constraints and a use case model, but proceeds by formalizing each use case through a set of UML collaboration diagrams, each corresponding to a use case scenario. Then, each collaboration diagram message is manually labelled with UI constraints (*inputData* and *outputData*) that identify the input and output message parameters for the UI. From the UI constraints it then automatically produces message constraints with UI widget information. Statechart diagrams are then derived from the UI labelled collaboration diagrams on a per use case basis. A statechart is created for each distinct class in a collaboration diagram. Then, state labeling and statechart integration are done incrementally, in order to obtain only one statechart per collaboration diagram, that is, per usage scenario. Elkoutbi's approach is then able to derive UI prototypes for every interface object defined in the class diagram.
Forbrig et al. (Wolff et al., 2005a; Wolff et al., 2005b; Javahery et al., 2007; Radeke et al., 2007; Forbrig et al., 2004; Reichart et al., 2004) developed an approach that interactively generates an abstract UI model, and then a concrete UI, by applying UI-patterns to elements of UI sub-models (e.g. task models). The approach starts by constructing a task model and a business objects model, complemented with a user model, that capture relevant information from the user (e.g.: typical tasks, its type, frequency and importance, preferences), and a device model, that captures relevant information about the device. Then, from the previous models, a set of selectable patterns is identified enabling its selection by the modeller in order to obtain more concrete models. This is not an automatic approach, but one that enables a computer assisted development of interactive applications by selecting different types of patterns at different levels of abstractions. Tools like DiaTask (Wolff et al., 2005b) and PIM Tool ("Patterns in Modelling" tool) (Radeke et al., 2007) enable this computer assisted approach.

2.5 Discussion of current approaches

Elkoutbi's and Martinez's approaches enable the semi-automatic generation of a UIP from non-UI models, but they do not produce an intermediate UIM. Also, the amount of work involved in the production of the demanded models makes the approaches of little use for software development teams.
Forbrig's approach facilitates the model transformation processes by making the modeller choose between a set of eligible patterns, but it is not an automatic generation approach.
The XIS/ProjectIT, just like the OO-Method/Olivanova and the ZOOM approach are able to produce a fully functional (executable) application, but the demanded input models are very time consuming and arduous to build. The need to attach a stereotype to every model element, in XIS, makes the models hard to read and build.
All except the XIS smart approach and partially the OO-Method demand the full construction of a UI model. The XIS smart approach enables the derivation of a UIM, called user interfaces view, by demanding the construction of three non-UI models, a domain model, a business entities model and a use case model. This approach to the UIM derivation is simpler than its full construction, but forces the modeller to repeat definitions that were already made in the domain model, by defining XIS business entities. XIS business entities select domain entities relations to provide a lookup or master/detail pattern to the UI needed for the interaction

inside the context of a use case (Silva, 2003; Silva et al., 2007). This way, the Business Entities view is the XIS way to define UI structure and functionality, though possible operations can be further restricted when associating the business entity to a use case.
It is not possible, in XIS, to specify complex behaviour - only predefined CRUD operations may be attached to Business Entities and to the connection between the use cases and business entities.
ZOOM and the OO-Method allow the definition of complex behaviour by using a formal specification language, ZOOM or OASIS respectively, though the OO-Method also provides a way that enables the definition of some behaviour without demanding the knowledge of OASIS from the software engineer.
From the previous survey and discussion the main drawbacks of existing approaches to UI automatic code generation have been identified, and are summarized below:

- In general, current approaches demand too much effort, from the modeller, in order to build the system models inputted to the approaches. They don't allow a gradual approach to system modelling if one wants to generate a (prototype) application to iteratively evaluate and refine the model. All models expected by one approach must be fully developed before code generation may be available, except with the OO-Method (Pastor et al., 2004; Molina, 2004; Pastor et al., 1997), to a certain point, because it may generate a concrete UI given only a structural model. But the OO-Method does not permit the specification of a use case driven system model.
- Most of the approaches demand the manual construction of a UI model from scratch, in order to be able to produce a concrete user interface for an interactive application. The exception is the XIS smart approach (Silva et al., 2007), that enables the generation of a user interface model from the core system model, but the generated UI is rather limited in what concerns its flexibility and the core system behaviour.
- Current approaches don't allow the generation of an executable prototype from the available system models, that would permit to interactively validate the model through a UI with the users and other stakeholders, and refine the model in a sequence of iterative steps.
- Most of the existing approaches don't take advantage of the specification of class state constraints (invariants) or of operations pre-conditions to enhance the usability of the generated UI. The exception is the ZOOM approach (Jia et al., 2005; Jia et al., 2007), and partially the OO-Method (Pastor et al., 2004; Molina, 2004; Pastor et al., 1997).
- Existing approaches don't take advantage of the use of constructs typically found in task models (e.g.: sequencing, alternative) for detailing use cases (Paternó, 2001).
- Existing approaches don't allow the definition of the semantic of operations at class level. Again, the exception is the ZOOM approach (Jia et al., 2005; Jia et al., 2007), and partially the OO-Method (Pastor et al., 2004; Molina, 2004; Pastor et al., 1997).
- With the partial exception of the OO-Method (Pastor et al., 2004; Molina, 2004; Pastor et al., 1997), existing approaches don't allow the definition of triggers, i.e. actions to be executed when certain events occur or certain conditions hold. Triggers activated by an operation's invocation are a way of modifying or adding behaviour to CRUD or other operations. Using triggers it is possible to specify business rules that involve several classes' operations. The OO-Method only allows the specification of condition activated triggers but not invocation activated triggers.

In the next section, a general presentation of the proposed approach is made, aiming the automatic generation of user interface models and prototypes from non-UI system models.

www.intechopen.com

3. Proposed approach to model-driven user interface generation

The proposed approach to model-driven UI generation and development (Cruz & Faria, 2007; Cruz & Faria, 2008; Cruz & Faria, 2009), illustrated in Fig.1, enables the automatic generation of user interface models (UIM) and executable user interface prototypes (UIP) from early, progressively enriched, non-UI system models.

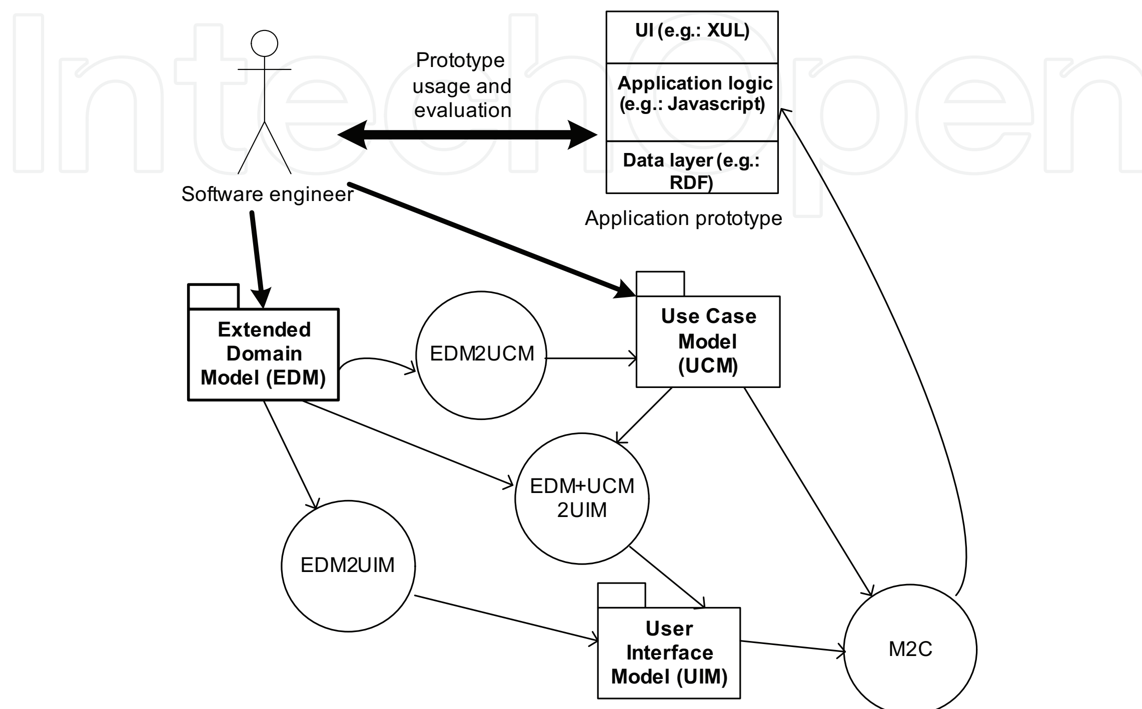


Fig. 1. General approach to UI generation.

In the first iterations, a simple domain model (DM) is constructed, represented by a UML class diagram, with classes (base domain entities), attributes and relationships. From this DM a simple UI can be automatically generated (by the EDM2UIM process, a model to model transformation, and model to code transformation - M2C -, in Fig. 1) supporting only the basic CRUD operations and navigation along the associations defined.
In subsequent iterations, the domain model is extended with additional features (to be explained in more detail in section 4) that allow the generation of richer user interfaces: OCL constraints, default values, derived attributes, derived entities (views), user-defined operations, and triggers. From this extended domain model (EDM), it is possible to generate validation routines from OCL class invariants and operations' pre-conditions, thus influencing what the user is able to do in the generated user interface. Derived classes allow the generation of UI forms with a better business tailored data structure.
Simultaneously, the modeller may develop a use case model (UCM), integrated with the EDM. This UCM will enable the separation of functionality by actor, and its customization (e.g.: hiding functionality for some actors). Corresponding UI models and prototypes are then automatically generated from both the EDM and UCM (EDM+UCM2UIM and M2C processes in Fig. 1). As will be explained in section 5, there is a full integration between the UCM and EDM, as use case specifications are established over the structural domain model.
On each iteration, the generated UI may be tuned by a UI designer in two points of the process: after having generated an abstract UIM, but before generating a concrete UI; and,

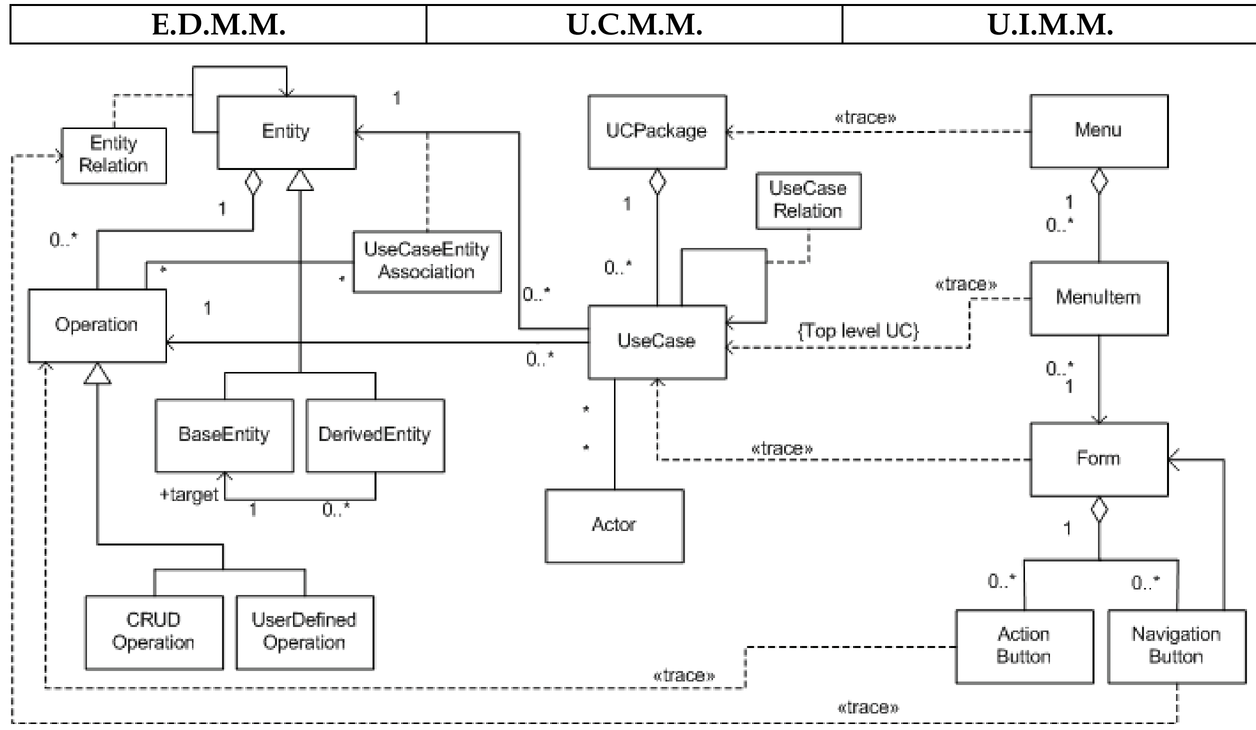


Fig. 2. Excerpt of the conceptual metamodels and their relations.

after generating a concrete UI in a XML-based UI description language (e.g.: XUL), which allows for the *a posteriori* customization and application of style sheets. A proof of concept tool has been developed for fully automating the EDM2UIM, EDM+UCM2UIM and M2C processes. The prototyped M2C process uses XUL to represent an executable UI description, JavaScript for the executable functionality and RDF to persist data.

Each of the models (EDM, UCM and UIM) presented in Fig. 1 is an instance of a defined metamodel, of which an excerpt is shown in Fig. 2 (EDMM, UCMM and UIMM, respectively). Elements in the user interface model are traced back to elements in the UCM or EDM, e.g.:

- A Menu in the UI traces back to a Use Case (UC) Package in the UCM;
- a Menu Item traces back to a top-level use case in the UCM, i.e. a use case that directly links to an actor;
- A Form can be traced back to a use case, which is always related to a base or derived domain Entity;
- An Action Button may trace back to a CRUD operation that may be identified in a use case, or to a user defined operation.

In the next two sections the mappings for deriving a UI model from one or both of the other models (EDM and UCM), as depicted in Fig. 1, are defined.

A set of rules has also been defined for transforming an EDM into a default UCM (EDM2UCM process), and these are briefly presented in section 6.

4. Automatic generation of a user interface model from an extended domain model

This section presents the rules defined to transform different elements of the extended domain model into appropriate user interface elements and their underlying functionality.

4.1 Extended domain model and transformation rules

Besides classes (domain entities), attributes and relationships, an extended domain model may contain the following elements:

- Class invariants: intra-object (over attributes of a single instance) or inter-object (over attributes of multiple instances of the same or related classes) constraints defined in a subset of OCL.
- User-defined operations: Operations defined in an Action Semantics-based action language, supplementing the basic CRUD operations (Create, Retrieve, Update and Delete).
- Derived attributes: Attributes whose values are defined by expressions in a subset of OCL, over attributes of self or related instances. A common special case is a reference to a related attribute, using a sequence of dot separated names.
- Default values: Initial attribute values defined in a subset of OCL.
- Derived classes (views): Classes that extend the domain model with non-persistent domain entities with a structure closer to the UI needs. Currently, each derived class must be related to a target base class, and is treated essentially as a virtual specialization of the base class, possibly restricted by a membership constraint and extended with derived attributes.
- Triggers: Actions to be executed before, after or instead of CRUD operations, or when a condition holds within the context of an instance of a class. By defining triggers, the modeller is able to modify the normal behaviour of CRUD operations, or define generic business rules.

The main transformation rules for generating a user interface model from an extended domain model are summarized in Table 1, and extend the rules for transforming simple domain models, previously addressed in (Cruz & Faria, 2008).

When the UIM/UIP is generated solely from the domain model, a special class named System has to be created and linked to the domain classes that should correspond to the application entry points. A more flexible approach is explained in section 5.

4.2 Illustrative example

To illustrate the transformation rules from an extended domain model (EDM) to a user interface model/prototype (UIM/UIP), a *Library System* example will be used. Fig. 3 depicts the extended domain model from our example. In order to be able to identify the application user interface entry points, the EDM must be rooted in a special class named *System*. This is a special class, with no attributes, that aggregates the base or derived entities that shall be directly accessed by the user. Each aggregation from *System* to a base entity class produces a window with a list of instances of the appropriate class, and each aggregation from *System* to a derived entity class produces a window with a list of instances of the derived class' target entity.

Transforming single classes

For each non-abstract entity class (base or derived) with self or inherited attributes, the EDM2UIM model transformer creates a form window. For instance, for the class *Book* (see Figs. 3 and 4), it is created a form with a label and an input field for each class attribute (attribute access modes are not being taken into account). The «ident» stereotype is used to mark attributes that are used for external identification (by the user).

EDM feature	Generated UI feature (UIM/UIP)
Base domain entity	Form with an input/output field for each attribute, and buttons and associated logic for the CRUD operations.
Inheritance	A field for each inherited attribute in the form generated for the specialized class.
To-many association, aggregation or composition	UI component in the source class form, with a list of the identifying attributes (explained in section 4.2) of the related instances of the target class, and buttons for adding new instances and for editing or removing the currently selected instance.
To-one association, aggregation or composition	Group box in the source class form, with a field for each identifying attribute of the related instance. If the related instance is not fixed by the navigation path followed so forth, then a button is also generated for selecting the related instance.
Enumerated type	Group of radio buttons for selecting one option.
Class invariant	Validation rule that is called when creating or updating instances of the class.
User-defined operation	Button and associated logic, within the form corresponding to the class where the operation is defined. Forms are also generated for entering the input parameters and displaying the result, in case they exist. The operation pre-condition determines when the button is enabled.
Derived attribute	Output-only field (calculated field).
Default value	Initial field value.
Derived entity (view)	Form with an input/output field for each attribute of the target class, an output-only field for each derived attribute, and buttons for the CRUD logic (over the target class).
Operation-Action Trigger	Logic that is executed before, after or instead of the CRUD operation that it refers to.
Condition-Action Trigger	Logic that is executed every time the condition holds, after creating or updating an instance of the class where the trigger is defined.

Table 1. EDM to UIM/UIP transformation rules.

In this example (see Fig. 4), to navigate to the *Book* window, the user has to select the *Book Collection* option in the *System* (root) window, and then press the *Add Book* button (to create a new instance), or select a *Book* instance and then press the *Edit Book* button (to view, update or delete an existing instance). In the first case, the user will have to fill in the appropriate fields, press the *Create/Update* button and then close the window or continue editing. In the second case, the user can update the relevant fields and press the *Create/Update* button to submit the changes, or press the *Delete* button to delete the instance and then close the window.

When a new or updated instance is submitted, it is checked that the values entered in the fields obey their declared data types, the identifying attributes (marked with the «ident» stereotype) are filled in, and the invariant constraints are satisfied.

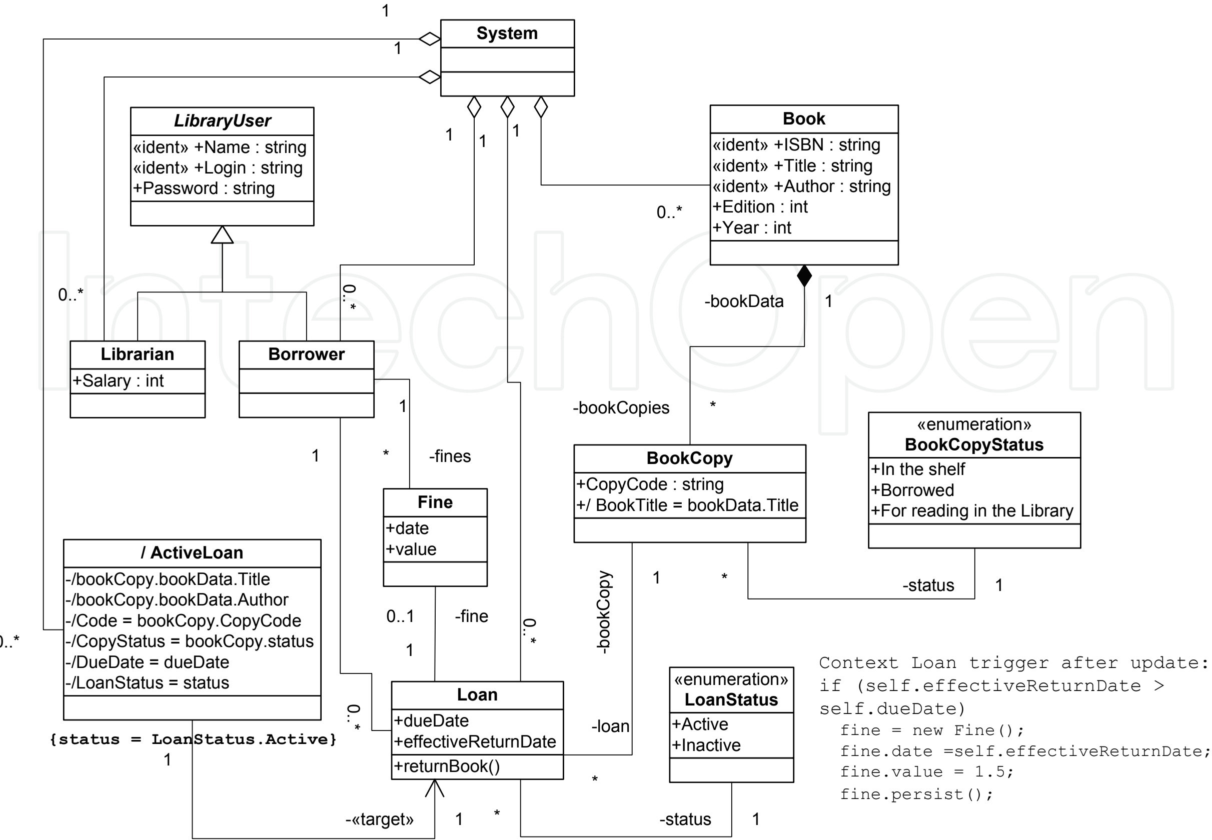


Fig. 3. Extended domain model (EDM) for a Library Management System (*LibrarySystem*), with an example trigger.

Transforming inheritance hierarchies

In our approach, only single inheritance is currently supported, and forms are generated only for the leaf classes of the inheritance hierarchy. Each leaf class inherits all the attributes and constraints from its ancestor classes, and then has the same treatment as single classes.

Transforming associations, aggregations and compositions

For each relationship between two classes, information about related objects and/or links to related objects are generated in each of the corresponding windows. The elements generated depend on the kind of relationship (composition is treated slightly differently from aggregation or association), its multiplicity (to-one and to-many are treated differently), and the navigation path followed. The information that is shown about related objects is the value of the identifying attributes (marked with the «ident» stereotype). If no attribute is marked with the «ident» stereotyped, all the attributes are considered identifying attributes. Role names are used to group the identifying attributes in the form generated. If a role name is not provided, it is used the class name.

In Fig. 4 the UI elements generated from the EDM's classes *Book* and *BookCopy*, and from the composition relationship between them, can be seen. The *Book* window presents a list of related *BookCopy* instances, and a set of buttons for editing (viewing or updating) or removing a previously selected instance, or adding a new instance. The *BookCopy* is accessed from the *Book* window (to edit or create a *BookCopy* instance), and presents the related *BookData* identified by ISBN, Title and Author, which are external identifiers («ident») in

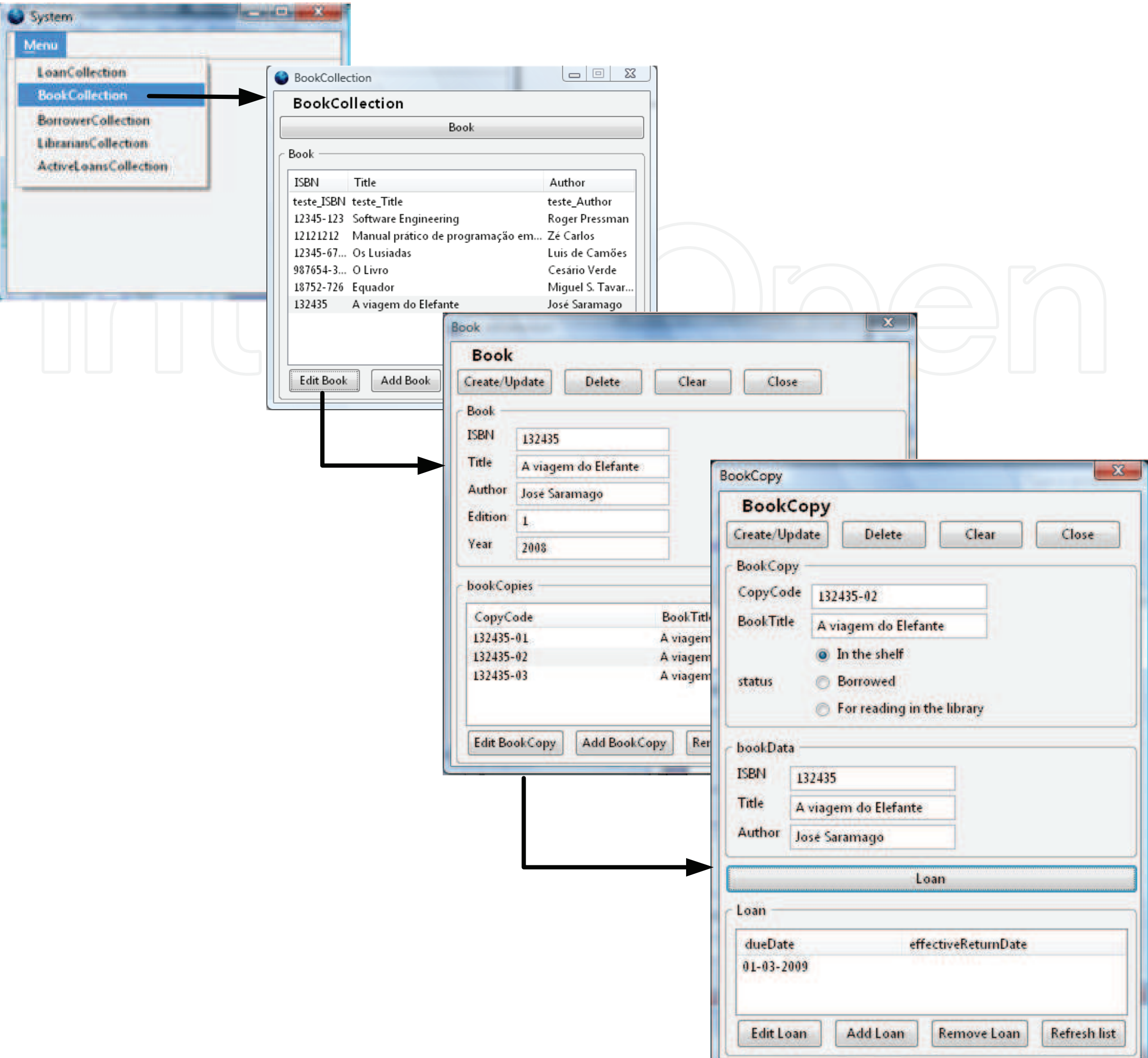


Fig. 4. Excerpt of the application prototype generated from the EDM in Fig. 3.

class *Book*. The *BookCopy* form also has a non-editable output field, BookTitle, generated from its derived attribute with the same name.

In the case of an aggregation or association relationship (instead of a composition relationship), as is the case of the one-to-many association between *BookCopy* and *Loan*, the list of related instances is only shown when requested by the user by pressing an expand/collapse button (see *BookCopy*'s form in Fig. 4).

When one is editing an object that has a related to-one object that is not in the navigation path followed so forth, the user can change the related instance through a *Select* button. This button gives access to a pop-up window with a list of instances (identified by their «ident» attributes), from which one can be selected. For example, the class *Loan* is the "many" side of two one-to-many relations. One can navigate to *Loan* from *BookCopy* or *Borrower* or one can navigate directly to *Loan* from the *System* root class (recall Figs. 3 and 4). Fig. 5 (a) shows the window that appears to the user when navigating to *Loan* directly from the *System* class. In this case, both the borrower that makes the loan and the lent book copy are selectable from the Loan window. Fig. 5 (b) shows the window that appears when navigating from

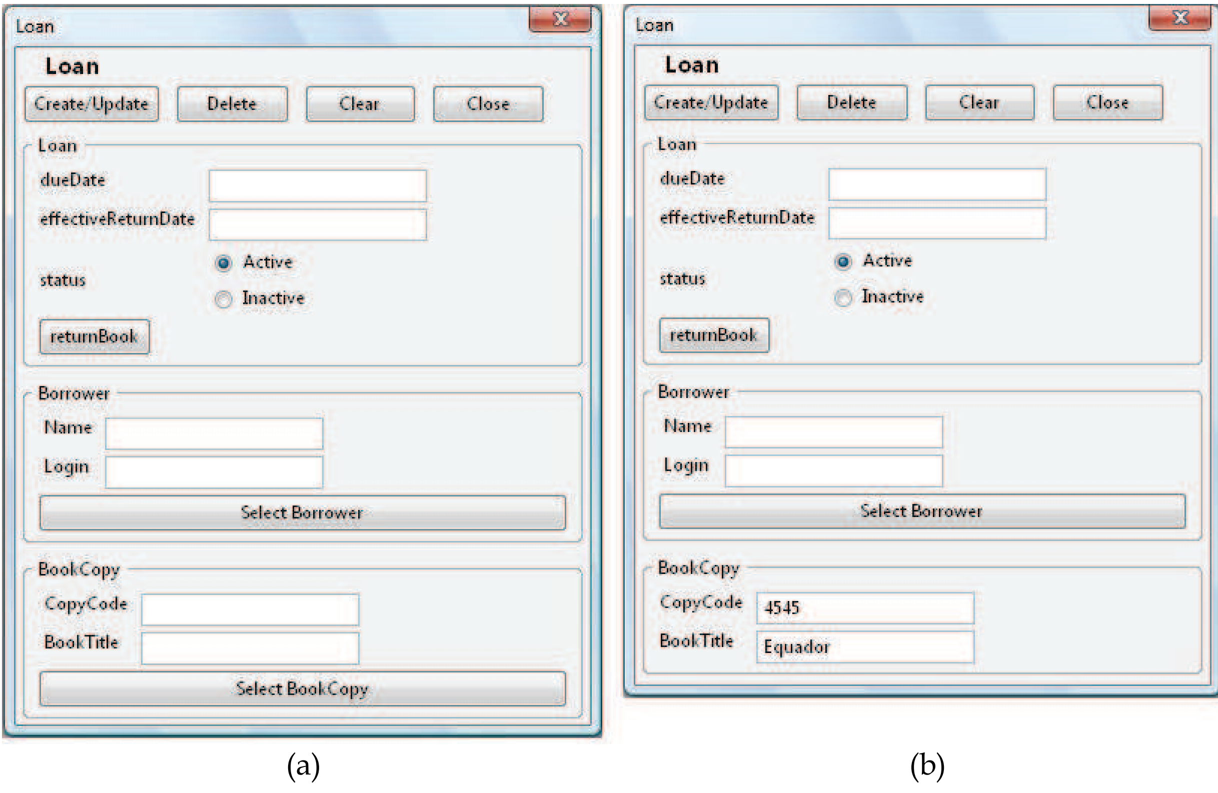


Fig. 5. (a) Window Loan that is shown when navigating directly to an instance of class *Loan*. (b) Window Loan, which is shown when navigating from a *BookCopy* instance to an instance of class *Loan*.

BookCopy to *Loan*. In this case, a given *BookCopy* instance has been previously selected, and thus the "Select BookCopy" button doesn't appear in the *Loan* window, and the field that identifies a book copy shows the referenced book copy. Similarly, when navigating from a borrower instance, the "Select Borrower" button wouldn't appear and the fields that identify a borrower would display the associated borrower.

Handling enumerated types

Enumerated types are defined in the model as classes with an «enumeration» stereotype. In Fig. 3, the UI elements that have origin in a class relation to an «enumeration» class can be seen in the *BookCopy*'s form window. The relation between class *BookCopy* and the enumerated type *BookCopyStatus* generated a list of radio buttons with the enumeration fields, in the *BookCopy* form. The role's name is used as an attribute, and each of the enumerated fields may be selected through a radio button.

Handling constraints

We can identify two kinds of business or domain constraints that may be specified in the domain model: - structural constraints; and, - non-structural constraints. Examples of the former are the multiplicity of the attributes or the uniqueness of classes' keys, and of the latter, are OCL constraints. Each kind of constraints may be further sub-divided into intra-object constraints, applied to attributes within the same object, and inter-object constraints, which may apply to attributes of different objects and/or classes.
The model transformer handles intra- and inter-object constraints, by generating data entry validation functions that are called every time a "Create/Update" button is pressed in the appropriate form.

Constraints may be specified, in the extended domain model, by using an OCL-like abstract language. Constraint expressions may have relational and logical operators, attribute references, constants, etc.

5. Automatic generation of a user interface model from extended domain and use case models

5.1 Use case model and transformation rules

To better allow the configuration of system functionality and enable its differentiation by actor, our approach allows the definition of a use case model (UCM) in close connection with the extended domain model (Cruz & Faria, 2009). This allows the modeller to define and organize the CRUD, user-defined or navigational operations over base or derived domain entities that are available for each actor (user role). The data manipulated in each use case is determined by the domain entity and/or operation associated with it. Several constraints are posed on the types of use cases and use case relationships that can be handled automatically.

Two categories of use cases are distinguished:

- Independent use cases: use cases that can be initiated directly, and so can be linked directly to actors (that initiate them) and appear as application entry points;
- Dependent use cases: use cases that can only be initiated from within other use cases, called source use cases, because they depend on the context set by the source use cases; the dependent use cases extend or are included by the source ones, according to their nature (optional or mandatory, respectively).

The types of independent use cases that can be defined in connection with the EDM are:

- List Entity: view the list of instances of an entity (usually only some attributes, marked as identifying attributes, are shown);
- Create Entity: create a new instance of an entity;
- Call StaticOperation: invoke a static user-defined operation defined in some entity; this includes entering the input parameters and viewing the results, when they exist.

The types of dependent use cases that can be defined in connection with the EDM are:

- Retrieve, Update and/or Delete Entity: view (retrieve) or edit (update or delete) an instance of the entity previously selected (in the source use case);
- Call InstanceOperation: invoke a user-defined operation over an instance of an entity previously selected (in the source use case); this includes entering the input parameters and viewing the results, when they exist;
- List Related Entity: view the list of (0 or more) instances of the target entity that are linked to a previously selected source object (in the source use case); in case of ambiguity, in this and in the next use case types, the link type (association) must also be specified;
- Create Related Entity: create a new instance of the target entity type and link it to a source object previously selected (in the direct or indirect source use case);
- Retrieve, Update and/or Delete Related Entity: view (retrieve) or edit (update or delete and unlink) the instance of the target entity type that is linked with a source object previously selected (in the direct or indirect source use case);
- Select Related Entity: select (and return to the source use case) an instance of the target entity that can be linked to a source object previously selected (in the source use case);
- Select and Link Related Entity: select an instance of the target entity and link it to the source object previously selected (in the source use case);

- Unlink Related Entity: unlink the currently selected instance of the target entity (in the source use case) from the currently selected source object (in the source use case).

The entity, operation(s), and link type (when needed) associated to each use case are specified with tagged-values.

The types of relationships that can be defined among use cases are illustrated in Fig. 6.

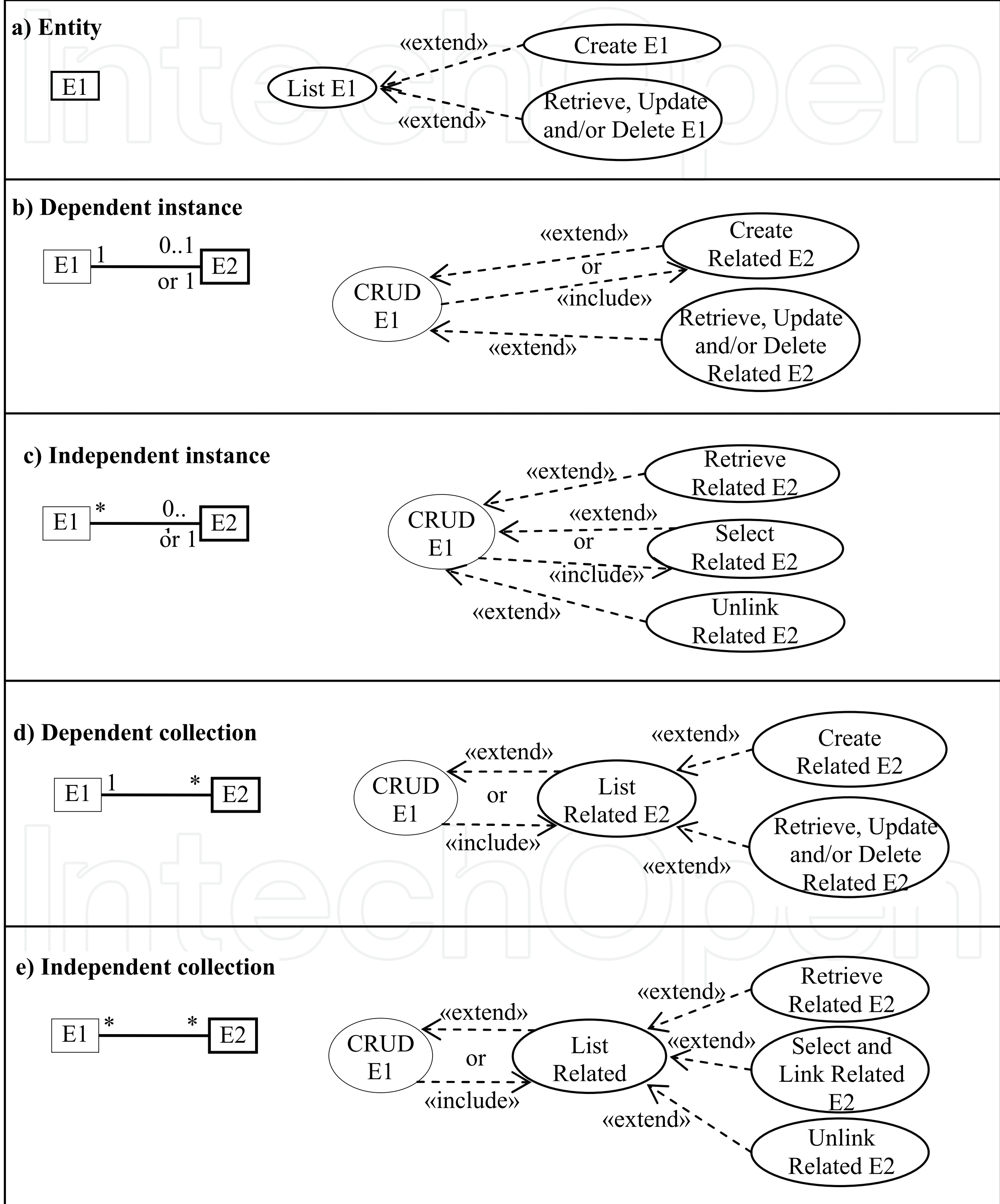


Fig. 6. Possible types of relationships among use cases for different domain model fragments (note: aggregations and compositions are treated similarly to associations).

Table 2 summarizes the rules for generating UI elements from the UCM. Their application is illustrated in the next section.

UCM feature	Generated UI feature (UIM/UIP)
Actor	Button in the application start window, linking to the actor's main window.
Use Case Package	Menu in the actor's main window, with a menu item for each use case that belongs to the package and is directly linked to the actor.
Use Case of type *List Entity* or *List Related Entity*	Form that displays the full list of instances or the list of related instances of the target entity, with buttons for the allowed operations (according to the dependent use cases). Only the identifying attributes are shown.
Use Case of type *Select Related Entity* or *Select and Link Related Entity*	Form that displays the list of candidate instances and allows selecting one instance. Only the identifying attributes are shown.
Use Case of type *CRUD Entity* or *CRUD Related Entity*	Form that displays the object attribute values, with buttons and functionality corresponding to the CRUD operations allowed. In the case of a related instance, the identifying attributes of the source object are shown but cannot be edited.
Use Case of type *Call User-Defined Operation*	Forms for entering and submitting input parameters and presenting output parameters, when they exist.
Extend relationship	Button in the form corresponding to the base use case that gives access to the extension.
Include relationship	If the included use case is of type "*List*...", it is generated a sub-window. Otherwise, it is generated a button in the source use case.

Table 2. UCM to UIM transformation rules.

5.2 Illustrative example

This subsection presents a refinement of the Library System example to illustrate the transformation rules from an extended domain model (EDM) and a use case model (UCM) to a user interface model/prototype (UIM/UIP) (Cruz & Faria, 2009). The constructed EDM is the same presented in section 4 (refer to Fig. 3). Such model has been developed in several iterations; an executable prototype has been automatically generated and tested at the end of each iteration.

After having a partial or complete EDM, the modeller may also develop a UCM. Fig. 7 illustrates an extract of a UCM that was developed for this system. Table 3 shows the entity types and operations associated (via tagged values) with some of the use cases. By applying the mapping rules described previously, the EDM+UCM2UIM process generates a UI model and then an executable prototype, part of which is shown in Fig. 8.

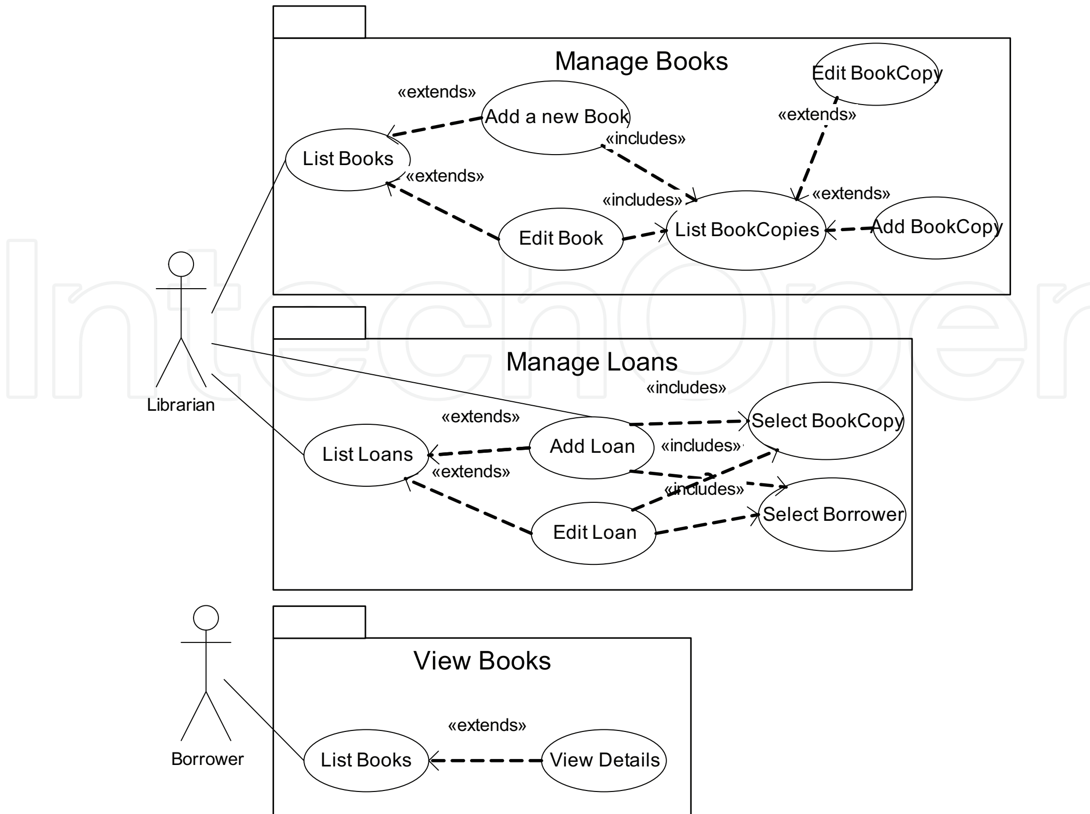


Fig. 7. Partial use case model (UCM) for the Library Management System.

Use case	Entity	Operation(s)
List Books	Book	List
Add a new Book	Book	Create
Edit Book	Book	Update
List BookCopies	BookCopy	List Related
Add BookCopy	BookCopy	Create Related
Edit BookCopy	BookCopy	Update, Delete
List Loans	Loan	List
Add Loan	Loan	Create
Edit Loan	Loan	Update
Select Borrower	Borrower	Select Related
Select BookCopy	BookCopy	Select Related
View Details	Book	Retrieve

Table 3. Entities and operations associated (via tagged values) with some of the use cases in Fig. 7.

Transforming actors, use case packages, and directly accessible use cases

Each actor originates a button in the application start window, and an actor's main window, which is accessed through the actor's selection button in the start window. In our example, the application start window is generated with two buttons for actor selection, "Librarian" and "Borrower". For each use case package where an actor has directly accessible use cases, a menu is generated in that actor's main window, having a menu item available for each

directly accessible use case. For example, the menu generated from the package "Manage Books" (see Fig. 8), has menu item "List Books" generated from the directly accessible use case with the same name.

Transforming use cases of type "List Entity" or "List Related Entity"

Every use case of type "List Entity" or "List Related Entity" is related to a base or derived entity in the extended domain model, and for each of these use cases the model transformer generates a form displaying a full list of instances or the list of related instances of the target domain model's entity. If there are dependent use cases, a button for each one of them is also generated, giving access to the allowed operations from the listing. In our example, "List Books" is a List Entity use case from which the "BookCollection" form has been generated (see Fig. 8). The "BookCollection" form also has buttons "Edit Book" and "Add a New Book" that were generated from the use cases with the same name included in the "List Books" use case.
An example of a List Related Entity is use case "List BookCopies", included in the "Edit Book" and in the "Add a New Book" use cases. In these use cases a Book is previously chosen or is created, setting the context for the next list related use case, that is use case "List BookCopies".

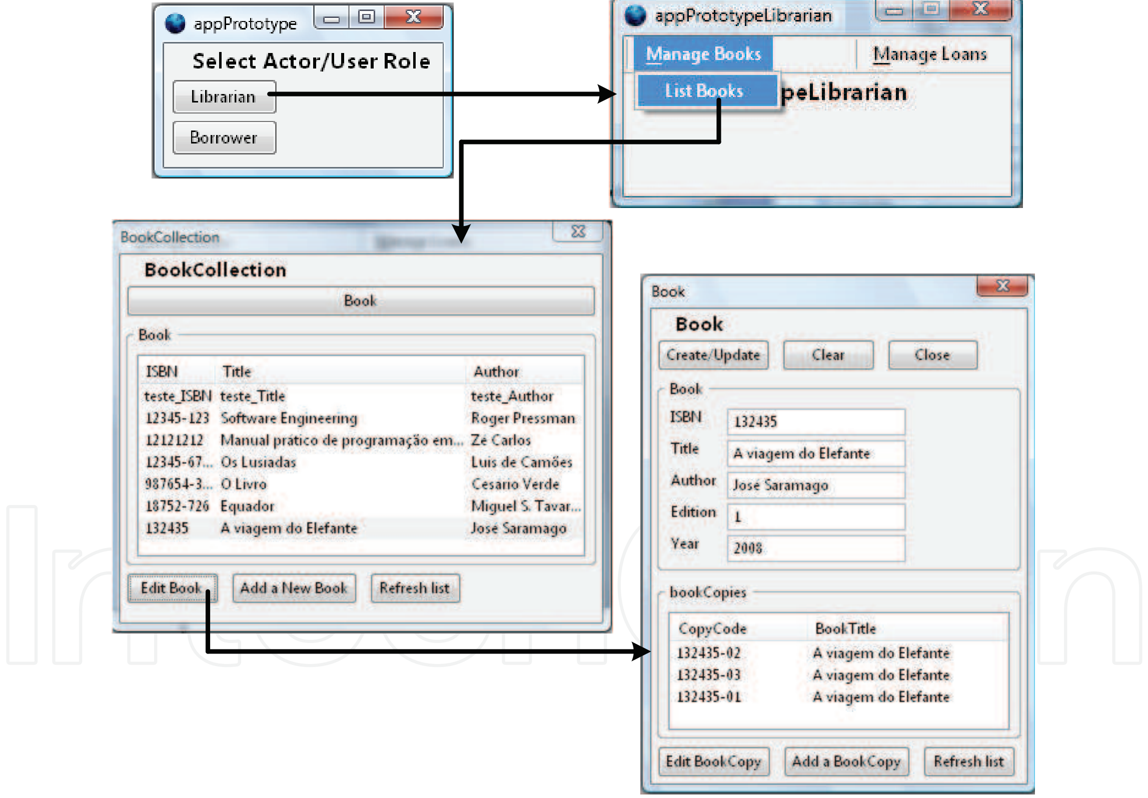


Fig. 8. Excerpt of the application prototype generated for a Librarian executing use cases List Books → Edit Book (that includes List BookCopies).

Transforming use cases of type "CRUD Entity" or "CRUD Related Entity"

Each use case of type "CRUD entity" or "CRUD related entity", that is, use cases that target an entity and a CRUD operation on that entity, generates a form displaying the attributes' values, with buttons and functionality for the CRUD operations allowed. In our example, a

CRUD entity use case is, for instance, use case “Edit Book”, which has associated tagged values Entity = “Book” and Operations = “Update” (see Table 3). An example of a CRUD related entity use case is “Edit BookCopy”.

Transforming use cases of type “Select Related Entity” or “Select and Link Related Entity”

In the LibrarySystem example “Select BookCopy” and “Select Borrower” are use cases of type “Select and Link Related Entity”, where an independent instance of BookCopy or Borrower, respectively, must be associated to an instance of Loan (refer to Fig. 5).
With the use case model, the modeller may choose not to give an actor the possibility to select a different borrower or book copy to loans.

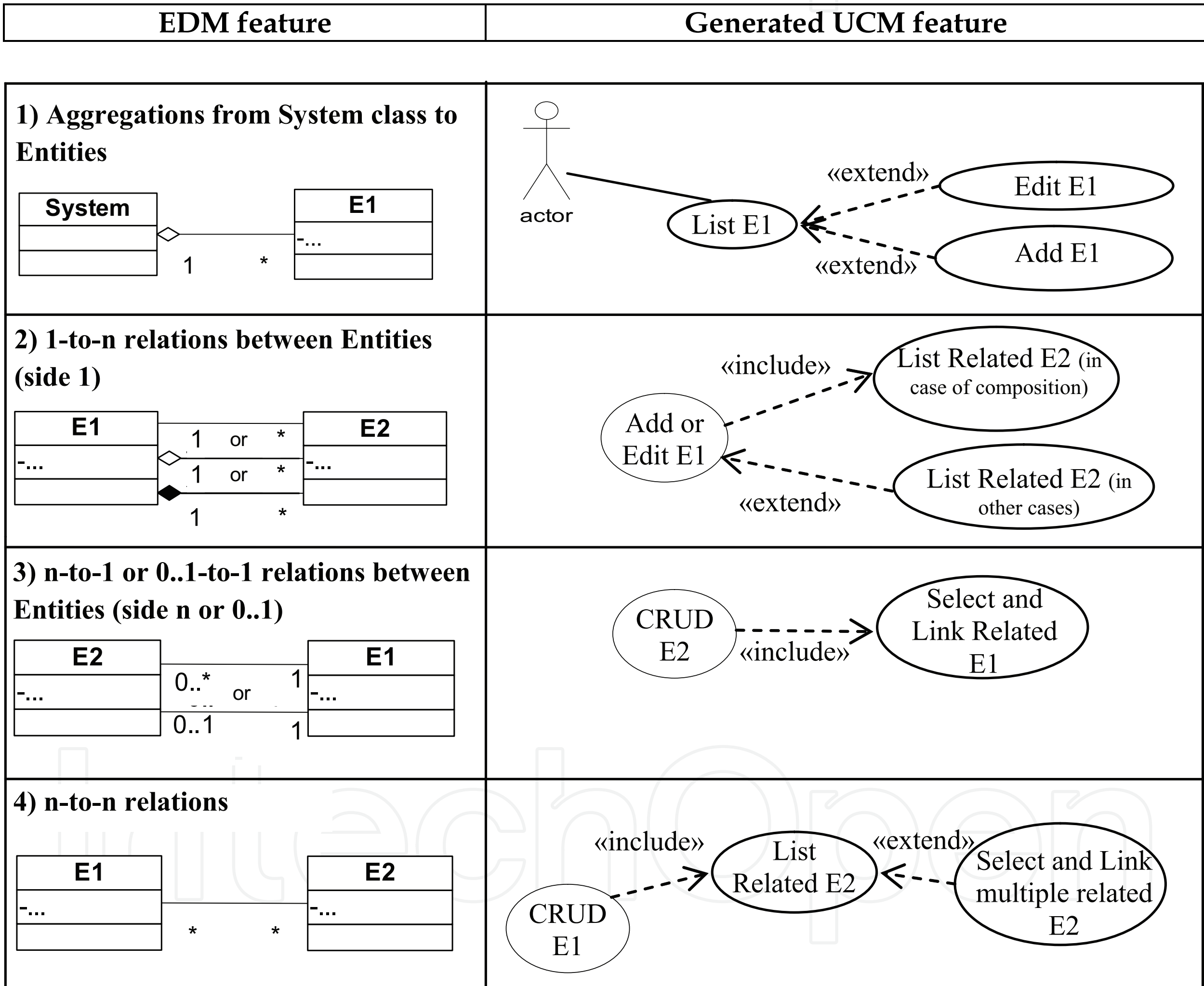


Fig. 9. Use case model fragments automatically derived from EDM’s patterns.

Transforming use cases of type “Call User Defined Operation”

A “Call User Defined Operation” use case generates a button in the form window corresponding to the entity where the operation is defined, and a form for entering parameters and another form for showing the operation’s result, if they exist. In our example, this situation appears in Loan. Class Loan defines operation returnBook, that is transformed to a button in the Loan form window, and a form for entering the operation’s

parameters. Since this operation, defined using an Action Semantics-like abstract language, returns no result, an output form is not generated.
When the operation returns, the entity form is refreshed to be able to show data modified by the operation in the instance's state.

6. Default use case model generation from extended domain model

As stated before, and according to the proposed approach (refer to section 3) a default UCM may be derived from the EDM facilitating the initial construction of the UCM. The default use case model has only one actor that has access to all the system functionality, and may serve as the basis for producing the intended use case model by creating new actors and eliminating or redistributing functions among actors.

Fig. 10. Partial default use case model generated from the EDM in Fig. 3.

Starting from the "System" entity an actor is created, linking to List Entity use cases, one for each aggregation from "system" to another base or derived entity. Fig. 10 partially shows the use case model that is generated by the EDM2UCM model-to-model transformation process.
Each List Entity use case shall have extensions for CRUD use cases (Add and Edit). A CRUD use case shall include use cases that list related entity instances. In Fig. 10, see, for example, use case "List Books" that links to the only actor and is extended by "Add Book" and "Edit Book". These last two use cases, that allow CRUD operations over Book, include use case "List Related BookCopies", which in turn is extended by use cases for adding and editing a book copy.

	XIS	**OO-Method**	**ZOOM**	**Elkoutbi et al./ Martinez et al.**	**Forbrig et al.**	**Our approach**
Is able to generate a fully functional interactive prototype	✓	✓	✓	---	---	✓
Requires/generates a UIM as a step for obtaining a concrete UI	Requires/ generates	Requires	Requires	Generate only UI state model	Requires	Generates / allows configura-tion
Is able to generate a UIM/UIP from non-UI system models	✓ (in smart approach)	(only from domain model)	---	✓ (non functional UIP)	---	✓
Is able to generate a UIM/UIP from domain model alone	---	✓	---	---	---	✓
Is able to generate a UIM/UIP from domain model + use case model	✓ (in smart approach)	---	---	---	---	✓
Allows the definition of triggers	---	✓ (partial)	---	---	---	✓
Assumes CRUD operations	✓	✓	---	---	---	✓
Generates code for user defined operations	---	✓	✓	---	---	✓
Takes advantage of formal constraints to generate features in the UI	---	✓ (partial)	---	---	---	✓

Table 4. Feature comparison between the current approaches and the proposed approach.

7. Results and contributions to the state of art

This section compares the presented approach to the ones surveyed in section 2, and discusses its similarities and distinguishing features. In table 4 a feature comparison between the current approaches, presented in section 2, and the approach proposed in this document is presented.

Unlike XIS, our approach doesn't demand the stereotyping of every model element, as the full model package is submitted to the transformation process.

XIS business entities are similar to our derived entities. Like in the XIS smart approach, the modeller must attach to each use case an Entity (base or derived) from the EDM. The difference is that, in our approach, relations between entities are inferred from the EDM, thus not being needed a separate business entities model to provide higher level entities to the UCM. The relation's selection provided by the XIS business entities model can be done, within our approach, in the UCM by modelling use cases for navigating only through the admitted relations.

Similarly to XIS and the OO-Method, in our approach CRUD operations are predefined.

In our approach user defined operations may be specified using an UML Action Semantics-based language.

Just like our approach, the OO-Method allows the definition of derived attributes, by assigning a calculation formula to the attributes.

So, the main contributions of the proposed approach, to the state of art are:

- To make possible to generate an application prototype from an incomplete system domain model or extended domain model;
- To make use of derived attributes and derived entities (views), in the EDM, to better specify "boundary" entities;
- To take advantage of class invariants and operation pre-conditions to generate validation routines in the generated application, enabling the enhancement of the usability of the generated UI by helping the user in entering valid data into forms, and by giving feedback identifying invalid data, or by disabling an operation's start button while its pre-condition doesn't hold;
- To make use of an action language to specify the semantic of operations at class level, and enable the definition of triggers activated either by the invocation of a CRUD operation or by the holding of a given state condition;
- To allow the usage of a use case model to specify several actors, or user profiles, enabling the hiding of possible functionality from some of the users;
- To derive a default use case model from an extended domain model, easing the process of developing a use case model integrated with the system EDM.

8. Conclusions and future work

The presented approach enables a gradual approximation to system modelling towards business forms-based applications, by being able to derive a default UI and an executable prototype from a domain model alone, an extended domain model or from an extended domain model and a use case model. It is also possible to have these initial models in different levels of abstraction or rigour, and refine them in an incremental and iterative manner.

As depicted in section 3, this approach is able to generate a UI model and prototype from the system's non-UI submodels, helping the modeller in creating a system model and facilitating the process of developing a UI for the final interactive system. The approach derives a default UI and an executable prototype from the system model, which comprises a domain model or extended domain model and, optionally, a use case model. This approach turns possible to interactively evaluate the system model with the end users, and to iteratively evaluate and refine the model. It also allows adding rigour and model elements to the system model, generating more complete, richer and refined UIs and executable prototypes that support an evolutionary model-driven development with the close participation of the end users.

Several benefits can be drawn from using the presented approach, as discussed in the previous section. Nevertheless, more results can be obtained with future work, namely in what concerns the flexibility of the generated UI.

The next step will be to support use case relations that recall HCI's task models, by properly stereotyping use case relations with «enables», «deactivates» or «choice», which allow the definition of use cases that are enabled by the execution of other use cases, use cases that are disabled by the execution of other use cases, and alternative use cases, respectively.

Another future development is the support for use cases that are not associated to an EDM class or class method, but may be associated to a given class attribute. This kind of use cases, together with the properly stereotyped use case relations, allows the modeller to define wich set of attributes must be set first, and which depend on other attributes, or are deactivated by setting other attributes.

This evolution of the proposed approach enables a higher degree of refinement in the use case model definition, allowing for greater flexibility in the generated UI model.

Other foreseen developments are the existence of use cases not directly associated to the EDM. This are parameterized use cases that collect information for session variables, and that must be aggregated, through «include» relations, in another use case that has access to all subordinate session variables. The aggregator use case is, then associated to an EDM operation binding session variables to the operation's parameters. Without loosing the tigh relation between use case model and extended domain model, this will enable the highest degree of flexibility in the use case model definition in order to better define what one wants to see generated in the UI model.

9. References

Cruz, A.M.R., Faria, J.P. (2007). Automatic generation of user interfaces from domain and use case models. In *Proceedings of the Sixth International Conference on the Quality of Information and Communication Technology* (QUATIC 2007), pp 208-212, Lisboa, Portugal, September 2007, IEEE.

Cruz, A.M.R., Faria, J.P. (2008). Automatic generation of interactive prototypes for domain model validation. In *Proceedings of the 3rd International Conference on Software Engineering and Data Technologies* (ICSoft 2008), vol. SE/GSDCA/MUSE, pp 206-213, Porto, Portugal, July 2008, INSTICC Press.

Cruz, A.M.R., Faria, J.P. (2009). Automatic generation of user interface models and prototypes from domain and use case models. In *Proceedings of the 4th International*

Conference on Software Engineering and Data Technologies (ICSoft 2009) , vol. 1, pp 169-176, Sofia, Bulgaria, July 2009, INSTICC Press.

Dix, A., Finlay, J., Abowd, G., Beale, R. (1998). *Human-Computer Interaction*. Prentice Hall, 2nd edition.

Elkoutbi, M.; Khriss, I.; Keller, R.K. (2006). Automated prototyping of user interfaces based on UML scenarios. *Journal of Automated Software Engineering*, 13(1):5-40, January.

Forbrig, P., Dittmar, A., Reichart, D., Sinnig, D. (2004). From models to interactive systems tool support and XIML. *In Proceedings of the First International Workshop MBUI 2004*, vol. 103-CEUR Workshop Proceedings, Funchal, Portugal. Available at http://ceur-ws.org.

Jacobson, I., Booch, G., Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

Jia, X., Steele, A., Liu, H., Qin, L., Jones, C. (2005). Using ZOOM approach to support MDD. *In Proceedings of the 2005 International Conference on Software Engineering Research and Practice* (SERP'05), Las Vegas, USA.

Jia, X., Steele, A., Qin, L., Liu, H., Jones, C. (2007). Executable visual software modelling - the ZOOM approach. Software Quality Control, 15(1):27-51.

Javahery, H., Sinnig, D., Seffah, A., Forbrig, P., Radhakrishnan, T. (2007). Task Models and Diagrams for Users Interface Design, chapter Pattern-Based UI Design: Adding Rigor with User and Context Variables, pages 97-108. Lecture Notes in Computer Science. Springer Berlin/Heidelberg.

Kelly, S., Tolvanen, Juha-Pekka (2008). *Domain Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press.

Kleppe, A., Warmer, J., Bast, W. (2003). *MDA Explained – The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional.

Martinez, A., Estrada, H., Sánchez, J., Pastor, O. (2002). From early requirements to user interface prototyping: A methodological approach. *In Proceedings of the 17th IEEE International Conference on A.S.E.*, pp 257-260.

Molina, P., Pastor, O., Marti, S., Fons, J., Insfrán, E. (2001). Specifying conceptual interface patterns in an object-oriented method with automatic code generation. *In Proceedings Second International Workshop on User Interfaces in Data Intensive Systems*, UIDIS 2001.

Molina, P.J., Hernández, J. (2003). Just-UI: Using patterns as concepts for IU specification and code generation. In Perspectives on HCI Patterns: Concepts and Tools (CHI'2003 Workshop).

Molina, P.J. (2004). User interface generation with Olivanova model execution system. In IUI '04: Proceedings of the 9th International Conference on Intelligent User Interfaces, pages 358-359, NY, USA. ACM.

Pastor, O., Insfrán, Pelechano, V., Romero, J., Merseguer, J. (1997). OO-METHOD: An OO software production environment combining conventional and formal methods. *In* CAiSE '97: *Proceedings of the 9th International Conference on Advanced Information Systems Engineering*, pages 145-158, London, UK. Springer-Verlag.

Pastor, O., Insfrán, E. (2003). OO-Method, the methodological support for OlivaNova model execution system. Technical report, Care Technologies. White paper. Available at http://www.care-t.com.

Pastor, O., Molina, J. (2007). *Model-driven Architecture in Practice – A software production environment based on Conceptual Modeling*. Springer-Verlag.

Pastor, O., Molina, J., Iborra, E. (2004). Automated production of fully functional applications with Olivanova model execution. ERCIM News No. 57, April 2004. Available at http://www.ercim.org/publication/Ercim News/enw57/pastor.html.

Paternó, F., 2001. Task Models in Interactive Software Systems. In *Handbook of Software Engineering and Knowledge Engineering, volume I, 2001.* World Scientific Publishing Co. Pte. Ltd., pp. 817–835.

Pinheiro da Silva, P., 2000. User interface declarative models and development environments: A survey. In *Interactive Systems - Design, Specification, and Verification: 7th International Workshop, DSV-IS 2000, Limerick, Ireland, June 2000. Revised Papers*, Springer Berlin / Heidelberg, *Lecture Notes in Computer Science* vol. 1946, pp. 207–226.

Pressman, R. S., 2005. *Software Engineering – A practitioner's approach*, 6th edition. Mc Graw Hill.

Reichart, D., Forbrig, P., Dittmar, A. (2004). Task models as basis for requirements engineering and software execution. *In Task Models and Diagrams for User Interface Design* TAMODIA, pages 51-58.

Radeke, F., Forbrig, P., Seffah, A., Sinnig, D. (2007). PIM Tool: Support for pattern-driven and model-based UI development. *In Task Models and Diagrams for User Interface Design* (TAMODIA 2006), volume 4385/2007 of Lecture Notes in Computer Science, pages 82-96. Springer Berlin/Heidelberg.

Saraiva, J., Silva, A. (2008). The ProjectITStudio. UMLModeler: A tool for the design and transformation of UML models. *In Proceedings of the 3rd Iberian Conference of Information Technologies and Systems* (CISTI 2008), Campus de Ourense, Ourense, Spain, Universidad de Vigo.

Silva, A. (2003). The XIS approach and principles. *In Proceedings of the 29th* EUROMICRO *Conference "New Waves in System Architecture"* (EUROMICRO '03), IEEE Computer Society.

Silva, A., Videira, C. (2008). *UML, Metodologias e Ferramentas CASE*, vol. 2 (in portuguese). Centro Atlântico, 2nd ed.

Silva, A.R., Saraiva, J., Silva, R., Martins, C. (2007). XIS - UML profile for extreme modeling interactive systems. *In Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software* (MOMPES 2007). IEEE, March.

Wolff, A., Forbrig, P., Dittmar, A., Reichart, D. (2005a). Linking GUI elements to tasks: supporting an evolutionary design process. *In Proceedings of the 4th International workshop on Task Models and Diagrams for User Interface Design* (TAMODIA '05), pages 27-34, New York, NY, USA, 2005. ACM.

Wolff, A., Forbrig, P., Reichart, D. (2005b). Tool support for model-based generation of advanced user-interfaces. In Andreas Pleuss, Jan Van den Bergh, Heinrich Hussmann, and Stefan Sauer, editors, *In Proceedings of the* MoDELS'05 *Workshop on Model Driven Development of Advanced User Interfaces*, Montego Bay, Jamaica, October.

User Interfaces

Edited by Rita Matrai

ISBN 978-953-307-084-1

Hard cover, 270 pages

Publisher InTech

Published online 01, May, 2010

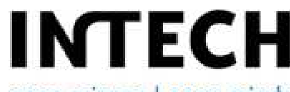
Published in print edition May, 2010

Designing user interfaces nowadays is indispensably important. A well-designed user interface promotes users to complete their everyday tasks in a great extent, particularly users with special needs. Numerous guidelines have already been developed for designing user interfaces but because of the technical development, new challenges appear continuously, various ways of information seeking, publication and transmit evolve. Computers and mobile devices have roles in all walks of life such as in a simple search of the web, or using professional applications or in distance communication between hearing impaired people. It is important that users can apply the interface easily and the technical parts do not distract their attention from their work. Proper design of user interface can prevent users from several inconveniences, for which this book is a great help.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Antonio Miguel Rosado da Cruz and Joao Pascoal Faria (2010). Automatic Generation of User Interface Models and Prototypes from Domain and Use Case Models, User Interfaces, Rita Matrai (Ed.), ISBN: 978-953-307-084-1, InTech, Available from: http://www.intechopen.com/books/user-interfaces/automatic-generation-of-user-interface-models-and-prototypes-from-domain-and-use-case-models



InTech Europe

University Campus STeP Ri

Slavka Krautzeka 83/A

51000 Rijeka, Croatia

Phone: +385 (51) 770 447

Fax: +385 (51) 686 166

www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai

No.65, Yan An Road (West), Shanghai, 200040, China

中国上海市延安西路65号上海国际贵都大饭店办公楼405单元

Phone: +86-21-62489820

Fax: +86-21-62489821

© 2010 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.